CSIE 3110: Formal Languages and Automata Theory

Tony Tan
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Table of contents

Lesson 0: Preliminaries

Theme: Some introductory material.

1 Problems impossible for computers

Consider the following problem, which we denote by Problem-A.

Problem-A	
Input:	Two files:
	The first file is a C++ program, denoted by `file-1.cpp`.
	The second file is a file with arbitrary extension, denoted by `file-2`.
Task:	Output `True`, if the C++ program file-1.cpp returns `True`
	when the input (to `file-1.cpp`) is the content of `file-2`.
	Otherwise, output `False`.

We will show that it is impossible to write a C++ program for Problem-A. Before we proceed, we consider the following two simpler problems.

Problem-B	
Input:	One file, a C++ program, denoted by `program.cpp`.
Task:	Output `True`, if the C++ program `program.cpp` returns `True`
	when the input is itself, i.e., the content of `program.cpp`.
	Otherwise, output `False`.

Problem-C	
Input:	One file, a C++ program, denoted by `program.cpp`.
Task:	Output `False`, if the C++ program `program.cpp` returns `True`
	when the input is itself, i.e., the content of `program.cpp`.
	Otherwise, output `True`.

Note that Problem-C is just the "negation" of Problem-B, thus, they are computationally "equivalent" in the sense that if we can write a C++ program for Problem-B, we can write another C++ program for Problem-C. Likewise, if we can write a C++ program for Problem-C, we can write another C++ program for Problem-B. On the other hand, Problem-A is computationally "more general" than Problem-B and Problem-C in the sense that if we can write a a C++ program for Problem-A, we can write another C++ program for Problem-B or for Problem-C.

We can show the following.

Theorem 0.1 *There is no C++ program for Problem-C. Therefore, there is no C++ program for Problem-B and Problem-A, as well.*

Proof. Suppose to the contrary that there is a C++ program for Problem-C, which we denote by `myprog.cpp`. We run `myprog.cpp` with input `myprog.cpp` itself and consider the output. There are two possibilities

- The output is `True`.

  Since `myprog.cpp` is a program for Problem-C, by definition of Problem-C, `myprog.cpp` does not return `True` on `myprog.cpp` itself. Thus, we arrive at a contradiction.

- The output is `False`.

  Since `myprog.cpp` is a program for Problem-C, by definition of Problem-C, `myprog.cpp` returns `True` on `myprog.cpp` itself. Again, we arrive at a contradiction.

In both cases, we arrive at a contradiction. Thus, there is no such C++ program `myprog.cpp` for Problem-C. ▮

Remark 0.2 Note that in the proof of Theorem 0.1 we do not use any fact about C++ program itself. The non-existence of C++ program for Problem-C is established by pure logic. So we can redefine all the problems Problem-A, Problem-B and Problem-C in terms of any other programming languages, and show that one can not possibly write a computer program (in that particular language) for any of them.

2 The notion of alphabets and languages

In this course we assume familiarity with basic terminology from discrete mathematics. See Appendix A. In addition, we will use the following terminology.

- An *alphabet* is a finite set of symbols. We usually use the symbol Σ to denote an alphabet.
- A (finite) string/word over Σ is a finite sequence of symbols from Σ.
- We will usually write $w = a_1 \dots a_n$ to denote a word whose label in position i is a_i. The length of w is denoted by $|w|$.
- We write ε to denote the *empty string/word*, i.e., the word of length 0.
- For an integer $n \geqslant 0$, Σ^n denotes the set of all the words over Σ of length n.
- Σ^* denotes the set of all *finite* words over Σ, i.e., $\Sigma^* = \bigcup_{n\geqslant 0} \Sigma^n$.
- A *language* L over Σ is a subset of Σ^*.

Note that a computer program (say, in C++) can be viewed as a string. It is important to notice also that the length of a computer program can only be finite, albeit it can be a very long string.

Appendix

A Reviews of some basic terminology from discrete mathematics

Set-theoretic terminology:

- A *set* is a collection of things, which are called its members or elements.

  $a \in X$ (read: a is in X, or a belongs to X) means a is a member or an element of X, whereas $a \notin X$ means a is not a member of X.

- We usually write a set X as $X = \{a | a \text{ satisfies some property } P\}$, which we read as "$X$ is the set of all elements a that satisfy property P."

  Sometimes, we will also write $X = \{a \in U | a \text{ satisfies some property } P\}$ to specify that the set X only contains elements from some set U. In this case, we read it as "X is the set of all elements in U that satisfy property P."

  For example, $P = \{a \in \mathbb{N} | a \text{ is a prime number}\}$ denotes the set of all prime numbers.

- The empty set is denoted by $\emptyset$, i.e., a set that does not contain anything.

- X is a *subset* of Y, denoted by $X \subseteq Y$, if every element of X is also an element of Y.

  X is a proper subset of Y, denoted by $X \subsetneq Y$, if $X \neq Y$ and $X \subseteq Y$.

- For two sets X and Y, we write $X \cap Y$ and $X \cup Y$ to denote their intersection and union, respectively.

- The cartesian product between two sets X and Y is the following.

$$X \times Y \quad := \quad \{(a, b) \mid a \in X \text{ and } b \in Y\}.$$

  We write X^n to denote $X \times \cdots \times X$, where X appears n time.

Relations and functions:

- A *relation* R from a set X to another set Y is a subset of $X \times Y$.

- A *binary relation* R over X is a subset of $X \times X$.

- An *n-ary relation* R over X is a subset of X^n.

- A relation R from X to Y is a *function* or a *mapping*, if for every $x \in X$, there is exactly one $y \in Y$ such that $(x, y) \in R$.

  In this case, we will say R is a function from X to Y, or R maps X to Y. We denote it by $R : X \to Y$.

- We usually use the letters $f, g, h, \ldots$ to represent functions. As usual, we write $f(x)$ to denote the element y in which $(x, y) \in f$.

- A function $f : X \to Y$ is an *injective* function, if for every $y \in Y$, there is at most one $x \in X$ such that $f(x) = y$. An injective functions is also called an *injection.*

- A function $f : X \to Y$ is a *surjective* function, if for every $y \in Y$, there is at least one $x \in X$ such that $f(x) = y$.

- A function $f : X \to Y$ is a *bijection*, if it is both injective and surjective.

B The use of quantifiers in mathematical statements

In this course it is important to be able to read mathematical/formal statements. It will take a while to get used to them. One important aspect of a formal statement is its use of "quantifiers."

Consider the following statement.

$$\text{Every student stays in a dormitory room.} \tag{1}$$

If we want to write in strict logical form, we will have to write it in the following way.

$$\underline{\text{For every}} \text{ student } x, \underline{\text{there is}} \text{ a dormitory room } y \text{ such that } x \text{ stays in } y.$$

"For every" and "there exists" in the above sentence are called quantifiers.

The negation of statement (1) is:

$$\underline{\text{There is}} \text{ student } x, \text{ such that } \underline{\text{for every}} \text{ dormitory room } y, \ x \textit{ does not} \text{ stay in } y. \tag{2}$$

Note also that neither (1) nor (2) are equivalent to the following sentence:

$$\underline{\text{There is}} \text{ student } x, \text{ such that } \underline{\text{for every}} \text{ dormitory room } y \text{ where } x \textit{ stays} \text{ in } y. \tag{3}$$

Some examples of deductions involving quantifiers. Suppose we know that the following is true:

$$\text{For every student } x, \text{ if } x \text{ is not from Taipei, then } x \text{ lives in dormitory.} \tag{4}$$

Now, suppose we also know that:

$$\text{John is a student, but he does not live in dormitory.} \tag{5}$$

From (4), we can deduce that John is from Taipei.

As another example, suppose we know that:

$$\text{Bob is a student and he is from Taipei.} \tag{6}$$

Can we conclude from (4) that Bob does not live in dormitory? No! It is because (4) does not give any us information about students from Taipei. It only gives us information about students not from Taipei.

Consider another example:

$$\text{Charlie is not from Taipei.} \tag{7}$$

Can we conclude from (4) that Charlie lives in dormitory? No! It is because (4) only gives us information about students. Note that (7) does not tell us whether Charlie is a student.

Lesson 1: Finite state automata

Theme: Deterministic and non-deterministic finite state automata.

1 Deterministic finite state automata

A *deterministic finite state automaton* (DFA) is a system $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$, where each component is as follows.

- Σ is an alphabet.
- Q is a *finite* set of states.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of *accepting* states.
- $\delta : Q \times \Sigma \to Q$ is the *transition* function.

In this case, we will say that "*$\mathcal{A}$ is a DFA over alphabet Σ*," or that "*the alphabet of $\mathcal{A}$ is Σ*."

Remark 1.1 A DFA $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ can be visualised as a directed graph where the vertices are elements of Q and there is an edge from p to p' labeled with a, if $\delta(p, a) = p'$. ▮

On input word $w = a_1 \cdots a_n$, the *run of $\mathcal{A}$ on w* is the sequence:

$$p_0 \; a_1 \; p_1 \; a_2 \; p_2 \; \cdots \; a_n \; p_n,$$

where $p_0 = q_0$ and $\delta(p_i, a_{i+1}) = p_{i+1}$, for each $i = 0, \dots, n-1$.

Sometimes we are interested in a run that does not start from the initial state. In that case, we can define the *run of $\mathcal{A}$ on w starting from state q* as the sequence defined as above, but with condition $p_0 = q$. That is,

$$p_0 \; a_1 \; p_1 \; a_2 \; p_2 \; \cdots \; a_n \; p_n,$$

where $p_0 = q$ and $\delta(p_i, a_{i+1}) = p_{i+1}$, for each $i = 0, \dots, n-1$.

A run is called an *accepting* run, if $p_0 = q_0$ and $q_n \in F$. We say that $\mathcal{A}$ *accepts* w, if there is an accepting run of $\mathcal{A}$ on w. The language of all words accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

A language L is called a *regular* language, if there is a DFA $\mathcal{A}$ such that $L(\mathcal{A}) = L$.

Remark 1.2 Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a DFA.

- The empty string ε is accepted by $\mathcal{A}$ if and only if $q_0 \in F$.
- For every word w, there is exactly *one* run of $\mathcal{A}$ on w. ▮

Theorem 1.3 *Regular languages are closed under boolean operations, i.e., intersection, union, and complement. More formally, it can be stated as follows.*

- *For every DFA $\mathcal{A}$ over alphabet Σ, there is a DFA $\mathcal{A}'$ over the same alphabet Σ such that $L(\mathcal{A}') = \Sigma^* - L(\mathcal{A})$.*
- *For every two DFA $\mathcal{A}_1$ and $\mathcal{A}_2$, there is a DFA $\mathcal{A}'$ such that $L(\mathcal{A}') = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.*
- *For every two DFA $\mathcal{A}_1$ and $\mathcal{A}_2$, there is a DFA $\mathcal{A}'$ such that $L(\mathcal{A}') = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$.*

Proof. (Closure under complement) Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a DFA. Consider the DFA $\mathcal{B} = \langle \Sigma, Q, q_0, Q - F, \delta \rangle$. That is, $\mathcal{B}$ is exactly the same as $\mathcal{A}$ with the difference only in the accepting states, where the accepting states in $\mathcal{A}$ become non-accepting in $\mathcal{B}$ and the non-accepting states in $\mathcal{A}$ become accepting in $\mathcal{B}$.

Obviously, for every word $w \in \Sigma^*$, the accepting run of $\mathcal{A}$ on w becomes non-accepting run of $\mathcal{B}$ on w. Vice versa, the non-accepting run of $\mathcal{A}$ on w becomes accepting run of $\mathcal{B}$ on w. Thus, $L(\mathcal{B}) = \Sigma^* - L(\mathcal{A})$.

(Closure under intersection) Let $\mathcal{A}_1 = \langle \Sigma, Q_1, q_{0,1}, F_1, \delta_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, q_{0,2}, F_2, \delta_2 \rangle$ be DFA. Consider the following DFA $\mathcal{B} = \langle \Sigma, Q, q_0, F, \delta \rangle$, where:

- $Q = Q_1 \times Q_2$.
- The initial state q_0 is $(q_{0,1}, q_{0,2})$.
- $F = F_1 \times F_2$.
- The transition function δ is defined as follows. For every $(p_1, p_2) \in Q_1 \times Q_2$, for every $a \in \Sigma$,

$$\delta((p_1, p_2), a) \quad = \quad (\delta_1(p_1, a), \delta_2(p_2, a))$$

We will show that $L(\mathcal{B}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. First, we show that $L(\mathcal{B}) \subseteq L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. Consider a word $w = a_1 \cdots a_n$, where each $a_i \in \Sigma$. Suppose $w \in L(\mathcal{B})$, and we denote its accepting run by:

$$(s_0, t_0)\ a_1\ (s_1, t_1)\ a_2\ (s_2, t_2)\ \cdots\ a_n\ (s_n, t_n).$$

By the definition of accepting run and the definition of $\mathcal{B}$,

$$s_0\ a_1\ s_1\ a_2\ s_2\ \cdots\ a_n\ s_n \quad \text{and} \quad t_0\ a_1\ t_1\ a_2\ t_2\ \cdots\ a_n\ t_n$$

are accepting runs of $\mathcal{A}_1$ and $\mathcal{A}_2$ on w, respectively, and hence, $w \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

Now, we show that $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) \subseteq L(\mathcal{B})$. Consider a word $w = a_1 \cdots a_n$, where each $a_i \in \Sigma$. Suppose $w \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. We denote the accepting run of $\mathcal{A}_1$ on w by:

$$s_0\ a_1\ s_1\ a_2\ s_2\ \cdots\ a_n\ s_n$$

and the accepting run of $\mathcal{A}_2$ on w by:

$$t_0\ a_1\ t_1\ a_2\ t_2\ \cdots\ a_n\ t_n.$$

By the definition of accepting run and the definition of $\mathcal{B}$,

$$(s_0, t_0)\ a_1\ (s_1, t_1)\ a_2\ (s_2, t_2)\ \cdots\ a_n\ (s_n, t_n).$$

is the accepting run of $\mathcal{B}$ on w. Hence, $w \in L(\mathcal{B})$.

(Closure under union) Similar to the intersection case, except that the set of accepting states become $(Q_1 \times F_2) \cup (F_1 \times Q_2)$. ∎

2 Non-deterministic finite state automata

A *non-deterministic finite state automaton* (NFA) is a system $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$, where each component is as follows.

- Σ is an alphabet.
- Q is a finite set of *states*.
- $q_0 \in Q$ is the *initial* state.
- $F \subseteq Q$ is the set of *accepting* states.
- $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*.

As before, we will say that "*$\mathcal{A}$ is an NFA over alphabet Σ*," or that "*the alphabet of $\mathcal{A}$ is Σ*."

On input word $w = a_1 \cdots a_n$, *a run of $\mathcal{A}$ on w* is a sequence:

$$q_0 \; a_1 \; q_1 \; a_2 \; q_2 \; \cdots \; a_n \; q_n,$$

where $(q_i, a_{i+1}, q_{i+1}) \in \delta$, for each $i = 0, \ldots, n-1$.[*] It is called *accepting* run, if $q_n \in F$. We say that $\mathcal{A}$ *accepts* w, if there is an accepting run of $\mathcal{A}$ on w. The language of all words accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$. A language L is an NFA language, if there is an NFA $\mathcal{A}$ such that $L = L(\mathcal{A})$, in which, we say that the language L is accepted by $\mathcal{A}$, or $\mathcal{A}$ accepts the language L.

Remark 1.4 NFA languages are closed under intersection and union. More formally, it can be stated as follows.

- For every two NFA $\mathcal{A}_1$ and $\mathcal{A}_2$, there is an NFA $\mathcal{A}'$ such that $L(\mathcal{A}') = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.
- For every two NFA $\mathcal{A}_1$ and $\mathcal{A}_2$, there is an NFA $\mathcal{A}'$ such that $L(\mathcal{A}') = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$.

Question: Why can we not conclude that NFA languages are closed under complementation directly from the definition of NFA? ▮

Theorem 1.5 *For every NFA $\mathcal{A}$, there is a DFA $\mathcal{A}'$ such that $L(\mathcal{A}) = L(\mathcal{A}')$.*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be an NFA. Consider the following DFA $\mathcal{A}' = \langle \Sigma, Q', q_0', F', \delta' \rangle$.

- $Q' = 2^Q$, i.e., the set of all subsets of Q, including $\emptyset$ and Q.
- The initial state q_0' is $\{q_0\}$, i.e., the set that contains only a single element which is the initial state q_0 of $\mathcal{A}$.
- F' consists of the subset $S \subseteq Q$ where $S \cap F \neq \emptyset$.
- The transition function $\delta : 2^Q \times \Sigma \to 2^Q$ is defined as follows.

  $$\delta'(S, a) \quad = \quad \{p \mid \text{ there is } q \in S \text{ such that } (q, a, p) \in \delta\}$$

  In other words, for every $S \in 2^Q$ and $a \in \Sigma$, we define $\delta'(S, a)$ to be the set T, where $p \in T$ if and only if there is $q \in S$ such that $(q, a, p) \in \delta$. By default, we define $\delta'(\emptyset, a) = \emptyset$.

We have the following two claims.

Claim 1 *For every word $w \in \Sigma^*$, where $w = a_1 \cdots a_n$, if there is a run of $\mathcal{A}$ on w:*

$$q_0 \; a_1 \; q_1 \; a_2 \; q_2 \; \cdots \; a_n \; q_n, \qquad \textit{where } q_0 \textit{ is the initial state of } \mathcal{A},$$

then the run of $\mathcal{A}'$ on w denoted by:

$$S_0 \; a_1 \; S_1 \; a_2 \; S_2 \; \cdots \; a_n \; S_n, \qquad \textit{where } S_0 \textit{ is the initial state of } \mathcal{A}',$$

is such that $q_i \in S_i$, for each $i = 0, 1, \ldots, n$.

[*]As in the case of DFA, we can define a run of $\mathcal{A}$ on w *starting from state q* as above, but starts from state q.

Proof. (of claim) The proof is by induction on the length of w, i.e., n. The base case, $n = 0$, holds trivially, since by the definition of $\mathcal{A}'$, its initial state q_0' is the set $\{q_0\}$.

For the induction hypothesis, assume that the claim holds for words of length n. For the induction step, let $w = a_1 \cdots a_n a_{n+1}$, i.e., of length $n+1$. Suppose there is a run of $\mathcal{A}$ on w:

$$q_0 \; a_1 \; q_1 \; a_2 \; q_2 \; \cdots \; a_n \; q_n \; a_{n+1} \; q_{n+1}$$

Applying the induction hypothesis on the word $a_1 \cdots a_n$, the run of $\mathcal{A}'$ on $a_1 \cdots a_n$ is:

$$S_0 \; a_1 \; S_1 \; a_2 \; S_2 \; \cdots \; a_n \; S_n,$$

is such that $q_i \in S_i$, for each $i = 0, 1, \ldots, n$.

Let $S_{n+1} = \delta'(S_n, a_{n+1})$. By the definition of run, $(q_n, a_{n+1}, q_{n+1}) \in \delta$. Since $q_n \in S_n$, by definition of δ', $q_{n+1} \in S_{n+1}$. Thus, the run of $\mathcal{A}'$ on $a_1 \cdots a_n a_{n+1}$ is:

$$S_0 \; a_1 \; S_1 \; a_2 \; S_2 \; \cdots \; a_n \; S_n \; a_{n+1} \; S_{n+1},$$

where $q_i \in S_i$, for each $i = 0, 1, \ldots, n+1$. ∎

Claim 2 *For every word $w \in \Sigma^*$, where $w = a_1 \cdots a_n$, if the run of $\mathcal{A}'$ on w is as follows:*

$$S_0 \; a_1 \; S_1 \; a_2 \; S_2 \; \cdots \; a_n \; S_n, \qquad \textit{where } S_0 \textit{ is the initial state of } \mathcal{A}'$$

then for every $q \in S_n$, there is a run of $\mathcal{A}$ on w:

$$q_0 \; a_1 \; q_1 \; a_2 \; q_2 \; \cdots \; a_n \; q_n, \qquad \textit{where } q_0 \textit{ is the initial state of } \mathcal{A}$$

such that $q_n = q$.

Proof. (of claim) The proof is very similar to the claim above, i.e., by induction on n. The base case, $n = 0$, holds trivially, since by the definition of $\mathcal{A}'$, its initial state is the set $\{q_0\}$.

For the induction hypothesis, assume that the claim holds for words of length n. For the induction step, let $w = a_1 \cdots a_n a_{n+1}$, i.e., of length $n+1$. Suppose the run of $\mathcal{A}'$ on w is as follows.

$$S_0 \; a_1 \; S_1 \; a_2 \; S_2 \; \cdots \; a_{n+1} \; S_{n+1}, \qquad \text{where } S_0 \text{ is the initial state of } \mathcal{A}'$$

Let $q \in S_{n+1}$. By the definition of run, $S_{n+1} = \delta'(S_n, a_{n+1})$. By the definition of δ', if $q \in S_{n+1}$, there is $p \in S_n$ such that $(p, a_{n+1}, q) \in \delta$.

Applying the induction hypothesis on the word $a_1 \cdots a_n$ and the state p, there is a run of $\mathcal{A}$ on $a_1 \cdots a_n$:

$$q_0 \; a_1 \; q_1 \; a_2 \; q_2 \; \cdots \; a_n \; q_n,$$

where $q_n = p$. Since $(p, a_{n+1}, q) \in \delta$, we extend the run to be:

$$q_0 \; a_1 \; q_1 \; a_2 \; q_2 \; \cdots \; a_n \; q_n \; a_{n+1} \; q_{n+1},$$

where $q_{n+1} = q$. ∎

Note that Claim 1 implies $L(\mathcal{A}) \subseteq L(\mathcal{A}')$ and Claim 2 implies $L(\mathcal{A}') \subseteq L(\mathcal{A})$. ∎

In view of Theorem 1.5, we can say that a language is regular if and only if it is accepted by an NFA.

Corollary 1.6 *NFA languages are closed under complement. That is, for every NFA $\mathcal{A}$ over alphabet Σ, there is a DFA $\mathcal{A}'$ over the same alphabet Σ such that $L(\mathcal{A}') = \Sigma^* - L(\mathcal{A})$.*

Remark 1.7 For every NFA $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$, we can always convert it into another NFA $\mathcal{A}' = \langle \Sigma, Q', q_0', F', \delta' \rangle$ such that $L(\mathcal{A}) = L(\mathcal{A}')$ and the initial state q_0' of $\mathcal{A}'$ does not have any incoming edge.

This is actually pretty straightforward. Let p be a state that is not in Q. The NFA $\mathcal{A}' = \langle \Sigma, Q', q_0', F', \delta' \rangle$ is defined as follows.

- $Q' = Q \cup \{p\}$.
- The initial state is p.
- The set of accepting states F' is as follows.

$$F \;:=\; \begin{cases} F, & \text{if } q_0 \notin F \\ F \cup \{p\}, & \text{if } q_0 \in F \end{cases}$$

- The transition relation δ' is defined as δ with the following extra transitions. For every $(q_0, a, q) \in \delta$, we have $(p, a, q) \in \delta'$. That is, state p behaves like state q_0.

Theorem 1.8 *Regular languages are closed under concatenation and Kleene star. More formally, it can be stated as follows.*

- *If L_1 and L_2 are regular languages, so is $L_1 L_2$.*
- *If L is a regular language, so is L^*.*

Proof. (Closure under concatenation) Let $\mathcal{A}_1 = \langle \Sigma, Q_1, q_{0,1}, F_1, \delta_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, q_{0,2}, F_2, \delta_2 \rangle$ be NFA for L_1 and L_2, respectively. We can assume that $Q_1 \cap Q_2 = \emptyset$. By Remark 1.7, we can assume that both $q_{0,1}$ and $q_{0,2}$ do not have incoming edge.

Define the following NFA $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$.

- $Q = Q_1 \cup Q_2$.
- $q_{0,1}$ is the initial state.
- The set F of accepting states is F_2.
- The transition relation δ is defined as $\delta_1 \cup \delta_2$ and the following extra transitions.
  - For every $(p, a, q) \in \delta_1$, where $q \in F_1$, we have transition $(p, a, q_{0,2})$ in δ.

It is not difficult to show that $L(\mathcal{A}) = L_1 L_2$.

(Closure under Kleene star) Let $\mathcal{A} = \langle \Sigma, Q_1, q_0, F, \delta \rangle$ be NFA for L. By Remark 1.7, we can assume that q_0 does not have incoming edge. Define the following NFA $\mathcal{A}' = \langle \Sigma, Q', q_0', F', \delta' \rangle$.

- $Q' = Q$.
- q_0 is the initial state.
- The set F' of accepting states is $\{q_0\}$.
- The transition relation δ' is defined as δ and the following extra transitions.
  - For every $(p, a, q) \in \delta$, where $q \in F$, we have transition (p, a, q_0) in δ'.

It is not difficult to show that $L(\mathcal{A}') = L^*$. ∎

3 Pumping lemma

In the following, for a word w and an integer $n \geqslant 0$, w^n obtained by repeating w for n number of times, i.e., $\underbrace{w \cdots w}_{n \text{ times}}$. By default, we define $w^0 = \epsilon$.

Lemma 1.9 (pumping lemma) *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be an NFA. Let $x \in L(\mathcal{A})$ be a word such that $|x| \geqslant |Q|$. Then, the word x can be divided into three parts u, v, w, i.e., $x = uvw$, such that $|v| \geqslant 1$ and for every integer $k \geqslant 0$, $uv^k w \in L(\mathcal{A})$.*

Proof. Let $x = a_1 \cdots a_n$ and $x \in L(\mathcal{A})$, where $n \geqslant |Q|$. Let the following be its accepting run:

$$p_0 \; a_1 \; p_1 \; a_2 \; p_2 \; \cdots \; a_n \; p_n$$

Since $n \geqslant |Q|$, there are $0 \leqslant i < j \leqslant n$ such that $p_i = p_j$.

Let $u = a_1 \cdots a_i$, $v = a_{i+1} \cdots a_j$ and $w = a_{j+1} \cdots a_n$. Then, for every integer $k \geqslant 0$, the following is an accepting run of $\mathcal{A}$ on $uv^k w$:

$$p_0 \; a_1 \; p_1 \; a_2 \; p_2 \; \cdots \; a_i \; p_i \; \underbrace{a_{i+1} \; p_{i+1} \; \cdots \; a_j \; p_j}_{\text{repeat } k \text{ times}} \; a_{j+1} \; p_{j+1} \; \cdots \; a_n \; p_n$$

∎

Lemma 1.9 can be restated as follows.

Lemma 1.10 (pumping lemma) *For every regular language L, there is an integer $n \geqslant 1$ such that for every word $x \in L$ with length $|x| \geqslant n$, there are u, v, w where $x = uvw$ and $|v| \geqslant 1$ and for every integer $k \geqslant 0$, $uv^k w \in L$.*

Note that from the proof of Lemma 1.9, we can easily deduce the following "more refined" pumping lemma.

Lemma 1.11 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be an NFA. Let $x \in L(\mathcal{A})$ be a word and $x = szt$, where $|z| \geqslant |Q|$. Then, the word z can be divided into three parts u, v, w such that $|v| \geqslant 1$ and for every integer $k \geqslant 0$, $suv^k wt \in L(\mathcal{A})$.*

Example 1.12 Using pumping lemma, we can show that all languages below are not regular languages.

- $L_1 = \{a^n b^n \mid n \geqslant 0\}$.
- $L_2 = \{a^n \mid n \text{ is a prime number}\}$.

Appendix: Concatenation and Kleene star

For two words u and v, $u \cdot v$ denotes the word obtained by *concatenating* v at the end of u. ($u \cdot v$ reads: u concatenates with v.) By default, $u \cdot \epsilon = \epsilon \cdot u = u$. We will usually omit $\cdot$ and simply write uv instead of $u \cdot v$.

In the following, let L_1, L_2 and L be languages. We define the following operators.

$$\begin{aligned} L_1 \cdot L_2 &:= \{uv \mid u \in L_1 \text{ and } v \in L_2\} && \text{(Concatenation)} \\ L^n &:= \{u_1 \cdots u_n \mid \text{each } u_i \in L\} \\ L^* &:= \bigcup_{n \geqslant 0} L^n && \text{(Kleene star)} \end{aligned}$$

As before, we usually write L_1L_2 to denote $L_1 \cdot L_2$, and L_1L_2 reads as L_1 concatenates with L_2.

Note that by default, for any set $X \subseteq \Sigma^*$, $X^0 = \{\epsilon\}$. Thus, $\emptyset^* = \{\epsilon\}$.

Lesson 2: Regular expressions

Theme: Regular expressions as alternative description of regular languages.

In the following we fix an alphabet Σ. *Regular expressions* (over Σ) are expressions built inductively as follows.

- $\emptyset$ is a regular expression.
- a is a regular expression, for every symbol $a \in \Sigma$.
- If e_1, e_2 are regular expressions, then so are $(e_1 \cdot e_2)$ and $(e_1 \cup e_2)$.
- If e is a regular expression, then so is $(e)^*$.

A regular expression e over Σ defines a language, denoted by $L(e)$, over the same alphabet as follows.

- If e is $\emptyset$, then $L(e) = \emptyset$.
- If e is a, where $a \in \Sigma$, then $L(e) = \{a\}$.
- If e is of the form $(e_1 \cdot e_2)$, where e_1 and e_2 are regular expressions, then $L(e) = L(e_1) \cdot L(e_2)$.
- If e is of the form $(e_1 \cup e_2)$, where e_1 and e_2 are regular expressions, then $L(e) = L(e_1) \cup L(e_2)$.
- If e is of the form $(e_1)^*$, where e_1 is a regular expression, then $L(e) = L(e_1)^*$.

Usually, we omit writing $\cdot$ in $(e_1 \cdot e_2)$, and instead, we simply write $(e_1 e_2)$. Also, when there is no ambiguity, we will omit writing the brackets and simply write $e_1 e_2$ and e_1^*, instead of $(e_1 e_2)$ and $(e_1)^*$.

The following theorem states that the class of languages defined by regular expressions is exactly the class of regular languages.

Theorem 2.1 *Regular expressions define precisely the class of regular languages. More formally, it can be stated as follows.*

- *For every regular expression e over Σ, $L(e)$ is a regular language.*
- *For every NFA $\mathcal{A}$, there is a regular expression e such that $L(e) = L(\mathcal{A})$.*

Proof. We first prove the first item. The proof is by induction on the regex e. The base case is when e is either $\emptyset$ or $a \in \Sigma$.

- When e is $\emptyset$, then $L(e) = \emptyset$.

  One can easily construct an NFA $\mathcal{A}$ that accepts nothing.
- When e is a, for some symbol $a \in \Sigma$, then $L(e) = \{a\}$.

  We can construct an NFA $\mathcal{A}$, that has only two states p and q, p is the initial state and there is only one accepting state q and δ contains only one transition (p, a, q).

For the induction step, we will prove the case where e is either of the form $\alpha \cdot \beta$, $\alpha \cup \beta$ or α^*.

By the induction hypothesis, there are NFA $\mathcal{A}_1$ and $\mathcal{A}_2$ that accept the languages $L(\alpha)$ and $L(\beta)$, respectively. By Remark 1.4 and Theorem 1.8 (in Lesson 1), there are NFAs for all the languages $L(\alpha \cdot \beta)$, $L(\alpha \cup \beta)$ and $L(\alpha^*)$.

We now prove the second item. Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be an NFA. Without loss of generality, we assume that $Q = \{1, \ldots, n\}$. For every $1 \leqslant i, j \leqslant n$ and $0 \leqslant k \leqslant n$, define the language $L(i, j, k)$ as follows.

$$L(i,j,k) \quad := \quad \left\{ \; w \in \Sigma^* \; \middle| \; \begin{array}{l} \text{there is a run of } \mathcal{A} \text{ on } w \text{ from state } i \text{ to state } j \\ \textit{without} \text{ passing any states } \geqslant k+1 \end{array} \right\}$$

That is, if $w \in L(i, j, k)$, there is a run of $\mathcal{A}$ on w from state i to j *without* passing through the states $k + 1, \ldots, n$.

We will first prove the following claim.

Claim 1 *For every $1 \leqslant i, j \leqslant n$ and $0 \leqslant k \leqslant n$, there is a regex e such that $L(e) = L(i, j, k)$.*

Proof. The proof is by induction on k. The base case is $k = 0$. For every $1 \leqslant i, j \leqslant n$, consider the set of symbols $\Gamma_{i,j} = \{a \mid (i, a, j) \in \delta\}$.

- If $\Gamma_{i,j} = \emptyset$, then $L(i, j, 0) = \emptyset$. The desired regex e is:

$$e \quad = \quad \begin{cases} \emptyset & \text{if } i \neq j \\ \emptyset^* & \text{if } i = j \end{cases}$$

- If $\Gamma_{i,j} \neq \emptyset$, assume $\Gamma_{i,j} = \{a_1, \ldots, a_t\}$. The desired regex e is:

$$e \quad = \quad \begin{cases} a_1 \cup \cdots \cup a_t & \text{if } i \neq j \\ a_1 \cup \cdots \cup a_t \cup \emptyset^* & \text{if } i = j \end{cases}$$

For the induction hypothesis, we assume that the claim holds for k. For the induction step, we will prove it for $k + 1$. Note the following identity:

$$L(i,j,k+1) \quad = \quad L(i,j,k) \;\cup\; \Big(L(i,k+1,k) \cdot L(k+1,k+1,k)^* \cdot L(k+1,j,k)\Big)$$

By the induction hypothesis, there are regexes that define each of $L(i, j, k)$, $L(i, k + 1, k)$, $L(k + 1, k+1, k)$, and $L(k+1, j, k)$. By the definition of regex, there is regex that define $L(i, j, k+1)$. ▮

To complete our proof, note that:

$$L(\mathcal{A}) \quad = \quad \bigcup_{q_f \in F} L(q_0, q_f, n)$$

By the claim above, for each $L(q_0, q_f, n)$, there is a regex that defines it. Taking the union of all of them, we have a regex for $L(\mathcal{A})$. ▮

Combining what we have learnt so far, we obtain three different, but equivalent, characterisations of regular languages, as stated below.

Corollary 2.2 *Let L be a language. The following are equivalent.*

- *L is accepted by a DFA.*
- *L is accepted by an NFA.*
- *L is defined by a regular expression.*

Remark 2.3 The term regular expressions are commonly abbreviated as *regex*. In most literatures and websites, the term "regex" are used more often than "regular expression." Due to its widespread applications, many modern programming languages now include libraries for regex. The following are some of them.

- Scala: `http://www.scala-lang.org/api/rc2/scala/util/matching/Regex.html`
- C++: `http://www.cplusplus.com/reference/regex/`
- Java: `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`
- Python: `https://docs.python.org/2/library/re.html`

Lesson 3: Context-free languages

Theme: Context-free grammars and their languages.

1 Context-free grammars

A *context-free grammar* (CFG) is a system $\mathcal{G} = \langle \Sigma, V, R, S \rangle$, where each component is as follows.

- Σ is a finite set of symbols called *terminals*.[*]
- V is a finite set of *variables*, and $V \cap \Sigma = \emptyset$.
- R is a finite set of *rules*, where each rule is of the form:

$$A \quad \to \quad w,$$

  where $A \in V$ and $w \in (V \cup \Sigma)^*$.
- S is a special variable from V called the *start variable*.

There is *no*(!) restriction that for each variable $A \in V$, there is only one rule $A \to w$ in R. We may have several rules, say $A \to w_1, A \to w_2, \ldots, A \to w_m$, all belong to R. In this case, we usually abbreviate them as:

$$A \quad \to \quad w_1 \;\mid\; w_2 \;\mid\; \cdots \;\mid\; w_m$$

Note also that we may have a rule of the form $A \to \varepsilon$.

We have a few important notations and terminology.

- Let $uAv \in (\Sigma \cup V)^*$ be a word in which a variable $A \in V$ appears. If there is a rule $A \to w$ in R, we say that uAv *yields* uwv, denoted as $uAv \Rightarrow uwv$.
- For strings $x, y \in (\Sigma \cup V)^*$, we say that x *derives* y, if either $x = y$, or there is a sequence $z_1, \ldots, z_n$ such that $x = z_1$, $y = z_n$ and

$$z_1 \Rightarrow z_2 \Rightarrow \cdots \Rightarrow z_n.$$

  We write $x \Rightarrow^* y$ to denote that x derives y.[†]
- For a variable A, $L(\mathcal{G}, A)$ denotes the language of all words over Σ that can be derived from variable A. Formally,

$$L(\mathcal{G}, A) \quad = \quad \{w \in \Sigma^* \;\mid\; A \Rightarrow^* w\}.$$

- $L(\mathcal{G})$ denotes the language $L(\mathcal{G}, S)$, i.e., the language of all words over Σ that can be derived from the start variable S. Formally,

$$L(\mathcal{G}) \quad = \quad \{w \in \Sigma^* \;\mid\; S \Rightarrow^* w\}.$$

  The language $L(\mathcal{G})$ is called the language generated/defined/derived from/by $\mathcal{G}$.

[*]Σ is actually the alphabet. Usually in context-free grammars elements in Σ are called terminals, instead of symbols.

[†]Sometimes we will say "y is derived from x" or "from x we can derive y" to mean "x derives y."

- A language L is called a *context-free language* (CFL), if there is a CFG $\mathcal{G}$ such that $L(\mathcal{G}) = L$.

Example 3.1 Consider the following CFG $\mathcal{G} = \langle \Sigma, V, R, S \rangle$, where:

- $\Sigma = \{a, b\}$.
- $V = \{S\}$.
- $R = \{S \to \varepsilon \mid aSb\}$.
- S is the start variable.

It can be shown that $L(\mathcal{G}) = \{a^n b^n \mid n \geqslant 0\}$.

Theorem 3.2 *Context-free languages are closed under union, concatenation and Kleene star.*

Proof. Let $\mathcal{G}_1 = \langle \Sigma, V_1, R_1, S_1 \rangle$ and $\mathcal{G}_2 = \langle \Sigma, V_2, R_2, S_2 \rangle$. First, we rename the variables in V_1 and V_2 such that $V_1 \cap V_2 = \emptyset$.

(Closure under union) Consider the CFG $\mathcal{G} = \langle \Sigma, V, R, S \rangle$ defined as follows.

- $V = V_1 \cup V_2 \cup \{S\}$, where S is a "new" variable, i.e., $S \notin V_1 \cup V_2$.
- $R = R_1 \cup R_2 \cup \{S \to S_1 | S_2\}$.
- S is the start variable.

Due to the rule $S \to S_1|S_2$, every word derived from S can also be derived from S_1 or S_2. Vice versa, every word that can be derived from S_1 or S_2 can also be derived from S. Thus, $L(\mathcal{G}) = L(\mathcal{G}_1) \cup L(\mathcal{G}_2)$.

(Closure under concatenation) Consider the CFG $\mathcal{G} = \langle \Sigma, V, R, S \rangle$ defined as follows.

- $V = V_1 \cup V_2 \cup \{S\}$, where S is a "new" variable, i.e., $S \notin V_1 \cup V_2$.
- $R = R_1 \cup R_2 \cup \{S \to S_1 S_2\}$.
- S is the start variable.

Due to the rule $S \to S_1 S_2$, every word w derived from S can be divided into two $w = uv$ such that u and v can be derived from S_1 and S_2, respectively. Vice versa, for every word u and v that can be derived from S_1 and S_2, respectively, the word uv can be derived from S. Thus, $L(\mathcal{G}) = L(\mathcal{G}_1) L(\mathcal{G}_2)$.

(Closure under Kleene star) Consider the CFG $\mathcal{G} = \langle \Sigma, V, R, S \rangle$ defined as follows.

- $V = V_1 \cup \{S\}$, where S is a "new" variable, i.e., $S \notin V_1$.
- $R = R_1 \cup \{S \to S_1 S | \varepsilon\}$.
- S is the start variable.

Due to the rule $S \to S_1 S|\varepsilon$, for every word w that can be derived from S, the following holds: Either $w = \varepsilon$ or w can be divided into finitely many words $w = v_1 \cdots v_n$ such that every v_i can be derived from S_1. Thus, $L(G) \subseteq L(\mathcal{G}_1)^*$.

Conversely, for every word $w \in L(\mathcal{G}_1)^*$, by definition of Kleene star, either $w = \varepsilon$ or w can be divided into finitely many words $w = v_1 \cdots v_n$ such that every v_i can be derived from S_1. Due to the rule $S \to S_1 S|\varepsilon$, we have the derivation $S \Rightarrow^* \underbrace{S_1 \cdots S_1}_{n \text{ times}}$. Since each v_i can be derived from one S_1, this means w can be derived from S. Thus, $L(\mathcal{G}_1)^* \subseteq L(\mathcal{G})$. Therefore, $L(\mathcal{G}) = L(\mathcal{G}_1)^*$. ∎

Theorem 3.3 *Every regular language is a context-free language.*

Remark 3.4 Modifying the CFG in the previous example, we can show that the following two languages are also context-free.

- $L_1 = \{a^k b^m c^n \mid k, n, m \geqslant 0 \text{ and } k = m\}$.
- $L_2 = \{a^k b^m c^n \mid k, n, m \geqslant 0 \text{ and } m = n\}$.

However, we will show later that $L_1 \cap L_2$ is not CFL. Hence, context-free languages are *not* closed under intersection. Combining this remark with Theorem 3.2, it is immediate that context-free languages are *not* closed under complement (why?). ▮

2 Derivation trees

A *derivation tree*, or a *parse tree*, of a CFG $\mathcal{G} = \langle \Sigma, V, R, S \rangle$ is a tree T in which:

- every vertex has a *label*, which is a symbol from $V \cup \Sigma \cup \{\varepsilon\}$;
- the label of an interior vertex is a variable from V;
- the label of a leaf vertex is either ε or a terminal from Σ;
- if an interior vertex has a label $A \in V$ and it has k children $n_1, \ldots, n_k$ (in the order from left to right) with labels $X_1, \ldots, X_k$, respectively, then $A \to X_1 \cdots X_k$ must be a rule in R.

If the label of the root is a variable A, and the leaf vertices of T are $n_1, \ldots, n_m$ (in the order from left to right) with labels $u_1, \ldots, u_m$, we say that T is a *derivation tree of $\mathcal{G}$ from variable A on word $u_1 \cdots u_m$*. When the label of the root is the start variable S, we will simply say T is a *derivation tree of $\mathcal{G}$ on $u_1 \cdots u_m$*.

Theorem 3.5 *Let $\mathcal{G} = \langle \Sigma, V, R, S \rangle$ be a CFG. For every variable $A \in V$, for every word $w \in \Sigma^*$, the following holds.*

$$A \Rightarrow^* w \quad \text{if and only if} \quad \text{there is a derivation tree of } \mathcal{G} \text{ from } A \text{ on } w.$$

In particular, $w \in L(\mathcal{G})$ if and only if there is a derivation tree of $\mathcal{G}$ on w.

3 Pumping lemma for context-free languages

Like regular languages, context-free languages also have property that their words can be "pumped."

Lemma 3.6 (pumping lemma) *Let $\mathcal{G} = \langle \Sigma, V, R, S \rangle$ be a CFG. Then, there is an integer N such that every $w \in L(\mathcal{G})$ with length $\geqslant N$ can be partitioned into:*

$$w \quad = \quad s\ x\ y\ z\ t$$

such that the following holds.

(P1) $|x| + |z| \geqslant 1$.

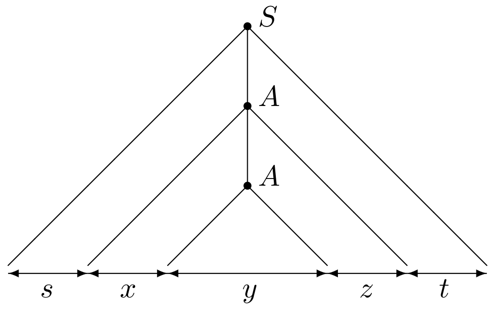
(P2) $|xyz| \leqslant N$.

(P3) For every $i \geqslant 0$, $sx^i yz^i t \in L(\mathcal{G})$.

Proof. Let $\mathcal{G} = \langle \Sigma, V, R, S \rangle$ be a CFG and let $n = |V|$. Let $m = \max_{A \to w \in R} |w|$, i.e., the maximum length of the string u over all the rule $A \to u$ in R. Intuitively, this means that in every derivation tree of $\mathcal{G}$, every node has at most m children. We define $N = m^n + 1$. In the following we will show that for every word $w \in L(\mathcal{G})$ with length $\geqslant N$ can be partitioned into $sxyzt$ such that (P1)–(P3) above hold.

Let $w \in L(\mathcal{G})$ and $|w| \geqslant N$. Thus, there is a derivation tree T of $\mathcal{G}$ on w. Since every node in T has at most m children and $|w| \geqslant N = m^n + 1$, the depth of the tree T is $\geqslant n + 1$.[‡]

Consider a leaf node ℓ with depth $\geqslant n+1$. The length of the path from the root node to ℓ is at least $n+1$,[§] which means there are at least $n+1$ internal nodes. Since there are only n variables, there is a variable A that appears at least twice in the path. See figure below.



We can choose variable A such that $|x| + |z| \geqslant 1$. Such variable exists. Otherwise, if $|x| + |z| = 0$, then we can omit the path from A to A, and thus, shorten it, and choose another path from the root node to a leaf node with length $\geqslant n+1$. Thus, (P1) holds.

Furthermore, we can also choose such variable A for which the "first" A has the minimal depth. That is, all the path from starting from the first A to the leaf nodes do not contain other variable that appear twice in the path. Thus, in this case $|x| + |y| + |z| \leqslant N$ and (P2) holds.

Finally, to show that (P3) holds, by repeating the variable A as many times as we want, i.e., for every integer $i \geqslant 0$, we repeat the derivation tree starting from variable A for as many as i times, we obtain the derivation tree for the word sx^iyz^it, which means that it is in $L(\mathcal{G})$. That is, (P3) holds. ∎

Using Lemma 3.6, we can show that the language $L = \{a^kb^kc^k \mid k \geqslant 0\}$ is not a CFL. Since L is the intersection of two CFL's $\{a^mb^mc^n \mid m,n \geqslant 0\}$ and $\{a^nb^mc^m \mid m,n \geqslant 0\}$, it also shows that CFL's are not closed under intersection.

We can also rewrite Lemma 3.6 as follows.

Lemma 3.7 (pumping lemma) *For every CFL L, there is an integer N such that every $u \in L(\mathcal{G})$ with length $\geqslant N$ can be partitioned into:*

$$u = s\ x\ y\ z\ t$$

such that the following holds.

- $|x| + |z| \geqslant 1$.
- $|xyz| \leqslant N$.
- *For every $i \geqslant 0$, $sx^iyz^it \in L$.*

[‡]Here we define the depth of the root node to be 0 and for every other node u, its depth is the depth of its parent plus 1. The depth of the tree T is the maximal depth over all its nodes.

[§]Here the length of a path is defined as the number of edges.

Lesson 4: Push-down automata

Theme: Push-down automata as a model of computation for context-free languages.

In the following we will have two alphabets Σ and Γ. A *push-down automaton* (PDA) is a system $\mathcal{A} = \langle \Sigma, \Gamma, Q, q_0, F, \delta \rangle$, where each of the component is as follows.

- Σ is a finite alphabet, called the *input* alphabet, whose elements are called *input symbols.*
- Γ is a finite alphabet, called the *stack* alphabet, whose elements are called *stack symbols.*
- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of accepting states.
- $\delta \subseteq Q \times \big(\Sigma \cup \{\varepsilon\}\big) \times \big(\Gamma \cup \{\varepsilon\}\big) \times Q \times \big(\Gamma \cup \{\varepsilon\}\big)$ is the transition relation.

We will usually write a transition $(p, x, y, q, z) \in \delta$ as:

$$(p, x, \mathsf{pop}(y)) \quad \to \quad (q, \mathsf{push}(z))$$

Intuitively, such transition means that when a PDA is in state p reading x from the input and the top of the stack is y, it can "pop" y from the top of the stack and moves to state q and push z into the stack. Here it is possible that x, y and z are the empty string ε.

Note that the fashion a symbol is written into and taken out of the stack is "Last In First Out" (LIFO), i.e., the last symbol that gets written into the stack has to come out first. It is also important to note that while the input is a word over Σ, its stack contains symbols from Γ.

We will now describe formally how PDA computes. Let $\mathcal{A} = \langle \Sigma, \Gamma, Q, q_0, F, \delta \rangle$ be a PDA. A *configuration* of $\mathcal{A}$ is a pair $(q, u) \in Q \times \Gamma^*$, where q is the state of $\mathcal{A}$ and u is the content of the stack. The *initial configuration* is (q_0, ε). A configuration is *accepting*, if the state component is one of the accepting states.

On input $w = a_1 \ldots a_m$, a *run* of a PDA *from a configuration* (q, u) is a sequence:

$$(p_0, v_0) \vdash_{b_1} (p_1, v_1) \vdash_{b_2} \cdots\cdots \vdash_{b_n} (p_n, v_n), \qquad (\dagger)$$

where

- $(p_0, v_0) = (q, u)$,
- $b_1 \cdots b_n = a_1 \cdots a_m$, i.e., some of the b_i's can be ε,
- for each $i = 1, \ldots, n$, there is $(p_i, x, \mathsf{pop}(y)) \to (p_{i+1}, \mathsf{push}(z)) \in \delta$ such that
  - $x = b_i$,
  - $v_i = sy$ and $v_{i+1} = sz$, for some $s \in \Gamma^*$.

  Note: Here $v_i = sy$ denotes that the content of the stack, where the top of the stack is y. When the transition $(p_i, x, \mathsf{pop}(y)) \to (p_{i+1}, \mathsf{push}(z))$ is applied, the PDA is in state p_i reads x from the input, "pops" y from the stack, and moves to state p_{i+1} and at the same time "pushes" z into the stack. Thus, the subsequent content v_{i+1} of the stack is sz.

We will also write the run (†) as:

$$(p_0, v_0) \vdash_w^* (p_n, v_n).$$

In this case we will also say that there is a run of $\mathcal{A}$ on w from (p_0, v_0) to (p_n, v_n).

A run is *accepting*, if it starts from the initial configuration and ends with an accepting configuration. The language accepted by $\mathcal{A}$, denoted by $L(\mathcal{A})$, consists of all the words for which it has an accepting run. Formally,

$$L(\mathcal{A}) \;=\; \big\{\, w \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w \big\}.$$

The following theorem states that CFL and PDA are actually equivalent.

Theorem 4.1

- *For every CFG $\mathcal{G}$, there is a PDA $\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{G})$.*
- *Vice versa, for every PDA $\mathcal{A}$, there is a CFG $\mathcal{G}$ such that $L(\mathcal{A}) = L(\mathcal{G})$.*

The proof is a bit technical and can be found in the appendix. One immediate consequence of Theorem 4.1 is that the intersection of a regular language and a CFL is a CFL.

Theorem 4.2 *If K is CFL and L is regular language, then the intersection $K \cap L$ is CFL.*

Appendix: Proof of Theorem 4.1

We are going to show that CFG and PDA define precisely the same class of languages. More precisely, we are going to show the following.

- For every CFG $\mathcal{G}$, there is a PDA $\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{G})$.
- For every PDA $\mathcal{A}$, there is a CFG $\mathcal{G}$ such that $L(\mathcal{A}) = L(\mathcal{G})$.

A From CFG to PDA

Allowing the PDA to push a string of symbols. First, we note that we can modify the definition of PDA to allow it to push a string of symbols to its stack. That is, the transitions can be of the form:

$$(p, x, \mathsf{pop}(y)) \;\to\; (q, \mathsf{push}(z)), \qquad\qquad \text{where } z \in \Gamma^*$$

Allowing such transition does not change the capability of a CFG. We can add "new" states $t_1, \ldots, t_m$, where m is the lenth of z and $z = a_1 \ldots a_m$. Each t_i is used to push the symbol a_i into the stack. More formally, the transition $(p, x, \mathsf{pop}(y)) \to (q, \mathsf{push}(z))$ can be replaced with the following transitions:

$$\begin{array}{rcl}
(p, x, \mathsf{pop}(y)) & \to & (t_1, \mathsf{push}(a_1)) \\
(t_1, \varepsilon, \mathsf{pop}(\varepsilon)) & \to & (t_2, \mathsf{push}(a_2)) \\
(t_2, \varepsilon, \mathsf{pop}(\varepsilon)) & \to & (t_3, \mathsf{push}(a_3)) \\
\vdots & & \vdots \\
(t_{i-1}, \varepsilon, \mathsf{pop}(\varepsilon)) & \to & (t_i, \mathsf{push}(a_i)) \\
\vdots & & \vdots \\
(t_{m-1}, \varepsilon, \mathsf{pop}(\varepsilon)) & \to & (t_m, \mathsf{push}(a_m)) \\
(t_m, \varepsilon, \mathsf{pop}(\varepsilon)) & \to & (q, \mathsf{push}(\varepsilon))
\end{array}$$

Left-most substitution property. Let $\mathcal{G} = \langle \Sigma, V, R, S\rangle$ be a CFG. Let $z_1 \Rightarrow z_2 \Rightarrow \cdots \Rightarrow z_n$ be a derivation. We say that the derivation has the *left-most substitution property*, if for every $i \in \{1, \ldots, n-1\}$, $z_i \Rightarrow z_{i+1}$ is obtained by applying a rule $A \to w$, where A is the left most variable in z_i. Intuitively, it means that we only substitute the left-most variable in our derivation.

For example, suppose we have grammars with the following rules: $A \to aABa$, $A \to SS$, $B \to aab$, $B \to SA$. The derivation $aABAaa \Rightarrow aSSBAaa$ has the left-most substitution property, because we substitute the left-most variable which is A. On the other hand, $aABAaa \Rightarrow aASAAaa$ does not, because we substitute variable B, which is not the left-most variable, and $aABAaa \Rightarrow aABSSaa$ does not either, because we substitute the second A, which is not the left-most.

Remark 4.3 Without loss of generality, we only need to consider derivations that has left-most substitution property, by simply substituting the left-most variable first.

Constructing a PDA from a CFG. Let $\mathcal{G} = \langle \Sigma, V, R, S\rangle$ be a CFG. Consider the following PDA $\mathcal{A} = \langle \Sigma, \Gamma, Q, q_0, F, \delta\rangle$ where each component is as follows.

- $\Gamma = \Sigma \cup V \cup \{\bot\}$.
- $Q = \{p, q\}$
- p is the initial state.
- $F = \{q\}$.
- δ consists of the following transitions. (Here w^r denotes the reverse of w.)
  - $(p, \varepsilon, \mathsf{pop}(\varepsilon)) \to (q, \mathsf{push}(S))$.
  - $(q, a, \mathsf{pop}(a)) \to (q, \mathsf{push}(\varepsilon))$, for every $a \in \Sigma$.
  - $(q, \varepsilon, \mathsf{pop}(A)) \to (q, \mathsf{push}(s^r))$, for every rule $A \to s \in R$.

  Notice that in the first and third transitions, $\mathcal{A}$ pushes a string of symbols to its stack.

We will show that $L(\mathcal{G}) = L(\mathcal{A})$. First, we show the following lemma.

Lemma 4.4 *For every derivation:*

$$u \;\Rightarrow\; \tilde{u}$$

there is a run of $\mathcal{A}$ on w:

$$(q, v) \;\vdash_w^*\; (q, \tilde{v}),$$

where

$$v^r \;=\; u \qquad \text{and} \qquad w\tilde{v}^r \;=\; \tilde{u} \tag{†}$$

Proof. Suppose $u \Rightarrow^* \tilde{u}$. Let m be the length of the derivation, which is denoted as follows.

$$u_0 \;\Rightarrow\; u_1 \;\Rightarrow\; \cdots \;\Rightarrow\; u_{m-1} \;\Rightarrow\; u_m \tag{1}$$

where $u_0 = u$ and $u_m = \tilde{u}$. We will prove the lemma by induction on m.

The base case is $m = 0$. In this case the derivation is a trivial derivation:

$$u_0 \;\Rightarrow^*\; u_0$$

The run:

$$(q,v) \ \vdash_{\varepsilon}^{*} \ (q,v), \qquad\qquad \text{where } v^r = u_0$$

satisfies property (†).

For the induction hypothesis, we assume that the lemma holds when the length of the derivation is $m-1$. We will prove the derivation of length m case for the induction step.

Consider a derivation as in (1). We may assume that it has the left-most substitution property. Suppose the derivation $u_0 \Rightarrow u_1$ is obtained by applying the rule $A \to s$, where variable A is the left-most variable in u_0. So, we can denote u_0 by xAy, where $x \in \Sigma^*$. and hence, $u_1 = xsy$. Since x contains only terminals, using transitions of the form $(q, a, \mathsf{pop}(a)) \to (q, \mathsf{push}(\varepsilon))$, where $a \in \Sigma$, there is a run:

$$(q, y^r A x^r) \ \vdash_x^* \ (q, y^r A). \tag{2}$$

By our construction of $\mathcal{A}$, there is a transition $(q, \varepsilon, \mathsf{pop}(A)) \to (q, \mathsf{push}(s^r))$. So, we have:

$$(q, y^r A) \ \vdash_{\varepsilon} \ (q, y^r s^r). \tag{3}$$

Moreover, x will not change during the derivation $u_0 \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_m$ (because x contains only terminals). Thus, the string u_i has prefix x, for every $i \in \{1,\ldots,m\}$. That is, $u_i = xu_i'$, for some $u_i' \in (\Sigma \cup V)^*$. So, the derivation is of the form:

$$xAy \ \Rightarrow \ xu_1' \ \Rightarrow \ \cdots \ \Rightarrow \ xu_m'.$$

Ignoring the first part of the derivation, we have:

$$xu_1' \ \Rightarrow \ \cdots \ \Rightarrow \ xu_m',$$

which means we also have derivation (since x contains only terminals):

$$u_1' \ \Rightarrow \ \cdots \ \Rightarrow \ u_m',$$

which has length $m-1$. By the induction hypothesis, there is a run:

$$(q, z_1) \ \vdash_t^* \ (q, z_2), \qquad\qquad \text{where } z_1^r = u_1' \text{ and } tz_2^r = u_{m+1}'. \tag{4}$$

Now, recall that $u_1 = xu_1' = xsy$, and hence, $z_1 = y^r s^r$. Combining the runs (2), (3) and (4), we have the following run:

$$(q, y^r A x^r) \ \vdash_x^* \ (q, y^r A) \ \vdash_{\varepsilon} \ (q, y^r s^r) \ \vdash_t^* \ (q, z_2)$$

which can be abbreviated as:

$$(q, y^r A x^r) \ \vdash_{xt}^* \ (q, z_2).$$

Since $tz_2^r = u_m'$, we have $xtz_2^r = xu_m' = u_m$. Since $u_0 = xAy$, this is the desired property (†). This completes the proof of Lemma 4.4. ∎

The next lemma is the converse direction of Lemma 4.4.

Lemma 4.5 *For every run of $\mathcal{A}$:*

$$(q,v) \ \vdash_w^* \ (q,\tilde{v}),$$

there is a derivation:

$$u \ \Rightarrow^* \ \tilde{u}$$

such that:

$$v^r \ = \ u \qquad \text{and} \qquad w\tilde{v}^r \ = \ \tilde{u} \tag{$\star$}$$

Proof. Suppose we have a run $(q, v) \vdash_w^* (q, \tilde{v})$. Let n be the length of the run, which is denoted as follows.

$$(q, v_0) \ \vdash_{b_1} \ (q, v_1) \ \vdash_{b_2} \ \cdots \ \vdash_{b_n} \ (q, v_n), \tag{5}$$

where $v_0 = v$, $v_n = \tilde{v}$ and $b_1 \cdots b_n = w$. We will prove the lemma by induction on n.

The base case is $n = 0$. In this case, the run is a trivial run:

$$(q, v_0) \ \vdash_\varepsilon^* \ (q, v_0)$$

Thus, the trivial derivation:

$$u \ \Rightarrow^* \ u \qquad\qquad \text{where } u = v_0^r$$

satisfies the desired property ($\star$).

For the induction hypothesis, we assume that the lemma holds when the length of the run is $n - 1$. We will prove the case when the length of the run is n for the induction step.

We consider a run of length n as in (5). We consider the step $(q, v_0) \vdash_{b_1} (q, v_1)$. There are two cases:

- $b_1 \neq \varepsilon$, i.e., b_1 is a symbol from Σ.

  By the construction of the PDA $\mathcal{A}$, it has to use the transition $(q, b_1, \mathsf{pop}(b_1)) \to (q, \mathsf{push}(\varepsilon))$, which means that the top of the stack v_0 is b_1. Therefore, $v_0 = v_1 b_1$.

  Now consider the run:
  $$(q, v_1) \ \vdash_{b_2} \ \cdots \ \vdash_{b_n} \ (q, v_n).$$

  This run is of length $n - 1$. By the induction hypothesis, we have a derivation:
  $$u_1 \ \Rightarrow^* \ u_2 \qquad\qquad \text{where } u_1 = v_1^r \text{ and } u_2 = b_2 \cdots b_n v_n^r.$$

  Adding b_1 in front of u_1 and u_2, we have:
  $$b_1 u_1 \ \Rightarrow^* \ b_1 u_2$$

  Now, $b_1 u_1 = b_1 v_1^r = v_0^r$ and $b_1 u_2 = b_1 b_2 \cdots b_{n+1} v_{n+1}^r$, which is the desired property ($\star$).

- $b_1 = \varepsilon$.

  By the construction of the PDA $\mathcal{A}$, it has to use the transition $(q, \varepsilon, \mathsf{pop}(A)) \to (q, \mathsf{push}(s^r))$, for some rule $A \to s$. This means that the top of the stack v_0 is A. We denote by $v_0 = yA$ and $v_1 = ys^r$, for some y.

  Consider the run:
  $$(q, v_1) \ \vdash_{b_2} \ \cdots \ \vdash_{b_n} (q, v_n).$$

  This run has length $n - 1$. By the induction hypothesis, there is a derivation:
  $$u_1 \ \Rightarrow^* \ u_2 \qquad\qquad \text{where } u_1 = v_1^r \text{ and } u_2 = b_2 \cdots b_n v_n^r$$

  Since $v_1 = ys^r$, and hence, $u_1 = sy^r$, we have a derivation:
  $$Ay^r \ \Rightarrow \ u_1 \ \Rightarrow^* \ u_2.$$

  By our notation, $v_0 = yA$ and $u_2 = b_2 \cdots b_n v_n^r = b_1 b_2 \cdots b_n v_n^r$, which is the desired property ($\star$).

This completes our proof of Lemma 4.5. ∎

Using the two lemmas above, we can show that $L(\mathcal{A}) = L(\mathcal{G})$ as stated below.

Theorem 4.6 *$L(\mathcal{G}) = L(\mathcal{A})$.*

Proof. We first prove $L(\mathcal{G}) \subseteq L(\mathcal{A})$. Let $S \Rightarrow^* w$, where $w \in \Sigma^*$. By Lemma 4.4, there is a run:

$$(q, S) \ \vdash_x^* \ (q, y), \qquad \text{where } xy^r = w.$$

Since w contains only symbols from Σ, so does y. If $y \neq \varepsilon$, using the transitions of the form $(q, a, \mathsf{pop}(a)) \to (q, \mathsf{push}(\varepsilon))$, we can extend the run until $y = \varepsilon$, which means that $x = w$.

By the construction of $\mathcal{A}$, we have the following accepting run on w:

$$(p, \varepsilon) \ \vdash_\varepsilon \ (q, S) \ \vdash_w^* \ (q, \varepsilon).$$

Thus, $w \in L(\mathcal{A})$.

Now we prove $L(\mathcal{G}) \supseteq L(\mathcal{A})$. Let $w \in L(\mathcal{A})$. So, there is an accepting run of $\mathcal{A}$ on w, which must start from the initial state p and end with the accepting state q after finishes reading the input word w. Thus, it has to start by using the transition $(p, \varepsilon, \mathsf{pop}(\varepsilon)) \to (q, \mathsf{push}(S))$. This means the run is of the form:

$$(p, \varepsilon) \ \vdash_\varepsilon \ (q, S) \ \vdash_w^* \ (q, \varepsilon).$$

By Lemma 4.5, there is a derivation:

$$u_1 \ \Rightarrow^* \ u_2, \qquad \text{where } u_1 = S \text{ and } u_2 = w.$$

Thus, $S \Rightarrow^* w$ and $w \in L(\mathcal{G})$. This completes the proof of Theorem 4.6. ∎

B From PDA to CFG

Let $\mathcal{A} = \langle \Sigma, \Gamma, Q, q_0, F, \delta \rangle$ be a PDA. Without loss of generality, we can assume the following.

- It has only one final state, say q_f. That is, $F = \{q_f\}$.
- The stack is empty before accepting an input word. More precisely, on every word w, if $\mathcal{A}$ accepts w, there is an accepting run of $\mathcal{A}$ on w from the initial configuration (q_0, ε) to a final configuration (q_f, ε) where the content of the stack is empty.
- In each transition, $\mathcal{A}$ either pushes a symbol into the stack or pops one from the stack, but it cannot do both. More precisely, every transition can only be of the forms:

$$\begin{array}{rcl} (p, x, \mathsf{pop}(y)) & \to & (q, \mathsf{push}(\varepsilon)) \\ (p, x, \mathsf{pop}(\varepsilon)) & \to & (q, \mathsf{push}(z)) \end{array}$$

Consider the following CFG $\mathcal{G} = \langle \Sigma, V, R, S \rangle$ where each component is as follows.

- $V = \{A_{p,q} \mid p, q \in Q\}$.
- A_{q_0,q_f} is the start variable.
- R consists of the following rules:

– For every state $p, q, r, s \in Q$ and every symbol $z \in \Gamma$ and every symbol $a, b \in \Sigma \cup \{\varepsilon\}$, if the following transitions are in δ:

$$\begin{array}{rcl} (p, a, \mathsf{pop}(\varepsilon)) & \to & (r, \mathsf{push}(z)) \\ (s, b, \mathsf{pop}(z)) & \to & (q, \mathsf{push}(\varepsilon)) \end{array}$$

then the following rule is in R:

$$A_{p,q} \ \to \ a \ A_{r,s} \ b \qquad (\mathcal{R}1)$$

– For every state $p, q, r \in Q$ and every symbol $a \in \Sigma \cup \{\varepsilon\}$, if the following transition is in δ:

$$(p, a, \mathsf{pop}(\varepsilon)) \ \to \ (r, \mathsf{push}(\varepsilon))$$

then the following rule is in R:

$$A_{p,q} \ \to \ a \ A_{r,q} \qquad (\mathcal{R}2)$$

– For every state $p, q, r \in Q$, we have the following rule in R:

$$A_{p,q} \ \to \ A_{p,r} \ A_{r,q} \qquad (\mathcal{R}3)$$

– For every $p \in Q$, we have the following rule in R:

$$A_{p,p} \ \to \ \varepsilon \qquad (\mathcal{R}4)$$

We will show that $L(\mathcal{A}) = L(\mathcal{G})$. First, we prove the following lemma.

Lemma 4.7 *For every derivation:*

$$A_{p,q} \ \Rightarrow^* \ w, \qquad \textit{where } w \in \Sigma^*$$

there is a run:

$$(p, \varepsilon) \ \vdash_w^* \ (q, \varepsilon)$$

Proof. Suppose $A_{p,q} \Rightarrow^* w$, where $w \in \Sigma^*$. Let the derivation of length m and denoted as follows.

$$A_{p,q} \ \Rightarrow \ w_1 \ \Rightarrow \ \cdots \ \Rightarrow \ w_m, \qquad \text{where } w_m = w. \qquad (6)$$

We will prove the lemma by induction on m. The base case is $m = 1$. So, we have a derivation:

$$A_{p,q} \ \Rightarrow \ w$$

The only rule that can be used in this case is rule $(\mathcal{R}4)$, which means $w = \varepsilon$ and $p = q$. Thus, there is a (trivial) run:

$$(p, \varepsilon) \ \vdash_\varepsilon^* \ (p, \varepsilon)$$

For the induction hypothesis, we assume the lemma holds for every derivation of length $\leqslant m-1$. We will prove the case of derivations of length m for the induction step.

Let the derivation be as in (6). We consider the rules applied on the step $A_{p,q} \Rightarrow w_1$. There are three cases:

- The rule applied is ($\mathcal{R}1$) type of rule, say:

$$A_{p,q} \quad \to \quad aA_{r,s}b$$

This means that $w_1 = aA_{r,s}b$, and hence, every w_i starts with a and ends with b, for every $i \in \{1, \ldots, m\}$. We denote each $w_i = aw_i'b$, for some w_i'. Thus, there is a derivation:

$$A_{r,s} \;\Rightarrow\; w_2' \;\Rightarrow\; \cdots \;\Rightarrow\; w_m'$$

This derivation has length $m - 1$. By the induction hypothesis, there is a run:

$$(r, \varepsilon) \quad \vdash_{w'_{m+1}} \quad (s, \varepsilon) \tag{7}$$

Since $A_{p,q} \to aA_{r,s}b$ is a rule in R, there are the following two transitions in δ:

$$(p, a, \mathsf{pop}(\varepsilon)) \to (r, \mathsf{push}(z)) \quad \text{and} \quad (s, b, \mathsf{pop}(z)) \to (q, \mathsf{push}(\varepsilon)) \tag{8}$$

Now, from the run (7), since the content of the stack never goes "below empty", we also have a run:

$$(r, z) \quad \vdash_{w'_{m+1}} \quad (s, z) \tag{9}$$

Using the transitions (8), we have:

$$(p, \varepsilon)) \vdash_a (r, z) \quad \text{and} \quad (s, z) \vdash_b (q, \varepsilon) \tag{10}$$

Combining the runs (9) and (10), we have:

$$(p, \varepsilon)) \;\vdash_a\; (r, z) \;\vdash^*_{w'_{m+1}}\; (s, z) \vdash_b \; (q, \varepsilon)$$

That is,

$$(p, \varepsilon)) \quad \vdash^*_{aw'_{m+1}b} \quad (q, \varepsilon)$$

Since $w = aw'_{m+1}b$, the run is as desired.

- The rule applied is ($\mathcal{R}3$) type of rule, say:

$$A_{p,q} \quad \to \quad A_{p,r}A_{r,q}, \qquad\qquad \text{for some } r \in Q.$$

Thus, $w_1 = A_{p,r}A_{r,q}$. This means $w = xy$ where $A_{p,r} \Rightarrow^* x$ and $A_{r,q} \Rightarrow^* y$. Both are derivations of length $\leqslant m - 1$. By the induction hypothesis, there are the following two runs:

$$(p, \varepsilon) \vdash^*_x (r, \varepsilon) \quad \text{and} \quad (r, \varepsilon) \vdash^*_y (q, \varepsilon)$$

Combining these two runs, we have:

$$(p, \varepsilon) \vdash^*_w (r, \varepsilon)$$

- The rule applied is ($\mathcal{R}2$) type of rule, say:

$$A_{p,q} \quad \to \quad aA_{r,q}, \text{for some } r \in Q \text{ and } a \in \Sigma$$

Thus, $w_1 = aA_{r,q}$. This means that every w_i starts with a, for every $i \in \{1, \ldots, m\}$. We denote each $w_i = aw'_i$, for some w'_i. Thus, there is a run:

$$A_{r,q} \;\Rightarrow\; w'_2 \;\Rightarrow\; \cdots \;\Rightarrow\; w'_m.$$

The length of this run is $m-1$. By the induction hypothesis, there is a run:

$$(r,\varepsilon) \quad \vdash^*_{w'_m} \quad (q,\varepsilon) \tag{11}$$

Now, since there is a rule $A_{p,q} \to aA_{r,q}$, there is the following transition in δ:

$$(p, a, \mathsf{pop}(\varepsilon)) \quad \to \quad (r, \mathsf{push}(\varepsilon))$$

Using this transition, we have:

$$(p,\varepsilon) \quad \vdash_a \quad (r,\varepsilon)$$

Combining this with run (11), we have run:

$$(p,\varepsilon) \;\vdash_a\; (r,\varepsilon) \;\vdash^*_{w'_m}\; (q,\varepsilon)$$

Thus, there is a run:

$$(p,\varepsilon) \quad \vdash^*_{aw'_m} \quad (q,\varepsilon)$$

This run is as required.

This completes the proof fo Lemma 4.7. ▮

The following lemma is the converse direction of the previous lemma.

Lemma 4.8 *For every run:*

$$(p,\varepsilon) \quad \vdash^*_w \quad (q,\varepsilon)$$

there is a derivation:

$$A_{p,q} \quad \Rightarrow^* \quad w$$

Proof. Suppose there is a run $(p,\varepsilon) \vdash^*_w (q,\varepsilon)$. Let the run has length n, denoted as follows.

$$(p_0,v_0) \;\vdash_{b_1}\; (p_1,v_1) \;\vdash_{b_2}\; \cdots \;\vdash_{b_n}\; (p_n,v_n) \tag{12}$$

where $p_0 = p$, $v_0 = \varepsilon$, $p_n = q$, $v_n = \varepsilon$ and $b_1 \cdots b_n = w$. We will prove the lemma by induction on n.

The base case is $n = 0$. In this case the run is a trivial run:

$$(p_0,\varepsilon) \quad \vdash^*_\varepsilon \quad (p_0,\varepsilon)$$

Applying the $(\mathcal{R}4)$ type of rule $A_{p_0,p_0} \to \varepsilon$, we have a derivation:

$$A_{p_0,p_0} \quad \Rightarrow \quad \varepsilon$$

For the induction hypothesis, we assume that the lemma holds for run of length $\leqslant n-1$. We will now prove it for runs of length n for the induction step.

Consider a run of length n as in (12). We consider the transition used in the step $(p_0,v_0) \vdash_{b_1} (p_1,v_1)$. There are two cases:

- The transition used is $(p_0, b_1, \mathsf{pop}(\varepsilon)) \to (p_1, \mathsf{push}(\varepsilon))$, i.e., the PDA $\mathcal{A}$ pops and pushes nothing to its stack. By definition of the CFG $\mathcal{G}$, there is a $(\mathcal{R}2)$ type of rule:

$$A_{p_0,p_n} \quad \to \quad b_1 A_{p_1,p_n} \tag{13}$$

Applying the induction hypothesis on the run:

$$(p_1, v_1) \ \vdash_{b_2} \ \cdots \ \vdash_{b_n} \ (p_n, v_n)$$

there is a derivation:

$$A_{p_1,p_n} \ \Rightarrow^* b_2 \cdots b_n$$

Moreover, using rule (13), we have derivation:

$$A_{p_0,p_n} \ \Rightarrow b_1 A_{p_1,p_n} \ \Rightarrow^* b_1 b_2 \cdots b_n$$

as desired.

- The transition used is of the form: $(p_0, b_1, \mathsf{pop}(\varepsilon)) \to (p_1, \mathsf{push}(z))$, for some $z \in \Gamma$.

  There are two more cases here.

  - For some $j \in \{2, \ldots, n\}$, $v_j = \varepsilon$.
    Thus, there is a run:

$$(p_0, v_0) \ \vdash_x^* \ (p_j, v_j) \ \vdash_y^* \ (p_n, v_n) \qquad\qquad \text{where } w = xy.$$

By induction hypothesis, there are derivations:

$$A_{p_0,p_j} \ \Rightarrow^* \ x \quad \text{and} \quad A_{p_j,p_n} \ \Rightarrow^* \ y$$

By construction of $\mathcal{G}$, there is a $(\mathcal{R}3)$ type of rule:

$$A_{p_0,p_n} \quad \to \quad A_{p_0,p_j} A_{p_j,p_n}.$$

Using this rule, we have the following derivation:

$$A_{p_0,p_n} \ \Rightarrow \ A_{p_0,p_j} A_{p_j,p_n} \ \Rightarrow^* \ xy$$

This is the desired derivation, since $w = xy$.

  - For every $i \in \{1, \ldots, n-1\}$, $v_i \neq \varepsilon$.
    This means that z is popped from v_{n-1} to obtain v_n and there are the following transitions:

$$(p_0, a, \mathsf{pop}(\varepsilon)) \to (p_1, \mathsf{push}(z)) \quad \text{and} \quad (p_{n-1}, b, \mathsf{pop}(z)) \to (p_n, \mathsf{push}(\varepsilon)) \tag{14}$$

where $b_1 = a$ and $b_n = b$. Since $v_n = \varepsilon$, this means $v_{n-1} = z$. In particular, every v_i starts with z, for every $i \in \{1, \ldots, n-1\}$. We denote by $v_i = zv_i'$, for some $i \in \{1, \ldots, n-1\}$. This means there is a run:

$$(p_1, z) \ \vdash_{w'}^* \ (p_{n-1}, z) \qquad\qquad \text{where } w = aw'b$$

Since during this run, z is untouched, we have a run:

$$(p_1, \varepsilon) \ \vdash_{w'}^* \ (p_{n-1}, \varepsilon)$$

This run has length $\leqslant n-1$. By the induction hypothesis, there is a derivation:

$$A_{p_1,p_{n-1}} \ \Rightarrow * \ w'$$

Due to the transitions in (14), there is a ($\mathcal{R}1$) type of rule:

$$A_{p_0,p_n} \quad \to \quad aA_{p_1,p_{n-1}}b$$

Using this rule, there is a derivation:

$$A_{p_0,p_n} \ \Rightarrow \ aA_{p_1,p_{n-1}}b \ \Rightarrow^* \ aw'b.$$

Since $aw'b = w$, this derivation is as desired.

This completes the proof of Lemma 4.8. ▮

Theorem 4.9 $L(\mathcal{A}) = L(\mathcal{G})$.

Proof. We first prove $L(\mathcal{G}) \supseteq L(\mathcal{A})$. Let $w \in L(\mathcal{A})$. So, there is an accepting run of $\mathcal{A}$ on w, which must start from the initial state q_0 and end with the accepting state q_f:

$$(q_0, \varepsilon) \quad \vdash_w^* \quad (q_f, \varepsilon)$$

By Lemma 4.8, we have:

$$A_{q_0,q_f} \quad \Rightarrow^* \quad w.$$

Thus, $w \in L(\mathcal{G})$.

Now we prove that $L(\mathcal{G}) \subseteq L(\mathcal{A})$. Let $w \in L(\mathcal{G})$. Thus,

$$A_{q_0,q_f} \quad \Rightarrow^* \quad w.$$

By Lemma 4.7, there is a run:

$$(q_0, \varepsilon) \quad \vdash_w^* \quad (q_f, \varepsilon).$$

Since this is an accepting run, $w \in L(\mathcal{A})$. This completes the proof of Theorem 4.9. ▮

Lesson 5: Turing machines

Theme: Turing machines as a model of general computation.

We reserve a special symbol $\sqcup$ called the *blank* symbol and $\lhd$ called the *left-end* symbol.

A *Turing machine* (TM) is a system $\mathcal{M} = \langle \Sigma, \Gamma, Q, q_0, q_{\text{acc}}, q_{\text{rej}}, \delta \rangle$, where each component is as follows.

- Σ is a finite alphabet, called the *input* alphabet, where $\sqcup, \lhd \notin \Sigma$.
- Γ is a finite alphabet, called the *tape* alphabet, where $\Sigma \subseteq \Gamma$ and $\sqcup, \lhd \in \Gamma$.
- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- $q_{\text{acc}}, q_{\text{rej}} \in Q$ are two special states called the *accept* and *reject* states, respectively.
- $\delta : (Q - \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma \to Q \times \Gamma \times \{\texttt{Left}, \texttt{Right}\}$ is the transition function.

The intuitive meaning of $\delta(p, a) = (q, b, \alpha)$ is as follows. When the head reads a symbol a, if $\mathcal{M}$ is in state p, it "writes" symbol b on top of a, enters state q, and the head moves left, if $\alpha = \texttt{Left}$, or moves right, if $\alpha = \texttt{Right}$.

Remark 5.1 We assume that the left-most cell of a Turing machine always contains the symbol $\lhd$. So, the input word to a Turing machine is always of the form $\lhd w$, where $w \in \Sigma^*$. For this reason, we require that any Turing machine *does not* write $\lhd$ on any other cell except on the left-most cell. It is also assumed that the head of the Turing machine always move right when it reads $\lhd$, i.e., for every transition $\delta(p, \lhd) = (q, b, \alpha)$, the move α is always $\texttt{Right}$.

To describe how a TM computes, we need a few terminologies. A configuration of $\mathcal{M}$ is a string C from $(Q \cup \Gamma)^*$ which contains $\underline{\textit{exactly one symbol}}$ from Q. We call such symbol the state of C. Intuitively, a configuration $C = \lhd a_1 \cdots a_{i-1}\ p a_i \cdots a_m$ means the content of the tape is:

$$\lhd a_1 \cdots a_{i-1} a_i \cdots a_m\ \sqcup \sqcup \sqcup \cdots$$

with the head reading a_i and the Turing machine is in state p.

On input word $w \in \Sigma^*$, the *initial* configuration of $\mathcal{M}$ on w is the string $\lhd q_0 w$. A configuration is called *accepting*, if it contains q_{acc}, and it is called *rejecting*, if it contains q_{rej}. A *halting* configuration is either an accepting or a rejecting configuration.

Let $C = \lhd a_1 \cdots a_{i-1}\ p a_i \cdots a_m$ be a configuration, where $a_1, \ldots, a_m \in \Gamma$ and $p \in Q$ such that $p \neq q_{\text{acc}}, q_{\text{rej}}$. The transition δ yields the subsequent configuration C', denoted by $C \vdash C'$, as follows.

- If $\delta(p, a_i) = (q, b, \texttt{Left})$ and $i \geqslant 2$, then $C' = \lhd a_1 \cdots a_{i-2}\ q a_{i-1} b a_{i+1} \cdots a_m$.
- If $\delta(p, a_i) = (q, b, \texttt{Left})$ and $i = 1$, then $C' = q \lhd b a_2 \cdots a_m$.
- If $\delta(p, a_i) = (q, b, \texttt{Right})$ and $i \leqslant m - 1$, then $C' = \lhd a_1 \cdots a_{i-1} b\ q a_{i+1} \cdots a_m$.
- If $\delta(p, a_i) = (q, b, \texttt{Right})$ and $i = m$, then $C' = \lhd a_1 \cdots a_{m-1} b\ q \sqcup$.

The *run* of $\mathcal{M}$ on w is the (possibly infinite) sequence:

$$C_0 \ \vdash\ C_1 \ \vdash\ C_2 \ \vdash\ \cdots, \qquad (1)$$

where C_0 is the initial configuration of $\mathcal{M}$ on w.

$\mathcal{M}$ stops when it reaches a halting configuration, i.e., when it reaches either q_{acc} or q_{rej}. If $\mathcal{M}$ halts in an accepting configuration, we say that $\mathcal{M}$ *accepts* w. If it halts in a rejecting configuration, we say that $\mathcal{M}$ *rejects* w.

Remark 5.2 Note that in our definition of Turing machine, we assume the left-end marker $\lhd$ and we require the transitions are defined so that the the head does not go beyond this marker. We make this assumption to prevent an overly technical presentation to address issue/question such as "what happens if the head fall off the left-end of the tape?" As we will see in our next lesson, the purpose of Turing machines is to serve as a precise mathematical model of algorithms where questions like that is irrelevant. In fact, throughout this class we will implicitly assume $\lhd$ as the left-end marker, so it is not necessary to mention it explicitly when we define a Turing machine.

However, we should note that in most textbook the definition of Turing machine does not require such left-end marker and this is fine too. Adding such a marker will not increase/decrease (at least in theory) what can be computed by Turing machines.

Important terminologies:

- We say that $\mathcal{M}$ *recognizes* a language L, if:

  (i) for every word $w \in L$, $\mathcal{M}$ accepts w;
  (ii) for every word $w \notin L$, $\mathcal{M}$ does not accept w.

  Note that $\mathcal{M}$ does not accept w have two meanings: either $\mathcal{M}$ rejects w, or $\mathcal{M}$ does not halt on w.

- We say that $\mathcal{M}$ *decides* a language L, if:

  (i) for every word $w \in L$, $\mathcal{M}$ accepts w;
  (ii) for every word $w \notin L$, $\mathcal{M}$ rejects w.

  Note that this implies $\mathcal{M}$ halts on every word $w \in \Sigma^*$.

- A language L is *recognizable/recursively enumerable* (r.e.), if there is a TM $\mathcal{M}$ that recognizes L.

- A language L is *decidable/recursive*, if there is a TM $\mathcal{M}$ that decides L.

  Otherwise, it is called *undecidable*.

Appendix

A Turing machines with `Stay` option

In some textbooks, Turing machines are defined such that the head can stay put, instead of moving `Left` or `Right`. Formally, a transition can be of the form:

$$(q,a) \to (p,b,\alpha), \qquad \text{where } \alpha \in \{\texttt{Left},\texttt{Right},\texttt{Stay}\}$$

If $\alpha = \texttt{Stay}$, then the head stays where it is. Such `Stay` option is obviously equivalent to making two moves: `Right`, and followed by `Left`, thus, does not add any power of computation.

B Encoding an arbitrary alphabet into the binary alphabet $\{0,1\}$

Turing machines are usually defined with arbitrary input and tape alphabets. It is not difficult to show that any alphabet can be "encoded" with binary alphabet.

Suppose $\Gamma = \{a_1, \ldots, a_n, \sqcup\}$. Each symbol a_i can then be encoded with a 0-1 string of length $\lceil \log_2 n \rceil$. For example, if $\Gamma = \{a_1, \ldots, a_5, \sqcup\}$, we can encode a_1 with 000, a_2 with 001, a_3 with 010, a_4 with 011, and a_5 with 100. We denote by $\langle a_i \rangle$ the encoding of the symbol a_i. For a word $w \in \Gamma^*$, $\langle w \rangle$ denotes the encoding of w by replacing each symbol a_i in w with $\langle a_i \rangle$. For example, if $w = a_1a_5a_2a_1$, $\langle w \rangle = \langle a_1 \rangle\langle a_5 \rangle\langle a_2 \rangle\langle a_1 \rangle = 000\ 100\ 001\ 000$.

We have the following proposition that shows that we can always assume that the Turing machines under consideration always work on tape alphabet $\Gamma = \{\lhd, 0, 1, \sqcup\}$, where $\lhd$ is the marker that marks the leftmost cell of the tape.

Proposition 5.3 *Let $\mathcal{M} = \langle \Sigma, \Gamma, Q, q_0, q_{acc}, q_{rej}, \delta \rangle$ be a TM, where $\Gamma = \{a_1, \ldots, a_n, \sqcup\}$. Let $K = \lceil \log_2 n \rceil$. Let $\langle a_i \rangle$ be an encoding of symbol a_i with 0-1 string of length K. There is a TM $\mathcal{M}' = \langle \{0,1\}, \{\lhd, 0, 1, \sqcup\}, Q', q_0', q_{acc}, q_{rej}, \delta' \rangle$ such that for every word $w \in \Sigma^*$, the following holds.*

$$\mathcal{M} \text{ accepts } w \quad \text{if and only if} \quad \mathcal{M}' \text{ accepts } \langle w \rangle$$

Intuitively, $\mathcal{M}'$ simulates $\mathcal{M}$ by reading the tapes by blocks of $\lceil \log_2 n \rceil$ cells. It then remembers the block that it reads in its states, and "simulates" the transitions of $\mathcal{M}$ accordingly.

Formally, $\mathcal{M} = \langle \{0,1\}, \{0,1,\sqcup\}, Q', q_0, q_{\text{acc}}, q_{\text{rej}}, \delta' \rangle$ is defined as follows. Let $\{0,1\}^{\leqslant K}$, i.e., the set of all 0-1 strings of length less than or equal to $K = \lceil \log_2 n \rceil$.

- $Q' = (Q \times \{0,1\}^{\leqslant K}) \ \cup \ (Q \times \{\underline{\mathsf{L}}_1, \ldots, \underline{\mathsf{L}}_K, \underline{\mathsf{R}}_1, \ldots, \underline{\mathsf{R}}_K\})$
  $\cup \ (Q \times \{\underline{\mathsf{L}}, \underline{\mathsf{R}}\} \times \{\underline{\mathsf{W}}\} \times \{0,1\}^{\leqslant K})$.
- $q_0' = (q_0, \epsilon)$.
- δ' is defined as follows.
  - For every $u \in \{0,1\}^{\leqslant K-1}$, for every $p \in Q - \{q_{\text{acc}}, q_{\text{rej}}\}$, δ' consists of the following transitions.

$$\begin{aligned} ((p,u),0) &\rightarrow ((p,u0),0,\texttt{Right}) \\ ((p,u),1) &\rightarrow ((p,u1),1,\texttt{Right}) \end{aligned}$$

  - For every $(q,a) \rightarrow (p,b,\texttt{Left}) \in \delta$, for every $d \in \{0,1,\sqcup\}$, δ' consists of the following transitions.

$$((q,\langle a \rangle),d) \rightarrow ((p,\underline{\mathsf{L}},\underline{\mathsf{W}},\langle b \rangle),d,\texttt{Left})$$

  - For every $(q,a) \rightarrow (p,b,\texttt{Right}) \in \delta$, for every $d \in \{0,1,\sqcup\}$, δ' consists of the following transitions.

$$((q,\langle a \rangle),d) \rightarrow ((p,\underline{\mathsf{R}},\underline{\mathsf{W}},\langle b \rangle),d,\texttt{Left})$$

  - For every $p \in Q$, for every $c \in \{0,1\}$, for every $v \in \{0,1\}^{\leqslant K-1}$ and $v \neq \epsilon$, for every $d \in \{0,1,\sqcup\}$, for every $\beta \in \{\underline{\mathsf{L}}, \underline{\mathsf{R}}\}$, δ' consists of the following transitions.

$$((p,\beta,\underline{\mathsf{W}},vc),d) \rightarrow ((p,\beta,\underline{\mathsf{W}},v),c,\texttt{Left})$$

- For every $p \in Q$, for every $d \in \{\lhd, 0, 1, \sqcup\}$, δ' consists of the following transitions.

$$((p, \underline{\mathsf{L}}, \underline{\mathsf{W}}, \epsilon), d) \quad \rightarrow \quad ((p, \underline{\mathsf{L}}_k), d, \texttt{Right})$$

- For every $p \in Q$, for every $d \in \{\lhd, 0, 1, \sqcup\}$, δ' consists of the following transitions.

$$((p, \underline{\mathsf{R}}, \underline{\mathsf{W}}, \epsilon), d) \quad \rightarrow \quad ((p, \underline{\mathsf{R}}_k), d, \texttt{Right})$$

- For every $p \in Q$, for every $d \in \{0, 1, \sqcup\}$, δ' consists of the following transitions.

$$((p, \underline{\mathsf{L}}, \underline{\mathsf{W}}, \epsilon), d) \quad \rightarrow \quad ((p, \underline{\mathsf{L}}_k), d, \texttt{Right})$$

- For every $p \in Q$, for every $i \in \{2, \ldots, k\}$, for every $d \in \{0, 1, \sqcup\}$, δ' consists of the following transitions.

$$\begin{array}{rcl} ((p, \underline{\mathsf{R}}_i), d) & \rightarrow & ((p, \underline{\mathsf{R}}_{i-1}), d, \texttt{Right}) \\ ((p, \underline{\mathsf{L}}_i), d) & \rightarrow & ((p, \underline{\mathsf{L}}_{i-1}), d, \texttt{Left}) \end{array}$$

- For every $p \in Q$, for every $d \in \{0, 1, \sqcup\}$, δ' consists of the following transitions.

$$\begin{array}{rcl} ((p, \underline{\mathsf{R}}_1), d) & \rightarrow & ((p, \epsilon), d, \texttt{Right}) \\ ((p, \underline{\mathsf{L}}_1), d) & \rightarrow & ((p, \epsilon), d, \texttt{Left}) \end{array}$$

All the other transitions not specified above are assumed to enter q_{rej}.

Lesson 6: Turing machines and the notion of algorithms

Theme: Turing machines and the notion of algorithms.

1 Multi-tape Turing machines

A multi-tape Turing machine is a Turing machine that has a few tapes. On each tape, the Turing machine has one head. Formally, it is defined as follows. Let $k \geqslant 1$. A k-tape Turing machine is a system $\mathcal{M} = \langle \Sigma, \Gamma, Q, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \delta \rangle$, where δ is the transition function:

$$\delta \quad : \quad (Q - \{q_{\mathrm{acc}}, q_{\mathrm{rej}}\}) \times \Gamma^k \;\to\; Q \times \Gamma^k \times \{\texttt{Left}, \texttt{Right}\}^k$$

A transition in δ is written in the form:

$$(q, a_1, \ldots, a_k) \to (p, b_1, \ldots, b_k, \alpha_1, \ldots, \alpha_k).$$

Intuitively, it means that if the TM is in state q, and on each $i = 1, \ldots, k$, the head on tape i is reading a_i, then it enters state p, and for $i = 1, \ldots, k$, the head on tape i writes the symbol b_i and moves according to α_i.

A *configuration* of $\mathcal{M}$ is a string of the form $(q, \triangleleft u_1, \ldots, \triangleleft u_k)$, where $q \in Q$, each u_i is a string over $\Gamma \cup \{\bullet\}$ and the symbol $\bullet$ appears exactly once in each u_i. The symbol $\bullet$ denotes the position of the head. As before, the symbol $\triangleleft$ is the left-end marker of each tape.

The *initial configuration* of $\mathcal{M}$ on input w is $(q_0, \triangleleft \bullet w, \triangleleft \bullet, \ldots, \triangleleft \bullet)$, i.e., the first tape initially contains the input word and all the other tapes are initially blank. The notion of "one step computation" $C \vdash C'$ is defined similarly as in the standard Turing machine. Likewise, the conditions of acceptance and rejection are defined as when the Turing machine enters the accepting and rejecting states, respectively.

Theorem 6.1 *For every k-tape TM $\mathcal{M}$, where $k \geqslant 2$, there is a single tape TM $\mathcal{M}'$ such that for every input word w, the following holds.*

- *If $\mathcal{M}$ accepts w, then $\mathcal{M}'$ accepts w.*
- *If $\mathcal{M}$ rejects w, then $\mathcal{M}'$ rejects w.*
- *If $\mathcal{M}$ does not halt on w, then $\mathcal{M}'$ does not halt on w.*

Proof. Let $\mathcal{M} = \langle \Sigma, \Gamma, Q, q_0, F, \delta \rangle$ be a k-tape TM with $k \geqslant 2$. We will design a TM $\mathcal{M}'$ that simulates the run of $\mathcal{M}$ on w using only one tape. The idea is that a configuration $(q, \triangleleft u_1, \ldots, \triangleleft u_k)$ (of $\mathcal{M}$) can be viewed as a string over the alphabet $Q \cup \Gamma \cup \{\bullet, \tilde{\triangleleft}\}$:

$$q \tilde{\triangleleft} u_1 \cdots \tilde{\triangleleft} u_k$$

This string can be stored in just one tape. Here $\tilde{\triangleleft}$ is a new symbol to represent the symbol $\triangleleft$ of $\mathcal{M}$.

As an algorithm, $\mathcal{M}'$ works as follows. On input word w, do the following.

- Let C be the string $q_0 \tilde{\triangleleft} \bullet w \tilde{\triangleleft} \bullet \cdots \tilde{\triangleleft} \bullet$, i.e., the initial configuration of $\mathcal{M}$ on w.
- While C is not a halting configuration of $\mathcal{M}$, do the following.
  - Scan the string C from left to right to find out the symbol read by each "head."

  - Move the head back to the beginning of the tape.
  - Change the state and the position of each head in C according to the transition function δ.

    This can be done by scanning the string C from left to right and when it encounters the symbol $\bullet$, it change its "position" according to δ.

- If C is an accepting configuration, ACCEPT. If C is a rejecting configuration, REJECT.

Note that $\mathcal{M}'$ uses only one string variable C, so in principle it suffices to use only one tape to store this string C. Note that when the head in tape-1 moves right to the new cell (i.e., the length of u_1 increases), $\mathcal{M}'$ has to "shift" right all the strings $\lhd u_2, \cdots, \lhd u_k$.

Alternatively we can also "compress" the content of each cell i on every tape into one symbol. For example, for $k = 3$, a configuration $(q, \lhd 01 \bullet 1\sqcup, \lhd 1 \bullet \sqcup, \lhd 0 \bullet 1\sqcup)$ can be encoded as:

$$q, \lhd\ (0,1,0)\ (1, \bullet\sqcup, \bullet 1)\ (\bullet 1, \sqcup, \sqcup)\ \sqcup)$$

where each $(0,1,0)(1,\bullet\sqcup,\bullet 1),(\bullet 1,\sqcup,\sqcup)$ is represented by a symbol in the tape alphabet of $\mathcal{M}'$. This encoding can be illustrated as follows.

tape-1	$\lhd$	0	1	$\bullet 1$	$\sqcup$
tape-2	$\lhd$	1	$\bullet\sqcup$	$\sqcup$	$\sqcup$
tape-3	$\lhd$	0	$\bullet 1$	$\sqcup$	$\sqcup$
the encoding	$\lhd$	$(0,1,0)$	$(1,\bullet\sqcup,\bullet 1)$	$(\bullet 1,\sqcup,\sqcup)$	$\sqcup$

More precisely, the string C is of the form $(q, \lhd a_1 a_2 \cdots a_n \sqcup)$, where each $a_i \in (\Gamma \cup \bullet\Gamma)^k$ and $\bullet\Gamma$ denotes a new alphabet whose symbols are used to represent $\bullet a$, for each $a \in \Gamma$. ▮

Remark 6.2 In the proof of Theorem 6.1, the TM $\mathcal{M}'$ simulates the run of $\mathcal{M}$ by remembering the "last" configuration C which contains the state q. Since the TM $\mathcal{M}$ has only some fixed number of states, the TM $\mathcal{M}'$ can actually remember the state q in its state. So, in principle the string C does not need to contain the state q. It suffices that C contains only the content of each tape, i.e., of the form: $\tilde{\lhd} u_1 \cdots \tilde{\lhd} u_k$.

Remark 6.3 Intuitively one can view a tape in TM as a variable in a computer program. A k-tape TM can be viewed as a computer program that uses k variables. Likewise, a computer program that uses k variables can be viewed as a k-tape TM. See the Appendix.

In view of Remark 6.3, to avoid being overly technical in our presentation, the term "Turing machine" and "algorithm" will often be used interchangeably. When we describe a TM (especially in the proofs in this and the subsequent lessons), we will often just describe an algorithm in some acceptable format. We will write capital ACCEPT to denote that the TM enters the accept state and REJECT to denote that the TM enters the reject state.

2 Some theorems on decidable and recognizable languages

Theorem 6.4

- *If a language L (over the alphabet Σ) is decidable, so is its complement $\Sigma^* - L$.*
- *If both a language L and its complement $\Sigma^* - L$ are recognizable, then L is decidable.*

Proof. The first item is trivial. If a language L is decidable, then by definition, there is a TM $\mathcal{M}$ that decides L. The TM that decides its complement can be obtained by simply switching the accepting and rejecting states: The accept state becomes reject state and the reject state becomes accept.

Now we prove the second item. Suppose L and its complement $\Sigma^* - L$ are recognizable. Let $\mathcal{M}_1$ be the TM that recognizes L and $\mathcal{M}_2$ be the TM that recognizes $\Sigma^* - L$. By Theorem 6.1, we may assume that they are both 1-tape TM.

W describe a 2-tape TM $\mathcal{M}'$ that decides L as follows. On input word w, do the following.

- Copy the input word w into the second tape.
- Run the TM $\mathcal{M}_1$ on the first tape and the TM $\mathcal{M}_2$ on the second tape simultaneously.
- If $\mathcal{M}_1$ accepts, then ACCEPT. If $\mathcal{M}_2$ accepts, then REJECT.

Note that for every word $w \in \Sigma^*$, either $w \in L$ or $w \in \Sigma^* - L$. Thus, by definition of $\mathcal{M}_1$ and $\mathcal{M}_2$, either w is accepted by $\mathcal{M}_1$ or by $\mathcal{M}_2$. That is, exactly one of $\mathcal{M}_1$ or $\mathcal{M}_2$ will accept w. If $\mathcal{M}_1$ accepts, it means $w \in L$, so the TM $\mathcal{M}'$ accepts. If $\mathcal{M}_2$ accepts, it means $w \in \Sigma^* - L$, so the TM $\mathcal{M}'$ rejects.

"To run $\mathcal{M}_1$ on the first tape and $\mathcal{M}_2$ on the second tape simultaneously," we mean that the head for tape-1 is used to "run" $\mathcal{M}_1$ and the head for tape-2 to "run" $\mathcal{M}_2$. More formally, at this stage the states of $\mathcal{M}'$ are of the form (p_1, p_2), where p_1 is a state of $\mathcal{M}_1$ and p_2 is a state of $\mathcal{M}_2$. According the symbols read by each head, it then applies the transitions in $\mathcal{M}_1$ and $\mathcal{M}_2$ on each head separately. This is very similar to the proof of Theorem 1.3 (in Lesson 1), where to prove that regular languages are closed under intersection, we can "run" two DFA simultaneously by taking the Cartesian product $Q_1 \times Q_2$ as the set of states. ∎

Theorem 6.5 *Decidable languages are closed under union, intersection, concatenation and Kleene star.*

Proof. In the following let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the TM that decide languages L_1 and L_2, respectively. We may assume that both are 1-tape TM. The proof is actually similar to the second item of Theorem 6.4.

(Closure under union) The TM $\mathcal{M}_\cup$ that recognizes $L_1 \cup L_2$ works as follows. On input word w, it runs $\mathcal{M}_1$ and then $\mathcal{M}_2$ on w. It accepts if and only if at least one of $\mathcal{M}_1$ or $\mathcal{M}_2$ accepts.

(Closure under intersection) Similar to union case.

(Closure under concatenation) The TM $\mathcal{M}_\cdot$ that recognizes $L_1 \cdot L_2$ works as follows. On input word w, enumerate all possible pairs (v_1, v_2) such that $v_1 v_2 = w$. On each pair (v_1, v_2), check if $\mathcal{M}_1$ accepts v_1 and $\mathcal{M}_2$ accepts v_2. $\mathcal{M}_\cdot$ accepts if and only if there is a pair (v_1, v_2) where v_1 is accepted by $\mathcal{M}_1$ and v_2 is accepted by $\mathcal{M}_2$.

Note that there are only $|w| + 1$ possible pairs (v_1, v_2) where $v_1 v_2 = w$. (It is possible that v_1 or v_2 is ε.)

(Closure under Kleene star) The TM $\mathcal{M}_\star$ that recognizes L_1^* works as follows. On input word w, enumerate all possible tuples $(v_1, \ldots, v_m)$ such that $v_1 \cdots v_m = w$. On each tuple $(v_1, \ldots, v_m)$, check if each v_i is accepted by $\mathcal{M}_1$. $\mathcal{M}_\star$ accepts if and only if there is a tuple $(v_1, \ldots, v_m)$ where each v_i is accepted by $\mathcal{M}_1$.

Note that we can easily write an algorithm that enumerates all possible tuples $(v_1, \ldots, v_m)$ where $v_1 \cdots v_m = w$. ∎

Theorem 6.6 *Recognizable languages are closed under union and intersection.*

Proof. In the following let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the TM that recognize languages L_1 and L_2, respectively. We may assume that both are 1-tape TM. The proof is actually similar to the second item of Theorem 6.4.

(Closure under union) The TM $\mathcal{M}_\cup$ that recognizes $L_1 \cup L_2$ works as follows. It has two tapes. First, it copies the input word into the second tape. Then, it runs $\mathcal{M}_1$ and $\mathcal{M}_2$ on the first and second tape simultaneously. It accepts if and only if at least one of $\mathcal{M}_1$ or $\mathcal{M}_2$ accepts.

By definition of $\mathcal{M}_1$ and $\mathcal{M}_2$, every word $w \in L_1 \cup L_2$ is accepted by at least one of $\mathcal{M}_1$ or $\mathcal{M}_2$. Thus, $\mathcal{M}_\cup$ recognizes the language $L_1 \cup L_2$ correctly.

(Closure under intersection) Similar to above. ∎

Remark 6.7 We should note that recognizable languages are also closed under concatenation and Kleene star. In fact, we can already prove it in this lesson, but the proof will be quite technical. So we will postpone the proof until Lesson 9. There we will use the notion of "non-deterministic" TM to obtain a neater and clearer proof.

We should stress that recognizable languages are *not* closed under complement. We will see this in Lesson 7.

Appendix

A An informal definition of algorithm

We define an algorithm (informally) as consisting of one "main" Boolean function of the form:

```
Boolean main (string w)
{       statement;
          ⋮
        statement;
}
```

and some (finite number of) functions of the form:

```
⟨value-type⟩ function ⟨function-name⟩ (⟨var-name⟩,...,⟨var-name⟩)
{       statement;
          ⋮
        statement;
}
```

Statements in the algorithm are of the following form:

- ⟨var-name⟩ := ⟨expression⟩;
- ⟨var-name⟩ := ⟨function-name⟩(⟨var-name⟩,...,⟨var-name⟩);
- **return** ⟨variable-name⟩/⟨some-value⟩;

- **if** ⟨`condition`⟩

```
{       statement;
          ⋮
        statement;
} else
{       statement;
          ⋮
        statement;
}
```

Note that we define our algorithm to mimic closely the C++ language so that we can be "convinced" every C++ program can be rewritten as our algorithm.

In our algorithm variables can only store Boolean or string values. Note that there is no `while`-loop, since it can be implemented as a recursive function.

We loosely define an "expression" as any reasonable "basic" computation which includes:

- Concatenating two strings.
- Shift left/right of a string.
- Change the symbol in a position in a string.

Since 0-1 strings can be used to represent numbers, "basic" computation also includes:

- Adding/subtracting/multiplying/dividing two numbers.
- Enumerating all the numbers between 1 and some number n.
- Measuring the length of a 0-1 string.
- Enumerating all the 0-1 strings with length between 1 and some number n.

"Condition" for `if` statement includes:

- Checking whether two numbers are equal, or whether one number is bigger than the other.
- Checking whether two strings are equal, or whether one string is lexicographically "bigger" than the other.

One can argue that every C++ computer program can be written as an algorithm defined above. Note that when we write an algorithm (or any computer program, in fact), it uses only a fixed number of variables (including variables for data structures such as linked list, arrays, etc). Multi-tape Turing machines and our definition of algorithms above are equivalent in the sense that a k-tape Turing machine can be viewed as an algorithm that uses k variables, and conversely, an algorithm that uses k variables can be viewed as a k-tape Turing machine.

Lesson 7: Universal Turing machines and the halting problem

Theme: Universal Turing machines and the halting problem.

1 The string representation of a Turing machine

Recall that a Turing machine is a system $\mathcal{M} = \langle \Sigma, \Gamma, Q, q_0, q_{\text{acc}}, q_{\text{rej}}, \delta \rangle$. In the following we will assume that $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, \sqcup\}$. Without loss of generality, we may also assume that $Q = \{0, 1, \ldots, n\}$ for some positive integer n.

We note the following.

- Each state $i \in Q$ can be written as a string in its binary form.
- Each transition $(i, a) \to (j, b, \alpha) \in \delta$ can be written as a string over the alphabet:

$$\{0,\ 1,\ (,\ ),\ \diamond, \to,\ \tilde{\sqcup},\ \mathtt{L},\ \mathtt{R}\}$$

where the symbol $\diamond$ represents the comma, $\tilde{\sqcup}$ represents $\sqcup$, and L, R represent `Left`, `Right`, respectively. For example, a transition $(5, \sqcup) \to (8, 1, \mathtt{Right})$ is written as the string:

$$(101 \diamond \tilde{\sqcup}) \to (1000 \diamond 1 \diamond \mathtt{R})$$

So, the whole system $\mathcal{M} = \langle \Sigma, \Gamma, Q, 0, q_{\text{acc}}, q_{\text{rej}}, \delta \rangle$ can be written as a string:

$$\lfloor \Sigma \rfloor \ \# \ \lfloor \Gamma \rfloor \ \# \ \lfloor Q \rfloor \ \# \ \lfloor 0 \rfloor \ \# \ \lfloor q_{\text{acc}} \rfloor \ \# \ \lfloor q_{\text{rej}} \rfloor \ \# \ \lfloor \delta \rfloor$$

where $\lfloor \cdot \rfloor$ denotes the string representing the component $\cdot$ and # the symbol separating two consecutive components.

For example, if $\mathcal{M}$ is a 1-tape TM where $Q = \{0, \ldots, 45\}$, 0 is the initial state, 3 is q_{acc} and 4 is q_{rej}, it is written as a string:

$$0 \diamond 1 \ \# \ 0 \diamond 1 \diamond \tilde{\sqcup} \ \# \ 45 \ \# \ 0 \ \# \ 3 \ \# \ 4 \ \# \ \underbrace{\cdots\cdots\cdots\cdots\cdots\cdots}_{\text{the list of the transitions}}$$

Recall that we use the symbol $\diamond$ to represent a comma here. Note also that we do not list all the states $0, \ldots, 45$. It suffices to write 45 to indicate that the states are all the numbers between 0 and 45.

This shows that every 1-tape TM (whose tape alphabet is $\Gamma = \{0, 1, \sqcup\}$) can be described as a string over the alphabet $\{0,\ 1,\ (,\ ),\ \diamond, \to,\ \tilde{\sqcup},\ \mathtt{L},\ \mathtt{R},\ \#\}$. Each of these symbols can be further encoded as 0-1 string of length 4. For example, 0 is encoded as 0000, 1 as 0001 and so on, as shown in the following table.

symbol	the encoding
0	0000
1	0001
(	0010
)	0011
$\diamond$	0100

symbol	the encoding
$\to$	0101
$\tilde{\sqcup}$	0110
L	0111
R	1000
#	1001

We denote by $\lfloor \mathcal{M} \rfloor$ the binary string obtained by such encoding and we call $\lfloor \mathcal{M} \rfloor$ *the binary string representation* of the Turing machine $\mathcal{M}$, or *the description of* $\mathcal{M}$.

Remark 7.1 Note that we can easily extend the definition of $\lfloor \mathcal{M} \rfloor$ for TM $\mathcal{M}$ with multiple tapes. In this course, $\lfloor \mathcal{M} \rfloor$ denotes the string representation of $\mathcal{M}$ where $\mathcal{M}$ can be any TM with multiple tapes.

Note that a string w over the alphabet $\{0,\ 1,\ (,\ ),\ \diamond, \rightarrow,\ \tilde{\sqcup},\ \mathtt{L},\ \mathtt{R},\ \#\}$ is the string representation of a Turing machine, if it is of the form:

$$u_1 \# u_2 \# u_3 \# u_4 \# u_5 \# u_6 \# u_7$$

that is, the symbol $\#$ appears exactly 6 times and each string u_i satisfies the following.

- u_1 is $0 \diamond 1$.
- u_2 is $0 \diamond 1 \diamond \tilde{\sqcup}$.
- u_3 is an integer n (written in binary form).
- u_4, u_5, u_6 are all the binary form of some numbers between 0 and n.
- u_7 is a string such that for every $(i, a) \in \{0, \ldots, n\} \times \{0, 1, \tilde{\sqcup}\}$, there is *exactly one* $(j, b, \alpha) \in \{0, \ldots, n\} \times \{0, 1, \tilde{\sqcup}\} \times \{\mathtt{L}, \mathtt{R}\}$ such that $(i \diamond a) \rightarrow (j \diamond b \diamond \alpha)$ appears in u_7.

We can easily write an algorithm/computer program that on input string w over the alphabet $\{0,\ 1,\ (,\ ),\ \diamond, \rightarrow,\ \tilde{\sqcup},\ \mathtt{L},\ \mathtt{R},\ \#\}$, it checks whether w satisfies all these properties. In similar manner, we can also write an algorithm/computer program that on input string w over $\{0, 1\}$, it checks whether w is the binary string representation of a Turing machine. This observation is summarized formally as the following proposition.

Proposition 7.2 *There is an algorithm $\mathcal{A}$ for the following problem.*

Verifying the description of a Turing machine	
Input:	A string w over the alphabet $\{0, 1\}$.
Task:	Output `True`, if w is indeed the description of a TM $\mathcal{M}$, i.e. $w = \lfloor \mathcal{M} \rfloor$.
	Output `False`, otherwise.

Note that the algorithm $\mathcal{A}$ depends on the 0-1 encoding of the symbols 0, 1, (,), $\diamond$, $\rightarrow$, $\tilde{\sqcup}$, L, R and #. Under different encoding, a different algorithm is needed for verifying the description of a Turing machine. Throughout this course we will assume the fixed encoding shown in the table above. Hence, we also assume a fixed algorithm $\mathcal{A}$ for verifying the description of a Turing machine.

2 Universal Turing machines

Definition 7.3 A *universal Turing machine* (UTM) is a Turing machine $\mathcal{U}$ that on input $\lfloor \mathcal{M} \rfloor \$ w$, where $w \in \{0, 1\}^*$, does the following.

- If $\mathcal{M}$ accepts w, then $\mathcal{U}$ accepts $\lfloor \mathcal{M} \rfloor \$ w$.
- If $\mathcal{M}$ rejects w, then $\mathcal{U}$ rejects $\lfloor \mathcal{M} \rfloor \$ w$.
- If $\mathcal{M}$ does not halt on w, then $\mathcal{U}$ does not halt on $\lfloor \mathcal{M} \rfloor \$ w$.

Intuitively, a UTM $\mathcal{U}$ works as follows. On input $\lfloor \mathcal{M} \rfloor \$ w$, it simulates $\mathcal{M}$ on w, i.e., it constructs the run of $\mathcal{M}$ on w. The way it works is actually similar to the proof of Theorem 6.1, except that now the TM $\mathcal{M}$ is given as part of the input. More precisely, on input word u, it does the following.

- Check if u is of the form $v\$w$, where $v, w \in \{0,1\}^*$.
- Check if v is indeed the description of a TM $\mathcal{M}$, i.e., $v = \lfloor \mathcal{M} \rfloor$, by using the algorithm in Proposition 7.2.

  If it is not, REJECT. Otherwise, continue.
- Construct the initial configuration of $\mathcal{M}$ on w and store it as a string C.
- while (C is not a halting configuration):
  - Compute the next configuration of C (by accessing the transition of $\mathcal{M}$).
- If C is an accepting configuration, ACCEPT. If C is a rejecting configuration, REJECT.

It is obvious that: (*i*) if $\mathcal{M}$ accepts w, then $\mathcal{U}$ accepts $\lfloor \mathcal{M} \rfloor\$w$; (*ii*) if $\mathcal{M}$ rejects w, then $\mathcal{U}$ rejects $\lfloor \mathcal{M} \rfloor\$w$ and (*iii*) if $\mathcal{M}$ does not halt on w, then $\mathcal{U}$ does not halt on $\lfloor \mathcal{M} \rfloor\$w$.

Similar to algorithm $\mathcal{A}$ in Proposition 7.2, a UTM is defined according to the encoding of the descriptions of the Turing machines. Since we assume that we only use one fixed encoding, throughout this course we will also assume a fixed UTM $\mathcal{U}$.

Remark 7.4 At this point we would like to clarify on the meaning of the phrases "run a TM $\mathcal{M}$ on w" and "simulate a TM $\mathcal{M}$ on w."

Intuitively, "run a TM $\mathcal{M}$ on w" means that we view a TM $\mathcal{M}$ as a procedure/function (written, say, in C++) and we "call" it with input w. On the other hand, "simulate a TM $\mathcal{M}$ on w" essentially means that we construct the run of $\mathcal{M}$ on w (assuming that we have access to the transitions of $\mathcal{M}$).[*]

3 The halting problem

We define the following languages:

$$\begin{aligned}
\mathsf{HALT} &:= \{\lfloor \mathcal{M} \rfloor\$w \mid \mathcal{M} \text{ accepts } w \text{ where } w \in \{0,1\}^*\}.\\
\mathsf{HALT}_0 &:= \{\lfloor \mathcal{M} \rfloor \mid \mathcal{M} \text{ accepts } \lfloor \mathcal{M} \rfloor\}.\\
\mathsf{HALT}'_0 &:= \{\lfloor \mathcal{M} \rfloor \mid \mathcal{M} \text{ does not accept } \lfloor \mathcal{M} \rfloor\}.
\end{aligned}$$

Note that we can use the UTM $\mathcal{U}$ to recognize the language HALT and we can also easily modify the UTM $\mathcal{U}$ to recognize the language HALT_0. We state this formally as the following proposition.

Proposition 7.5 *The language HALT_0 and HALT are recognizable.*

Theorem 7.6 *HALT'_0 is undecidable.*

Proof. Suppose to the contrary that HALT'_0 is decidable. Let $\mathcal{B}$ be the TM that decides HALT'_0. We examine whether $\mathcal{B}$ accept its own description $\lfloor \mathcal{B} \rfloor$. There are two cases.

- If $\mathcal{B}$ accepts $\lfloor \mathcal{B} \rfloor$.

  Since $\mathcal{B}$ decides HALT'_0, this means $\lfloor \mathcal{B} \rfloor \in \mathsf{HALT}'_0$. By the definition of HALT'_0, $\mathcal{B}$ does not accept $\lfloor \mathcal{B} \rfloor$. A contradiction.

[*]Sometimes these two phrases are used interchangeably, since the end results (in terms of accept/reject) are usually the same. However, the two processes have different run time.

- If $\mathcal{B}$ rejects $\lfloor \mathcal{B} \rfloor$.

  Since $\mathcal{B}$ decides HALT'_0, this means $\lfloor \mathcal{B} \rfloor \notin \mathsf{HALT}'_0$. By the definition of HALT'_0, $\mathcal{B}$ accepts $\lfloor \mathcal{B} \rfloor$. A contradiction.

Both cases yield contradiction. Thus, there is no such TM $\mathcal{B}$ that decides HALT'_0, i.e., HALT'_0 is undecidable. ∎

We should note that Theorem 7.6 actually states the same thing as Theorem 0.1 in Lesson 0. The only difference is that Theorem 7.6 is formulated in term of the Turing machines while Theorem 0.1 is formulated in term of the C++ programs.

Corollary 7.7 *HALT_0 and HALT are undecidable.*

Proof. It follows immediately from Theorem 7.6. ∎

Recall that if both L and its complement $\overline{L} = \Sigma^* - L$ are recognizable, then both are decidable. Then, the following corollary follows immediately from Proposition 7.5 and Theorem 7.6.

Corollary 7.8 *The language $\overline{\mathsf{HALT}'_0}$ is not recognizable.*

Lesson 8: Reducibility

Theme: Reductions as a tool to prove undecidability.

1 Reductions

Definition 8.1 Let $F : \Sigma^* \to \Sigma^*$ be a function from Σ^* to Σ^*. We say that *a Turing machine* $\mathcal{M}$ *computes the function* F is a 2-tape Turing machine that accepts every word $w \in \Sigma^*$ and when it halts, the content of its second tape is $F(w)$.

Note that for $\mathcal{M}$ to compute F, the content of the first tape can be anything when it halts. That is, on every word w, $\mathcal{M}$ accepts w with the accepting run:

$$(q_0, \bullet w, \bullet) \ \vdash \ \cdots \ \vdash \ (q_{\text{acc}}, u, \bullet F(w))$$

for some string u (which denotes the content of the first tape).

Note that due to the equivalence between multi-tape Turing machines and 1-tape Turing machines, the Turing machine $\mathcal{M}$ that computes F is not necessarily 2-tape. It can be any multi-tape Turing machine with a designated tape that contains the output string.

Definition 8.2 A function $F : \Sigma^* \to \Sigma^*$ is *computable*, if there is a Turing machine $\mathcal{M}$ that computes it.

Definition 8.3 A language L_1 is *mapping reducible* to another language L_2, denoted by:

$$L_1 \ \leqslant_m \ L_2,$$

if there is a computable function F such that for every $w \in \Sigma^*$:

$$w \in L_1 \quad \text{if and only if} \quad F(w) \in L_2$$

The function f is called *mapping reduction.*

Sometimes we omit the word "mapping" and call it simply "reducible" or "reduction," instead of "mapping reducible" or "mapping reduction." Intuitively $L_1 \leqslant_m L_2$ means that L_2 is "computationally more general," or "more general" than L_1 and that a Turing machine that decides L_2 can be used to decide L_1.

Definition 8.4 A language L_1 is *Turing reducible* to another language L_1, denoted by $L_1 \leqslant_T L_2$, if by assuming that L_2 is decidable by a TM $\mathcal{M}_2$, there is a TM $\mathcal{M}_1$ that decides L_1 using $\mathcal{M}_2$ as a "subroutine."

Moreover, we also assume that $\mathcal{M}_2$ decides L_2 in *one* step. We call $\mathcal{M}_1$ a TM *with oracle access to* L_2.

Remark 8.5 Some observations:

- If $L_1 \leqslant_m L_2$, then $L_1 \leqslant_T L_2$.
- If $L_1 \leqslant_T L_2$ and L_1 is undecidable, then L_2 is also undecidable.

2 Some variants of the halting problem

In the following for a Turing machine $\mathcal{M}$, we denote by $L(\mathcal{M})$ the set of all words accepted by $\mathcal{M}$.

We will show that the following languages are undecidable.

- $L_0 := \{\lfloor\mathcal{M}\rfloor \mid L(\mathcal{M}) = \emptyset\}$.

  That is, $\lfloor\mathcal{M}\rfloor \in L_0$ if and only if $\mathcal{M}$ does not accept any word.
- $L_1 := \{\lfloor\mathcal{M}\rfloor \mid L(\mathcal{M}) = \{0,1\}^*\}$.

  That is, $\lfloor\mathcal{M}\rfloor \in L_1$ if and only if $\mathcal{M}$ accepts every word.
- $L_2 := \{\lfloor\mathcal{M}\rfloor \mid \mathcal{M}$ accepts the empty word $\epsilon\}$

  That is, $\lfloor\mathcal{M}\rfloor \in L_2$ if and only if $\mathcal{M}$ accepts the empty word ϵ.
- $L_3 := \{\lfloor\mathcal{M}\rfloor \mid \mathcal{M}$ accepts the word $1101\}$.
- $L_4 := \{\lfloor\mathcal{M}\rfloor \mid L(\mathcal{M}) = \{a^n b^n \mid n \geqslant 0\}\}$.
- $L_5 := \{\lfloor\mathcal{M}\rfloor \mid L(\mathcal{M})$ is a regular language$\}$.

In the following we will illustrate how to prove that L_0 and L_4 are undecidable by both mapping and Turing reductions. The proof can then be generalized easily to establish the so called *Rice's theorem.*

Proof that L_0 is undecidable (via mapping reduction). We are going to show that $\mathsf{HALT} \leqslant_m \overline{L}_0$, where $\overline{L}_0$ is the complement of L_0. The reduction is as follows.

INPUT: $\lfloor\mathcal{M}\rfloor\$w$.

- Construct a TM $\mathcal{K}_{\mathcal{M},w}$ that works as follows.

  INPUT: $u \in \Sigma^*$.
  - Run $\mathcal{M}$ on w.
  - If $\mathcal{M}$ accepts w, ACCEPT.
  - If $\mathcal{M}$ rejects w, REJECT.

  (Note: ACCEPT and REJECT above are inside $\mathcal{K}_{\mathcal{M},w}$, thus, they are supposed to mean $\mathcal{K}_{\mathcal{M},w}$ accepts and rejects its input string u, respectively.)
- Output $\lfloor\mathcal{K}_{\mathcal{M},w}\rfloor$.

The language accepted by $\mathcal{K}_{\mathcal{M},w}$ is as follows.

$$L(\mathcal{K}_{\mathcal{M},w}) \quad := \quad \begin{cases} \Sigma^*, & \text{if } \mathcal{M} \text{ accepts } w \\ \emptyset, & \text{if } \mathcal{M} \text{ does not accept } w \end{cases}$$

Thus, $\mathcal{M}$ accepts w if and only if $L(\mathcal{K}_{\mathcal{M},w}) \neq \emptyset$. By definition of L_0 and HALT, $\lfloor\mathcal{M}\rfloor\$w \in \mathsf{HALT}$ if and only if $\lfloor\mathcal{K}_{\mathcal{M},w}\rfloor \notin L_0$. Since HALT is undecidable, $\overline{L}_0$ is undecidable, and therefore, L_0 is undecidable.

Proof that L_0 is undecidable (via Turing reduction). Assume to the contrary that L_0 is decidable. Let $\mathcal{M}_0$ be a TM that decides L_0. The following algorithm, denoted by $\mathcal{M}^*$, decides HALT.

```
INPUT: $\lfloor \mathcal{M} \rfloor \$ w$.
• Construct a TM $\mathcal{K}_{\mathcal{M},w}$ that works as follows.
    INPUT: $u \in \Sigma^*$.
    – Run $\mathcal{M}$ on $w$.
    – If $\mathcal{M}$ accepts $w$, ACCEPT. (Note: ACCEPT here is for $\mathcal{K}_{\mathcal{M},w}$ to accept $u$.)
    – If $\mathcal{M}$ rejects $w$, REJECT. (Note: REJECT here is for $\mathcal{K}_{\mathcal{M},w}$ to reject $u$.)
• Run $\mathcal{M}_0$ on $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor$.
• If $\mathcal{M}_0$ accepts $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor$, REJECT. (Note: REJECT here is for $\mathcal{M}^*$ to reject $\lfloor \mathcal{M} \rfloor \$ w$.)
• If $\mathcal{M}_0$ rejects $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor$, ACCEPT. (Note: ACCEPT here is for $\mathcal{M}^*$ to accept $\lfloor \mathcal{M} \rfloor \$ w$.)
```

Now that the language $L(\mathcal{K}_{\mathcal{M},w})$ is:

$$L(\mathcal{K}_{\mathcal{M},w}) \quad := \quad \begin{cases} \Sigma^*, & \text{if } \mathcal{M} \text{ accepts } w \\ \emptyset, & \text{if } \mathcal{M} \text{ does not accept } w \end{cases}$$

Thus, $\lfloor \mathcal{M} \rfloor \$ w \in \mathsf{HALT}$ if and only if $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor \notin L_0$. Since $\mathcal{M}_0$ is supposed to decide L_0, our algorithm $\mathcal{M}^*$ above decides HALT, which contradicts the fact that HALT is undecidable. Therefore, there is no such Turing machine $\mathcal{M}_0$ that decides L_0, which means L_0 is undecidable.

Proof that L_4 is undecidable (via mapping reduction). First, note that the language $\{a^n b^n | n \geqslant 0\}$ is decidable and we denote by $\mathcal{A}$ the TM that decides it.

We now show that $\mathsf{HALT} \leqslant_m L_4$. The reduction is as follows.

```
INPUT: $\lfloor \mathcal{M} \rfloor \$ w$.
• Construct a TM $\mathcal{K}_{\mathcal{M},w}$ that works as follows.
    INPUT: $u \in \Sigma^*$.
    – Run $\mathcal{A}$ on $u$. (to check if $u$ is of the form $a^n b^n$, for some $n \geqslant 0$.)
    – If $\mathcal{A}$ rejects $u$, REJECT.
    – If $\mathcal{A}$ accepts $u$, then run $\mathcal{M}$ on $w$.
        * If $\mathcal{M}$ accepts $w$, ACCEPT.
        * If $\mathcal{M}$ rejects $w$, REJECT.
  (Note: ACCEPT and REJECT above are inside $\mathcal{K}_{\mathcal{M},w}$, thus, they are supposed to mean
  $\mathcal{K}_{\mathcal{M},w}$ accepts and rejects its input string $u$, respectively.)
• Output $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor$.
```

The language accepted by $\mathcal{K}_{\mathcal{M},w}$ is as follows.

$$L(\mathcal{K}_{\mathcal{M},w}) \quad := \quad \begin{cases} \{a^n b^n \mid n \geqslant 0\}, & \text{if } \mathcal{M} \text{ accepts } w \\ \emptyset, & \text{if } \mathcal{M} \text{ does not accept } w \end{cases}$$

Thus, $\mathcal{M}$ accepts w if and only if $L(\mathcal{K}_{\mathcal{M},w}) = \{a^n b^n \mid n \geqslant 0\}$. By definition of HALT and L_4, $\mathcal{M}\$ w \in \mathsf{HALT}$ if and only if $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor \in L_4$. Since HALT is undecidable, L_4 is undecidable too.

Rice's theorem. As mentioned earlier, the statement that all languages L_0–L_4 are undecidable is actually a special case of the so called *Rice's theorem* which states as follows. Let P be a set of the descriptions of Turing machines. We say that P is a *property*, if for for every Turing machines $\mathcal{M}_1$ and $\mathcal{M}_2$, if $L(\mathcal{M}_1) = L(\mathcal{M}_2)$, then either both $\lfloor \mathcal{M}_1 \rfloor$ and $\lfloor \mathcal{M}_2 \rfloor$ are in P, or both are not in P. Intuitively, for a set P to be a property, the criteria for the membership of $\lfloor \mathcal{M} \rfloor$ depends on the language $L(\mathcal{M})$, and *not* on the string $\lfloor \mathcal{M} \rfloor$ itself. A property P is called a *trivial* property, if it is either $\emptyset$ or contains all the descriptions of the Turing machines.

Theorem 8.6 (Rice's theorem) *For a property P, if P is not a trivial property, then P is undecidable.*

Proof. The proof is actually a straightforward generalization of the proof of the undecidablity of L_4 above.

Let P be a non-trivial property. We first assume that P does *not* contain the description of a Turing machine $\lfloor \mathcal{M} \rfloor$ where $L(\mathcal{M}) = \emptyset$. Let $\mathcal{A}$ be a Turing machine where $\lfloor \mathcal{A} \rfloor \in P$. Such $\mathcal{A}$ exists since P is not a trivial property.

We show that $\mathsf{HALT} \leqslant_m P$ by the following reduction.

```
INPUT: $\lfloor \mathcal{M} \rfloor \$ w$.
• Construct a TM $\mathcal{K}_{\mathcal{M},w}$ that works as follows.
      INPUT: $u \in \Sigma^*$.
    – Run $\mathcal{A}$ on $u$. (to check if $u \in L(\mathcal{A})$)
    – If $\mathcal{A}$ rejects $u$, REJECT.
    – If $\mathcal{A}$ accepts $u$, then run $\mathcal{M}$ on $w$.
        * If $\mathcal{M}$ accepts $w$, ACCEPT.
        * If $\mathcal{M}$ rejects $w$, REJECT.
• Output $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor$.
```

We examine the language $L(\mathcal{K}_{\mathcal{M},w})$.

- If $\mathcal{M}\$w \in \mathsf{HALT}$, then $L(\mathcal{K}_{\mathcal{M},w}) = L(\mathcal{A})$, and hence, $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor \in P$.

  This is because P is a property and $\lfloor \mathcal{A} \rfloor \in P$.
- If $\mathcal{M}\$w \notin \mathsf{HALT}$, then $L(\mathcal{K}_{\mathcal{M},w}) = \emptyset$, and hence, $\lfloor \mathcal{K}_{\mathcal{M},w} \rfloor \notin P$.

  This is because P is a property and we assume that P does not contain the description of a Turing machine $\lfloor \mathcal{M} \rfloor$ where $L(\mathcal{M}) = \emptyset$.

Thus,

$$\mathcal{M}\$w \in \mathsf{HALT} \qquad \text{if and only if} \qquad \lfloor \mathcal{K}_{\mathcal{M},w} \rfloor \in P$$

This proves that $\mathsf{HALT} \leqslant_m P$. Hence, P is undecidable.

Next, we consider the case where P contains the description of a Turing machine $\lfloor \mathcal{M} \rfloor$ where $L(\mathcal{M}) = \emptyset$. In this case the complement of P, denoted by $\overline{P}$, does *not* contain the description of a Turing machine $\lfloor \mathcal{M} \rfloor$ where $L(\mathcal{M}) = \emptyset$. We have shown above that $\overline{P}$ is undecidable. Hence, P is also undecidable. ∎

Remark 8.7 Note that when a property P is trivial, it is decidable. If $P = \emptyset$, it is obviously decidable. When P contains all the descriptions of the Turing machines, we can use the algorithm $\mathcal{A}$ in Proposition 7.2 to decide P.

3 Some undecidable problems concerning CFL

Consider the following problem

CFL-Intersection	
Input:	Two CFG's $\mathcal{G}_1 = \langle \Sigma, V_1, R_1, S_1 \rangle$ and $\mathcal{G}_2 = \langle \Sigma, V_2, R_2, S_2 \rangle$, where $\Sigma = \{0, 1\}$.
Task:	Output `True`, if $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) \neq \emptyset$. Otherwise, output `False`.

This problem can be viewed as a language:

$$\mathsf{CFL\text{-}Intersection} \;:=\; \{\lfloor \mathcal{G}_1 \rfloor \$ \lfloor \mathcal{G}_2 \rfloor \mid L(\mathcal{G}_1) \cap L(\mathcal{G}_2) \neq \emptyset\}$$

where $\lfloor \mathcal{G} \rfloor$ denotes the encoding of the CFG $\mathcal{G}$ as a string over some fixed alphabet. For example, we can encode a CFG over Σ as a string over the alphabet $\Sigma \cup \{0, 1, \langle, \rangle, \rightarrow, \diamond, \#\}$ as follows. Let $\mathcal{G}$ be a CFG with n variables. The variables can be represented as $\langle i \rangle$, where i is an integer (written in binary) between 1 and n. Each rule, say for example, $S \rightarrow 0X11$ is written as $\langle 0 \rangle \rightarrow 0\langle 3 \rangle 11$ (assuming that S is represented as 0 and X as 3).

Theorem 8.8 *The problem* CFL-Intersection *is undecidable.*

Proof. We will show that HALT $\leqslant_m$ CFL-Intersection. In the proof we assume that HALT contains only the string $\lfloor \mathcal{M} \rfloor \$ w$ where $\mathcal{M}$ is a *1-tape* Turing machine and $\mathcal{M}$ accepts w. Such HALT is still undecidable.

We first make a few observations. First, for a Turing machine $\mathcal{M}$, we can add a "new" state q_{loop} such that on every input word w, if $\mathcal{M}$ rejects w, instead of entering the q_{rej}, $\mathcal{M}$ enters q_{loop} and loops forever. By adding one more state, if necessary, we also assume that for every word w, if $\mathcal{M}$ accepts w, the run of $\mathcal{M}$ on w has odd "length." That is, the run is:

$$C_0 \;\vdash\; C_1 \;\vdash\; C_2 \;\vdash\; C_3 \;\vdash\; \cdots \;\vdash\; C_n \tag{1}$$

where n is odd. So, for every input word w, the following holds.

(P1) If $\mathcal{M}$ accepts w, then the run is finite, i.e., $C_0 \vdash C_1 \;\vdash\; \cdots \;\vdash\; C_n$ and n is odd.

(P2) If $\mathcal{M}$ does not w, then the run is infinite.

Recall that the states of a Turing machines $\mathcal{M}$ are represented as numbers written in binary form. Thus, the run (1) can be viewed as a string over the alphabet $\{\vdash, 0, 1, \tilde{\sqcup}, [,]\}$, where we write $[i]$ to represent the state in the configuration.

We will present an algorithm $\mathcal{A}$ such that on input $\lfloor \mathcal{M} \rfloor \$ w$, it constructs two CFG $\mathcal{G}_1$ and $\mathcal{G}_2$ such that the following holds. If the run of $\mathcal{M}$ on w is finite:

$$C_0 \;\vdash\; C_1 \;\vdash\; C_2 \;\vdash\; C_3 \;\cdots\; \vdash\; C_n \tag{2}$$

then $L(\mathcal{G}_1) \cap L(\mathcal{G}_2)$ contains exactly one word and that word is:

$$C_0 \;\vdash\; C_1^r \;\vdash\; C_2 \;\vdash\; C_3^r \;\vdash\; \cdots \;\vdash\; C_n^r \tag{3}$$

where C_i^r denotes the reverse of C_i. We call the string in (3) *the reverse representation* of the run in (2).

We define $\mathcal{G}_1$ as a CFG that generates words of the form:

$$u_0 \;\vdash\; u_1 \;\vdash\; u_2 \;\vdash\; u_3 \;\cdots\; \vdash\; u_n \tag{4}$$

such that the following holds.

(a) n is an odd number.

(b) u_0 is the initial configuration of $\mathcal{M}$ on w.

(c) $u_{i-1} \vdash u_i^r$, for each odd i in between 1 and n.

That is, the string in 4 is almost the reverse representation of the run, except that the steps on even i are not guaranteed to obey the transitions of $\mathcal{M}$ and the last string u_n is not guaranteed to be an accepting configuration, i.e., to contain $[q_{\text{acc}}]$.

Note that the rules for $\mathcal{G}_1$ can be defined as:

$$\begin{array}{lcl} S & \to & X \vdash Y \\ X & \to & C_0 \vdash C_1^r \end{array} \qquad \begin{array}{lcl} Y & \to & Z \vdash Y \\ Y & \to & \varepsilon \end{array}$$

Variable Z generates words of the form: $u \vdash v^r$. The rules for generating words of this form can be constructed by accessing the transitions of $\mathcal{M}$. For example, for each transition $(p, a) \to (q, b, \texttt{Left})$, where $a, b \in \{0, 1, \sqcup\}$, we have the following rules:

$$\begin{array}{lcl} Z & \to & 0\ Z\ 0 \\ Z & \to & 1\ Z\ 1 \\ Z & \to & \tilde{\sqcup}\ Z\ \tilde{\sqcup} \\ Z & \to & 0[p]a\ T\ b0[q] \\ Z & \to & 1[p]a\ T\ b1[q] \\ Z & \to & \sqcup[p]a\ T\ b\sqcup[q] \end{array} \qquad \begin{array}{lcl} T & \to & 0\ T\ 0 \\ T & \to & 1\ T\ 1 \\ T & \to & \tilde{\sqcup}\ T\ \tilde{\sqcup} \\ T & \to & \vdash \\ & & \\ & & \end{array}$$

Recall that the states p and q are written in binary form. Variable Y generates words of the form: $u_0 \vdash u_1 \vdash u_2 \vdash u_3 \cdots \vdash u_n$, where $u_{i-1} \vdash u_i^r$, for each odd i in between 1 and n.

Next, we define $\mathcal{G}_2$ as a CFG that generates words of the form (4) such that:

(d) $u_{i-1}^r \vdash u_i$, for each even i in between 1 and n.

(e) The last string u_n contains $[q_{\text{acc}}]$.

In other words, $\mathcal{G}_2$ generates words of the form (4) where the steps on even i are guaranteed to obey the transitions of $\mathcal{M}$ and the last string u_n contains $[q_{\text{acc}}]$. Again, the rules for generating words of this form can be constructed by accessing the transitions of $\mathcal{M}$.

To summarize these observations, we present the following reduction for establishing $\mathsf{HALT} \leqslant_m$ $\mathsf{CFL\text{-}Intersection}$. On input $\lfloor \mathcal{M} \rfloor \$ w$, do the following.

- Add some new states to $\mathcal{M}$ so that (P1) and (P2) hold.
- Construct the CFG $\mathcal{G}_1$ that generates words of the form (4) where (a)–(c) hold.
- Construct the CFG $\mathcal{G}_2$ that generates words of the form (4) where (d) holds.
- Output $\lfloor \mathcal{G}_1 \rfloor \$ \lfloor \mathcal{G}_2 \rfloor$.

To prove that the reduction is correct, observe that if $\mathcal{M}$ accepts w, then the run is finite. By the construction of $\mathcal{G}_1$ and $\mathcal{G}_2$, the reverse representation of the run is in $L(\mathcal{G}_1) \cap L(\mathcal{G}_2)$. Likewise, if a string $u_0 \vdash u_1 \vdash \cdots \vdash u_n$ is a word in $L(\mathcal{G}_1) \cap L(\mathcal{G}_2)$, then by the construction of $\mathcal{G}_1$ and $\mathcal{G}_2$, it is the reverse representation of the run of $\mathcal{M}$ on w. Thus, by definition, w is accepted by $\mathcal{M}$.

Note that the CFG $\mathcal{G}_1$ and $\mathcal{G}_2$ are defined over the terminals $0, 1, \sqcup, \vdash, [,]$, but each of them can encoded properly as 0-1 string. ∎

Next, we consider the following problem.

CFL-Universality	
Input:	A CFG $\mathcal{G} = \langle \Sigma, V, R, S \rangle$, where $\Sigma = \{0, 1\}$.
Task:	Output `True`, if $L(\mathcal{G}) = \Sigma^*$. Otherwise, output `False`.

Similar to CFL-Intersection, the problem CFL-Universality can be viewed as language.

Theorem 8.9 *The problem* CFL-Universality *is undecidable.*

Proof. The proof is quite similar to Theorem 8.8. We will describe an algorithm that on input $\lfloor \mathcal{M} \rfloor \$ w$, it construct a CFG $\mathcal{G}$ such that it generates all strings that are *not*(!) the run of $\mathcal{M}$ on w.

Note that a word:

$$u_0 \ \vdash \ u_1 \ \vdash \ u_2 \ \vdash \ u_3 \ \cdots \ \vdash \ u_n \tag{5}$$

is not the reverse representation of the run $\mathcal{M}$ on w, if at least one of the following holds.

(C1) The symbol $\vdash$ appears even number of times.

(C2) u_0 is not the initial configuration.

(C3) For some $0 \leqslant i \leqslant n$, the string u_i is not a string that represents a configuration, i.e., it does not contain a state or the states appear at least twice or the brackets [and] do not appear "properly" or inside the bracket [and] is not a state of $\mathcal{M}$.

(C4) For some $0 \leqslant i \leqslant n - 1$, the string $u_i \vdash u_i$ is not according to the transitions of $\mathcal{M}$.

(C5) For some $o \leqslant i \leqslant n - 1$, the string u_i is not the reverse of the string u_{i+1} (disregarding the state symbol and the symbols next to the state in both u_i and u_{i+1}).

(C6) The last string u_n does not contain q_{acc}.

For each of the conditions (C1)–(C6), we can construct a CFG $\mathcal{G}$ that generate all the strings that satisfy the condition. (It is useful to recall that CFL are closed union.) Here, as before, we fix an encoding of the terminals $0, 1, \sqcup, \vdash, [,]$ by 0-1 string.

Now, to complete our proof, we present the reduction for establishing HALT $\leqslant_T$ CFL-Universality. The following algorithm assumes that there is an algorithm for checking whether $L(\mathcal{G}) = \Sigma^*$. On input $\lfloor \mathcal{M} \rfloor \$ w$, do the following.

- Construct the CFG $\mathcal{G}$ that generates words of the form (4) such that at least one of (C1)–(C6) holds.
- If $L(\mathcal{G}) = \Sigma^*$, then REJECT.
  If $L(\mathcal{G}) \neq \Sigma^*$, then ACCEPT.

We will show that this algorithm decides HALT. Note that if $\mathcal{M}$ accepts w, then by definition, the run of $\mathcal{M}$ on w is finite. By the construction of $\mathcal{G}$, the reverse representation of this run is not in $L(\mathcal{G})$. Thus, $L(\mathcal{G}) \neq \Sigma^*$. Conversely, $L(\mathcal{G})$ is the set of all the strings that are not the reverse representation of the run of $\mathcal{M}$ on w. So, if $L(\mathcal{G}) \neq \Sigma^*$, then there is a string that is not in $L(\mathcal{G})$. This means that there is a string that is the reverse representation of the run of $\mathcal{M}$ on w, which means that $\mathcal{M}$ accepts w. This completes the proof. ∎

Corollary 8.10 *The problem* CFL-Subset *defined below is undecidable.*

CFL-Subset	
Input:	*Two CFG's* $\mathcal{G}_1$ *and* $\mathcal{G}_2$.
Task:	*Output* `True`, *if* $L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2)$. *Otherwise, output* `False`.

Lesson 9: Non-deterministic Turing machines

Theme: Non-deterministic Turing machines.

A non-deterministic Turing machine (NTM) $\mathcal{M} = \langle \Sigma, \Gamma, Q, q_0, q_{\text{acc}}, q_{\text{rej}}, \delta \rangle$ is defined as the standard Turing machine, with the exception that δ is now a relation where there maybe one or two transitions applicable on every pair $(p, a) \in Q \times \Gamma$.

More precisely, for every pair $(p, a) \in Q \times \Gamma$, the following holds. Either there is exactly one (q, b, α) such that:

$$(p, a) \to (q, b, \alpha) \ \in \ \delta$$

or there are exactly two (q_1, b_1, α_1) and (q_2, b_2, α_2) such that:

$$(p, a) \to (q_1, b_1, \alpha_1) \ \in \ \delta \qquad \text{and} \qquad (p, a) \to (q_2, b_2, \alpha_2) \ \in \ \delta$$

The standard Turing machine is a special case of non-deterministic Turing machine where for every $(p, a) \in Q \times \Gamma$, there is exactly one (q, b, α) such that $(p, a) \to (q, b\alpha) \in \delta$. For this reason, the standard Turing machine is also called deterministic Turing machine. To avoid potential confusion, from now we will always specify whether a Turing machine is a deterministic or non-deterministic. We use the abbreviation DTM for deterministic Turing machine.

As before, the initial configuration of $\mathcal{M}$ on input word w is $\triangleleft q_0 w$. For two configurations C, C', the notion of "one step computation" $C \vdash C'$ is defined similarly as in the standard Turing machine. A *run* of $\mathcal{M}$ on input w is a sequence:

$$C_0 \vdash C_1 \vdash \cdots$$

where C_0 is the initial configuration on w. A run is accepting/rejecting, if it ends up in an accepting/rejecting configuration, respectively. However, due to non-determinism, for each C there are two configurations C_1 and C_2 such that $C \vdash C_1$ and $C \vdash C_2$. Thus, for every input word w, there are <u>many</u> runs of $\mathcal{M}$ on w. Some may be accepting, some may be rejecting, and some other may not halt.

The following is the definition of acceptance/rejection and recognizable/decidable languages in terms of non-deterministic Turing machines.

- An NTM $\mathcal{M}$ *accepts* w, if <u>there is</u> an accepting run of $\mathcal{M}$ on w.
- An NTM $\mathcal{M}$ *rejects* w, if <u>all</u> runs of $\mathcal{M}$ on w are rejecting.
- An NTM $\mathcal{M}$ *does not halt on* w, if $\mathcal{M}$ <u>neither accept nor reject</u> w, i.e., $\mathcal{M}$ does not accept w and it also does not reject w.

  Equivalently, we can say that $\mathcal{M}$ does not halt on w, if there is <u>no</u> accepting run of $\mathcal{M}$ on w and there is an infinite run of $\mathcal{M}$ on w.
- A language L is decided by an NTM $\mathcal{M}$, if:
  - for every $w \in L$, $\mathcal{M}$ accepts w;
  - for every $w \notin L$, $\mathcal{M}$ rejects w.
- A language L is recognized by an NTM $\mathcal{M}$, if:
  - for every $w \in L$, $\mathcal{M}$ accepts w;
  - for every $w \notin L$, $\mathcal{M}$ does not accept w.

Theorem 9.1 *For every language NTM $\mathcal{M}$, there is DTM $\mathcal{M}'$ such that for every input word w, the following holds.*

- *If $\mathcal{M}$ accepts w, then $\mathcal{M}'$ accepts w.*
- *If $\mathcal{M}$ rejects w, then $\mathcal{M}'$ rejects w.*
- *If $\mathcal{M}$ does not halt on w, then $\mathcal{M}'$ does not halt on w.*

In other words, $\mathcal{M}$ and $\mathcal{M}'$ are equivalent.

Proof. Let $\mathcal{M}$ be an NTM. The DTM $\mathcal{M}'$ works by simulating $\mathcal{M}$ on the input word as follows.
On input word w, do the following.

- Let C_0 be the initial configuration of $\mathcal{M}$ on w.
- Let $S = \{C_0\}$, i.e., a set that contains only one element C_0.
- while ($S \neq \emptyset$) or (S does not contain an accepting configuration):
  - Delete all the rejecting configurations from S.
  - Compute the next configurations of each element in S and store them all in S.
    That is, construct the following set and store it back in S:
    $$\{\ C \mid \text{there is } C' \in S \text{ such that } C' \vdash C\}$$
- If S contains an accepting configuration, ACCEPT.
  If $S = \emptyset$, REJECT.

For each input word w, the DTM $\mathcal{M}'$ behaves as follows.

- If $\mathcal{M}$ accepts w.
  By definition, there is an accepting run of $\mathcal{M}$ on w. This means that at one point the set S will contain an accepting configuration, so $\mathcal{M}'$ will come out of the `while`-loop and accepts w.
- If $\mathcal{M}$ rejects w.
  By definition, all the runs of $\mathcal{M}$ on w are rejecting. Since rejecting configurations are deleted from the set S, this means that at one point the set S will become $\emptyset$, so $\mathcal{M}'$ will come out of the `while`-loop and rejects w.
- If $\mathcal{M}$ does not halt on w.
  This means that there is no accepting run and there is an infinite run. So, the set S will never contain an accepting configuration.
  Moreover, since there is an infinite run, the set S will never be empty. Thus, the `while`-loop in $\mathcal{M}'$ will also loop forever, which means $\mathcal{M}'$ will not halt too on w.

This completes the proof of Theorem 9.1. ∎

Theorem 9.2 *Recognizable languages are closed under concatenation and Kleene star.*

Proof. Let L_1 and L_2 be recognizable languages and let $\mathcal{M}_1$ and $\mathcal{M}_2$ be DTM that recognize them. We assume that $\Sigma = \{0, 1\}$.

(Closure under concatenation) We present an NTM $\mathcal{M}$ that recognizes the language L_1L_2. It has two "new" symbols $\tilde{O}$ and $\tilde{1}$. On input word w, do the following.

- Scan the input word from left to right.
- On each symbol $a \in \{0,1\}$, the TM non-deterministically chooses whether to move right or write $\tilde{a}$.

  That is, it moves from left to right and "guesses" a position in the input word. It marks the guessed position with $\tilde{a}$.

  Intuitively, we can view this as $\mathcal{M}$ "guesses" a partition $w = v_1 v_2$.
- Let v_1 and v_2 be the portion of the word to the left and to the right of the marker, where the first symbol of v_2 is $\tilde{a}$.
- Run $\mathcal{M}_1$ on v_1. If it accepts, then run $\mathcal{M}_2$ on v_2 (after the symbol $\tilde{a}$ is changed back to a).
- If $\mathcal{M}_2$ accepts v_2, ACCEPT.

Obviously, if the input word w indeed belongs to $L_1 L_2$, then there is a partition $w = v_1 v_2$ such that $\mathcal{M}_1$ accepts v_1 and $\mathcal{M}_2$ accepts v_2. Hence, $w \in L_1 L_2$ if and only if $\mathcal{M}$ accepts w.

(Closure under Kleene star) The proof is similar to the concatenation case. We present an NTM $\mathcal{M}$ that recognizes the language L_1^* as follows. On input word w, it guesses a partition $w = v_1 \cdots v_k$, for some $k \geqslant 1$ and run $\mathcal{M}_1$ on each v_i. It accepts w if and only if $\mathcal{M}_1$ accepts each v_i. ∎

Appendix

A An informal definition of non-deterministic algorithm

One can view a "non-deterministic" algorithm as an algorithm as defined in the appendix in Lesson 7, with an additional special variable z and an instruction of the following form:

$$z \ := \ 0 \parallel 1; \tag{1}$$

This instruction means "randomly assign variable z with either 0 or 1."

A non-deterministic algorithm A "accepts" an input word w, if on every instruction of the form (1), variable z can be assigned with 0 or 1 such that A will "return true." Note that the instruction (1) can be encountered more than once during the execution of algorithm A. For example, it may appear inside a `while`-loop.

As an example, we consider the problemSAT defined as follows.

SAT	
Input:	A propositional formula φ.
Task:	Output `True`, if φ has a satisfying assignment. Otherwise, output `False`.

The following is an example of a non-deterministic algorithm that decides SAT. On input formula φ, do the following.

- Let $x_1, \ldots, x_n$ be the variables in φ.
- For each $i = 1, \ldots, n$ do:
  - $z \ := \ 0 \parallel 1;$
  - If $z == 1$, then assign x_i with `True`.

  - If $z == 0$, then assign x_i with `False`.
- Check if the formula φ evaluates to true under the assignment.
- If the formula evaluates to `True`, then ACCEPT.

  If the formula evaluates to `False`, then REJECT.

We argue that the algorithm above decides SAT.

- If φ has a satisfying assignment, then there is a possibility that the algorithm ACCEPTS.

  Formally, this means that *there is* an accepting run, hence, by definition, φ is accepted by the algorithm.

- If φ does not have any satisfying assignment, then the algorithm always REJECTS.

  Formally, this means that *all* runs are rejecting runs, hence, by definition, φ is rejected by the algorithm.

Note that in the algorithm above, the assignment to the variables is done by invoking $z := 0|1$, which can be viewed as "guessing" an assignment. If there is such an assignment, then there is a possibility that the guess is correct, and we can assume that the NTM always guesses "correctly," i.e., it always makes a guess that leads to the accept state. If there is no such assignment, then there is no possiblity that the guess is correct, i.e., all guesses lead to the reject state.

Note also that the non-deterministic algorithm above is different from the standard *deterministic* algorithm for SAT which works as follows. On input formula φ, do the following.

- Let $x_1, \ldots, x_n$ be the variables in φ.
- Enumerate all possible assignment to the variables $x_1, \ldots, x_n$.
- If there is an assignment under which the formula φ evaluates to true, then ACCEPT.

  If there is no such assignment, then REJECT.

Lesson 10: Basic complexity classes

Theme: Classification of languages/problems according to number of steps (time) and cells (space) needed by Turing machines to decide them.

In the following let $\mathbb{N}$ denote the set of natural numbers $\{0, 1, 2, \ldots\}$. Recall the big 'Oh' notation $f(n) = O(g(n))$ which means that there is $c, n_0 \in \mathbb{N}$ such that for every $n \geqslant n_0$,

$$f(n) \quad \leqslant \quad c \cdot g(n).$$

Recall also that for a word $w \in \Sigma^*$, $|w|$ denotes the length of w.

1 Polynomial time complexity

We say that a DTM/NTM $\mathcal{M}$ runs in time $O(n^k)$, if there is $c, n_0 \in \mathbb{N}$ such that for every word $w \in \Sigma^*$ with $|w| \geqslant n_0$, every run of $\mathcal{M}$ on w has length $\leqslant c|w|^k$. That is, for every run of $\mathcal{M}$ on w with $|w| \geqslant n_0$:

$$C_0 \ \vdash \ C_1 \ \vdash \ \cdots \ \vdash \ C_N \qquad\qquad \text{where } C_N \text{ is an accepting/rejecting configuration,}$$

we have $N \leqslant c|w|^k$.

The notion that $\mathcal{M}$ runs in time $O(n^k)$ is the same for DTM and NTM. The only difference is that a DTM only has one run for each input word w, whereas NTM can have many runs for each input word w. In both cases, we can only say that $\mathcal{M}$ runs in time $O(n^k)$, if for each input word w, the length of every run of $\mathcal{M}$ on w is $\leqslant c|w|^k$.

Definition 10.1 A *DTM/NTM $\mathcal{M}$ decides/accepts a language L in time $O(n^k)$*, if $\mathcal{M}$ decides L and it runs in time $O(n^k)$.

Here it is useful to recall that a DTM/NTM $\mathcal{M}$ decides a language L, if for every word $w \in \Sigma^*$, $\mathcal{M}$ accepts w if and only if $w \in L$.

We define the class $\textsc{Dtime}[n^k]$ and the class $\mathbf{P}$ as follows.

$$\begin{aligned} \textsc{Dtime}[n^k] &:= \{L \mid \text{there is a DTM } \mathcal{M} \text{ that decides } L \text{ in time } O(n^k)\} \\ \mathbf{P} &:= \bigcup_{k \geqslant 1} \textsc{Dtime}[n^k] \end{aligned}$$

Note that the class $\mathbf{P}$ is closed under complement, union and intersection.

The non-deterministic counterpart is defined as follows.

$$\begin{aligned} \textsc{Ntime}[n^k] &:= \{L \mid \text{there is an NTM } \mathcal{M} \text{ that decides } L \text{ in time } O(n^k)\} \\ \mathbf{NP} &:= \bigcup_{k \geqslant 1} \textsc{Ntime}[n^k] \end{aligned}$$

The following class is also an important class in complexity theory.

$$\mathbf{coNP} \ := \ \{L \mid \Sigma^* - L \in \mathbf{NP}\}$$

2 Space complexity

We say that a DTM/NTM $\mathcal{M}$ uses space $O(n^k)$ if the following holds.

- $\mathcal{M}$ halts on every word $w \in \Sigma^*$, i.e., it either accepts or rejects w.
- There is $c, n_0 \in \mathbb{N}$ such that for every word $w \in \Sigma^*$ with length $|w| \geqslant n_0$, *each configuration* in every run of $\mathcal{M}$ on w has length $\leqslant c|w|^k$.

In other words, for every run of $\mathcal{M}$ on w:

$$C_0 \vdash C_1 \vdash \cdots \vdash C_N \qquad \text{where } C_N \text{ is an accepting/rejecting configuration,}$$

the length $|C_i| \leqslant c|w|^k$, for each $i = 0, \ldots, N$.

Again, note that the notion of $\mathcal{M}$ uses space $O(n^k)$ is the same for DTM and NTM. The only difference is that a DTM has only one run for each input word w, whereas NTM can have many runs for each input word w. In both cases, we can only say that $\mathcal{M}$ uses space $O(n^k)$, if for each input word w, for every run of $\mathcal{M}$ on w, the length of each configuration in the run is always $\leqslant c|w|^k$.

Definition 10.2 We say that *a DTM/NTM $\mathcal{M}$ decides/accepts a language L in space $O(n^k)$*, if $\mathcal{M}$ decides L and it uses space $O(n^k)$.

We define the class DSPACE$[n^k]$ and **PSPACE** as follows.

$$\begin{array}{rcl} \textsc{Dspace}[n^k] & := & \{L \mid \text{there is a DTM } \mathcal{M} \text{ that decides } L \text{ in space } O(n^k)\} \\ \textbf{Pspace} & := & \bigcup_{k\geqslant 1} \textsc{Dspace}[n^k] \end{array}$$

Note that the class **PSPACE** is closed under complement, union and intersection.

The non-deterministic counterpart is defined as follows.

$$\begin{array}{rcl} \textsc{Nspace}[n^k] & := & \{L \mid \text{there is an NTM } \mathcal{M} \text{ that decides } L \text{ in space } O(n^k)\} \\ \textbf{NPspace} & := & \bigcup_{k\geqslant 1} \textsc{Nspace}[n^k] \\ \textbf{coNPspace} & := & \{L \mid \Sigma^* - L \in \textbf{NPspace}\} \end{array}$$

3 Logarithmic space complexity

Another interesting classes are **LOG** and **NLOG**. We say that a language L is in **LOG**, if there is a 2-tape DTM $\mathcal{M}$ that decides L and a constant c such that for every input word w:

- The first tape always contains only the input word w, i.e., $\mathcal{M}$ never changes the content of the first tape.
- $\mathcal{M}$ uses space $\leqslant c \cdot \log(|w|)$ in its second tape.

Likewise, we say that a language L is in **NLOG**, if there is a 2-tape NTM $\mathcal{M}$ that decides L such that the above two conditions are satisfied.

4 Some classic complexity results

Obviously, we have $\mathbf{Log} \subseteq \mathbf{NLog}$, $\mathbf{P} \subseteq \mathbf{NP}$, and $\mathbf{Pspace} \subseteq \mathbf{NPspace}$.

Proposition 10.3

- $\mathbf{Log} \subseteq \mathbf{P}$.
- $\mathbf{NP} \subseteq \mathbf{Pspace}$.

Deterministic/non-deterministic time/space hierarchy theorem states that for every $k \geqslant 1$, the following holds.

$$\begin{array}{ll} \mathrm{Dtime}[n^k] \subsetneq \mathrm{Dtime}[n^{k+1}] & \mathrm{Dspace}[n^k] \subsetneq \mathrm{Dspace}[n^{k+1}] \\ \mathrm{Ntime}[n^k] \subsetneq \mathrm{Ntime}[n^{k+1}] & \mathrm{Nspace}[n^k] \subsetneq \mathrm{Nspace}[n^{k+1}] \end{array}$$

Some classic results in complexity theory are: (We will not prove them in the class.)

- $\mathbf{NLog} \subseteq \mathbf{P}$.
- If $L \in \mathrm{Nspace}[n^k]$, then $\Sigma^* - L \in \mathrm{Nspace}[n^k]$.
- $\mathrm{Nspace}[n^k] \subseteq \mathrm{Dspace}[n^{2k}]$.

The second bullet implies that $\mathrm{coNspace}[n^k] = \mathrm{Nspace}[n^k]$, and hence, $\mathbf{NPspace} = \mathbf{coNPspace}$. The third bullet implies that $\mathbf{NPspace} = \mathbf{Pspace}$.

Combining all these inclusions together, we obtain:

$$\mathbf{Log} \subseteq \mathbf{NLog} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{Pspace}$$

From the deterministic/non-deterministic space hierarchy, it is also known that $\mathbf{Log} \subsetneq \mathbf{Pspace}$ and $\mathbf{NLog} \subsetneq \mathbf{Pspace}$. So, we know that at least one of the inclusions must be strict, but we don't know which one.

Lesson 11: NP-complete languages

Theme: Polynomial time reductions and NP-complete languages/problems.

1 Polynomial time reduction

Recall the definition of reduction in Lesson 11: $L_1 \leqslant_m L_2$, if there is a computable function F such that for every $w \in \Sigma^*$:

$$w \in L_1 \quad \text{if and only if} \quad F(w) \in L_2$$

We say that a TM $\mathcal{M}$ computes F in time $O(g(n))$, if there is a constant $c > 0$ such that on every word w, $\mathcal{M}$ computes $F(w)$ in time $\leqslant cg(|w|)$. If $g(n) = n^k$ for some $k > 0$, such a function F is called *polynomial time computable* function.

Definition 11.1 A language L_1 is *polynomial time reducible* to another language L_2, denoted by $L_1 \leqslant_p L_2$, if there is a polynomial time computable function F such that for every $w \in \Sigma^*$:

$$w \in L_1 \quad \text{if and only if} \quad F(w) \in L_2$$

Such a function F is called polynomial time reduction, also known as *Karp reduction.*

2 NP-complete languages

Definition 11.2 Let L be a language.

- L is $\mathbf{NP}$*-hard*, if for every $L' \in \mathbf{NP}$, $L' \leqslant_p L$.
- L is $\mathbf{NP}$*-complete*, if $L \in \mathbf{NP}$ and L is $\mathbf{NP}$-hard.

Recall that a propositional formula (Boolean expression) with variables $x_1, \ldots, x_n$ is in Conjunctive Normal Form (CNF), if it is of the form: $\bigwedge_i \bigvee_j \ell_{i,j}$ where each $\ell_{i,j}$ is a literal, i.e., a variable x_k or its negation $\neg x_k$. It is in 3-CNF, if it is of the form $\bigwedge_i \big(\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}\big)$. A formula φ is satisfiable, if there is an assignment of Boolean values `True` or `False` to each variables in φ that evaluates to `True`.

SAT	
Input:	A propositional formula φ in CNF
Task:	Output `True`, if φ is satisfiable. Otherwise, output `False`.

Theorem 11.3 *SAT is* $\mathbf{NP}$*-complete.*

3-SAT	
Input:	A propositional formula φ in 3-CNF
Task:	Output `True`, if φ is satisfiable. Otherwise, output `False`.

Theorem 11.4 *3-SAT is* $\mathbf{NP}$*-complete.*

3 More NP-complete problems

We need a few terminologies. Let $G = (V, E)$ be a (undirected) graph.

- G is 3-colorable, if we can color the vertices in G with 3 colors (every vertex must be colored with one color) such that no two adjacent vertices have the same color.
- A set $C \subseteq V$ is a clique in G, if every pair of vertices in C are adjacent.
- A set $W \subseteq V$ is a vertex cover, if every edge in E is adjacent to at least one vertex in W.
- A set $I \subseteq V$ is independent, if every pair of vertices in I are non-adjacent.
- A set $D \subseteq V$ is dominating, if every vertex in V is adjacent to at least one vertex in D.

All the following problems are **NP**-complete.

3-color	
Input:	A (undirected) graph $G = (V, E)$.
Task:	Output `True`, if G is 3-colorable. Otherwise, output `False`.

Clique	
Input:	A (undirected) graph $G = (V, E)$ and an integer $k \geqslant 0$ in binary form.
Task:	Output `True`, if G has a clique of size $\geqslant k$. Otherwise, output `False`.

Independent-Set	
Input:	A (undirected) graph $G = (V, E)$ and an integer $k \geqslant 0$ in binary form.
Task:	Output `True`, if G has an independent set of size $\geqslant k$. Otherwise, output `False`.

Vertex-Cover	
Input:	A (undirected) graph $G = (V, E)$ and an integer $k \geqslant 0$ in binary form.
Task:	Output `True`, if G has a vertex cover of size $\leqslant k$. Otherwise, output `False`.

Dominating-Set	
Input:	A (undirected) graph $G = (V, E)$ and an integer $k \geqslant 0$ in binary form.
Task:	Output `True`, if G has an dominating set of size $\leqslant k$. Otherwise, output `False`.