Notes for ComS 331
Theory of Computing

These notes will be updated as needed throught the course.

Gianfranco Ciardo

September 10, 2017

Logistics

Typeset your homeworks using LaTeX.

Make sure the pdf file generated form the LaTeX file has your name and the homework number prominently displayed at the top.

Write in terse, correct, and complete sentences: you are writing “technical English”!

Read your assignments carefully. . . twice.

You do not get any partial credit for solving the wrong problem correctly.

The safest (and often the fastest) way to contact the instructor is to use e-mail: “ciardo@iastate.edu”.

The class is generally considered both challenging and rewarding.

Chapter 1

Background

We assume the notion of *set*. At times we can define a set by listing its elements, $S = \{a, b, c\}$. Other times, we need to use a "rule", $\mathbb{N} = \{0, 1, 2, \ldots\}$. Often, a set is defined by giving a property which determines whether a particular element is in the set or not: $L = \{\underbrace{a \cdots a}_{n \text{ times}}\underbrace{b \cdots b}_{n \text{ times}} : n \in \mathbb{N}\}$. Sometimes "|" is used instead of ":"; either symbol is read "such that" in this context.

The *Cartesian product* or *cross-product* ($S \times T$) of two sets S and T is the set $S \times T = \{(a, b) : a \in S, b \in T\}$. The order of the elements in the pair (a, b) matters.

Given a set V, an (*undirected*) graph over V is the pair (V, E), where $E \subseteq \{\ \{a, b\} : a, b, \in V\}$. We call V the *vertices* or *nodes* and E the *edges* or *arcs*. If $\{a, a\} = \{a\} \in E$, we call it a *(self)-loop* on a.

A *path* from a to b is a sequence of edges $(e_1, \ldots, e_m)$ such that, for $i = 1, \ldots, m$, $e_i = \{a_{i-1}, a_i\}$, with $a = a_0$ and $b = a_m$. Alternatively, we can define a path from a to b by focusing on its nodes: a path is a sequence of nodes $(a_0, \ldots, a_m)$ such that, for $i = 1, \ldots, m$, $\{a_{i-1}, a_i\} \in E$, with $a = a_0$ and $b = a_m$. Either way, the *length* of the path is m, the number of edges (not nodes!) in it; in particular, (a) is a *(trivial)* path of length 0. A *cycle* on a is a non-trivial path from a to a; thus, the shortest possible cycle is (a, a), provided $\{a\} \in E$, of length 1. A graph is *connected* if, for every pair of vertices $a, b \in V$, there is a path between them. A graph is a *clique* if, for every pair of vertices $a, b \in V$, there is an edge between them.

A *directed graph* is a pair (V, E) where, now, $E \subseteq V \times V$. An edge (a, a) is a *self-loop* on a. The definitions of path, length of a path, and cycle are analogous to the undirected ones.

A directed graph is *weakly connected* if the underlying graph where we ignore arc directions is connected. It is *strongly connected* if, for each pair of nodes a and b, there is a (directed) path from a to b. A *directed acyclic graph (DAG)* is a directed graph containing no cycles.

A *(rooted) tree* is a special type of directed graph. Several equivalent definitions exist for it:

1. A directed graph with one special node, the *root* r, such that there is exactly one path from r to each node.

2. A directed graph is a tree rooted at r if it is either (a) a single node r or (b) a graph (V, E) where $V = \{r\} \cup V_1 \cup \cdots \cup V_n$, all disjoint sets, and $E = \{(r, r_1,), \ldots, (r, r_n)\} \cup E_1 \cup \cdots \cup E_n$ and, for $i = 1, \ldots, n$, $r_i \in V_i$, $E_i \subseteq V_i \times V_i$, and (V_i, E_i) is a tree rooted at r_i.

An *unrooted tree* is then an undirected tree without a root, or, more precisely, an acyclic connected undirected graph.

A (binary) *relation* over S and T is a set $R \subseteq (S \times T)$. The sets S and T might coincide.

A relation is *reflexive* if $\forall a \in S, (a, a) \in R$.

A relation is *symmetric* if $\forall a, b \in S, (a, b) \in R \Rightarrow (b, a) \in R$.

A relation is *antisymmetric* if $\forall a, b \in S, (a, b) \in R \wedge (b, a) \in R \Rightarrow a = b$.
Or, in other words, $a \neq b \wedge (a, b) \in R \Rightarrow (b, a) \notin R$.

A relation is *transitive* if $\forall a, b, c \in S, (a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$.

A relation is an *equivalence* relation iff it is reflexive, symmetric, and transitive. Example: $x \equiv y$ iff $x \bmod 3 = y \bmod 3$. Example: $x \equiv y$ iff x and y can reach each other in a directed graph.

A relation is a (partial) *order* relation iff it is reflexive, antisymmetric, and transitive. Example: vectors on $\mathbb{R}^2$ and the $\geq$ comparison.

An order relation is a *total order* relation iff $\forall a, b \in S, (a, b) \in R \vee (b, a) \in R$. Example: $\mathbb{Z}$ and the $\geq$ comparison.

A *function* $f \subseteq (S \times T)$ is a relation satisfying $\forall a \in S, \forall b, c \in T, (a, b) \in f \wedge (a, c) \in f \Rightarrow b = c$.

A function $f \subseteq (S \times T)$ is total iff $\forall a \in S, \exists b \in T, (a, b) \in f$. Unless stated otherwise, all the functions we discuss are assumed to be total.

We normally write a function as $f : S \to T$ rather than $f \subseteq (S \times T)$, and write $f(a) = b$ rather than $(a, b) \in f$. We call S the *domain* and T the *range* of f.

A function $f : S \to T$ is *one-to-one* or an *injection* iff $\forall a, b \in S, f(a) = f(b) \Rightarrow a = b$, that is, each element in the range has at most one corresponding element in the domain.

A function $f : S \to T$ is *onto* or a *surjection* iff $\forall b \in T, \exists a \in S, f(a) = b$, that is, each element in the range has at least one corresponding element in the domain.

A function $f : S \to T$ is a *bijection* iff it is one-to-one and onto, that is, each element in the domain has exactly one corresponding element in the range, and vice versa.

We say that two sets S and T are *equinumerous* if there exists a bijection $f : S \to T$ between them.

The *cardinality* of a set is the number of elements in it, if this number is finite. For example, the cardinality of $S = \{a, b, c\}$, indicated as $|S|$, is three. An infinite set is *countably infinite* if it is equinumerous to the natural numbers $\mathbb{N}$, *uncountably infinite* otherwise. A set is *discrete*, or *countable*, iff it is finite or countably infinite.

Great care must be taken when comparing the cardinalities of infinite sets.

1.1 Basic proof techniques

Several proof techniques occur frequently enough to deserve a particular mention.

1.1.1 Contradiction

To prove that A is true by contradiction, assume that $\neg A$ is true, and derive a statement B which is known, or can be proven, to be false, following a sequence of logical steps.

A common mistake is to derive A directly without using $\neg A$. Then, A is indeed proved, but not by contradiction.

Example: prove that the set of prime numbers is infinite. Let S be the set of prime numbers, and assume, by contradiction, that it is finite: $S = \{p_1, p_2, \dots p_n\}$. Now, consider the number $x = p_1 p_2 \cdots p_n + 1$. Then, we can ask: "Is x divisible by p_1?". No because the remainder of the division of x by p_1 is 1. "Is x divisible by p_2?". No because the remainder of the division of x by p_2 is 1. And so on. Thus, either x is itself prime, or it is divisible by other primes. Thus, S cannot contain all primes.

1.1.2 Induction

The *principle of (strong) mathematical induction* states that, if $A \subseteq \mathbb{N}$ is the set of natural numbers for which a certain property P holds, that is, $A = \{n : n \in \mathbb{N}, P(n)\}$, if $0 \in A$, and if $\{0, 1, \dots n\} \subseteq A$ implies that $n + 1 \in A$, then $A = \mathbb{N}$.

When giving a proof by induction, always write explicitly (1) the variable on which the induction is being performed, (2) the basis, (3) the induction hypothesis, and (4) the inductive step.

Example: prove that $\sum_{i=1}^{n} i = n(n+1)/2$. We prove it *by induction on* n.

Basis: The property holds for $n = 0$, since $\sum_{i=1}^{0} i = 0$ and $0(0+1)/2 = 0$.

Induction hypothesis: Assume that $\forall m \leq n, \sum_{i=1}^{m} i = m(m+1)/2$.

Inductive step: Then, the property holds for $n+1$ as well, since $\sum_{i=1}^{n+1} i = n + 1 + \sum_{i=1}^{n} i = n + 1 + n(n+1)/2 = (n^2 + 3n + 2)/2 = (n+1)((n+1)+1)/2$.

With weak induction, we only need to assume that $n \in A$ implies $n + 1 \in A$, but this is not really different from strong induction.

1.1.3 Pigeonhole principle

The pigeonhole principle states that, if S and T are nonempty finite sets and $|S| > |T|$, then there is no one-to-one function $f : S \to T$. It can be proved by induction on $|T|$.

Basis: If $|T| = 1$ and $|S| > 1$, $\exists s_1, s_2 \in S$, $s_1 \neq s_2, f(s_1) = f(s_2)$, hence f is not one-to-one.

Induction hypothesis: Assume that $\forall m \leq n, m \geq 1, \forall T, S, |T| = m, |S| > |T|, \forall f, f : S \to T$, f is not one-to-one.

Inductive step: Consider two sets T and S, satisfying $|T| = n + 1$ and $|S| > |T|$, and a function $f, f : S \to T$. Since T is nonempty, consider $t \in T$. There are three possibilities.
(1) If $\exists s_1, s_2 \in S, s_1 \neq s_2, f(s_1) = f(s_2) = t$, then f is not one-to-one.
(2) If $\exists! s \in S, f(s) = t$, define a function $g : S - \{s\} \to T - \{t\}$, satisfying $\forall r \in S - \{s\}, g(r) = f(r)$. Then g is a function from a set $S - \{s\}$, with more than n elements, to a set $T - \{t\}$, with n elements, hence, by induction hypothesis it is not one-to-one, that is, $\exists r_1, r_2 \in S - \{s\}, r_1 \neq r_2, g(r_1) = g(r_2)$.

But then, $g(r_1) = f(r_1) = f(r_2) = g(r_2)$, hence f is not one-to-one either.
(3) If $\nexists s \in S, f(s) = t$, define $g : S \to T \setminus \{t\}$. Also in this case the inductive hypothesis implies that g, hence f, is not one-to-one.

Exercise: using the pigeonhole principle, prove that, if a finite graph contains arbitrarily long paths, it must contain a cycle.

1.1.4 Diagonalization principle

The diagonalization principle states that, if $R \subseteq (A \times A)$ is a binary relation on a set A, which can be infinite, and we define the diagonal set for R, $D = \{a \in A : (a, a) \notin R\}$, and the row sets $R_a = \{b \in A : (a, b) \in R\}$, then D is different from any R_a. This can be proven by contradiction.

Example on a small set: $A = \{1, 2, 3, 4\}$, and

$R =$

	1	2	3	4
1	y	y	n	n
2	y	y	y	n
3	n	y	n	n
4	n	y	n	y

that is, $R_1 = \{1, 2\}$, $R_2 = \{1, 2, 3\}$, $R_3 = \{2\}$, $R_4 = \{2, 4\}$. Then, $D = \overline{\{1, 2, 4\}} = \{3\}$.

Example: prove that $2^{\mathbb{N}}$, the set of all sets of natural numbers, is not countable. By contradiction, assume that $2^{\mathbb{N}}$ is countably infinite. Then, we can define a bijection $f : \mathbb{N} \to 2^{\mathbb{N}}$. Consider then the set $D = \{n \in \mathbb{N} : n \notin f(n)\}$. Since D is a set of natural numbers, $D \in 2^{\mathbb{N}}$, hence $\exists! d \in \mathbb{N}, D = f(d)$. Then, we can ask whether d is in D. If $d \in D$, then, by definition of D, $d \notin f(d)$, but $D = f(d)$, hence $d \notin D$, a contradiction. If $d \notin D$, then, by definition of D, $d \in f(d)$, but $D = f(d)$, hence $d \in D$, a contradiction. Hence a bijection f cannot exists, and $2^{\mathbb{N}}$ must be uncountable.

$R =$

	$f(0)$	$f(1)$	$f(2)$	$f(3)$	$\cdots$
0	y	y	n	n	
1	y	y	n	n	
2	y	y	y	n	
3	n	y	n	n	
4	n	y	n	y	
$\vdots$					

where $R(i, f(j)) = y \Leftrightarrow i \in f(j)$.

This is a very important result.

Exercise: using the diagonalization principle, prove that the set $\mathbb{R}$ of real numbers is uncountable (hint: it is enough to show that $[0, 1]$ is uncountable).

1.2 Formal languages

An *alphabet* is a finite set of *symbols*, $\Sigma = \{a, b, c\}$.

A *string over* Σ is a finite sequence of symbols from Σ, $x = aabcac$.

The length of a string is the number of symbols in it, counted with multiplicity. The length of aab is three. We write $|aab| = 3$. We also write $|w|_\alpha$ to mean the number of occurrences of $\alpha \in \Sigma$ in the string w, thus $|aab|_a = 2$ and $|aab|_b = 1$.

Σ^n is the set of all strings of length n.

ϵ is the string of length zero, or *empty string*.

Then $\Sigma^0 = \{\epsilon\}$.

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$ is the set of all strings over Σ. The operator "*" is called *Kleene star*.

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \cdots = \Sigma^* \setminus \{\epsilon\}$ is the set of all nonempty strings over Σ.

NOTE the difference between "infinite" and "arbitrarily large". Σ^* contains strings whose length is arbitrarily large, but not infinite (indeed, strings are finite by definition).

We can *concatenate* two strings $x = aab$ and $y = acb$ to form a string $xy = aabacb$. If $x \in \Sigma^*$ and $y \in \Gamma^*$, $xy \in (\Sigma \cup \Gamma)^*$.

Then, $|xy| = |x| + |y|$.

The *reverse* x^R of a string x is the string read in reverse order. Formally, we can define it by induction: $\epsilon^R = \epsilon$ and $\forall a \in \Sigma, \forall w \in \Sigma^*, (aw)^R = w^R a$. Then $\forall a \in \Sigma, a^R = a$ and $\forall x \in \Sigma, (x^R)^R = x$.

A string $w \in \Sigma^*$ such that $w = w^R$ is called a *palindrome*. For example, ϵ, $aabaa$, and bb are palindromes. Remember: there are odd-length and even-length palindromes!

Given a string $x \in \Sigma^*$, assume that two strings $u, v \in \Sigma^*$ satisfy $x = uv$. Then, we say that u is a *prefix* and v is a *suffix* of x. In particular, ϵ and x are both a prefix and a suffix of x.

A *language* L over the alphabet Σ is a set of strings, that is, a subset of Σ^*: $L \subseteq \Sigma^*$.

The concatenation of two languages L_1 and L_2 is the language containing all the possible concatenations of one string from L_1 with one string from L_2: $L_1 L_2 = \{xy : x \in L_1 \wedge y \in L_2\}$.

We can then define $L^0 = \{\epsilon\}$, and $L^n = LL^{n-1}$.

The *star closure* of a language L is $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \{w_1 w_2 \cdots w_n : n \in \mathbb{N}, \forall i, 1 \leq i \leq n, w_i \in L\}$, while the *positive closure* is $L^+ = L^1 \cup L^2 \cup \cdots$. Since $L^0 = \{\epsilon\}$, $L^+ = L^* \Leftrightarrow \epsilon \in L$.

The *complement* of language is $\overline{L} = \Sigma^* \setminus L$, that is, the set of strings over Σ not in L.

1.2.1 Finite representation of languages

We are interested in languages that can be represented in a finite way, using some kind of representation we agree upon. This does not mean we restrict ourselves to finite languages. For example, $L = \{a, b, c\}^* \setminus \{a, b\}^*$ is an infinite language represented in a finite way, using the *metasymbols* "{", "}", ",", "*", and "\", plus the three symbols from the alphabet Σ, a, b, and c. The representation itself is a language over an alphabet containing Σ, plus a finite set S of "metasymbols".

Σ^* is countably infinite: we can enumerate its elements in increasing length, and lexicographically within strings of equal length: $\Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, \cdots\}$. Analogously, $(\Sigma{\cup}S)^*$ is countably infinite, so we can only describe a countably infinite number of languages in a finite way.

How many languages exists over $\Sigma = \{a, b, c\}$? A language is a subset of Σ^*, so the set of languages over $\Sigma = \{a, b, c\}$ is 2^{Σ^*}. Σ^* is countably infinite, but we already know that $2^{\mathbb{N}}$ is not countable, hence so is 2^{Σ^*}.

Hence, an uncountable number of languages over Σ cannot be described finitely.

1.3 Grammars

If a language is finite, we can define it by listing its elements; if it is infinite but "simple", we can define it by describing a boolean property that its elements satisfy. In general though, these mechanisms are not appropriate for more complex languages. One way to describe such languages might be a *grammar*.

A grammar G is defined as $G = (V, T, S, P)$, where

- V is an alphabet, called the *variables*, or *nonterminals*.
- T is an alphabet, $V \cap T = \emptyset$, called the *terminal symbols* or *terminals*.
- $S \in V$ is the *start* variable.
- $P \subseteq ((V \cup T)^* \setminus T^*) \times (V \cup T)^*$ is a set of *productions*. The key property is that there is at least one variable on the left-hand-side of a production.

Given a string of the form $w = uxv$ and a production of the form $x{\rightarrow}y$, we can *apply* the production to w and obtain a new string: $w \Longrightarrow uyv$.

We say that w *derives* uyv, or that uyv is derived from w, by applying the production $x{\rightarrow}y$.

As usual we indicate the star and positive closures with $\overset{*}{\Longrightarrow}$ and $\overset{+}{\Longrightarrow}$, respectively.

The *language generated by a grammar* $G = (V{,}T{,}S{,}P)$ is the set of strings $L(G) = \{w{\in}T^* : S \overset{*}{\Longrightarrow} w\}$.

If $w \in L(G)$, we say that $S \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow \cdots \Longrightarrow w_n \Longrightarrow w$ is a *derivation* of w. A string w might have multiple derivations. S, w_1, $\ldots$, w_n are called *sentential forms*.

Example: consider $G = (\{S\}, \{a, b\}, S, \{S{\rightarrow}aSb, S{\rightarrow}\epsilon\})$. Then, the derivations are of the form

$$S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow \cdots \Longrightarrow a^nSb^n \Longrightarrow a^nb^n$$

and $L(G) = \{a^nb^n : n \in \mathbb{N}\}$.

1.4 Automata

Grammars give us a way to talk about languages: $L(G)$ is the language generated by a grammar G. Determining what strings are generated by a grammar can be a difficult task. We can then hope to give a more algorithmic definition of how to recognize a language.

We do this by introducing the notion of *automaton*, which can be thought as a formalization of the concept of a digital computer. An automaton reads an *input* string from some alphabet Σ, left to right, without changing it. It might have a *temporary storage*, containing a finite or infinite number of cells, each able to store a symbol from an alphabet, and it has a *finite control*, that is, the automaton is, at any time, in exactly one of a finite number of *states*. At any *move*, the *transition function* determines the new state of the automaton, according to the current *configuration*, that is, the current state, the current input symbol, and the current contents of the temporary storage.

An automaton can be *deterministic* or *nondeterministic*, according to whether, in each configuration, exactly (or at most) one move is possible, or not.

An automaton has also some way to generate an *output*.

If the only output of an automaton M is either "yes" or "no", we say that M is an *accepter*, and we define the language accepted by M as $L(M) = \{w \in \Sigma^* : \ M$ outputs "yes" on input $w\}$.

Alternatively, M could generate strings from an alphabet Γ as output. Then, we say that M is a *translator*, and we say that M translates w into z if z is the output of M on input w: $M : \Sigma^* \rightarrow \Gamma^*$.

Chapter 2

Regular languages

The easiest class of languages we study is the *regular languages*.

2.1 Regular expressions

We define the language of *regular expressions over* Σ recursively as:

- $\emptyset$ is a regular expression.
- $\forall a \in \Sigma, a$ is a regular expression.
- If α and β are regular expressions, then $(\alpha\beta)$ is a regular expression.
- If α and β are regular expressions, then $(\alpha + \beta)$ is a regular expression.
- If α is a regular expression, then α^* is a regular expression.
- Nothing else is a regular expression.

Hence the language of regular expressions over Σ is a subset of $(\Sigma \cup \{(,), +, ^*, \emptyset\})^*$, where $\{(,), +, ^*, \emptyset\}$ is the set of metasymbols.

We can assign a meaning to each regular expression, by associating it to a language over Σ, according to the following rules:

- $L(\emptyset) = \emptyset$.
- $\forall a \in \Sigma, L(a) = \{a\}$.
- $L((\alpha\beta)) = L(\alpha)L(\beta)$.
- $L((\alpha + \beta)) = L(\alpha) \cup L(\beta)$.
- $L(\alpha^*) = L(\alpha)^*$.

When appropriate, we will drop sets of parenthesis, assuming, as usual, that star has precedence over concatenation, and concatenation has precedence over sum. For example, if $\Sigma = \{a, b, c\}$, $a + b(ac + cc)^*$, is a regular expression indicating the language $L(a+b(ac+cc)^*) = \{a, b, bac, bcc, bacac, baccc, \ldots\}$.

Different regular expressions can indicate the same language, we then say that they are *equivalent*: $a + aa + a^* \equiv a^*$, $(a^*b^*)^* \equiv (a + b)^*$.

We will use the shorthand notations:

$$\forall\alpha, \alpha^+ \triangleq \alpha\alpha^*$$

$$\epsilon \triangleq \emptyset^*$$

These *are not* new rules for regular expressions, they are just abbreviations.

Applications: syntax of an identifier, integer constant, or a real constant in a programming language.

2.2 Deterministic finite automata

A deterministic finite automaton (DFA) is a tuple $M = (Q, \Sigma, \delta, s_0, F)$ where

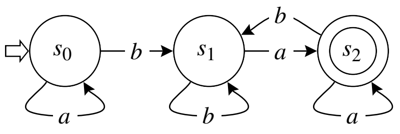
- Q is a finite set of states.
- Σ is an alphabet.
- $\delta : (Q \times \Sigma) \to Q$ is the *transition function*.
- $s_0 \in Q$ is the *initial state*.
- $F \subseteq Q$ is the set of *final states* (also called *accepting states*).

δ defines what the next state is going to be, given the current state and the current input character.

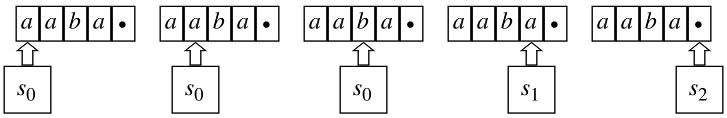
Example: Consider the following DFA. $M = (\{s_0, s_1, s_2\}, \{a, b\}, \delta, s_0, \{s_2\})$, where δ is given by the following table:

	a	b
s_0	s_0	s_1
s_1	s_2	s_1
s_2	s_2	s_1

It is easier to describe a DFA graphically. The states are the nodes of a graph. There is an arc from s_i to s_j labeled α iff $\delta(s_i, \alpha) = s_j$. The initial state is indicated with an incoming arrow. The final states are indicated with a double circle. Since δ is a total function, each node has exactly $|\Sigma|$ arcs leaving from it, each labeled with a different symbol from Σ.



The following picture depicts the execution of our DFA on the string *aaba*.



It is easier to describe a computation textually. First, we need to define the *configurations of a DFA*. A configuration $(q, aw) \in (Q \times \Sigma^*)$ indicates that the DFA is in state q, that the current input symbol is a, and that the remaining input symbols are w. A configuration (q, ϵ) indicates that the DFA is in state q and that the entire input string has been consumed.

Then, we say that (q, aw) *yields* (p, w) *in one step* iff $\delta(q, a) = p$, and write $(q, aw) \underset{M}{\vdash} (p, w)$. We can drop the subscript M when the automaton is clear from the context.

In our example, $(s_0, aaba) \underset{M}{\vdash} (s_0, aba) \underset{M}{\vdash} (s_0, ba) \underset{M}{\vdash} (s_1, a) \underset{M}{\vdash} (s_2, \epsilon)$.

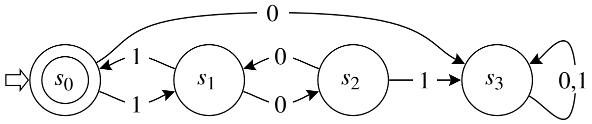
We can also write $(s_0, aaba) \overset{4}{\underset{M}{\vdash}} (s_2, \epsilon)$ and, as usual, we denote the reflexive and transitive closure of $\vdash$ with $\overset{*}{\vdash}$. Hence, we can write $(s_0, aaba) \overset{*}{\underset{M}{\vdash}} (s_2, \epsilon)$.

We can also extend δ, and allow it to have strings, not just symbols, as second parameter, using the recursive definition: $\delta(s, \epsilon) = s$ and $\delta(s, aw) = \delta(\delta(s, a), w)$.

Finally, we can define the set of strings *accepted* by a DFA $M = (Q, \Sigma, \delta, s_0, F)$ as $L(M) = \{w \in \Sigma^* : \delta(s_0, w) \in F\}$ or, equivalently, $L(M) = \{w \in \Sigma^* : (s_0, w) \overset{*}{\underset{M}{\vdash}} (q, \epsilon) \wedge q \in F\}$.

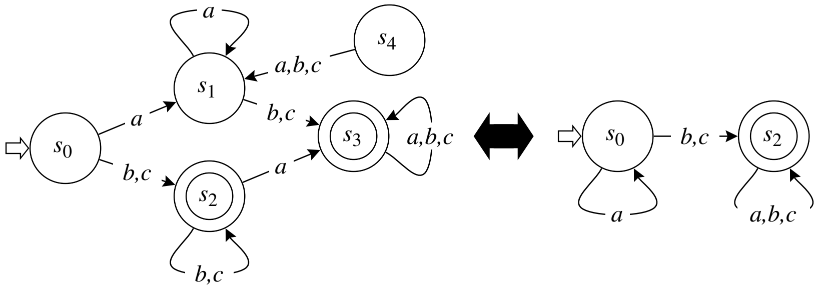
Question: What is the language accepted by our DFA? Answer: $a^*b^+a^+(b^+a^+)^* = a^*(b^+a^+)^+ = (a+b)^*ba^+$ The second expression is preferable, it is simpler and more easily understood: the set of string having at least one b and ending in a.

Example: The following DFA accepts $L = (1(00)^*1)^* = \epsilon + 1(00+11)^*1$.



Note how the *trap* state s_3 is not final, and, once entered, it cannot be exited. This is useful when no possible completion exist for the string which has been partially read. A single trap state is sufficient in a DFA. For example, if we have already read $x = 10001$, the string $w = xy$ cannot be in L, no matter what y is.

Example: The following two DFA are *equivalent*, they accept the same language $L = a^*(b+c)(a+b+c)^*$, which we can also write as $(a+b+c)^*(b+c)(a+b+c)^*$ or $(a+b+c)^*(b+c)a^*$.



Observing the DFA on the left, note that

- State s_4 cannot be reached by the initial state, so it can be removed.
- Once state s_2 is reached, only final states can be reached, so $\delta(s_2, a)$ could have been defined as s_2 instead of s_3, and s_2 and s_3 could have been merged.
- State s_1 can be merged with s_0.

2.2.1 A language that cannot be generated by a DFA

$L = \{a^n b^n : n \in \mathbb{N}\}$ cannot be accepted by a DFA.

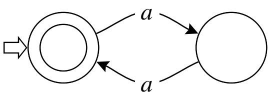
Proof by contradiction: assume that $M = (Q, \{a, b\}, \delta, s_0, F)$ exists such that $L = L(M)$.

Since Q is finite, it must be true that there exist $i, j \in \mathbb{N}$, $i \neq j$, $1 \leq i, j \leq |Q| + 1$, such that $\delta(s_0, a^i) = \delta(s_0, a^j)$. This follows from the pigeonhole principle: the function $f(n) \equiv \delta(s_0, a^n)$ defined over the integers $1, \ldots |Q|+1$ is a function from a set with $|Q|+1$ elements to a set with $|Q|$ elements, so it cannot be one-to-one.

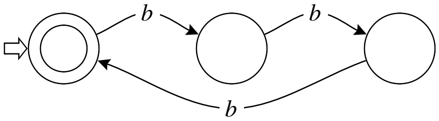
But, $\delta(s_0, a^i) = \delta(s_0, a^j) = s$ means that, after reading a^i or a^j starting from state s_0, the DFA reaches the same state s. Then, after reading b^i starting from state s, the DFA is in state $\delta(s, b^i) = t$.

We then have two cases. If $t \in F$, M accepts $a^i b^i$ (right), but also $a^j b^i$ (wrong). If $t \notin F$, M rejects $a^j b^i$ (right), but also $a^i b^i$ (wrong). Either way, $L(M) \neq L$, hence no DFA can accept L.

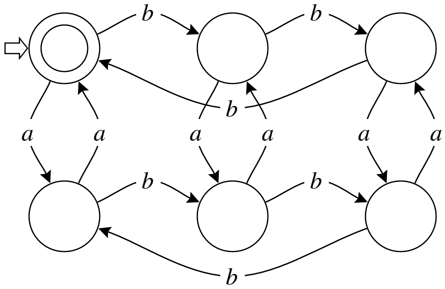
At this point, it useful to examine a few regular languages and DFA that can generate them:



$L_1 = \{a^{2n} : n \in \mathbb{N}\}$:



$L_2 = \{b^{3m} : m \in \mathbb{N}\}$:



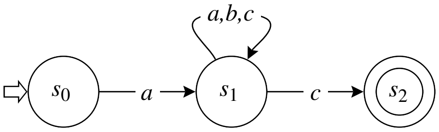
$L_3 = \{w : |w|_a = 2n, |w|_b = 3m, n, m \in \mathbb{N}\}$:

Thus, L_3 is also known as the *interleaving* of L_1 and L_2.

2.3 Nondeterministic finite automata

Consider the language $a(a+b+c)^*c$, that is, the set of strings of a, b, and c starting in a and ending in c.

At first, we may come up with this solution:

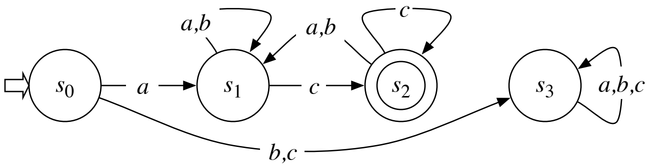


but this is not a DFA because

- There are two arcs leaving s_1 labeled with the same symbol, c: $\delta(s_1, c)$ can be either s_1 or s_2!
- There is no arc labeled b or c leaving s_0, and no arc labeled a, b, or c leaving s_2: what is the value of $\delta(s_0, b)$, $\delta(s_0, c)$, $\delta(s_2, a)$, $\delta(s_2, b)$, and $\delta(s_2, c)$?

The first point is the most important issue (the second could be fixed by adding a trap state): we want to know that when we see a c, it is the last symbol in the string.

We have two solutions. Either we modify the automaton, so that it is indeed a correct DFA:



or we allow nondeterminism, by introducing a new type of automaton.

A nondeterministic finite automaton (NFA) is a tuple $M = (Q, \Sigma, \delta, s_0, F)$ where

- Q is a finite set of states.
- Σ is an alphabet.
- $\delta : (Q \times (\Sigma \cup \{\epsilon\})) \to 2^Q$ is the *transition function.*
- $s_0 \in Q$ is the *initial state.*
- $F \subseteq Q$ are the *final states.*

Hence, the only difference between DFA and NFA is the form of the transition function.

For $a \in \Sigma$, $\delta(s, a) = \{s_1, s_2, \ldots, s_n\}$ means that any of those states can be reached from state s by consuming the input a.

For $a \in \Sigma$, $\delta(s, a) = \emptyset$ means that no state can be reached from state s by consuming the input a. This does not necessarily imply that (s, aw) is a *dead configuration*, because a ϵ-transition might still be possible.

Trivially, $\delta(s, \epsilon)$ always contains s itself, since the NFA can stay in place without consuming an input symbol. If, in addition, $\delta(s, \epsilon)$ contains another state s_1, the NFA can change state from s to s_1 without consuming any input, hence $(s, aw) \vdash (s_1, aw)$.

Note that we can have a sequence of ϵ-transitions.

We can extend δ to its reflexive and transitive closure, $\delta : (Q \times \Sigma^*) \to 2^Q$, defined by $s_1 \in \delta(s, w) \Leftrightarrow$ there is a path from s to s_1 labeled with w. This is computable because, even if there are cycles labeled only with ϵ, we do not need to follow them: we can recognize and avoid them.

An equivalent recursive definition is possible, but cumbersome.

We define the language accepted by a NFA as $L(M) = \{w : \delta(s_0, w) \cap F \neq \emptyset\}$

This if fundamental to nondeterminism: a string w is accepted by a NFA M iff at least one of the states that can be reached from s_0 on w is a final state.

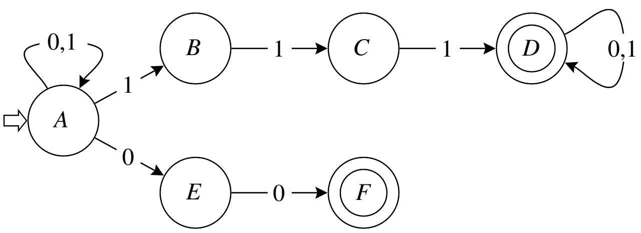
We can think of a NFA as an automaton that can clone itself every time it must make a choice. As soon as one of the clones reaches a final state after having consumed the last character, it alerts all the other clones that the process has completed. On the other hand, if a clone reaches a dead configuration, or reaches a nonfinal states after having consumed the last character, it simply dies off. So the string is not accepted iff none of the clones accepts it.

This is a fundamental asymmetry in the behavior of nondeterministic automata: they accept a string if *one* of the clones accepts it, they reject a string if *all* of the clones reject it.

2.3.1 More examples of NFA

Consider the language of the strings over $\Sigma = \{0, 1\}$ with three consecutive 1's or ending with two consecutive 0's.

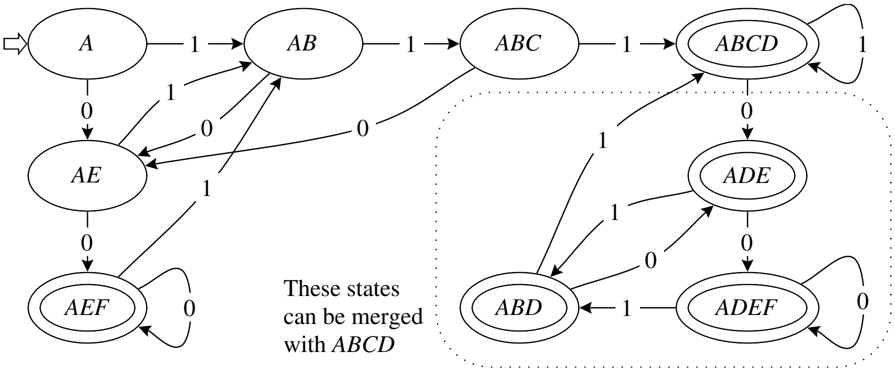
This language is easily described using a NFA $M_n = (\{A, B, C, D, E, F\}, \{0, 1\}, \delta_n, A, \{D, F\})$:



If we want to use a DFA, we need to define the states of the DFA as the sets of states in which the NFA could be.

That is, the NFA, starting from state A, can be in states A, B, or C after reading the string 011. Hence, the DFA should be in state ABC after reading 011.

Thus, $M_d = (\{A, AE, AEF, AB, ABC, ABCD, ADE, ABD, ADEF\}, \{0, 1\}, \delta_d, \{A\}, \{AEF, ABCD, ADE, ABD, ADEF\})$.



We can actually ignore unreachable states and even simplify further the DFA, since states $\{A, B, C, D\}$, $\{A, B, D\}$, $\{A, D, E\}$, and $\{A, D, E, F\}$ can be merged into a single state (once we know that the string contains three consecutive 1's, we don't care whether it also ends with two consecutive 0's.)

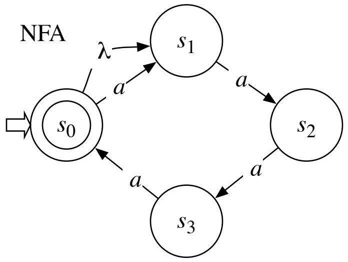
This example illustrates two important concepts which we study next:

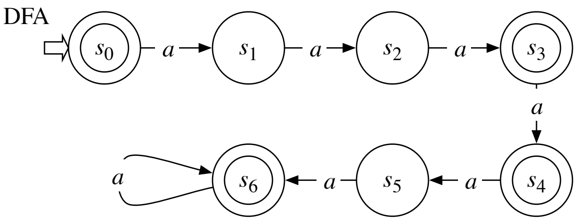
- How to obtain a DFA equivalent to a given NFA.
- How to minimize the number of states of a given DFA.

But first, let consider an example of NFA where the use of ϵ-transitions allows to obtain a compact NFA. Consider the language

$$L = \{w \in \{a\}^* : \exists k, l \in \mathbb{N}, |w| = 3k + 4l\} = \overline{\{a, a^2, a^5\}}$$

This is just a generalization of the "mod n" DFA we have seen before. The difficulty is that we do not know what k and l are.





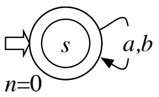
2.3.2 Importance of nondeterminism

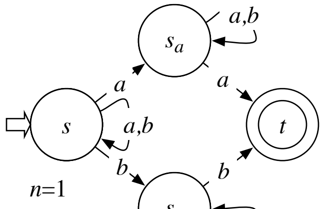
Nondeterminism is a fundamental concept. It allows us to express certain properties of a language in a very natural way.

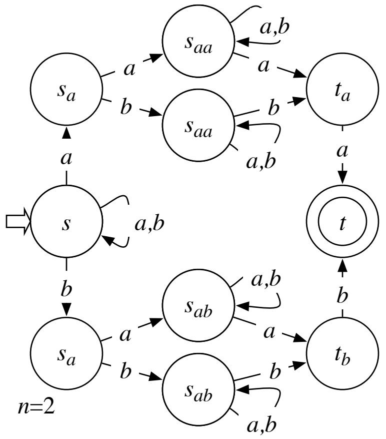
Example: Given a $n \in \mathbb{N}$, and $\Sigma = \{a, b\}$, derive a DFA and an NFA that accept the language $L_n = \{xwyw^R : x, y, w \in \Sigma^*, |w| = n\}$. Note that each n identifies a different language:

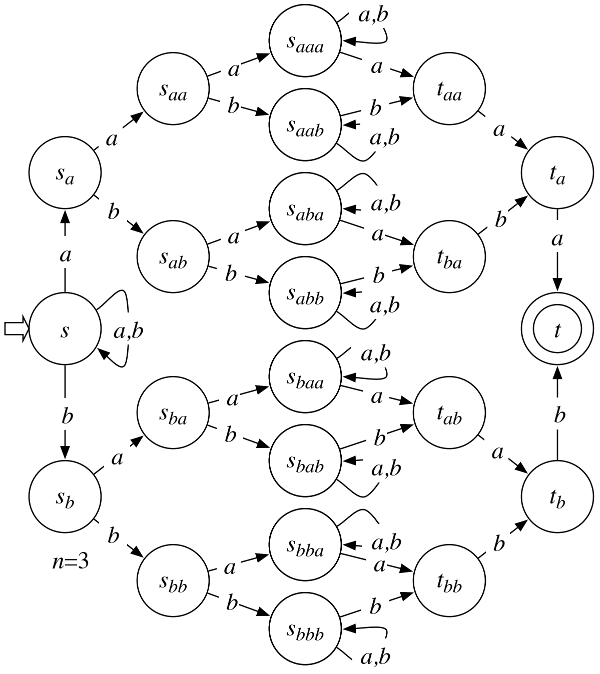
- $L_0 = (a + b)^*$.
- $L_1 = (a + b)^*a(a + b)^*a + (a + b)^*b(a + b)^*b$.
- $L_2 = (a + b)^*aa(a + b)^*aa + (a + b)^*ab(a + b)^*ba + (a + b)^*ba(a + b)^*ab + (a + b)^*bb(a + b)^*bb$.
- $L_3 = (a + b)^*aaa(a + b)^*aaa + (a + b)^*aab(a + b)^*baa + (a + b)^*aba(a + b)^*aba + (a + b)^*abb(a + b)^*bba + (a + b)^*baa(a + b)^*aab + (a + b)^*bab(a + b)^*bab + (a + b)^*bba(a + b)^*abb + (a + b)^*bbb(a + b)^*bbb$.

It is easier to start with the NFA:







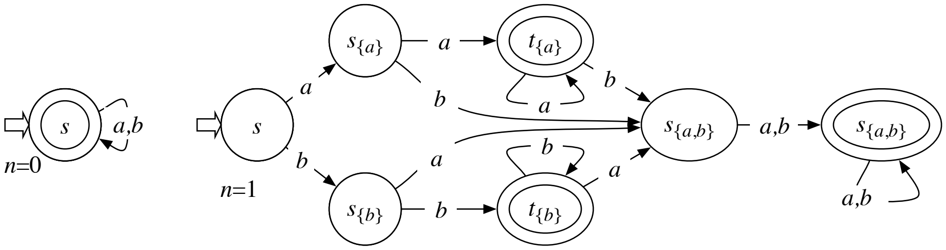


The number of states for the NFA recognizing L_n is $1+2+2^2+2^3+\cdots+2^{n-1}+2^n+2^{n-1}+\cdots+2^2+2+1 = \sum_{i=0}^{n} 2^i + \sum_{i=0}^{n-1} 2^i = 2^{n+1} - 1 + 2^n - 1 = 3 \cdot 2^n - 2$. So, for $n = 0$, the NFA has $2^0 3 - 2 = 1$ state; for $n = 1$, it has $3 \cdot 2^1 - 2 = 4$ states; for $n = 2$, it has $3 \cdot 2^2 - 2 = 10$ states; for $n = 3$, it has $3 \cdot 2^3 - 2 = 22$ states.

It is easy to see the pattern. While the number of states grows exponentially, the structure of the transition graph is quite regular. This example also shows the importance of a good choice of names for the states of an automaton. State s_α signifies that we have read $x\alpha$ and that we expect to read $\beta y \beta^R \alpha^R$ next, for some $\beta \in \Sigma^*$, $|\alpha\beta| = n$. State t_α signifies that we have read $x\alpha^R \beta y \beta^R$ and that we expect to read α next.

Hence, the state is used to record the previously seen input in a compact (finite!) way.

When we try to derive a DFA for this language, we are faced with a problem: how can we decide that we are starting to read w? The answer is that we can't! We need to record, in the state of the DFA, all the patterns of length n that we have seen. Furthermore, the DFA cannot determine whether it is reading the last n characters until it reaches the end of the input. This results in an enormous number of states, since there are $2^{(2^n)}$ sets of strings of length n.



2.4 Equivalence of DFA and NFA

Two FA M_1 and M_2 with alphabet Σ are equivalent if they accept the same language: $L(M_1) = L(M_2)$, that is $\forall w \in \Sigma^*, w \in L(M_1) \Leftrightarrow w \in L(M_2)$.

Theorem Given a NFA $M = (Q, \Sigma, \delta, s_0, F)$, there exists a DFA $M' = (Q', \Sigma, \delta', S'_0, F')$ such that $L(M) = L(M')$.

Proof Define an "extension" function $E : Q \to 2^Q$ as:

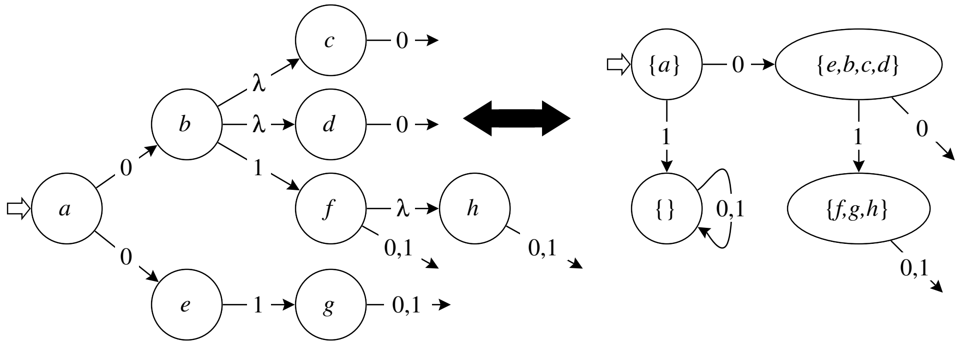
$$\forall s \in Q, \quad E(s) = \{t \in Q : (s, \epsilon) \overset{*}{\underset{M}{\vdash}} (t, \epsilon)\}$$

That is, $E(s)$ is the set of states that the NFA can reach from s without consuming any input symbol. Hence $E(s) \supseteq \{s\}$.

Then, the DFA is defined as follows:

- $Q' = 2^Q$: the states of the DFA are the subsets of Q. Some of these states may be unreachable, that's not important.
- $S'_0 = E(s_0)$: the initial state of the DFA corresponds to the initial state of the NFA plus any state that the NFA can reach from the initial state without consuming any input symbol.
- $F' = \{S \in 2^Q : S \cap F \neq \emptyset\}$: any state of the DFA corresponding to a subset of Q and containing one or more final states is itself a final state.
- $\forall S' \in Q', \forall a \in \Sigma, \quad \delta'(S', a) = \bigcup_{\exists s \in S', \exists t \in Q, \delta(s,a)=t} E(t)$: the state reached in the DFA by reading a while in state S' corresponds to the set of states in which the NFA could be after reading an a and then, possibly, performing any number of ϵ-transitions while in any of the states in S'.

Example: Assume that the NFA is initially in state a:



then, after reading a 0, the NFA could be in state b, c, d, or e. If the second symbol is a 1, the NFA could be in state f or h (if it was in b), or in state g, (if it was in e). Note that the NFA hangs on input 1 if the state is c or d.

We need to prove that the two automata recognize the same language:

$$\forall w \in \Sigma^*, \left(\exists s \in F, (s_0, w) \overset{*}{\underset{M}{\vdash}} (s, \epsilon)\right) \Leftrightarrow \left(\exists S \in F', (S'_0, w) \overset{*}{\underset{M'}{\vdash}} (S, \epsilon)\right)$$

Claim $\forall w \in \Sigma^*, \forall s, t \in Q, (s, w) \overset{*}{\underset{M}{\vdash}} (t, \epsilon) \Leftrightarrow \left((E(s), w) \overset{*}{\underset{M'}{\vdash}} (T, \epsilon) \wedge t \in T\right)$

If this claim is true, so it the theorem, since the claim implies the theorem when $s = s_0$ and t is a final state of the NFA M.

We prove the claim by induction on the length of w.

Basis: If $|w| = 0$, w is equal ϵ, and $(s, \epsilon) \overset{*}{\underset{M}{\vdash}} (t, \epsilon) \Leftrightarrow t \in E(s) = T$ (and we know that, by definition, $(E(s), \epsilon) \overset{*}{\underset{M'}{\vdash}} (E(s), \epsilon)$).

Inductive hypothesis: Assume that,

$$\forall w \in \Sigma^*, |w| \leq n, \forall s, t \in Q, (s, w) \overset{*}{\underset{M}{\vdash}} (t, \epsilon) \Leftrightarrow \left((E(s), w) \overset{*}{\underset{M'}{\vdash}} (T, \epsilon) \wedge t \in T\right).$$

Inductive step: We need to prove the claim when $|w| = n + 1$.

If $|w| = n + 1$, then $\exists v \in \Sigma^*, \exists a \in \Sigma, w = va$. Let us first prove the claim in one direction:

$(s, va) \overset{*}{\underset{M}{\vdash}} (t, \epsilon) \Rightarrow$

$\exists r_1, r_2 \in Q, (s, va) \overset{*}{\underset{M}{\vdash}} (r_1, a) \overset{1}{\underset{M}{\vdash}} (r_2, \epsilon) \overset{*}{\underset{M}{\vdash}} (t, \epsilon) \Rightarrow$

$(s, v) \overset{*}{\underset{M}{\vdash}} (r_1, \epsilon) \wedge (r_1, a) \overset{1}{\underset{M}{\vdash}} (r_2, \epsilon) \wedge (r_2, \epsilon) \overset{*}{\underset{M}{\vdash}} (t, \epsilon) \Rightarrow$

$\underbrace{\exists R_1 \in 2^Q, (E(s), v) \overset{*}{\underset{M'}{\vdash}} (R_1, \epsilon) \wedge r_1 \in R_1}_{\text{by Inductive hypothesis}} \wedge r_2 \in \delta(r_1, a) \wedge t \in E(r_2) \Rightarrow$

$(E(s), v) \overset{*}{\underset{M'}{\vdash}} (R_1, \epsilon) \wedge t \in \delta'(R_1, a) \Rightarrow$

$(E(s), v) \overset{*}{\underset{M'}{\vdash}} (R_1, \epsilon) \wedge \exists T \in 2^Q, (R_1, a) \overset{1}{\underset{M'}{\vdash}} (T, \epsilon) \wedge t \in T \Rightarrow$

$(E(s), va) \overset{*}{\underset{M'}{\vdash}} (R_1, a) \overset{1}{\underset{M'}{\vdash}} (T, \epsilon) \wedge t \in T \Rightarrow$

$(E(s), va) \overset{*}{\underset{M'}{\vdash}} (T, \epsilon) \wedge t \in T$

The proof in the other direction is exactly analogous:

$(E(s), va) \overset{*}{\underset{M'}{\vdash}} (T, \epsilon) \wedge t \in T \Rightarrow$

$\exists R_1 \in 2^Q, (E(s), va) \overset{*}{\underset{M'}{\vdash}} (R_1, a) \overset{1}{\underset{M'}{\vdash}} (T, \epsilon) \wedge t \in T \Rightarrow$

$(E(s), v) \overset{*}{\underset{M'}{\vdash}} (R_1, \epsilon) \wedge (R_1, a) \overset{1}{\underset{M'}{\vdash}} (T, \epsilon) \wedge t \in T \Rightarrow$

$(E(s), v) \overset{*}{\underset{M'}{\vdash}} (R_1, \epsilon) \wedge t \in \delta'(R', a) \Rightarrow$

$(E(s), v) \overset{*}{\underset{M'}{\vdash}} (R_1, \epsilon) \wedge \exists r_1 \in R, r_2 \in \delta(r_1, a), t \in E(r_2) \Rightarrow$
$(s, v) \overset{*}{\underset{M}{\vdash}} (r_1, \epsilon) \wedge (r_1, a) \overset{1}{\underset{M}{\vdash}} (r_2, \epsilon) \wedge (r_2, \epsilon) \overset{*}{\underset{M}{\vdash}} (t, \epsilon) \Rightarrow$
$(s, va) \overset{*}{\underset{M}{\vdash}} (t, \epsilon)$. □

2.5 DFA minimization

So far we have been interested in deriving a FA for a given language, regardless of its size (number of states).

We have seen that, if we come up with a NFA, we can transform it into a DFA, although, possibly, with a much larger number of states.

We have also seen an example of a family of languages for which the smallest DFA has an exponentially larger number of states than the smallest NFA for the same language.

We are now going to explore the concept of "smallest" in more detail, by giving an algorithm to minimize the number of states in a DFA.

Statement of the problem Given a DFA $M = (Q, \Sigma, \delta, s_0, F)$, find an equivalent DFA $M' = (Q', \Sigma, \delta', s'_0, F')$ that accepts the same language and has the minimum number of states:

$$L(M) = L(M') \wedge \forall M'' = (Q'', \Sigma, \delta'', s''_0, F''), L(M) = L(M'') \Rightarrow |Q''| \geq |Q'|$$

Definition Given a DFA $M = (Q, \Sigma, \delta, s_0, F)$, two states $s_1, s_2 \in Q$ are *distinguishable* if there is a string $w \in \Sigma^*$ such that $\delta(s_1, w) \in F$ and $\delta(s_2, w) \notin F$ or vice versa.

Algorithm The states of the minimized DFA are macro-states (sets of states of the original DFA). After dealing with the two extreme case, the algorithm starts optimistically with two macro-states, one final and one nonfinal, and splits (refines) them as needed.

Proof of correctness It is clear that the states of M' are a partition of the states of M. We prove that the algorithm is correct by proving three claims:

Claim 1: all states of M' are *distinguishable*. This is true because, initially, Q' contains only $(Q \setminus F)$ and F, and we split a state in Q' only if its states can reach different group of states. The proof is by induction based on the transitive property of the transition function.

Claim 2: M' accepts the correct language, that is, $L(M) = L(M')$. This is true because, by the way we defined δ', $\forall w = a_1 a_2 \cdots a_n \in \Sigma^*$,

$$(s_0, a_1 a_2 \cdots a_n) \overset{1}{\underset{M}{\vdash}} (s_1, a_2 \cdots a_n) \overset{1}{\underset{M}{\vdash}} \cdots \overset{1}{\underset{M}{\vdash}} (s_n, \epsilon) \Leftrightarrow$$

$$(Q(s_0), a_1 a_2 \cdots a_n) \overset{1}{\underset{M}{\vdash}} (Q(s_1), a_2 \cdots a_n) \overset{1}{\underset{M}{\vdash}} \cdots \overset{1}{\underset{M}{\vdash}} (Q(s_n), \epsilon)$$

where $Q(s)$ is the state of M' containing the state s of M.

Claim 3: M' is minimal. We prove this by contradiction. Assume that there is a DFA $M'' = (Q'', \Sigma, \delta'', s''_0, F'')$ such that $L(M'') = L(M)$ and $|Q''| < |Q'|$. Since all the states $Q_1, Q_2, \ldots Q_n$ in Q' are reachable, $\exists w_1, w_2, \ldots w_n \in \Sigma^*$ such that $\delta(s'_0, w_i) = Q_i, 1 \leq i \leq n$ and $w_i \neq w_j$ whenever $i \neq j$, since we are dealing with a DFA.

If $|Q''| < |Q'|$, $\exists i, j, i \neq j, 1 \leq i, j \leq n$ such that $\delta''(s''_0, w_i) = \delta''(s''_0, w_j)$.

All the states of M' are distinguishable, hence, $\exists w \in \Sigma^*, \delta'(Q_i, w) \in F' \wedge \delta'(Q_j, w) \notin F'$ or vice versa.

Hence, $\delta'(s'_0, w_i w) = \delta'(Q_i, w) \in F' \wedge \delta'(s'_0, w_j w) = \delta'(Q_j, w) \notin F'$, while $\delta''(s''_0, w_i w) = \delta''(s''_0, w_j w)$. Hence M'' either rejects both $w_i w$ and $w_j w$, or accepts both of them, while it should accept one and reject the other: a contradiction.

Example Minimize the following DFA, which recognizes $\{a^i b^j : \exists k, i + j = 2k, i, j, k \in \mathbb{N}\}$.

$Q' = \{ \ \{s_0, s_2, s_5\} \ , \ \{s_1, s_3, s_4, s_6\} \ \}$

on a: $\delta(\{s_0, s_2, s_5\}, a) = \{s_1, s_6\}$: no split

on a: $\delta(\{s_1, s_3, s_4, s_6\}, a) = \{s_0, s_6\}$: split into $\{s_1\}$ and $\{s_3, s_4, s_6\}$.

$Q' = \{ \ \{s_0, s_2, s_5\} \ , \ \{s_1\} \ , \ \{s_3, s_4, s_6\} \ \}$

on b: $\delta(\{s_0, s_2, s_5\}, b) = \{s_3, s_4\}$: no split

we do not need to check $\{s_1\}$ further, since it cannot be split

on b: $\delta(\{s_3, s_4, s_6\}, b) = \{s_2, s_5, s_6\}$: split into $\{s_3, s_4\}$ and $\{s_6\}$.

$Q' = \{ \ \{s_0, s_2, s_5\} \ , \ \{s_1\} \ , \ \{s_3, s_4\} \ , \ \{s_6\} \ \}$

on a: $\delta(\{s_0, s_2, s_5\}, a) = \{s_1, s_6\}$: split into $\{s_0\}$ and $\{s_2, s_5\}$.

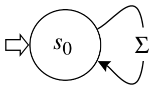
$Q' = \{ \ \{s_0\} \ , \ \{s_2, s_5\} \ , \ \{s_1\} \ , \ \{s_3, s_4\} \ , \ \{s_6\} \ \}$

No further split is needed since:

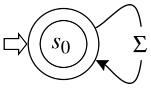
$\delta(\{s_0\}, a) = \{s_1\}$
$\delta(\{s_2, s_5\}, a) = \{s_6\}$
$\delta(\{s_1\}, a) = \{s_0\}$
$\delta(\{s_3, s_4\}, a) = \{s_6\}$
$\delta(\{s_0\}, b) = \{s_4\} \subseteq \{s_3, s_4\}$
$\delta(\{s_2, s_5\}, b) = \{s_3, s_4\}$
$\delta(\{s_1\}, b) = \{s_2\} \subseteq \{s_2, s_5\}$
$\delta(\{s_3, s_4\}, b) = \{s_2, s_5\}$

DFA minimization algorithm:

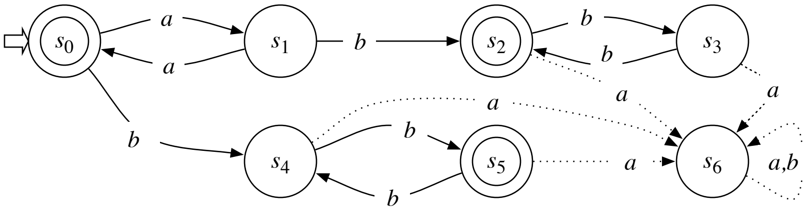
```
eliminate all states in $Q$ not reachable from $s_0$;
if all remaining states are nonfinal, $L(M) = \emptyset$ and the minimized DFA is
```

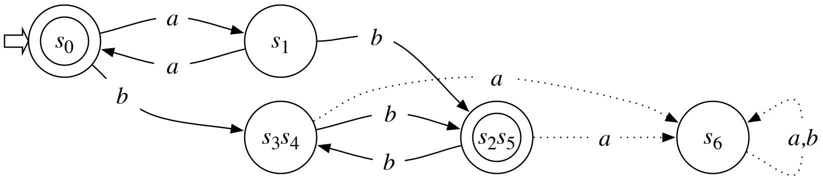


```
if all remaining states are final, $L(M) = \Sigma^*$ and the minimized DFA is
```



```
• otherwise we can conclude that $\emptyset \subset L(M) \subset \Sigma^*$, and we must do the following
$Q' \leftarrow \{(Q - F), F\}$;          • start with the smallest possible case, two macro-states
repeat
  $CHANGE \leftarrow false$;          • used to stop the algorithm
  for each $(K, a) \in Q' \times \Sigma$ do
    $S \leftarrow \emptyset$;
    for each $q \in K$ do
      $S \leftarrow S \cup \{\delta(q, a)\}$          • find states reachable from macro-state $K$ on $a$
    end for;
    if not $\exists K' \in Q'$ such that $S \subseteq K'$ then
• macro-state $K$ must be split because it contains states going to different macro-states on $a$
      partition $K$ into the smallest number $r \geq 2$ of sets $K_1, K_2, \ldots K_r$ such that
      $\forall i = 1, \ldots r,\ \exists K' \in Q',\ \forall q \in K_i,\ \delta(q, a) \in K'$;
• each state in macro-state $K_i$ goes to the same macro-state $K'$ on $a$, and the partition of $K$ contains
  two macro-states $K_i$ and $K_j$ only if they go to different macro-states in $Q'$
      $Q' \leftarrow Q' \setminus \{K\} \cup \{K_1, K_2, \ldots K_r\}$;
      $CHANGE \leftarrow true$;
    end if;
  end for;
until $CHANGE = false$;
$F' \leftarrow \{K \in Q' : K \cap F \neq \emptyset\}$;
• we could also say $F' \leftarrow \{K \in Q' : K \subseteq F\}$ because a macro-state $K$ is final if it contains only final
  states; it is nonfinal if it contains only nonfinal states; no other case is possible, since we start with
  macro-states $Q \setminus F$ and $F$ and we never merge macro-states
$s'_0 \leftarrow K \in Q'$ such that $s_0 \in K$;          • there is only one such $K$
for each $(K, a) \in Q' \times \Sigma$ do
  $\delta'(K, a) \leftarrow K' \in Q'$ such that $\exists q \in K,\ \exists q' \in K',\ \delta(q, a) = q' \wedge q \in K \wedge q' \in K'$;
• well defined: all the states in $K$ go to states in $K'$ on $a$; if not, we would have split $K$ further
end for;
```





2.6 Closure properties for FA

We can now talk about FA, without differentiating between DFA and NFA.

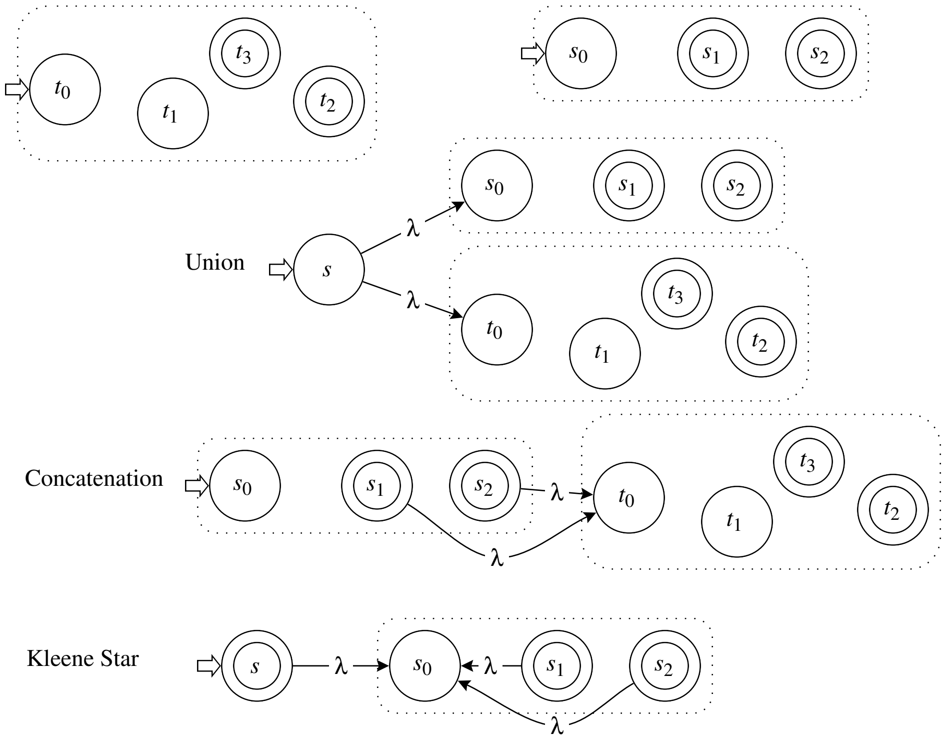
A set $S \subseteq T$ is *closed* with respect an n-ary operator $\circ : T^n \to T$ iff $\forall s_1, \ldots, s_n \in S, \circ(s_1, \ldots, s_n) \in S$.

We are interested in their *closure properties* with respect to an operator f, that is: given any two FA M_1 and M_2, is there a FA for the language $f(L(M_1), L(M_2))$?

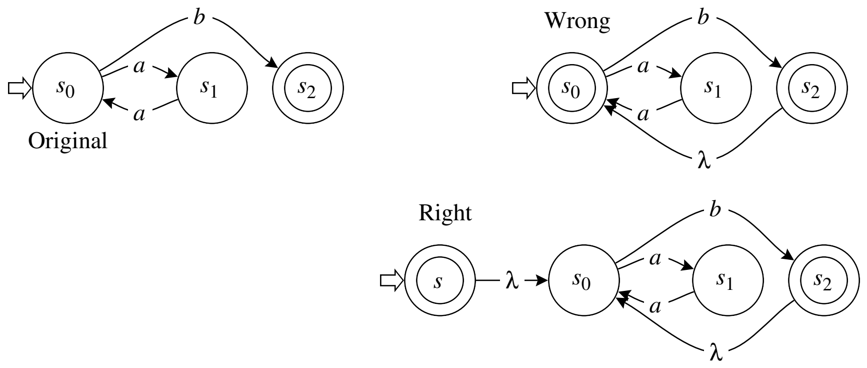
Theorem The class of languages recognized by FA is closed under

- Union: $\exists M : L(M) = L(M_1) \cup L(M_2)$.
- Concatenation: $\exists M : L(M) = L(M_1) \cdot L(M_2)$
- Kleene star: $\exists M : L(M) = L(M_1)^*$
- Complementation: $\exists M : L(M) = \overline{L(M_1)} = \Sigma^* \setminus L(M_1)$
- Intersection: $\exists M : L(M) = L(M_1) \cap L(M_2)$
- Difference: $\exists M : L(M) = L(M_1) \setminus L(M_2)$
- Reversal: $\exists M : L(M) = L^R = \{w : w^R \in L(M_1)\}$
- Homomorphism: given a function $\phi : \Sigma \to \Gamma^*$, $\exists M : L(M) = \phi(L(M_1)) = \{w_1 w_2 \cdots w_n \in \Gamma^* : \exists a_1 \cdots a_n \in L(M_1), w_1 = \phi(a_1), \ldots, w_n = \phi(a_n)\}$
- Right quotient: $\exists M : L(M) = L_1/L_2$, where L_1/L_2 is defined as $\{x \in \Sigma^* : \exists y \in L_2, xy \in L_1\}$, that is, the strings that, followed by a string in L_2, form a string in L_1. Left quotient is analogous, but the string in L_2 precedes, instead of follows.

Proof We prove the first three properties by sketching graphically how to obtain the resulting FA:



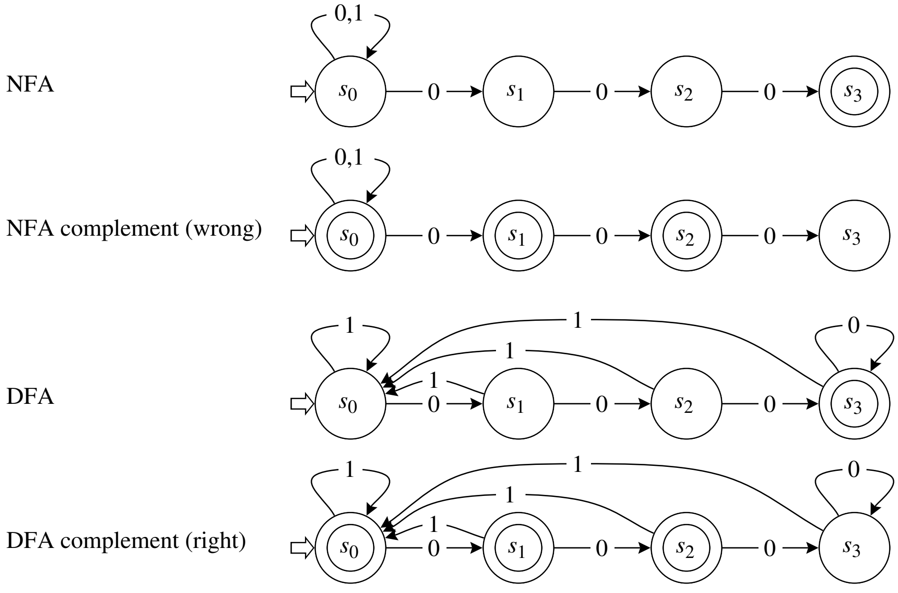
The state s in the Kleene star construction is essential. If we start in s_0 instead, after making it final, and s_0 is not a final state in the original language, we might accept some strings which should be rejected. For example, consider $L(M_1) = (aa)^*b$.



For complementation see exercise in homeworks. We show that, given a DFA $M_1 = (Q, \Sigma, \delta, s_0, F)$, the DFA $M = (Q, \Sigma, \delta, s_0, Q \setminus F)$ accepts $\Sigma^* \setminus L(M_1)$, since:

$$\forall w \in \Sigma^*, w \in L(M) \Leftrightarrow \delta(s_0, w) \in F \Leftrightarrow \delta(s_0, w) \notin Q \setminus F \Leftrightarrow w \notin L(M').$$

It is important to remember that the construction is correct only on DFA, not NFA:



The first NFA accepts (0+1)*000, the strings of 0's and 1's ending with three zeros. The "complement NFA" accepts $(0 + 1)^*$, hence it does not accept the complement of the language. The same process is shown for a DFA instead, yielding a correct "complement DFA", which accepts any string not ending in 000.

For intersection, it is sufficient to apply De Morgan's law:

$$L(M_1) \cap L(M_2) = \overline{\overline{L(M_1) \cap L(M_2)}} = \overline{\overline{L(M_1)} \cup \overline{L(M_2)}}.$$

For difference, it is sufficient to observe that $L(M_1) \setminus L(M_2) = L(M_1) \cap \overline{L(M_2)}$.

For reversal, see exercise in homeworks. Given the DFA $M_1 = (Q, \Sigma, \delta, s_0, F)$ (but this works with an NFA as well), define the NFA $M = (Q \cup \{s_x\}, \Sigma, \delta', s_x, \{s_0\})$ such that $L(M) = L^R$ as follows: $\delta(s,a) = t \;\Rightarrow\; s \in \delta'(t,a)$ and $\delta(s_x, \epsilon) = F$. That is, reverse all the arcs, make the initial state final, the final states nonfinal, and add a new initial state s_x with ϵ-transitions to each original final state. To prove that $w \in L(M_1) \Leftrightarrow w^R \in L(M)$, consider the paths on the graphs of the two FA.

For homomorphism, see exercise in the homeworks. Given the DFA $M_1 = (Q, \Sigma, \delta, s_0, F)$, define the NFA $M = (Q', \Gamma, \delta', s_0, F)$ as follows:

$$Q' = Q \cup \{(s,a,i) : s \in Q, a \in \Sigma, i \in \{1, \ldots, |\phi(a)|\}\}$$

If $|\phi(a)| > 1$, $\phi(a) = \gamma_1\gamma_2 \cdots \gamma_{|\phi(a)|}$,

$$\forall s \in Q, \delta(s,a) = t \Rightarrow (s,a,1) \in \delta'(s,\gamma_1) \wedge (s,a,2) \in \delta'((s,a,1),\gamma_2) \wedge \cdots \wedge t \in \delta'((s,a,1),\gamma_{|\phi(a)|})$$

If $|\phi(a)| = 1$, $\phi(a) = \gamma$ or $|\phi(a)| = 0$, $\phi(a) = \epsilon$,

$$\forall s \in Q, \delta(s,a) = t \;\Rightarrow\; t \in \delta'(s, \phi(a))$$

That is, if there is an arc from s to t with label a in M_1, there is a path from s to t with label $\phi(a)$ in M. □

2.7 Regular languages

Definition A language L is regular iff it can be represented by a regular expression R: $L = L(R)$.

Theorem A language is regular iff it is accepted by a finite automaton.

Proof In two parts: first, let's prove that if a language L is regular, $L = L(R)$ then there is a FA M such that $L = L(M)$. By induction, on the number of operators in R:

- $R = \emptyset \Rightarrow L(R) = \emptyset$, and we know how to build a FA that recognizes $\emptyset$.
- $R = a \in \Sigma \Rightarrow L(R) = \{a\}$, and we know how to build a FA that recognizes $\{a\}$.
- $R = (\alpha + \beta)$, or $R = (\alpha \cdot \beta)$, or $R = \alpha^* \Rightarrow L(R) = L(\alpha) \cup L(\beta)$, or $L(R) = L(\alpha) \cdot L(\beta)$, or $L(R) = L(\alpha)^*$, and we know how that the languages accepted by FA are closed under union, concatenation, and Kleene star.

The proof that if a language L is accepted by a DFA M, $L = L(M)$, then there exists a regular expression R such that $L = L(R)$ is more complex. We will show how to "build" a regular expression corresponding to M.

Without loss of generality, assume that the set of states of $M = (Q, \Sigma, \delta, s_1, F)$ is $Q = \{s_1, s_2, \ldots s_n\}$.

Define $R(i,j,k) = \{w \in \Sigma^* : (s_i, w) \overset{*}{\underset{M}{\vdash}} (s_j, \epsilon) \wedge M$ visits only intermediate states with index at most k in this computation$\}$, for $0 \le k \le n$. If $w = a_1a_2 \cdots a_{|w|}$, then

$$\underbrace{(s_i, a_1a_2\cdots a_{|w|}) \overset{1}{\underset{M}{\vdash}} (q_{i_1}, a_2\cdots a_{|w|}) \overset{1}{\underset{M}{\vdash}} (q_{i_2}, a_3\cdots a_{|w|}) \overset{1}{\underset{M}{\vdash}} \cdots \overset{1}{\underset{M}{\vdash}} (s_j, \epsilon)}_{\text{in } |w| \text{ steps}}$$

and $q_1, q_2, \ldots, q_{|w|-1} \in \{s_1, s_2, ..., s_k\}$.

Each $R(i,j,k)$ is a language (thus we have defined $n \times n \times (n+1)$ languages), and, clearly,

$$L(M) = \bigcup_{j \in F} R(s_1, j, n)$$

since, when $k = n$, there is no restriction on the intermediate states that can be visited.

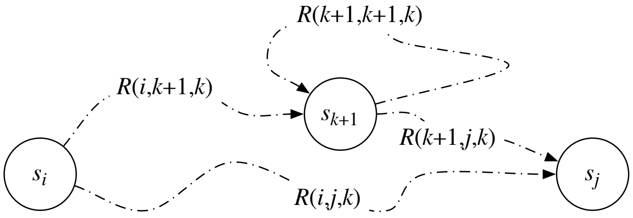
We will prove, by induction on k, that all the $R(i,j,k)$ are regular languages (we will give regular expressions for each of them).

Basis $k = 0$:
if $i \neq j$ and $\forall a \in \Sigma, \delta(s_i, a) \neq s_j$, $R(i,j,0) = \emptyset$
if $i \neq j$ and $\exists a_1, a_2, \ldots a_l \in \Sigma, \delta(s_i, a_1) = \delta(s_i, a_2) = \cdots = \delta(s_i, a_l) = s_j$, $R(i,j,0) = \{a_1, a_2, \ldots a_l\}$
if $i = j$, add ϵ to above sets: if $i = j$ and $\forall a \in \Sigma, \delta(s_i, a) \neq s_j$, $R(i,j,0) = \{\epsilon\}$, if $i = j$ and $\exists a_1, a_2, \ldots a_l \in \Sigma, \delta(s_i, a_1) = \delta(s_i, a_2) = \cdots = \delta(s_i, a_l) = s_j$, $R(i,j,0) = \{\epsilon, a_1, a_2, \ldots a_l\}$
All these sets are regular.

Inductive hypothesis Assume that $\forall i, j \in Q, R(i,j,k)$ is regular.

Inductive step There are two ways to go from i to j without visiting states with index larger than $k+1$: we can avoid visiting state $k+1$ as well, or we can visit it. In the second case, we can visit it any number of times.

$$R(i,j,k+1) = R(i,j,k) \cup R(i,k+1,k) \cdot R(k+1,k+1,k)^* \cdot R(k+1,j,k)$$



$\Rightarrow R(i,j,k+1)$ is regular as well. □

2.8 Pumping theorem for regular languages

Theorem Let L be a regular language over Σ. Then, $\exists n \in \mathbb{N}$ such that $\forall x \in L$, if $|x| \geq n$, then it is possible to decompose x into three strings, $\exists u, v, w \in \Sigma^*$, $x = uvw$, $|uv| \leq n$, $|v| > 0$, such that "v pumps", $\forall k \in \mathbb{N}, uv^k w \in L$.

Proof If L is finite, the theorem is vacuously true, just choose n greater than the length of any string in L, so that no x satisfying $|x| > n$ exists.

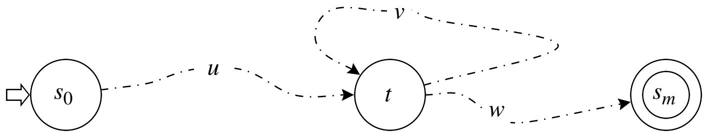
If L is infinite, consider a DFA $M = (Q, \Sigma, \delta, s_0, F)$ recognizing L and a string $a_1 a_2 \cdots a_m = x \in L$, $|x| = m \geq |Q|$. Then

$$(s_0, a_1 a_2 \cdots a_m) \overset{1}{\underset{M}{\vdash}} (s_1, a_2 \cdots a_m) \overset{1}{\underset{M}{\vdash}} \cdots \overset{1}{\underset{M}{\vdash}} (s_{m-1}, a_m) \overset{1}{\underset{M}{\vdash}} (s_m, \epsilon)$$

At least one state among the first $|Q|+1$ states encountered, $\{s_0, s_i, \ldots, s_{|Q|}\}$, must be repeated, define t to be such a state:

$$t = s_p, \quad p = \min_{1\le j\le |Q|}\{\exists i \in \mathbb{N} : i < j \wedge s_i = s_j\}$$

Then, $(s_0, uvw) \overset{|u|}{\underset{M}{\vdash}} (t, vw) \overset{|v|}{\underset{M}{\vdash}} (t, w), \overset{|w|}{\underset{M}{\vdash}} (s_m, \epsilon)$ where $uvw = a_1 \cdots a_m$, $vw = a_{i+1} \cdots a_m$, $w = a_{j+1} \cdots a_m$.



But then, $uw \in L$, $uvvw \in L$, and so on. □

Very important observations
$|uv| \le n$: pumping must happen soon.
By the same argument we can prove a very similar theorem where $|vw| \le n$.
$v \ne \epsilon$: otherwise the theorem is always trivially true.
$\forall x, |x| \ge n$: all the strings beyond a certain length must pump.
$k = 1$ gives the original string in the language; $k = 0$ and $k > 1$ all must result in a string in the language.

The pumping lemma is useful to prove that a given language is NOT regular. It can never be used to show that a language IS regular (there are nonregular languages that pump!).

Example Prove that $\{a^n b^n : n \in \mathbb{N}\}$ is not regular. By contradiction. Assume that L is regular, then $\exists m \in \mathbb{N}, \forall x \in L, |x| > m$, $\exists u, v, w \in \{a, b\}^*, x = uvw, |uv| \le m, |v| > 0$, $\forall k \in \mathbb{N}, uv^k w \in L$.

Consider $a^m b^m$, and decompose it in any possible way into uvw satisfying $|uv| \le m, |v| > 0$. This implies that $v = a^i$ for some $i > 0$. Then, $uv^0 w = a^{m-i} b^m$ should be in L as well, but it is not.

Note that, if we did not use the fact $|uv| \le m$, we would have to consider three cases: $v = a^i$, $v = a^i b^j$, $v = b^j$. The same result can still be proved, but with more effort.

Example Prove that $\{a^{n^2} : n \in \mathbb{N}\}$ is not regular. By contradiction. Assume that L is regular, then $\exists m \in \mathbb{N}, \forall x \in L, |x| > m$, $\exists u, v, w \in a^*, x = uvw, |uv| \le m, |v| > 0$, $\forall k \in \mathbb{N}, uv^k w \in L$.

Consider a^{m^2}, and decompose it in any possible way into uvw satisfying $|uv| \le m, |v| > 0$. Then, $u = a^j$, $v = a^i$, $i > 0$, and $w = a^{m^2-i-j}$. (in practice, only the length i of v is important). We can then write $x = uvw = a^j a^i a^{m^2-i-j}$ and, $\forall k \in \mathbb{N}, a^j a^{ki} a^{m^2-i-j} = a^{m^2+i(k-1)} \in L$, that is, $\forall k \in \mathbb{N}, m^2 + i(k-1)$ should be a square.

Considering $k = im^2 + 1$, however, we see that $m^2 + i(k-1) = (i^2+1)m^2$ cannot be a square, because $i^2 + 1$ is not a square for $i > 0$.

Example $L = \{a^k b^n c^n b^m : k, n, m \in \mathbb{N}\}$. It is much easier to show that L is not regular by first intersecting it with $L(b^+c^+)$, thus obtaining $L' = L \cap L(b^+c^+) = \{b^n c^n : n > 0\}$. Then, $L'' = L' \cup \{\epsilon\} = \{b^n c^n : n \in \mathbb{N}\}$ is obtained from $L''' = \{a^n b^n : n \in \mathbb{N}\}$ through the homomorphism $a \to b$ and $b \to c$. We know that L''' is nonregular, hence L cannot be regular either.

Example Prove that $L = \{ww^R : w \in \{a, b\}^*\}$ is not regular. By contradiction. Assume that L is regular, then $\exists n \in \mathbb{N}, \forall x \in L, |x| > n$, $\exists u, v, w \in \{a, b\}^*, x = uvw, |uv| \le n, |v| > 0$, $\forall k \in \mathbb{N}, uv^k w \in L$.

Trick question: when is $L = \{ww^R : w \in \Sigma^*\}$ regular? When $|\Sigma| = 1$.

Consider $a^n bba^n$, and decompose it in any possible way into uvw satisfying $|uv| \leq n, |v| > 0$. This implies that $v = a^i$ for some $i > 0$. Then, $uv^0w = a^{n-i}bba^n$ should be in L as well, but it is not.

Example Prove that $L = \{w \in \{a,b\}^* : |w|_a < |w|_b\}$ is not regular. By contradiction. Assume that L is regular, then $\exists n \in \mathbb{N}, \forall x \in L, |x| > n$, $\exists u, v, w \in \{a,b\}^*, x = uvw, |uv| \leq n, |v| > 0$, $\forall k \in \mathbb{N}, uv^kw \in L$.

Consider $a^n b^{n+1}$, and decompose it in any possible way into uvw satisfying $|uv| \leq m, |v| > 0$. This implies that $v = a^i$ for some $i > 0$. Then, $uv^2w = a^{n+i}b^{n+1}$ should be in L as well, but it is not, since $i > 0$ implies that $n + i \geq n + 1$.

If the language had been $\{w\{a,b\}^* : |w|_a > |w|_b\}$, a good string to choose would be $a^{n+1}b^n$, since, by pumping v zero times we get a string not in the language.

2.9 Right and left linear grammars

A right linear grammar is a grammar $G = (V, T, S, P)$ where the productions P are of the form $A \to xB$ or $A \to x$, with $B \in V$ and $x \in T^*$.

Since the initial sentential form, S, has exactly one nonterminal, and since al productions have at most one nonterminal, all sentential forms have exactly one nonterminal, with the exception of the last one, which has none.

A left linear grammar is exactly analogous, but productions are of the form $A \to Bx$ or $A \to x$.

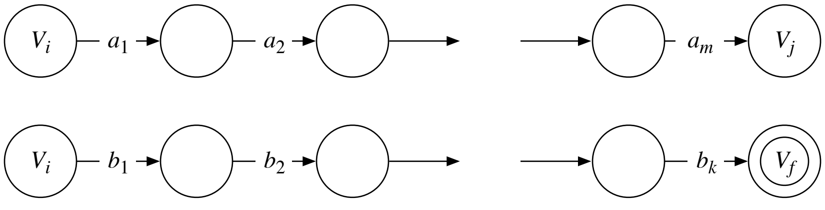
A regular grammar is either a right linear grammar or a left linear grammar.

Theorem If $G = (V, T, S, P)$ is a regular grammar, then $L(G)$ is regular.

Proof Let's consider the case of a right linear grammar, the case of a left linear grammar is exactly analogous. Assume that $V = \{V_0, V_1, \ldots, V_n\}$ and that $S = V_0$.

We build a NFA $M = (V \cup \{V_f\}, T, \delta, V_0, \{V_f\})$ with $V_f \notin V$.

For each production $V_i \to a_1a_2 \cdots a_mV_j \in P$, we must have $V_j \in \delta(V_i, a_1a_2 \cdots a_m)$ and for each production $V_i \to b_1b_2 \cdots b_k \in P$, we must have $V_f \in \delta(V_i, b_1b_2 \cdots b_m)$.



In particular, if $V_i \to \epsilon$, we have a ϵ-transition from V_i to V_f.

It is easy to see that $L(G) = L(M)$. □

Theorem If L is regular, then there is a regular grammar $G = (V, T, S, P)$ such that $L = L(G)$.

Proof Since L is regular, we can assume that there is a NFA $M = (Q, \Sigma, \delta, s_0, F)$ such that

$L = L(M)$. Then define $G = (Q, \Sigma, s_0, P)$ so that

$$t \in \delta(s, a) \;\Leftrightarrow\; s \to at \in P$$

and

$$t \in F \;\Leftrightarrow\; t \to \epsilon \in P$$

Then, it is easy to show by induction on n that

$$(s_i, a_1 \cdots a_n) \overset{*}{\underset{M}{\vdash}} (s_n, \epsilon) \;\Leftrightarrow\; s_i \overset{*}{\underset{G}{\Longrightarrow}} a_1 \cdots a_n s_n$$

Finally, $a_1 \cdots a_n \in L \;\Leftrightarrow\; (s_i, a_1 \cdots a_n) \overset{*}{\underset{M}{\vdash}} (s_n, \epsilon) \wedge s_n \in F \;\Leftrightarrow\; (s_n \to \epsilon) \in P$, hence we can complete the derivation $s_0 \overset{*}{\underset{G}{\Longrightarrow}} a_1 \cdots a_n s_n \overset{1}{\underset{G}{\Longrightarrow}} a_1 \cdots a_n$ if and only if $a_1 \cdots a_n \in L$. □

2.10 Summary on regular languages

Theorem There is an algorithm to determine whether $w \in L$ if L is regular, assuming L is specified as a regular expression or equivalent representation (DFA, NFA, regular grammar).

Proof First, derive a DFA M for L, then follow the computation of M on w. □

Theorem There is an algorithm to determine whether a regular language L is empty, finite, infinite, equal Σ^*.

Proof First, derive a minimized DFA M for L.

If M consists of a single nonfinal state, $L = \emptyset$.

If M consists of a single final state, $L = \Sigma^*$.

If M contains a cycle, L is infinite. Otherwise, it is finite. □

Theorem There is an algorithm to determine whether two regular languages L_1 and L_2 are equal (contain the same strings).

Proof We could think to "try various strings on both", but this empirical testing approach cannot prove that $L_1 = L_2$ (although it can prove $L_1 \neq L_2$, if we find a $w \in L_1$ such that $w \notin L_2$, or vice versa).

The right way to prove this is simply to observe that $L = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$ is empty if and only if $L_1 = L_2$. But L is regular if L_1 and L_2 are, hence we can test whether $L = \emptyset$. □

OPTIONAL: encoding languages with decision diagrams

It is obvious that any *finite* language $\mathcal{L}$, that is, any finite set of strings $\mathcal{L} = \{w_1, ..., w_n\}$, is regular: we can simply build an NFA M_{nondet} that

- has an initial state s_0,
- for each string $w_i = a_{i,1} \cdots a_{i,m_i} \in \mathcal{L}$, has m_i states $s_{i,1}, ..., s_{i,m_i}$, such that $\delta(s_{i,j-1}, a_{i,j}) = \{s_{i,j}\}$, for $1 \leq j \leq m_i$, where we let $s_0 \equiv s_{i,0}$, and
- the only final states are s_{i,m_i}, for each string $w_i = a_{i,1} \cdots a_{i,m_i} \in \mathcal{L}$.

M_{nondet} is an NFA and not a DFA because, in general, two strings w_i and w_j in $\mathcal{L}$ can start with the same symbol $a_{i,1} = a_{j,1}$. However, it is easy to see how we can transform M_{nondet} into a DFA M_{merge} that has the shape of a *tree*, by simply merging $s_{i,1}$ with $s_{j,1}$, $s_{i,2}$ with $s_{j,2}$, up to $s_{i,l}$ with $s_{j,l}$, if $a_{i,1} = a_{j,1}$, $a_{i,2} = a_{j,2}$, $a_{i,l} = a_{j,l}$, but $a_{i,l+1} \neq a_{j,l+1}$, for all strings w_i and w_j in $\mathcal{L}$. Of course, we also need to add all the missing transitions out of each state, making them go to a trap state (this of course is in addition to the tree structure of M_{merge}).

The resulting DFA M_{merge} is also called a *decision tree*, for obvious reasons: at each state, we decide the next state (child) to visit based on the next character. One way to think of M_{merge} is that it is the DFA is obtained from the NFA M_{nondet} by maximally merging the common *prefixes* for the strings in $\mathcal{L}$. The DFA M_{merge}, however, is far from being minimized. For example, it has as many final states as there are strings in $\mathcal{L}$, while all the *terminal* final states, i.e., those whose only outgoing transitions go to the trap state, could be merged. Furthermore, just merging the final state would not result in a minimized DFA, as there may be common *suffixes* that should be exploited, just as we exploited common prefixes.

Of course, we could just run the DFA M_{merge} through the minimization algorithm seen in class, but this assumes that, as we discussed so far, the language $\mathcal{L}$ is given to us by *explicitly* specifying its elements one-by-one, so that the cost of building M_{nondet} and then M_{merge}, or even just M_{merge} directly, is linear in the size of $\mathcal{L}$. Our goal is instead to build and manipulate a minimized DFA for such a finite language $\mathcal{L}$ in less than $O(|\mathcal{L}|)$ time and memory, when possible (it won't always be, the techniques we will discuss are *heuristics*, meaning that they tend to work well in many cases, but not in all cases). This is important because there are many applications where the size of $\mathcal{L}$ is enormous ($|\mathcal{L}|$ is of the order of 10^{50} or even more — that's a one followed by 50 zeros!!!), but the minimized DFS for $\mathcal{L}$ contains relatively few states (often their number is just a low-degree polynomial in $\log |\mathcal{L}|$).

2.11 Binary decision diagrams

Especially in the verification of VLSI circuits, an important and common case arises where $\mathcal{L} \subseteq \mathbb{B}^L$, with $\mathbb{B} = \{0, 1\}$, that is, all the strings in the language $\mathcal{L}$ are composed of 0's and 1's and have length *exactly* equal to L.

Such a set of L-tuples can then be stored using an *ordered binary decision diagram*, which is a directed acyclic edge-labeled multi-graph such that:

- The only *terminal* nodes can be $\mathbf{0}$ and $\mathbf{1}$, and are at *level* 0: we write $\mathbf{0}.lvl = \mathbf{1}.lvl = 0$.
- A *nonterminal* node p is at a *level* k, with $L \geq k \geq 1$: we write $p.lvl = k$.
- A nonterminal node p at level k has two outgoing edges pointing to *child* nodes $p[0]$ and $p[1]$.
- The level of the children is lower than that of p: $p[0].lvl < p.lvl$ and $p[1].lvl < p.lvl$.

Then, a node p at level k encodes the function $v_p : \mathbb{B}^L \rightarrow \mathbb{B}$ defined recursively by

$$v_p(x_L, ..., x_1) = \begin{cases} p & \text{if } k = 0 \\ v_{p[x_k]}(x_L, ..., x_1) & \text{if } k > 0, \end{cases}$$

where the x_k are the boolean *variables* corresponding to the levels. If we then consider v_p to be the *indicator function* for the set $\mathcal{S}_p = \{\mathbf{i} \in \mathbb{B}^L : v_p(\mathbf{i}) = 1\}$, then it obvious that a node p encodes a set of L-tuples as well.

As defined, ordered binary decision diagrams are not *canonical*, that is, different ones can encode the same function. To eliminate this possibility, we place additional requirements on the form of the diagram, resulting in either the *quasi-reduced ordered binary decision diagrams (QROBDDs)* or the *fully-reduced ordered binary decision diagrams (FROBDDs)*. Both types require that

- There are no *duplicates*: if $p.lvl = q.lvl = k > 0$, $p[0] = q[0]$, and $p[1] = q[1]$, then $p = q$.

Then, in the quasi-reduced case, there is *no level skipping*:

- The only *root* nodes with no incoming arcs are at level L.
- If a node p is at level $k > 0$, its children $p[0]$ and $p[1]$ are at level $k - 1$.

While, in the *fully-reduced* case, there is *maximum level skipping*:

- There is no *redundant* node p at level $k > 0$ satisfying $p[0] = p[1]$.

It is then easy to see that a QROBDD root node p encoding the function v_p is exactly the minimized DFA M_{min} accepting the language $\mathcal{L} = \mathcal{S}_p$, where p is the initial state and $\mathbf{1}$ is the only final state (of course, after adding the missing transitions from the terminal nodes $\mathbf{0}$ and $\mathbf{1}$ to the trap $\mathbf{0}$). The FROBDD node q encoding the same function $v_q = v_p$ is instead not quite a DFA, as it is obtained from the minimized DFA M_{min} by eliminating (skipping) all the redundant nodes (states). Thus, the number of nodes in the FROBDD is always less or equal to the number of nodes in the equivalent QROBDD, although this is achieved at the cost of having to actually store the level of a node in the node itself, while the level is implicitly equal to the distance from the root node, thus there is no need to store this information, in the case of QROBDDs.

In practical applications, we often need to store sets of non-binary L-tuples from a finite domain, that is, subsets of $\mathcal{X}_{pot} = \mathcal{X}_L \times \cdots \times \mathcal{X}_1$, where each *local index set* $\mathcal{X}_k$, for $L \geq k \geq 1$, is of the form $\{0, 1, ..., n_k - 1\}$. This natural extension from a binary to a *multi-way* choice at each nonterminal level results in the QROMDDs and FROMDDs, whose definition is left as an exercise. Fig. 2.1 shows how the decision diagram encoding the set of tuples

$$\{1000, 1010, 1100, 1110, 1210, 2000, 2010, 2100, 2110, 2210, 3010, 3110, 3200, 3201, 3202, 3210, 3211, 3212\}$$

looks in four different canonical representations:

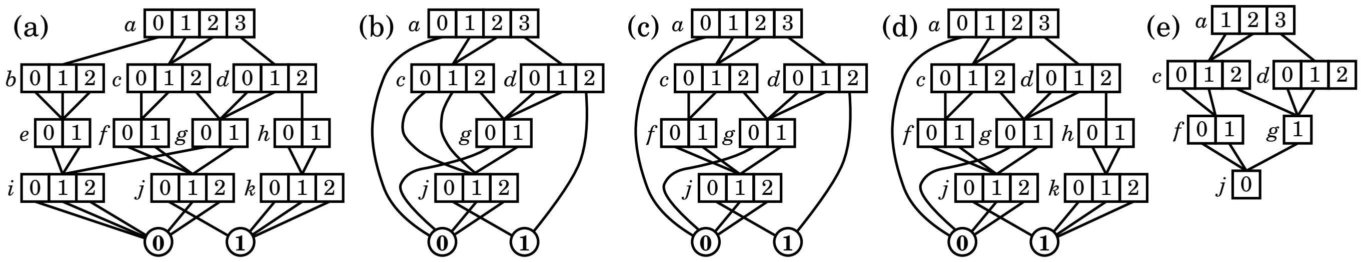


Figure 2.1: An example of multi-way decision diagram.

(a) is a QROMDD.

(b) is a FROMDD (redundant nodes b, e, f, h, i, and k have been removed).

(c) is a variant of QROMDD where "long edges" pointing to terminal nodes are allowed (thus redundant nodes b, e, and i, all encoding the empty set, and nodes h and k, encoding the full set, have been removed, but redundant node f is still present).

(d) is a variant of QROMDD where only "long edges" pointing to the terminal node **0** are allowed.

Finally, Fig. 2.1(e), shows how a different, more compact, graphical representation can be used: nodes are represented in "sparse format", by omitting local indices corresponding to edges that point to node **0** (as is the case for $a[0]$), and showing the local indices corresponding to edges that point to node **1**, but omitting these edges, as well as the terminal node **1** itself (as is the case for $d[2]$). Such a graphical representation can be used for the canonical forms shown in (b), (c), and (d); in particular, (e) is the compact graphical representation of (c).

2.12 Project idea: sequence diagrams

Almost all classes of decision diagrams published in the literature focus on the efficient encoding and manipulation of finite, but enormous, sets of strings (tuples) of a *fixed* length L.

We aim at extending this idea to allow the encoding of finites sets $\mathcal{L}$ of strings of *arbitrary* but finite length. If the longest string in $\mathcal{L}$ is of length L, one way (although maybe not the best) to achieve this goal is by "padding" each string of length $L' < L$ with $L - L'$ occurrences of a special character. However, even this simple approach is still not so obvious: do we pad at the beginning? at the end? anywhere along the path?

The paramount goal is of course to have a *memory-efficient* encoding, but having a *canonical* encoding is a close second, since canonicity is usually essential to have *time-efficient* manipulation algorithms (e.g., being able to efficiently compute operations such as union, intersection, or concatenation of languages encoded as sequence diagrams).

Chapter 3

Context-free languages

Context-free languages are the next step up in the hierarchy of languages.

Recall the definition of a grammar. A grammar G is a tuple $G = (V, T, S, P)$, where

- V is an alphabet, called the *variables*, or *nonterminals*.
- T is an alphabet, $V \cap T = \emptyset$, called the *terminal symbols* or *terminals*.
- $S \in V$ is the *start* variable.
- $P \subseteq ((V \cup T)^* \setminus T^*) \times (V \cup T)^*$ is a set of *productions*. The key property is that there is at least one variable on the left-hand-side of a production.

We say that uxv derives uyv, written $uxv \Longrightarrow uyv$, if P contains a production $x{\to}y$.

As usual we indicate the star and positive closures with $\overset{*}{\Longrightarrow}$ and $\overset{+}{\Longrightarrow}$, respectively.

The *language generated* by $G = (V, T, S, P)$ is the set of strings $L(G) = \{w \in T^* : S \overset{*}{\Longrightarrow} w\}$.

3.1 Context-free grammars

A context-free grammar (CFG) is a grammar $G = (V, T, S, P)$ where $P \subseteq V \times (V \cup T)^*$ or, in other words, the productions are of the form

$$X{\to}y, \quad X \in V, \ \ y \in (V \cup T)^*.$$

Then, $u \underset{G}{\overset{1}{\Longrightarrow}} v$ iff $\exists x, y, z \in (V \cup T)^*, \exists A \in V$, $\exists A{\to}y \in P$, $u = xAz \wedge v = xyz$ such that $xAz \overset{A\to y}{\Longrightarrow} xyZ$.

The language generated by G is $L(G) = \{w \in T^* : S \underset{G}{\overset{*}{\Longrightarrow}} w\}$.

Definition L is a context-free language (CFL) iff it is generated by a CFG G: $L = L(G)$.

Observation We call CFGs "context-free" because a production can be applied independent of its context: whenever A appears in a sentential form, any production $A{\to}x$ can be applied. This is different from, for example, the English grammar:

<pronoun> <verb> $\implies$ I <verb> $\implies$ I are

<verb> can go to “are”, or <pronoun> can go to “I”, but only in the right context.

Theorem Every regular language is a CFL.

Proof Every regular language can be expressed as a right linear grammar, which is, by definition, a special case of CFG. □

Theorem There are CFLs that are not regular.

Proof We know that $L = \{a^n b^n : n \in \mathbb{N}\}$ is not regular. But it is easy to see that $G = (\{S\}, \{a, b\}, S, \{S \to aSb, S \to \epsilon\}$ is a CFG such that $L(G) = L$. □

Hence, we just extended the set of languages we know.

Example Consider the grammar $G = (\{S\}, \{a, b\}, S, \{S \to aSb, S \to SS, S \to \epsilon\})$.

$$L(G) = \{w \in \{a, b\}^* : |w|_a = |w|_b \wedge \forall x, y, xy = w, |x|_a \geq |x|_b\}$$

If we think of a as open parenthesis, “(”, and of b as close parenthesis, ”)”, $L(G)$ is the language of balanced parentheses: at no point we can close a parenthesis which is not yet open. Clearly, this language is related to programming languages.

Example A CFG to generate $\{ww^R : w \in \{a, b\}^*\}$ is:

$$(\{S\}, \{a, b\}, S, \{S \to aSa | bSb | \epsilon\}).$$

For example, $S \implies aSa \implies aaSaa \implies aabSbaa \implies aabbaa$.

Example We know how to generate $L = \{a^n b^m : n = m\}$: we simply need the rules $S \to aSb$ and $S \to \epsilon$. How do we generate $L' = \{a^n b^m : n \neq m\}$? Consider that $L' = a^+ L \cup L b^+$. Since a^+ and b^+ are regular expressions, we can find right (or left) linear grammars for them:

$$G_a = (\{X\}, \{a\}, X, \{X \to aX, X \to a\})$$

and

$$G_b = (\{Y\}, \{b\}, Y, \{Y \to bY, Y \to b\})$$

Then, we can form the union and concatenation of a finite number of CFLs very simply:

$$G' = (\{S', S, X, Y\}, \{a, b\}, S', \{S' \to XS, S' \to SY, S \to aSb, S \to SS, S \to \epsilon, X \to aX, X \to a\})$$

We can then immediately state an important closure theorem for CFLs:

Theorem CFLs are closed under union, concatenation, and Kleene star.

Proof Given two languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$, where $G_1 = (V_1, T_1, S_1, P_1)$ and $G_2 = (V_2, T_2, S_2, P_2)$, and $V_1 \cap V_2 = \emptyset$ and $S \notin V_1 \cup V_2$:

- $L_1 \cup L_2$ is generated by the grammar $(\{S\} \cup V_1 \cup V_2, T_1 \cup T_2, S, P_1 \cup P_2 \cup \{S \to S_1 | S_2\})$
- $L_1 \cdot L_2$ is generated by the grammar $(\{S\} \cup V_1 \cup V_2, T_1 \cup T_2, S, P_1 \cup P_2 \cup \{S \to S_1 S_2\})$
- L_1^* is generated by the grammar $(\{S\} \cup V_1, T_1, S, P_1 \cup \{S \to S_1 S | \epsilon\})$

In addition to right-linear and left-linear grammars, we can also define *linear* grammars, which are CFGs where the right-hand side of a production can contain at most one variable.

Thus, any sentential form will also contain exactly one variable, and a string in the language is generated as soon as we use a production containing *no* variables on the right-hand side.

The fundamental property of linear grammars is that, at each step of the derivation, we have no choice about *which* variable to expand (there is only one), although we might still have a choice about *how* to expand it (if there are multiple productions with that variable on the left-hand side).

3.2 Leftmost and rightmost derivations

Given a CFG $G = (V, T, S, P)$ and a string $w \in L(G)$, a leftmost derivation of w is

$$S = \alpha_0 \overset{L}{\underset{G}{\Longrightarrow}} \alpha_1 \overset{L}{\underset{G}{\Longrightarrow}} \alpha_2 \overset{L}{\underset{G}{\Longrightarrow}} \alpha_n = w$$

where each step is leftmost, that is:

$$\alpha_i = \beta A \gamma \overset{L}{\underset{G}{\Longrightarrow}} \beta x \gamma = \alpha_{i+1} \qquad \wedge \quad \beta \in T^* \quad \wedge \quad A \to x \in P \quad \wedge \quad \gamma \in (V \cup T)^*$$

Theorem $S \overset{*}{\underset{G}{\Longrightarrow}} w$ if and only if $S \overset{*\ L}{\underset{G}{\Longrightarrow}} w$.

Proof By induction on the position where the first non leftmost step occurs in the derivation. Informally, if the first k steps are leftmost and step $k+1$ is not:

$$S \overset{k\ L}{\Longrightarrow} \alpha A \beta B \gamma \overset{(B \to \delta)}{\Longrightarrow} \alpha A \beta \delta \gamma \overset{i}{\Longrightarrow} \alpha A \sigma \overset{(A \to \epsilon)}{\Longrightarrow} \alpha \epsilon \sigma \overset{j}{\Longrightarrow} w$$

we can reorder the steps so that we ensure that (at least) the first $k+1$ steps are leftmost:

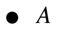
$$S \overset{k\ L}{\Longrightarrow} \alpha A \beta B \gamma \overset{(A \to \epsilon)}{\Longrightarrow} \alpha \epsilon \beta B \gamma \overset{(B \to \delta)}{\Longrightarrow} \alpha \epsilon \beta \delta \gamma \overset{i}{\Longrightarrow} \alpha \epsilon \sigma \overset{j}{\Longrightarrow} w. \qquad \square$$

Analogously, we can talk about rightmost derivations.

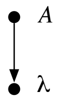
3.3 Parse trees

Given a CFG $G = (V, T, S, P)$, a parse tree for this grammar is defined recursively as follows:

- $\forall A \in V \cup T$, a single node labeled A is a parse tree with root A and yield A

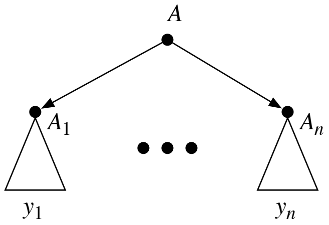


- $\forall A \to \epsilon \in P$,



is a parse tree with root A and yield ϵ

- $\forall A \to A_1 \cdots A_n \in P$, $\forall$ parse trees with root $A_1, \ldots A_n$ and yield $y_1, \ldots y_n$,

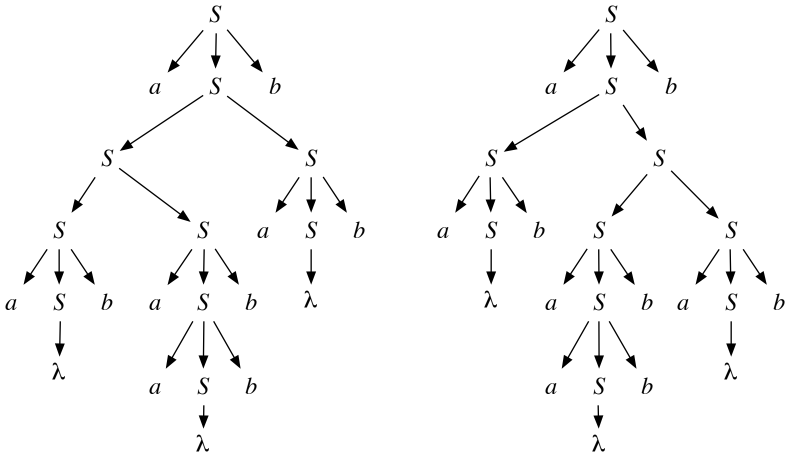


is a parse tree with root A and yield $y_1 \cdots y_n$ (the order of the subtrees is important!).

- Nothing else is a parse tree.

In particular, a parse tree with root S and yield $w \in T^*$ is called a *derivation tree* for the string $w \in L(G)$.

Example Let the productions be $P = \{S \to aSb|SS|\epsilon\}$. Here are two possible derivation trees for the string *aabaabbabb*.



A derivation tree is less specific (less constrictive) than a derivation:

- Given a derivation there is a unique derivation tree for it.

- Given a derivation tree there is at least one derivation for it.

3.4 Ambiguous grammars

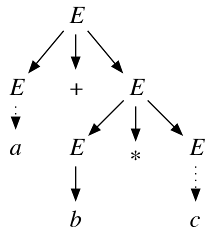
Definition A CFG G is ambiguous iff there is a $w \in L(G)$ such that there are two distinct derivation trees for it.

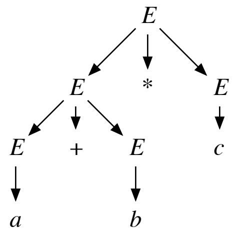
Definition A CFL L is (inherently) ambiguous iff any CFG G such that $L = L(G)$ is ambiguous.

For example, consider the language of arithmetic expressions over the variables $\{a, b, c\}$:

$$G = (\{E\}, \{(,), *, +, a, b, c\}, E, \{E \to a|b|c|E + E|E * E|(E)\})$$

The string $a + b * c$ has two derivation trees:





The one on the left corresponds to the interpretation $a+(b*c)$, while the one on the right corresponds to the interpretation $(a + b) * c$.

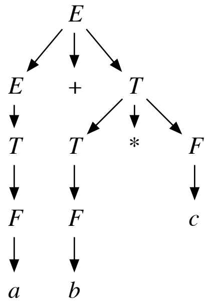
The fact that we normally assume the former meaning is due to an arbitrarily agreed upon precedence of multiplication over addition, but this is not captured by the grammar.

Hence this grammar is ambiguous.

However, we can define a non-ambiguous grammar for the same language:

$$G' = (\{E, T, F\}, \{(,), *, +, a, b, c\}, E, \{E{\rightarrow}T|E + T, T{\rightarrow}F|T * F, F{\rightarrow}(E)|a|b|c\}).$$

Strictly speaking, we should prove that G' is non-ambiguous. It is easy to see, however, that string $a + b * c$ has a unique derivation tree:



Hence, the initial grammar G was ambiguous, but, since the same language can be generated by a nonambiguous grammar, the language itself is not inherently ambiguous.

We do not prove that inherently ambiguous languages exist, but they do. For example, $\{a^i b^j c^k : i, j, k \in \mathbb{N}, i = j \vee j = k\}$ is inherently ambiguous.

3.5 Determining whether $w \in L(G)$

Given a CFG $G = (V, T, S, P)$, we would like to answer two important questions:

1. Can the string w be generated by G? $\quad w \in L(G)$?
2. If the answer to the previous question is affirmative, that is, if $w \in L(G)$, can we determine a derivation for w?

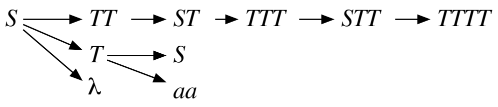
The answer for the second question is trivially true: if we know that $w \in L(G)$, we can enumerate all derivations of length one, two, three, etc. Eventually, we will find a derivation for w, hence the algorithm will halt (this exhaustive search is very inefficient, but let's not worry about that for now).

However, if $w \notin L(G)$, an exhaustive search does not terminate, so this is not an algorithm to determine whether $w \in L(G)$.

In particular, certain types of productions cause trouble. Consider for example the grammar with productions

$$S \rightarrow T|TT|\epsilon \qquad T \rightarrow aa|S$$

If we ask whether aaa can be generated by this grammar, the answer is clearly negative, because no odd-length string of a's can be generated. However, an exhaustive search would not terminate:



This exemplifies some of the problems:

- The computation along the derivation reaching the sentential form $TTTT$ cannot be halted, because any of those T's can "disappear", due to the ϵ productions for S: $T \Longrightarrow S \Longrightarrow \epsilon$. Hence, a sentential form of length n is not guaranteed to generate only strings of length n or greater.
- The computation along the path $S \Longrightarrow T \Longrightarrow S \Longrightarrow T \cdots$ is clearly not going to help us going to any string in the language, but detecting this situation is not trivial.

Theorem If a CFG $G = (V, T, S, P)$ does not contain UNIT PRODUCTIONS (of the form $A \rightarrow B$) and ϵ-PRODUCTIONS (of the form $A \rightarrow \epsilon$), then exhaustive search is an algorithm, that is, it can be made to stop: if $w \in L(G)$, a derivation for w will be found; if $w \notin L(G)$, we can determine this fact after having explored all derivations to a depth of $2 \cdot |w| - 1$ steps (each derivation either increases the length of the sentential form by at least one symbol, or it transforms a variable into at least one terminal, hence $2 \cdot |w| - 1$ is the worst case number of steps required to generate w).

3.6 Grammar transformations

We have seen how unit productions and ϵ-productions are undesirable. We now show how to eliminate them.

Note. Of course, if $\epsilon \in L(G)$, we cannot eliminate all ϵ productions, since we might need to be able to generate ϵ. However, we can treat this as a special case, using a new starting symbol S' and productions $S' \rightarrow S|\epsilon$. Then, S only needs to generate non-empty strings. Hence, in this section, we assume that $\epsilon \notin L(G)$.

3.6.1 Eliminating left-recursion

First, we need to learn how to transform a grammar so that it does not contain left-recursive productions, that is, productions of the form $A \to Ax$, with $x \in (V \cup T)^*$. Of course, we can simply delete productions $A \to A$, since this does not change the language generated by the grammar.

Given a CFG $G = (V, T, S, P)$, partition the rules for a given variable A into:

(1) $A \to Ax_1 | \cdots | Ax_n$

(2) $A \to y_1 | \cdots | y_m$, where $y_1, \ldots, y_m$ do not start with A.

Then, obviously, we can say that any and only strings w of the form $(y_1 + \cdots + y_m)(x_1 + \cdots + x_n)^*$ can be derived by A. We can then transform G into G' so that A derives the same strings, but without left-recursion: $G' = (V \cup \{Z\}, T, S, P')$, with $Z \notin V$ and P' is obtained by eliminating productions of type (1) and (2) from P and adding these productions instead:

(3) $A \to y_1 | \cdots | y_m | y_1 Z | \cdots | y_m Z$

(4) $Z \to x_1 | \cdots | x_n | x_1 Z | \cdots | x_n Z$

3.6.2 Eliminating useless variables

Definition Given a CFG $G = (V, T, S, P)$, a variable $A \in V$ is **useful** iff $S \overset{*}{\Longrightarrow} xAy \overset{*}{\Longrightarrow} w = xuy \in T^*$. It is useless otherwise.

Note Do not confuse "useless" with "redundant". We might still be able to remove a useful variable and not change the language generated by G, but not in all cases.

There are two reasons for being useless:

1. A cannot derive a string: $\nexists u \in T^*,\ A \overset{*}{\Longrightarrow} u$.
2. A cannot be the only variable in a sentential form derived from S: $\nexists x, y \in T^*,\ S \overset{*}{\Longrightarrow} xAy$.

We can eliminate all variables of the first type using an iterative algorithm:

```
$V' \leftarrow \emptyset$;
repeat
  $V' \leftarrow V' \cup \{A \in V : \exists u \in (T \cup V')^*,\ A \to u \in P\}$;
until "$V'$ does not change";
```

If $S \notin V'$, then $L(G) = \emptyset$, otherwise, we now eliminate all variables left in V' of the second type, with a second step, where we keep only the variables "reachable from S":

```
$V'' \leftarrow S$;
repeat
  $V'' \leftarrow V'' \cup \{A \in V' : \exists B \in V'', \exists x, y \in (T \cup V')^*,\ B \to xAy \in P\}$;
until "$V''$ does not change";
```

Finally, we can simply set P'' to be all productions in P except those containing variables not in V''.

The simplified grammar is then (V'', T, S, P'').

3.6.3 Eliminating ϵ-productions

We can now remove all productions of the form $A \to \epsilon$ from a grammar $G = (V, T, S, P)$.

Definition A variable $A \in V$ is **nullable** iff $A \overset{*}{\Longrightarrow} \epsilon$.

We can find the set V_{null} of nullable variables as follows:

```
$V_{null} \leftarrow \{A \in V : A \to \epsilon \in P\}$;
repeat
  $V_{null} \leftarrow V_{null} \cup \{A \in V : \exists A_1, \ldots, A_n \in V_{null},\ A \to A_1 \cdots A_n \in P\}$;
until "$V_{null}$ does not change";
```

Then, we can eliminate the null productions from P, provided that, for any production $A \to x_1 \cdots x_m \in P, m \geq 1, x_i \in V \cup T$, we add all the productions obtained from it by eliminating any combination of nullable variables (except that, if each x_i is nullable, we do not eliminate all of them, since this would result in the production $A \to \epsilon$.

For example, given the production $A \to aBbCcDd$, where B and C are nullable, we will have to have the following productions in the final set:
$A \to aBbCcDd$
$A \to abCcDd$
$A \to aBbcDd$
$A \to abcDd$
Note that, if, for example, C can only produce the null string, it will become useless, hence the first two productions will be removed if we simplify the grammar.

3.6.4 Eliminating unit-productions

We can now remove productions of the form $A \to B$ from a grammar $G = (V, T, S, P)$. We only need to be concerned with the case $A \neq B$, since the production $A \to A$ can always be removed without changing the language.

First, we need to find, for any $A \in V$, the set of variables V_A obtainable from A through unit productions:

$$V_A = \{B \in V \setminus \{A\} : A \overset{+}{\Longrightarrow} B\}$$

(we can do this using a simple iterative algorithm).

Then, we can build a new grammar $G' = (V, T, S, P')$ where:

1. $\forall A \to x \in P, x \notin V$, add $A \to x$ to P'.
2. $\forall A, \forall B \in V_A, \forall B \to x \in P'$, add $A \to x$ to P'.

In conclusion, given a CFG G such that $\epsilon \notin L(G)$, we now know how to obtain an equivalent CFG G' that

- does not have useless variables, terminals, or productions.
- does not have ϵ- or unit-productions.

(and remember that, if $\epsilon \in L(G)$, we can simply add the production $S \to \epsilon$ as a special case.

3.6.5 Normal forms

At times it is important to know that the productions of a CFG are of a particular form. This will be often useful later when proving certain theorems about CFGs.

We simply state the two most important **normal forms**, without proving that they are always achievable (the proof is straightforward).

Theorem (CNF — Chomsky Normal Form) Given a a CFG $G = (V, T, S, P)$ such that $\epsilon \notin L(G)$ it is always possible to transform it in an equivalent grammar $G' = (V', T, S, P')$ whose productions are of the form
$A \to a$, $a \in T$ or
$A \to BC$, $B, C \in V'$

This implies that a string $w \in L(G)$ requires exactly $2|w| - 1$ steps for its derivation.

Theorem (GNF — Greibach Normal Form) Given a a CFG $G = (V, T, S, P)$ such that $\epsilon \notin L(G)$ it is always possible to transform it in an equivalent grammar $G' = (V', T, S, P')$ whose productions are of the form
$A \to ax$, $a \in T$, $x \in V'^*$

This implies that a string $w \in L(G)$ requires exactly $|w|$ steps for its derivation.

3.7 The CYK algorithm

We have already seen an algorithm that answers the question $w \in L(G)$ for a CFG G: exhaustive search applied to G (after we eliminate ϵ-productions and unit-productions). However, exhaustive search has exponential complexity.

We now present an algorithm that has $O(|w|^3)$ complexity.

Assume that G is in Chomsky Normal Form, that is, all productions are of the type $A \to a$ or $A \to BC$.

Define a_i to be the i-th symbol in w, that is, $w = a_1 a_2 \cdots a_n$.

Define $w_{i,j}$, for $i \geq j$, to be the substring of w starting at position i and ending at position j: $w_{i,j} = a_i a_{i+1} \cdots a_j$.

Hence, $w = w_{1,n}$.

Define $V_{i,j}$ to be the set of variables that can derive $w_{i,j}$:

$$V_{i,j} = \{A \in V : A \overset{*}{\Longrightarrow} w_{i,j}\}$$

We are going to build these sets iteratively:

```
for $i = 1$ to $n$ do
  $V_{i,i} \leftarrow \{A \in V : A \to a_i \in P\}$;
for $l = 1$ to $n - 1$ do
  for $i = 1$ to $n - l$ do
    $V_{i,i+l} \leftarrow \{A \in V : A \to BC \in P \wedge \exists k, i \leq k < i + l, B \in V_{i,k}, C \in V_{k+1,l}\}$;
```

We are really only interested in $V_{1,n}$, but, to obtain it, we build $n(n+1)/2$ sets:

$$V_{1,1},\ldots,V_{n,n},V_{1,2},\ldots,V_{n-1,n},\ldots,V_{1,n-1},V_{2,n},V_{n,n}$$

To compute set $V_{i,j}$, we must examine all the possible k between i (included) and j (excluded), hence the $O(|w|^3)$ complexity. Note that we also need to search whether $A{\to}BC$ is a legal production, but this has a constant cost for each given grammar.

Then, $w \in L(G) \Leftrightarrow S \in V_{1,n}$, where S is the starting symbol of the grammar.

3.8 Nondeterministic pushdown automata

Why can't FA recognize $\{a^n b^n : n \in \mathbb{N}\}$?

Because they cannot "count" up to an arbitrarily large n.

How does a CFG generate $\{a^n b^n : n \in \mathbb{N}\}$?

By "remembering" to generate a b every time an a is generated. If we consider the Greibach Normal Form grammar for this language, it has the production

$$S{\to}aSB|\epsilon, B{\to}b$$

and a leftmost derivation in this grammar looks like:

$$S \Longrightarrow aSB \Longrightarrow aaSBB \Longrightarrow aaaSBBB\cdots$$

A nondeterministic pushdown automaton (NPDA) is the machine we need to accept a CFL following the same idea. It can be thought of as a NFA with the addition of one data structure: a stack.

Definition An NPDA is an automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ where:

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $\delta : (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \to \{S \subseteq Q^* : |S| < \infty\}$ (i.e., finite subsets of $Q \times \Gamma^*$) is the transition function
- $q_0 \in Q$ is the initial state
- $z \in \Gamma$ is the initial content of the stack
- $F \subseteq Q$ is a set of final states

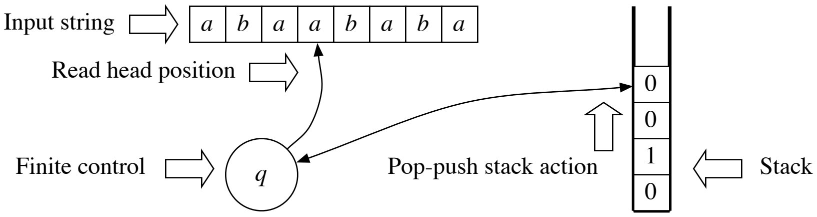
If $(p, \beta_1 \cdots \beta_n) \in \delta(q, a, \alpha)$, with $q, p \in Q$, $a \in \Sigma \cup \{\epsilon\}$, and $\beta_1 \cdots \beta_n \in \Gamma^*$, it means that:

if M is in state q, the next input symbol is a (if $a \in \Sigma$) or regardless of the next input symbol (if $a = \epsilon$), and the symbol on the top of the stack is α

then, M might move to state p, consume the input symbol a (if $a \in \Sigma$), pop the top of the stack (thus removing α), and push $\beta_n, \beta_{n-1}, \ldots, \beta_1$ on the top of the stack in the order (thus leaving β_1 on the top of the stack).

We then write

$$(q, ax, \alpha\gamma) \vdash (p, x, \beta_1 \cdots \beta_n \gamma)$$



The language accepted by a NPDA is defined without regard to the contents of the stack, as long as the entire input string is consumed and (one of the clones of) M is in a final state after doing so:

$$L(M) = \{w \in \Sigma^* : \exists p \in F, \exists u \in \Gamma^*, (q_0, w, z) \overset{*}{\vdash} (p, \epsilon, u)\}.$$

Note that, if the stack becomes empty at any point before the entire input string is read, the computation halts, since the δ function is defined only when the top of the stack is a symbol in Γ.

However, it is common to accept with an empty stack, that is, often the last move of the NPDA removes the last symbol on the stack. Indeed, there are even definitions of NPDA where this must happen. The two definitions are equivalent, that is, if an NPDA M defined as we did accepts L, then we can always find an NPDA M' that accepts L and leaves an empty stack at the end.

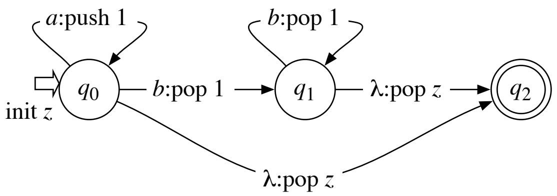
Example An NPDA for the language $\{a^n b^n : n \in \mathbb{N}\}$ is:

$$M = (\{q_0, q_1, q_2\}, \{a, b\}, \{0, 1\}, \delta, q_0, 0, \{q_2\})$$

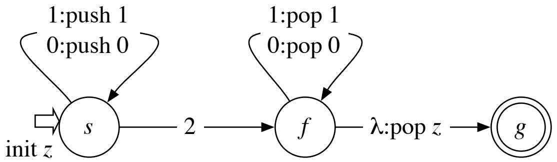
where:

- $\delta(q_0, \epsilon, 0) = \{(q_2, \epsilon)\}$: needed to accept ϵ
- $\delta(q_0, a, 0) = \{(q_0, 10)\}$: start counting a's
- $\delta(q_0, a, 1) = \{(q_0, 11)\}$: keep counting a's
- $\delta(q_0, b, 1) = \{(q_1, \epsilon)\}$: start matching b's
- $\delta(q_1, b, 1) = \{(q_1, \epsilon)\}$: keep matching b's
- $\delta(q_1, \epsilon, 0) = \{(q_2, \epsilon)\}$: stop

We will use a graphical representation for NPDA, analogous to that of FA with the addition of the operations to manipulate the stack. The initial state must be labeled with "init z", stating that z is the initial symbol on the stack, and the transition corresponding to $(p, \beta_1 \cdots \beta_n) \in \delta(q, a, \alpha)$ is labeled "a: pop α push $\beta_1 \cdots \beta_n$". However, if $\alpha = \beta_n$, we simply write "a: push $\beta_1 \cdots \beta_{n-1}$"; if, in addition, $n = 1$, we simply write "a". Instead, if $n = 0$, we simply write "a: pop α". Hence, in our example, we obtain:

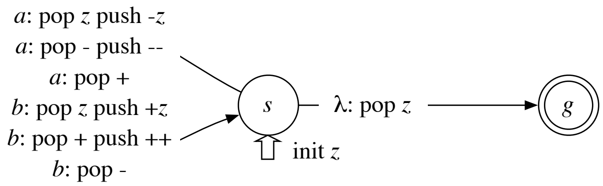


Example An NPDA for the language $\{w2w^R : w \in \{0,1\}^*\}$ is:



Note that, to recognize the language $\{ww^R : w \in \{0,1\}^*\}$, we simply need to change the label "2" of the transition from s to f into "ϵ". This, however, introduces nondeterminism, since now, at every step before reading the entire string and while still in state s, the NPDA can either remain in state s or move to state f. We stress that is due to the fact that the NPDA cannot determine where "the middle of the string" is.

Example A NPDA for the language $\{w \in \{a,b\}^* : |w|_a = |w|_b\}$ is:



Note that, effectively, the entire processing of w occurs in state s.

3.9 Equivalence of CFGs and NPDA

We are now ready to prove that CFGs and NPDA are equivalent.

Theorem Let G be a context-free grammar, then there is an NPDA M such that $L(G) = L(M)$.

Proof Without loss of generality, we can assume that $G = (V, T, S, P)$ is in Greibach Normal Form, (its productions are of the form $A \rightarrow au$ with $a \in T$ and $u \in V^*$, and, in addition, if $\epsilon \in L(G)$, P also contains the production $S \rightarrow \epsilon$). Then, $M = (\{q_0, q_1, q_f\}, T, V \cup \{z\}, \delta, q_0, z, \{q_f\})$, where

- $\{q_0, q_1, q_f\}$ is the set of states (independent of G!!!)
- T is the input alphabet
- $V \cup \{z\}$ is the stack alphabet ($z \notin V$)
- δ is the transition relation, built as follows:
    - initialize δ to an empty set

- add (q_1, Sz) to $\delta(q_0, \epsilon, z)$
- for each rule $A \rightarrow au$ in P, add (q_1, u) to $\delta(q_1, a, A)$
- add (q_f, ϵ) to $\delta(q_1, \epsilon, z)$

- q_0 is the initial state
- z is the initial stack symbol
- $\{q_f\}$ is the set of final states.

We need to prove that $S \overset{*\ L}{\underset{G}{\Longrightarrow}} \alpha \in T^*$ iff $(q_0, \alpha, z) \overset{*}{\underset{M}{\vdash}} (q_f, \epsilon, \epsilon)$, where "$L$" indicates a leftmost derivation.

We can then prove by induction on the number of steps k that

$$\forall x \in V^*,\ S \overset{k\ L}{\Longrightarrow} a_1 a_2 \cdots a_k x \ \Leftrightarrow\ (q_1, a_1 a_2 \cdots a_k, Sz) \overset{k}{\vdash} (q_1, \epsilon, xz).$$

This also applies to the case where $x = \epsilon$, hence $w \in L(G) \Leftrightarrow S \overset{*\ L}{\Longrightarrow} w \Leftrightarrow (q_1, w, Sz) \overset{*}{\vdash} (q_1, \epsilon, z)$. Adding to this the fact that $(q_0, w, z) \vdash (q_1, w, Sz)$ and that $(q_1, \epsilon, z) \vdash (q_f, \epsilon, \epsilon)$, we conclude that $w \in L(G) \Leftrightarrow w \in L(M)$.

Now for the proof by induction:

Basis: $S \overset{0\ L}{\Longrightarrow} x \ \Leftrightarrow\ x = S$, that is $S \overset{0\ L}{\Longrightarrow} S$. Since $(q_1, \epsilon, Sz) \overset{0}{\vdash} (q_1, \epsilon, Sz)$, the basis holds.

Inductive hypothesis: Assume that, for any $i \leq k$,

$$S \overset{i\ L}{\Longrightarrow} a_1 a_2 \cdots a_i x \ \Leftrightarrow\ (q_1, a_1 a_2 \cdots a_i, Sz) \overset{i}{\vdash} (q_1, \epsilon, xz).$$

Inductive step: We now need to show that

$$S \overset{k+1\ L}{\Longrightarrow} a_1 a_2 \cdots a_{k+1} x \ \Leftrightarrow\ (q_1, a_1 a_2 \cdots a_{k+1}, Sz) \overset{k+1}{\vdash} (q_1, \epsilon, xz).$$

Consider the last leftmost step in $S \overset{k+1\ L}{\Longrightarrow} a_1 a_2 \cdots a_{k+1} x$:

$$S \overset{k\ L}{\Longrightarrow} a_1 a_2 \cdots a_k C_1 C_2 \cdots C_{m'} \overset{1\ L}{\Longrightarrow} a_1 a_2 \cdots a_k a_{k+1} \underbrace{B_1 \cdots B_{m''} C_2 \cdots C_{m'}}_{x}$$

Then, $C_1 \rightarrow a_{k+1} B_1 \cdots B_{m''} \in P$ and $(q_1, B_1 \cdots B_{m''}) \in \delta(q_1, a_{k+1}, C_1)$.

But, by inductive hypothesis,

$$S \overset{k\ L}{\Longrightarrow} a_1 a_2 \cdots a_k C_1 C_2 \cdots C_{m'} \ \Leftrightarrow\ (q_1, a_1 a_2 \cdots a_k, Sz) \overset{k}{\vdash} (q_1, \epsilon, C_1 C_2 \cdots C_{m'} z)$$

which implies $(q_1, a_1 a_2 \cdots a_k a_{k+1}, Sz) \overset{k}{\vdash} (q_1, a_{k+1}, C_1 C_2 \cdots C_{m'} z)$.

Applying the transition $(q_1, B_1 \cdots B_{m''}) \in \delta(q_1, a_{k+1}, C_1)$, we obtain

$(q_1, a_1 a_2 \cdots a_k a_{k+1}, Sz) \overset{k}{\vdash} (q_1, a_{k+1}, C_1 C_2 \cdots C_{m'} z) \overset{1}{\vdash} (q_1, \epsilon, B_1 \cdots B_{m''} C_2 \cdots C_{m'} z) \equiv (q_1, \epsilon, \alpha)$. □

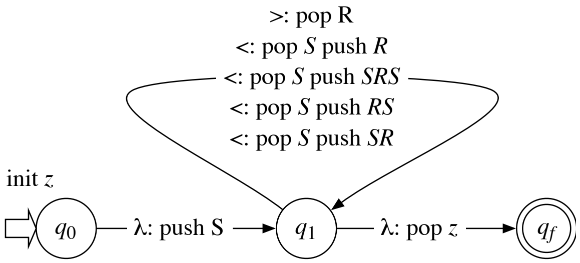
Example Consider the grammar of non-empty balanced "<" and ">", generated by

$$G = (\{S\}, \{<, >\}, S, \{S \rightarrow <> \mid < S > \mid SS\})$$

The equivalent Greibach Normal Form is:

$$G = (\{S, R\}, \{<, >\}, S, \{S \to\ < R \mid\ < SR \mid\ < SRS \mid\ < RS, R \to\ >\})$$

and the resulting NPDA is



In the following, we will find easy to use NPDA in a certain normal form:

Theorem Given a NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$, there exists a NPDA $M' = (Q', \Sigma, \Gamma, \delta', q_0, z, F')$ such that $L(M) = L(M')$ and

- M' has a single final state: $F' = \{q_f\}$.
- z is removed from the bottom of the stack only when the stack becomes empty (and, of course, no more moves are possible after that).
- q_f is entered only when the stack becomes empty.
- All transitions either decrease or increase the height of the stack by one, that is, the transition function contains only moves of the form $(p, \epsilon) \in \delta(q, a, A)$ and $(p, BC) \in \delta(q, a, A)$.

Proof Left as an exercise.

To complete the proof of equivalence between CFGs and NPDA, we need a theorem in the other direction:

Theorem Given a NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$, there exists a CFG $G = (V, \Sigma, S, P)$ such that $L(M) = L(G)$.

Proof We can assume that the NPDA satisfies the normal form of the previous theorem. Then:

- $V = \{\boxed{q_iAq_j} : q_i, q_j \in Q \wedge A \in \Gamma\}$ (this defines $|Q|^2 \cdot |\Gamma|$ variables; some of them might be useless, but this is not a problem).
- $S = \boxed{q_0zq_f}$ (the starting symbol of the grammar encodes the fact that, starting from state z, the NPDA must read a string that will make it remove z and reach state q_f in doing so).
- The productions of G are defined as follows:
  - (POP) $\forall (q_j, \epsilon) \in \delta(q_i, a, A)$, add $\boxed{q_iAq_j} \to a$ to P.
  - (PUSH) $\forall (q_{k_1}, BC) \in \delta(q_i, a, A), \quad \forall q_{k_2}, q_j \in Q, \quad$ add $\boxed{q_iAq_j} \to a \boxed{q_{k_1}Bq_{k_2}}\boxed{q_{k_2}Cq_j}$ to P.

Intuition: the variable $\boxed{q_iAq_j}$ must be able to generate all the strings that make the NPDA go from state q_i to state q_j while removing A (and all other symbols possibly pushed on the stack in the process of removing A). The meaning of the productions that simulate a pop is then clear; to understand the productions for a push, just think that, if the the NPDA in in state q_i with an A on the top of the stack and an a in input, it can read a, pop A, push BC, and go to state q_{k_1}; but the effect of removing A, that is, the presence of BC on the stack, must also be removed from the stack; after a while B and its effect will be removed, but only by reading a string that makes the NPDA go from q_{k_1} to some q_{k_2} (hence the $\boxed{q_{k_1}Bq_{k_2}}$), and, after another while, C and its effect will be removed, but only by reading a string that makes the NPDA go from that state q_{k_2} to state q_j (hence the $\boxed{q_{k_2}Cq_j}$).

We must then show that

$$\forall q_i, q_j \in Q, \forall u \in \Sigma^*, \forall A \in \Gamma, \forall X \in \Gamma^*,\ (q_i, u, AX) \overset{*}{\underset{M}{\vdash}} (q_j, \epsilon, X) \ \Leftrightarrow\ \boxed{q_iAq_j} \overset{*}{\Longrightarrow} u$$

Hence, this will also prove, in particular, that,

$$(q_0, u, z) \overset{*}{\underset{M}{\vdash}} (q_f, \epsilon, \epsilon) \ \Leftrightarrow\ \boxed{q_0zq_f} \overset{*}{\Longrightarrow} u \quad \text{that is,} \quad u \in L(M) \ \Leftrightarrow\ u \in L(G)$$

We prove this first in one direction (NPDA $\Rightarrow$ CFG) by induction on the number of steps in the computation of the NPDA.

Basis: If $(q_i, u, AX) \overset{1}{\underset{M}{\vdash}} (q_j, \epsilon, X)$, the NPDA must have performed a pop, that is, $u \in \Sigma$ and $(q_j, \epsilon) \in \delta(q_i, u, A)$, hence $\boxed{q_iAq_j} \to u \in P$, and we can write $\boxed{q_iAq_j} \overset{1}{\underset{G}{\Longrightarrow}} u$.

Inductive Hypothesis: Assume that what we want to prove is true for computations up to length n.

Inductive step: We must consider the case $(q_i, u, AX) \overset{n+1}{\underset{M}{\vdash}} (q_j, \epsilon, X)$. Consider the first move: if $n > 0$, it must be a push, otherwise $(q_i, u, AX) \vdash (q, y, X)$ would hang when $X = \epsilon$, where $u = ay$. A push can then be formalized as:

$$(q_i, u, AX) \equiv (q_i, ay_1y_2, AX) \vdash (q_{k_1}, y_1y_2, BCX) \overset{n_1}{\vdash} (q_{k_2}, y_2, CX) \overset{n_2}{\vdash} (q_j, \epsilon, X),$$

with $|u| = n+1$, $|y_1| = n_1$, $|y_2| = n_2$, and, of course, $n_1 + n_2 = n$. Then

- $(q_{k_1}, BC) \in \delta(q_i, a, A)$, which implies $\boxed{q_iAq_j} \to a\boxed{q_{k_1}Bq_{k_2}}\boxed{q_{k_2}Cq_j}$
- $(q_{k_1}, y_1, BCX) \overset{n_1}{\vdash} (q_{k_2}, \epsilon, CX)$, which implies, by I.H., $\boxed{q_{k_1}Bq_{k_2}} \overset{*}{\Longrightarrow} y_1$
- $(q_{k_2}, y_2, CX) \overset{n_2}{\vdash} (q_j, \epsilon, X)$, which implies, by I.H., $\boxed{q_{k_2}Cq_j} \overset{*}{\Longrightarrow} y_2$

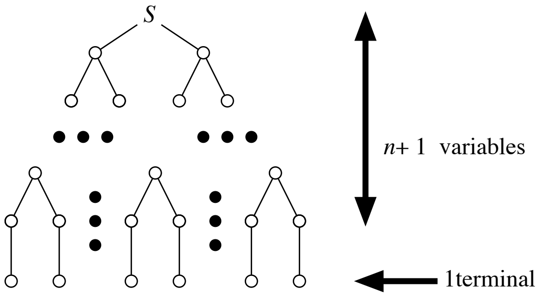
Hence, $\boxed{q_iAq_j} \Longrightarrow a\boxed{q_{k_1}Bq_{k_2}}\boxed{q_{k_2}Cq_j} \overset{n_1+n_2}{\Longrightarrow} ay_1y_2$

The proof in the other direction is analogous. □

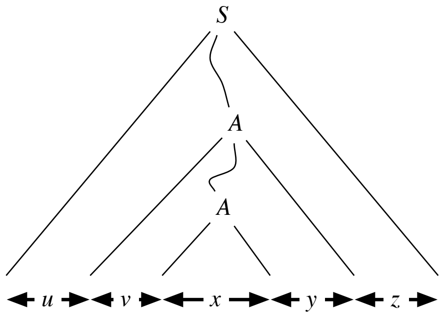
3.10 Pumping lemma for CFLs

Theorem Let L be a CFL. Then there exists an integer m such that any string $w \in L$ satisfying $|w| \geq m$ pumps, that is, it can be decomposed into $w = uvxyz$, with $|vy| > 0$ and $|vxy| \leq m$, and, for any $k \in \mathbb{N}$, $uv^k xy^k z \in L$.

Proof Assume that a CFG $G = (V, T, S, P)$ generating $L \setminus \epsilon$ is in Chomsky Normal Form. Then, any derivation tree for a string of length 2^n has at least one path of length $n+1$ (where $n+1$ nodes are variables and the last one is a terminal):



Hence, if we consider a string w of length at least $2^{|V|}$, its parse tree must contain at least one path where a variable A appears at least twice:



Then, $S \overset{*}{\Longrightarrow} uAz \overset{*}{\Longrightarrow} uvAyz \overset{*}{\Longrightarrow} uvxyz = w$, but this implies that $A \overset{*}{\Longrightarrow} vAy$ and $A \overset{*}{\Longrightarrow} x$ hence $A \overset{*}{\Longrightarrow} v^k xy^k$ and $S \overset{*}{\Longrightarrow} uv^k xy^k z$ for any $k \in \mathbb{N}$.

Why can we say that $vy \neq \epsilon$? Because a production of the form $A \rightarrow BC$ must be used for the first A, in order to generate a second A in the path. Neither B nor C derives just ϵ (because the grammar is in Chomsky Normal Form. At most, the second A could be the leftmost symbol produced by BC (in which case $v = \epsilon$) or the rightmost one (in which case $y = \epsilon$), but not both v and y can be ϵ.

Why can we say that $|vxy| \leq m = 2^{|V|}$? Because, if this is not true, the subtree under the first A must have other repeated variables on a single path in addition to the two A we know of. Then, this path is either in the left or the right subtree for the first A, and we can then apply the same reasoning to it, until we find a subtree with yield $|vxy| \leq m = 2^{|V|}$ rooted at a variable repeated in the subtree. □

Note Again, we stress that this pumping lemma can only be used to prove that a given language is NOT context-free, never to prove that it is context-free.

Example Prove that $L = \{a^n b^n c^n : n \in \mathbb{N}\}$ is not context free. By contradiction, if it were context-free, there would be an m such that any string $w \in L$, $|w| \geq m$ would pump. We can choose $w = a^m b^m c^m$ and show that it cannot be decomposed into $w = uvxyz$, $vy \neq \epsilon$, $|vxy| \leq m$, so that $uv^k xy^k z \in L$ for all $k \in \mathbb{N}$. This is simple, since the restriction $|vxy| \leq m$ implies that v and y can only span at most two out the three runs, that is, if they contain a's they cannot contain c's, and vice versa. Hence when we pump them, if they contain a's, the number of a's will change but that of c' won't, or vice versa. If they contain only b's, the same problem arises. Hence L cannot be context-free.

Example Prove that $L = \{ww : w \in \{a,b\}^*\}$ is not context free. By contradiction, if it were context-free, there would be an m such that any string $w \in L$, $|w| \geq m$ would pump. We can choose $w = a^m b^m a^m b^m$ and show that it cannot be decomposed into $w = uvxyz$, $vy \neq \epsilon$, $|vxy| \leq m$, so that $uv^k xy^k z \in L$ for all $k \in \mathbb{N}$. This is simple, since the restriction $|vxy| \leq m$ implies that v and y can only span at most two out the four runs. If they span only the first two runs, or the last two runs, pumping them zero times will result in a string $a^i b^j a^m b^m$, or $a^m b^m a^i b^j$, where at least one among i and j is not m. If they span the second and third run, analogously, pumping them zero times will result in a string $a^m b^i a^j b^m$, where at least one among i and j is not m.

Example Prove that $L = \{a^r b^s : r = s^2\}$ is not context free. By contradiction, if it were context-free, there would be an m such that any string $w \in L$, $|w| \geq m$ would pump. We can choose $w = a^{m^2} b^m$ and show that it cannot be decomposed into $w = uvxyz$, $vy \neq \epsilon$, $|vxy| \leq m$, so that $uv^k xy^k z \in L$ for all $k \in \mathbb{N}$. Clearly, we must have $v = a^K$ and $y = b^L$ for some $K > 0$ and $L > 0$, to have any hope of pumping. But if we pump once, we obtain $a^{m^2+K} b^{m+L}$ where $m^2 + K$ should equal $(m+L)^2$. However, this implies $m^2 + K = m^2 + 2mL + L^2$, or $K = 2mL + L^2$, and this is impossible because $K < m$.

3.11 Closure properties of CFLs

We already know that CFLs are closed under union, concatenation, and Kleene star.

The pumping lemma allows us to prove that:

Theorem CFLs are not closed under intersection and complementation.

Proof It is easy to see that $L_1 = \{a^n b^n b^m : n, m \in \mathbb{N}\}$ and $L_2 = \{a^n b^m b^m : n, m \in \mathbb{N}\}$ are CFLs. However, $L_1 \cap L_2 = \{a^n b^n c^n : n \in \mathbb{N}\}$ which we just proved not to be a CFL. To show that closure does not hold for complementation either, simple observe that closure under complementation would imply closure under intersection, since $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, and CFLs are closed under union. $\square$.

While closure with respect to intersection cannot be assured in general, we are able to show that closure is guaranteed if one of the two languages is not just context-free, but regular:

Theorem Given a CFL L_1 and a regular language L_2, $L_1 \cap L_2$ is a CFL.

Proof We can assume to have an NPDA $M_1 = (Q_1, \Sigma_1, \Gamma, \delta_1, s_1, z, F_1)$ such that $L_1 = L(M_1)$ and a DFA $M_2 = (Q_2, \Sigma_2, \delta_2, s_2, F_2)$ such that $L_2 = L(M_2)$. Then, we can build $M = (Q_1 \times Q_2, \Sigma_1 \cap \Sigma_2, \Gamma, \boxed{s_1, s_2}, \delta, z, F_1 \times F_2)$ where δ is defined as follows:

$$(\boxed{p_1, p_2}, \beta) \in \delta(\boxed{q_1, q_2}, a, \alpha) \;\Leftrightarrow\; (p_1, \beta) \in \delta_1(q_1, a, \alpha) \wedge (a = \epsilon \wedge p_2 = q_2 \vee a \in \Sigma_1 \cap \Sigma_2 \wedge \delta_2(q_2, a) = p_2)$$

We should now show that

$$(\boxed{q_1, q_2}, w, z) \overset{*}{\underset{M}{\vdash}} (\boxed{p_1, p_2}, \epsilon, \gamma) \Leftrightarrow (q_1, w, z) \overset{*}{\underset{M_1}{\vdash}} (p_1, \epsilon, \gamma) \wedge (q_2, w) \overset{*}{\underset{M_2}{\vdash}} (p_2, \epsilon)$$

This is easily proved by induction on the number of steps of M. □

Note We observe that this construction works for a NPDA and a DFA, but not for two NPDA, because we would not be able to manage the stack if the two NPDA decided to change the height of the stack by a different amount (e.g., one wants to pop and the other wants to push).

3.12 Concluding remarks about CFLs

Given a CFL expressed as a CFG $G = (V, T, S, P)$, there is an algorithm to determine whether:

- $L(G) = \emptyset$: simply eliminate all useless variables; $L(G) = \emptyset$ iff none remains.
- $L(G)$ is infinite : simply eliminate all useless variables and ϵ- and unit-productions, then check whether there is a cycle in the directed graph having an arc from A to B iff $A \rightarrow xBy \in P$.
- $w \in L(G)$: use the CYK algorithm.

Note that we will prove later on that there is no algorithm to determine whether $L(G_1) = L(G_2)$ when G_1 and G_2 are two arbitrary CFGs.

3.13 Deterministic pushdown automata

We have seen that, for FA, nondeterminism is not essential (although it is a handy feature to reduce the number of states).

For PDA, instead, nondeterminism is essential: the class of deterministic pushdown automata we now define is less powerful than the NPDA.

Definition A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ is deterministic (DPDA) iff:

$$\forall q \in Q, \forall a \in \Sigma \cup \{\epsilon\}, \forall b \in \Gamma, |\delta(q, a, b)| \leq 1$$

and, in addition, if $\delta(q, \epsilon, b) \neq \emptyset$, then $\forall c \in \Sigma, \delta(q, c, b) = \emptyset$.

This implies that, at every step, at most one legal move exists.

A CFL L is said to be deterministic (DCFL) iff there is a DPDA M such that $L = L(M)$.

We do not prove that there are CFLs that are not deterministic, but we give some examples:

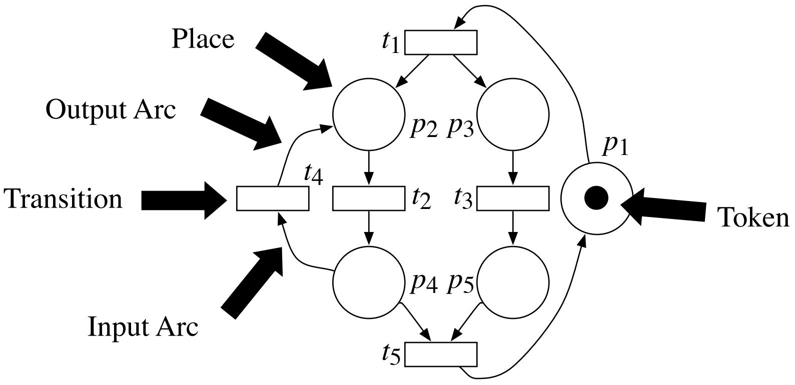
- $\{a^n b^n : n \in \mathbb{N}\} \cup \{a^n b^{2n} : n \in \mathbb{N}\}$
- $\{a^m b a^n b a^k : m = n \vee n = k\}$
- $\{ww^R : w \in \{a, b\}^*\}$

Note that the first two languages can be expressed as the union of two DCFLs, hence this shows that DCFLs are not closed under union.

OPTIONAL: Petri net languages

Petri net languages are rarely considered in an introductory course on formal languages. We study them, although very briefly, because Petri nets are an excellent formalism to model concurrent systems, an increasingly important topic in a computer science.

A Petri net is a finite directed bipartite graph with two sets of nodes, $P = \{p_1, p_2, ...p_{|P|}\}$, a set of *places*, and $T = \{t_1, t_2, ...t_{|T|}\}$, a set of *transitions*. The arcs $A \subseteq (P \times T) \cup (T \times P)$ are either *input arcs*, if they connect a place to a transition, or *output arcs*, if they connect a transition to a place. A place p connected to t via an input arc is an *input place* for t, and a place p connected to t via an output arc is an *output place* for t. Places can contain *tokens*. A *marking* $\mu \in \mathbb{N}^{|P|}$ is a vector describing the number of tokens in each place, it is the "state" of the Petri net. It is common to represent markings using *bag notation*, that is, if $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, the marking $[0, 0, 2, 3, 0, 1]$ is written as $[p_3^2, p_4^3, p_5]$. The initial marking is denoted by $\mu^{[0]}$. Places are drawn as circles and transitions are drawn as rectangles.



A transition t is *enabled* in a marking μ if each of its input places contains at least one token in μ (its input bag I_t satisfies $I_t \leq \mu$, elementwise). A marking μ is a *trap* if no transition is enabled in it. An enabled transition can *fire*. The firing causes a change of marking from μ to μ', where μ' is obtained from μ by removing one token from each input place and adding one token to each output place, we write $\mu \xrightarrow{t} \mu'$. We can extend this relation to a finite sequence of transitions $s = (t_{i_0}, \ldots, t_{i_n}) \in T^*$:

$$\mu \xrightarrow{s} \mu' \;\Leftrightarrow\; \exists \mu_1, \ldots, \mu_n \in \mathbb{N}^{|P|},\; \mu \xrightarrow{t_{i_0}} \mu_1 \xrightarrow{t_{i_1}} \mu_2 \xrightarrow{t_{i_2}} \ldots \xrightarrow{t_{i_{n-1}}} \mu_n \xrightarrow{t_{i_n}} \mu'$$

To generate a language using a Petri net, we define a *labeling function* $\sigma : T \to (\Sigma \cup \{\epsilon\})$, which associates a symbol $a \in \Sigma$ or the empty string ϵ to each transition of the Petri net. When $t \in T$ fires, $\sigma(t)$ is emitted. We can extend σ to sequences of transitions: $\forall s = (t_{i_1}, \ldots t_{i_n}) \in T^*, \sigma(s) = \sigma(t_{i_1}) \cdots \sigma(t_{i_n})$. There are three types of labeling:

- **Free**: each transition is associated to a different symbol $a \in \Sigma$.
- **Non-ϵ**: the same $a \in \Sigma$ can label multiple transitions, but ϵ is not a legal label.
- **Unrestricted**: multiple transitions can be labeled with the same $a \in \Sigma \cup \{\epsilon\}$.

We consider two different definitions of the language generated by a Petri net:

- **L-type** languages: given an alphabet Σ, a Petri net $M = (P, T, A, \mu^{[0]})$, a finite set of final markings $F \subseteq \mathbb{N}^{|P|}$, and a labeling function $\sigma : T \to (\Sigma \cup \{\epsilon\})$,

$$L^{\mathrm{L}}(\Sigma, M, F, \sigma) = \{\sigma(s) : s \in T^* \wedge \mu^{[0]} \xrightarrow{s} \mu \wedge \mu \in F\}.$$

In other words, a string $w \in \Sigma^*$ is in the language if it is emitted by a firing sequence that, starting from the initial marking, leads to a final marking.

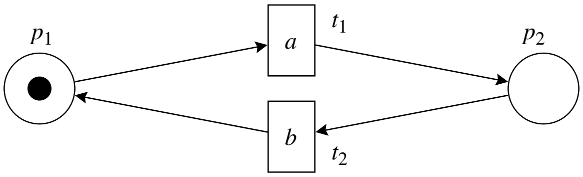
- **T-type** languages: given an alphabet Σ, a Petri net $M = (P, T, A, \mu^{[0]})$ and a labeling function $\sigma : T \to (\Sigma \cup \{\epsilon\})$,

$$L^{\mathrm{T}}(\Sigma, M, \sigma) = \{\sigma(s) : s \in T^* \wedge \mu^{[0]} \xrightarrow{s} \mu \wedge \mu \text{ is a trap}\}.$$

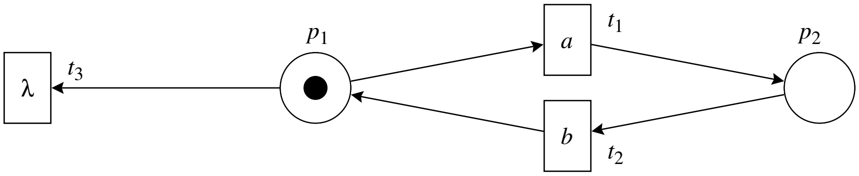
In other words, a string $w \in \Sigma^*$ is in the language if it is emitted by a firing sequence that, starting from the initial marking, leads to a trap.

Clearly, if we let $\mathcal{L}_x^y$ be the class of y-type languages (where y is L or T) that can be generated by a Petri net with x labeling (where x can be F, N, or U, to indicate free, non-ϵ, and unrestricted, respectively), we have the following inclusions: $\mathcal{L}_F^L \subseteq \mathcal{L}_N^L \subseteq \mathcal{L}_U^L$ and $\mathcal{L}_F^T \subseteq \mathcal{L}_N^T \subseteq \mathcal{L}_U^T$. Also, $\mathcal{L}_U^L \subseteq \mathcal{L}_U^T$: given an unrestricted L-type PN, we can obtain an equivalent unrestricted T-type PN by adding a control place (input and output to each of the original transitions) and $|F|$ "killer" transitions, each labeled with ϵ and having input bags equal each to one of the final markings plus the control place, and an empty output bag: a trap state is clearly reached whenever one of these killer transitions fires, since the control token is removed, and these transition can fire only in markings that are final in the original L-type PN.

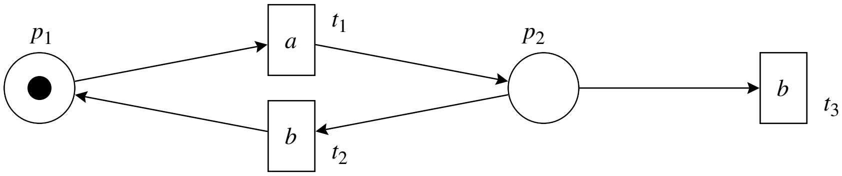
Example A free L-type PN that generates $L = (ab)^*$, by setting $F = \{[p_1]\}$:



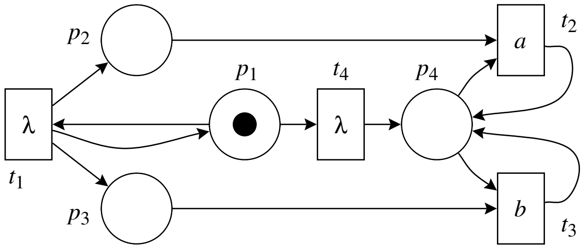
Example An unrestricted T-type PN that generates $L = (ab)^*$:



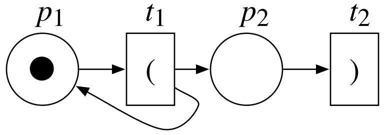
Example A free T-type PN that generates $L = (ab)^+$:



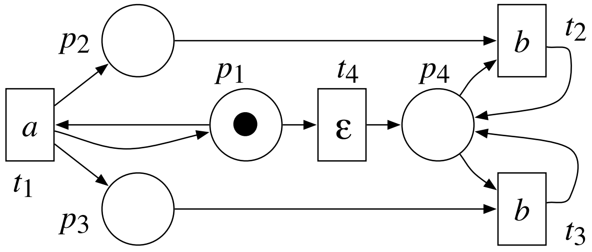
Example An unrestricted T-type PN that generates $L = \{w \in \{a, b\}^* : |w|_a = |w|_b\}$:



Example A free L-type PN that generates the language of balanced parentheses, $L = \{w \in \{(,)\}^* : |w|_{(} = |w|_{)} \wedge \forall x, (\exists y, xy = w) \Rightarrow (|x|_{(} \geq |x|_{)})\}$, by setting $F = \{[p_1]\}$:



Example An unrestricted T-type PN that generates $\{a^n b^{2n} : n \in \mathbb{N}\}$:



Theorem All regular languages can be generated by non-ϵ L-type languages.

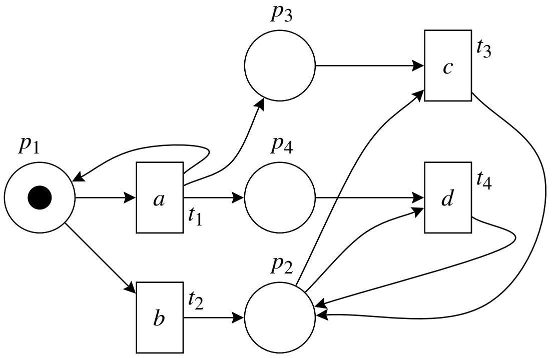
Proof If L is regular, a DFA $M = (Q, \Sigma, \delta, s_0, F)$ exists for it. Then, the PN $M' = (Q, T, A, [s_0])$, with labeling σ and final markings F', where

- $T = \{\boxed{q_1, q_2, a} : \delta(q_1, a) = q_2\}$.
- $A = \{(q_1, \boxed{q_1, q_2, a}) : q_1 \in P \wedge \boxed{q_1, q_2, a} \in T\} \cup \{(\boxed{q_1, q_2, a}, q_2) : q_2 \in P \wedge \boxed{q_1, q_2, a} \in T\}$
- $\sigma(\boxed{q_1, q_2, a}) = a$
- $F' = \{[q] : q \in F\}$

clearly accepts the same language. □

Theorem Some free L-type or T-type PN languages are not context-free.

Proof Consider the language $L = \{a^n bw : w \in \{c, d\}^* \wedge |w|_c = |w|_d = n\}$, which can be proved not to be context-free using the pumping lemma. This language is generated by the following PN, either considered as an L-type with $F = \{[p_2]\}$ or as a T-type (the only trap marking is $[p_2]$ anyway):



Theorem Some context-free languages are not non-ϵ PN languages (of any type).

Proof This proof involves a pumping-lemma-like argument. Consider the language

$$L = \{ww^R : w \in \{a, b\}^*\},$$

clearly context-free. By contradiction, assume that there is a PN with k places that generates L under either the T-type or L-type rule. Then, after generating w_1 or w_2, $w_1 \neq w_2$, $|w_1| = |w_2|$, the PN must be in two different markings (otherwise it would either generate both $w_1 w_1^R \in L$ and $w_2 w_1^R \notin L$, or neither of the two. This implies that the PN must be able to reach $2^{|w|}$ different markings in $|w|$ firings, since each firing generates one symbol in Σ. But the PN can have at most $|w| \cdot r$ tokens after $|w|$ firings, where r is the maximum net number of tokens generated at each firing (total number of output tokens minus total number of input tokens for any transition). Hence, only at most $\binom{|w| \cdot r + k - 1}{k - 1}$ markings can be reached in $|w|$ firings, and, for a sufficiently large $|w|$,

$$\binom{|w| \cdot r + k - 1}{k - 1} < 2^{|w|}$$

since the LHS is $O(|w|^{k-1})$, a polynomial in $|w|$, while the RHS is exponential in $|w|$. □

3.14 Closure properties for PNLs

We need to discuss separately closure with respect to the commonly used language operators for each combination of labeling and type of PNL.

In several cases, closure is an open problem, that is, nobody knows whether a particular class of PN languages is closed under a given operator.

Definition Given two languages L_1 and L_2, the *concurrency* operation on them is:

$$L = L_1 || L_2 = \{x_1 y_1 x_2 y_2 \cdots x_n y_n : x_1 x_2 \cdots x_n \in L_1 \wedge y_1 y_2 \cdots y_n \in L_2\}$$

At times, this operation is also called *interleaving*, and has obvious applications in the theory of operating systems, among other fields.

In many cases, free labeling is a problem simply because it does not allow us to combine two PNs whose transitions share some labels. But even if we limit ourselves to unrestricted labeling, things are not simple. This is what we know about closure for either L-type or T-type PN with unrestricted labeling:

- Concatenation: closed
- Intersection: closed
- Reversal: closed
- Complementation: we don't know
- Kleene star: not closed
- Concurrency: closed (note that CFLs are not closed under concurrency)

Chapter 4

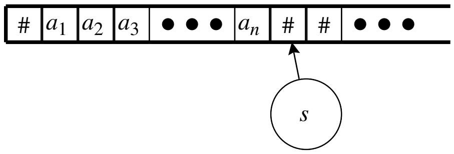
Turing machines

We now move to the most general type of machine (and class of languages).

Definition A Turing machine (TM) is an automaton $M = (Q, \Sigma, \delta, s)$ where:

- Q is a finite set of states; $h \notin Q$ is an additional "halting state" that every TM must have.
- Σ is a finite alphabet; $\# \in \Sigma$ is the "blank symbol"; $L \notin \Sigma$ (for left) and $R \notin \Sigma$ (for right) are two reserved symbols that are meaningful to every TM, but they are not in the alphabet.
- $s \in Q$ is the initial state.
- $\delta : Q \times \Sigma \to (Q \cup \{h\}) \times (\Sigma \cup \{L, R\})$ is the (deterministic) transition function.

The input to M is written on a one-way infinite read-write tape initialized to #'s except for a finite portion. If we want to present the input $w = a_1 a_2 \cdots a_n \in (\Sigma \setminus \{\#\})^*$ to M, we use the convention:



That is, w is surrounded by two #'s, and put in the leftmost possible position, and the read-write head is placed immediately to the right of w.

A configuration of M is an element of $(Q \cup \{h\}) \times \Sigma^* \times \Sigma \times (\Sigma^*(\Sigma \setminus \{\#\}) \cup \epsilon)$: (q, x, a, y) means that M is in state q, that the head is on a, that the entire contents of the tape to the left of the head is x, and the entire contents of the tape up to the first # (excluded) of the infinite run of #'s on the tape is y. We also use the notation $(q, x\underline{a}y)$, where x or y are omitted if they equal ϵ.

If M is in the configuration $(q_1, x_1\underline{a_1}y_1)$, with $q_1 \in Q$, one step in the computation of M is

$$(q_1, x_1\underline{a_1}y_1) \vdash (q_2, x_2\underline{a_2}y_2)$$

where exactly one of the following moves might have been performed:

- (WRITE): $\delta(q_1, a_1) = (q_2, a_2)$, where $a_2 \in \Sigma \wedge x_1 = x_2 \wedge y_1 = y_2$

- (GO LEFT): $\delta(q_1, a_1) = (q_2, L)$, where $x_1 = x_2a_2 \wedge (a_1 = \# \wedge y_1 = \epsilon \wedge y_2 = \epsilon \vee a_1y_1 \neq \# \wedge y_2 = a_1y_1)$

- (GO RIGHT): $\delta(q_1, a_1) = (q_2, R)$, where $x_2 = x_1a_1 \wedge (y_1 = \epsilon \wedge a_2 = \# \vee y_1 = a_2y_2 \neq \epsilon)$

Alternatively, the step $(q_1, x_1\underline{a_1}y_1) \vdash \mathit{HANG}$ is performed when:

- (HANG): $\delta(q_1, a_1) = (q_2, L)$, where $x_1 = \epsilon$

The computation of M on w is a sequence $C_1 \vdash C_2 \vdash C_3 \cdots$ where $C_1 = (s, \#w\underline{\#})$ and, for $i > 1$, each C_i is the configuration obtained from the previous one according to the transition function. There are three cases:

- (HALTING COMPUTATION): $(s, \#w\underline{\#}) \overset{*}{\vdash} (h, x\underline{a}y)$

- (HANGING COMPUTATION): $(s, \#w\underline{\#}) \overset{*}{\vdash} \mathit{HANG}$

- (NON-HALTING COMPUTATION): the remaining case, h is never entered and M never hangs.

4.1 Turing-decidable vs. Turing-acceptable languages

Given a TM $M = (Q, \Sigma, \delta, s)$, there are two ways of defining a language L:

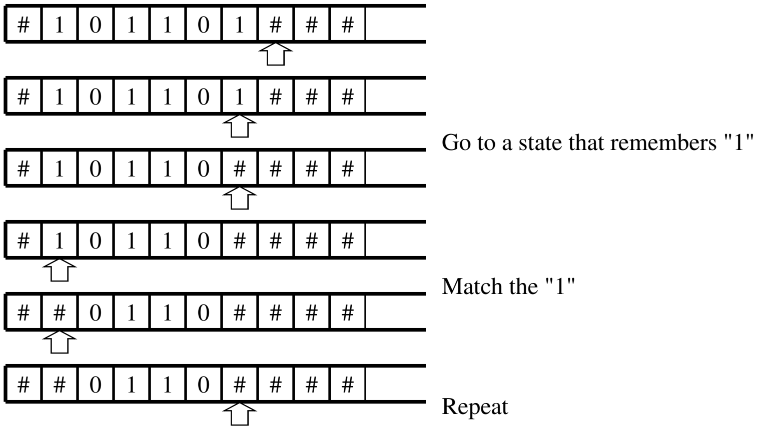
- M **decides** L iff $\forall w \in L, (s, \#w\underline{\#}) \overset{*}{\vdash} (h, \#Y\underline{\#})$ and $\forall w \notin L, (s, \#w\underline{\#}) \overset{*}{\vdash} (h, \#N\underline{\#})$ (of course, $\{Y, N\} \subset \Sigma$).

- M **accepts** L iff $\forall w \in L, (s, \#w\underline{\#}) \overset{*}{\vdash} (h, x\underline{a}y)$, we write this as $M \searrow w$, and, $\forall w \notin L$, M does not halt, or hangs, we write this as $M \nearrow w$.

Definition A language L is Turing-decidable iff there is a TM M that decides L.

Definition A language L is Turing-acceptable iff there is a TM M that accepts L.

Example A TM to decide or accept the language L of the palindromes of even length over $\{0, 1\}$ could go through the following steps:



At the end the TM that decides L must write Y and halt, if it finds out that the input was a palindrome of even length, or it must write N and halt, otherwise. The TM that accepts L must simply halt in the first case, or go into an infinite loop, in the latter.

Definition Let TDL and TAL be the sets of Turing-decidable and Turing-acceptable languages, respectively.

Theorem $TDL \subseteq TAL$.

Proof If $L \in TDL$, there is a TM M that decides L. Simply modify M into M' that behaves exactly like M except that, instead of halting, it moves to the left and tests whether the symbol under the head is Y or N. In the former case it halts, in the latter it goes into an infinite loop. $\square$.

4.2 Turing-computable functions

Since TMs can leave output on the tape, we can use them to compute functions, not just accept or decide languages.

Definition $M = (Q, \Sigma, \delta, s)$ computes $f : \Sigma_0^* \to \Sigma_1^*$ iff,

$$\forall w \in \Sigma_0^*, (s, \#w\underline{\#}) \overset{*}{\vdash} (h, \#f(w)\underline{\#})$$

Of course, this implies $\Sigma_0 \subseteq \Sigma \setminus \{\#\}$ and $\Sigma_1 \subseteq \Sigma \setminus \{\#\}$.

Note Effectively, deciding a language L is the same as computing the indicator function $f : \Sigma^* \to \{Y, N\}$ where $f(w) = Y$ iff $w \in L$.

Note We can easily extend this definition to a machine M that requires k inputs and generate l outputs:

$$\forall w_1 \in \Sigma_1^*, \ldots, \forall w_k \in \Sigma_k^*, (s, \#w_1\# \cdots \#w_k\underline{\#}) \overset{*}{\vdash} (h, \#f_1(w_1, \ldots, w_k)\# \ldots \#f_l(w_1, \ldots, w_k)\underline{\#})$$

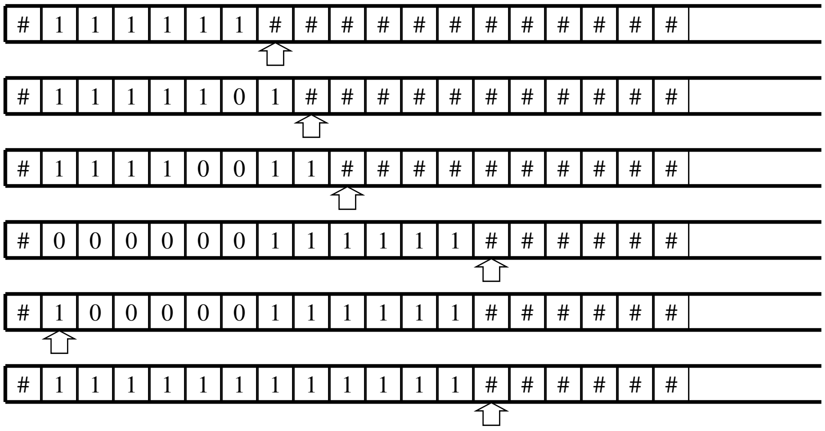
Note We can even allow a variable number of arguments (as long as the input is bounded), we simply need to "tell" the TM their number using this convention for the initial configuration:

$$(s, \#w_1\# \cdots \#w_k\#\underbrace{11 \cdots 1}_{k}\underline{\#}).$$

Example To compute the "increment by one" function, $f(1^n) = 1^{n+1}$, for $n \geq 0$, we can simply define a TM that (1) changes # to 1 and (2) move right and halts: $\delta(s, \#) = (s, 1)$, $\delta(s, 1) = (h, R)$. Note that we do not need (ever) to worry about what happens if the input is not in the correct format. We only need to be sure that the TM computes the correct output and halts if it is presented with an input in the correct format.

Example A TM for the "decrement by one if not zero" function, $f(1^n) = 1^{n-1}$, for $n > 0$, $f(\epsilon) = \epsilon$, is obtained by: $\delta(s, \#) = (t, L)$, $\delta(t, 1) = (h, \#)$, $\delta(t, \#) = (h, R)$.

Example A TM to compute the "doubling" function, $f(1^n) = 1^{2n}$, is also simple, but it shows how low-level and "slow" TMs are with respect to real computers. We simply show some intermediate steps in the computation:



```
# 1 1 1 1 1 1 # # # # # # # # # # # #
# 1 1 1 1 1 0 1 # # # # # # # # # # #
# 1 1 1 1 0 0 1 1 # # # # # # # # # #
# 0 0 0 0 0 0 1 1 1 1 1 1 # # # # # #
# 1 0 0 0 0 0 1 1 1 1 1 1 # # # # # #
# 1 1 1 1 1 1 1 1 1 1 1 1 # # # # # #
```

Note We can also talk about functions of the type $f : \mathbb{N}^k \to \mathbb{N}$, since, as we just did, we can simply express natural numbers using unary notation: the number n is represented by the string 1^n.

Definition We say that a function $f : \mathbb{N}^k \to \mathbb{N}$ is Turing-computable iff there is a TM M that computes f.

Note For FA and NPDA, the best analog we have for "computing a function" $f : \mathbb{N}^k \to \mathbb{N}$ is that we can feed the string $w_1 \cdots w_k w$ and ask that the FA or the NPDA accepts it iff $w = f(w_1, \ldots, w_k)$. E.g., $\{a^n b^m c^{n+m} : n, m \in \mathbb{N}\}$ is the CFL for addition. In practice, if we wanted to use an NPDA to add numbers, we could submit the strings $a^n b^m c^0$, $a^n b^m c^1$, $a^n b^m c^2$, and so on, until we find the answer $(n + m)$ we want. This would be very slow, but not essentially different, if we just want to talk about computational power.

Note Given our convention in presenting the input and requiring the output for a TM, it is easy to see how we can combine TMs that compute functions. If M_1 computes f_1 and M_2 computes f_2, a TM M computing $f_2 \circ f_1$ is easily obtained by letting M_1 go the initial state of M_2 instead of h (of course the sets of states of the two machines must be disjoint).

4.3 Turing's thesis

We are now going to try to modify and extend the definition of TMs we gave, in the hope to extend its power. Since we will not succeed, we might be inclined to think that TMs are indeed the most powerful computational device, and that any effort to extend it is destined to fail.

More precisely, Turing's thesis can be stated as follows:

> Turing machines are the formalization of our concept of algorithm or mechanical computation. No computational procedure is considered an algorithm unless it can be described by a Turing machine.

While this is not a mathematical result (it is not a result we can prove), there is strong evidence that is a correct belief. This is related to Church's thesis which can be stated as follows:

> All interesting (i.e., not intentionally limited) discrete models of computation are equally powerful, and they are all equivalent to Turing machines.

Since TMs are very low-level objects, we will often talk about algorithms instead, in the sense that, if we agree that we would be able to write a computer program for a given task (for example), then we know that we would also be able to define a TM for the same task. However, we must be careful: at times, something can "sound" like an algorithm, but it is not. Only by defining very clearly what we are talking about we can be sure that, indeed, the task could be carried on by a TM.

The extension/variations we consider are:

- TMs that both move and write in a single move: $\delta : Q \times \Sigma \to (Q \cup \{h\}) \times \Sigma \times \{L, R, S\}$, where S means "stay".

  This simply allows to be "faster" than the TMs we defined by a factor no greater than two.

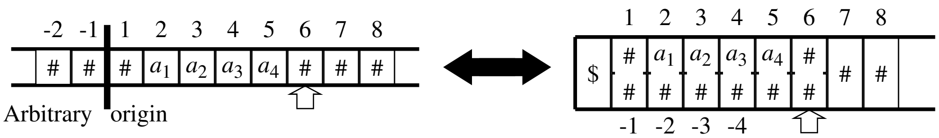
- TMs that are prevented from hanging by definition. For example, we could say that a move left when the TM is at the leftmost position results in a move to the right, or in no move.

  We can ensure that our TM never hangs by going to find the leftmost # and changing it to a new symbol, \$, such that $\delta(q, \$)$ never results in a move to the left, for any $q \in Q$.

- TMs that cannot write #'s. This is really a restriction, not an extension, but it does not limit the power of the TM, because we can just use a different symbol to mean a "dirty blank", as opposed to the otherwise equivalent "original blank" that can be found but not written.

- TMs with a two-way infinite tape.

  Here the idea is that we can "fold" the tape around an arbitrary point using a more complex alphabet for the one-way infinite tape TM, which must then keep track of which side of the tape the two-way infinite TM is on, by (at most) doubling the number of states:



  Note that the doubling of the tape can only be carried on a finite portion, so that the remaining infinite portion is still the default (single) #. Then, if the two-way infinite tape TM is $M = (Q, \Sigma, \delta, s)$, the equivalent one-way infinite tape TM is

$$M' = (Q' \cup Q \times \{+, -\}, \Sigma \cup \overline{\Sigma} \cup (\Sigma \times \Sigma) \cup (\overline{\Sigma} \times \Sigma) \cup (\Sigma \times \overline{\Sigma}) \cup \{\$\}, \delta', s').$$

  If the input is $w = w_1 \cdots w_n$ (to both M and M'), the first thing M' must do is to go from $(s', \#w_1\underline{\#})$ to $([s, +], \$[\#, \#][w_1, \#] \cdots [w_n, \#]\underline{[\#, \#]})$. Then, if $\delta(q, a) = (p, b)$, for any $x \in \Sigma$:

$$\delta'([q,+],[a,x]) = \begin{cases} ([p,+],L) & \text{if } b = L \\ ([p,+],R) & \text{if } b = R \\ ([p,+],[b,x]) & \text{if } b \in \Sigma \end{cases} \qquad \delta'([q,-],[x,a]) = \begin{cases} ([p,-],R) & \text{if } b = L \\ ([p,-],L) & \text{if } b = R \\ ([p,-],[x,b]) & \text{if } b \in \Sigma \end{cases}$$

  In addition, we need to set the mechanism to change tracks:

$$\delta'([q,+],\$) = ([q,-],R) \qquad \delta'([q,-],\$) = ([q,+],R)$$

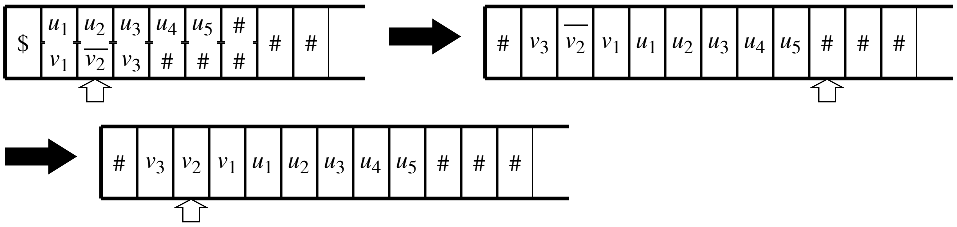
and to extend the tape:

$$\delta'([q,+],\#) = ([q,+],[\#,\#]) \qquad \delta'([q,-],\#) = ([q,-],[\#,\#])$$

Finally, when M halts, M' must mark the position where this happens:

$$\delta'([h,+],[a_+,a_-]) = ([h,+],[\overline{a_+},a_-]) \qquad \delta'([h,-],[a_+,a_-]) = ([h,-],[a_+,\overline{a_-}])$$

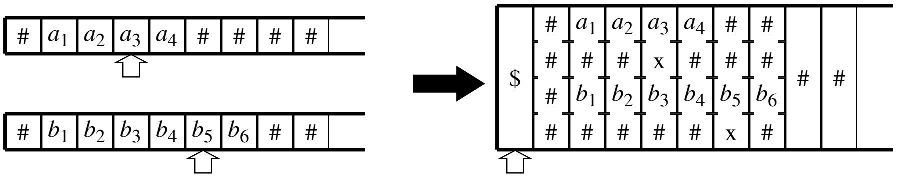
then put the tape in single track format and return the head to the position where the head of M would be:



$ | u1 u2 u3 u4 u5 # | v1 v2 v3 # # # | # #
# v3 v2 v1 u1 u2 u3 u4 u5 # # #
# v3 v2 v1 u1 u2 u3 u4 u5 # # #

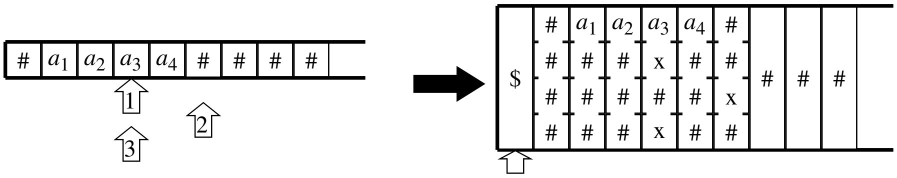
- TMs with multiple tapes.

  The construction for simulating these with a single tape TM is similar to the previous one, we use a more complex alphabet:



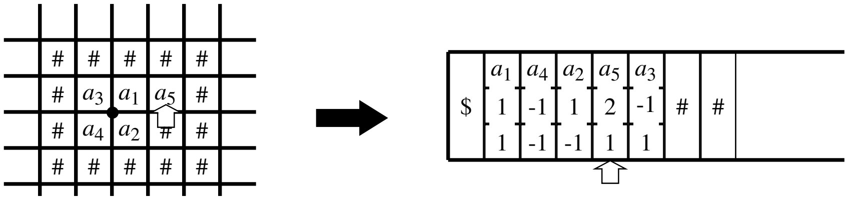
- TMs with multiple heads on a single tape.

  This is almost the same as the multitape extension. However, we must be careful about the definition of a TM with k heads when $\delta(q,[a_1,\ldots,a_k]) = (q,[b_1,\ldots,b_k])$ and there are some $b_i \in \Sigma$, $b_j \in \Sigma$, $b_i \neq b_j$, and heads i and j are actually on the same cell. In this case, a reasonable policy is, for example, to give priority to the highest-numbered head (i.e., write b_j on the tape iff $j > i$). We can certainly model this behavior with our single tape TM:

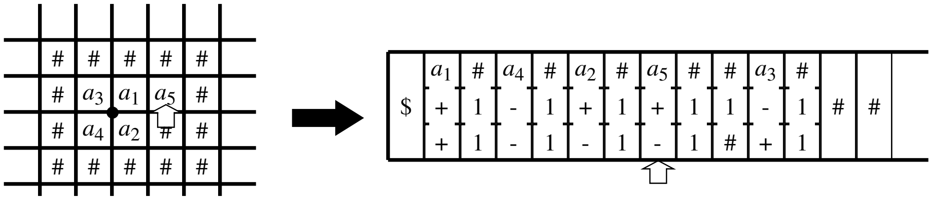


- TMs with a multidimensional tape.

  Here we need to define an indexing for the cells in a k-dimension space. By assigning an arbitrary origin (and positive/negative directions), we can say that each cell has index $(i_1,\ldots,i_k) \in (\mathbb{N}^+)^k$. Representing this on a single tape, however, is not trivial because the k indices are not bounded, so we cannot store them in a single cell. In other words, this solution is wrong:



We need to use a variable-length encoding (e.g., unary) to represent the index in each dimension:



- Nondeterministic TMs.

  We will show how even nondeterminism is not a real extension in the next section.

4.4 Nondeterministic TMs

Definition A nondeterministic TM (NDTM) is an automaton $M = (Q, \Sigma, \delta, s)$ defined analogously to a deterministic TM (DTM from now on), except for the transition function:

- $\delta : Q \times \Sigma \to 2^{(Q \cup \{h\}) \times (\Sigma \cup \{L,R\})}$.

Unlike DTMs, though, we can't use NDTMs to compute a function or to decide a language: if I want to use a NDTM to compute f, and at the end I see that multiple clones halted with different values on the tape, how do I choose the correct answer? Analogously, if some clones say Y and some say N, is w really in the language decided by M?

Hence, we only use a NDTM M to accept a language: if any clone halts on w, then $w \in L(M)$.

To stress the difference between deterministic and nondeterministic thinking, consider the language $L = \{1^n : \exists i, j > 1, n = i \cdot j\}$ of composite numbers. A DTM to solve this problem can start at 2 and try all numbers i up to $n-1$, and test whether they divide n. If it finds one, then $1^n \in L$. Of course, we can use more sophisticated filters or stop at $\sqrt{n}$, but the basic approach is the same. A NDTM can simply "guess" a pair $(i, j) \in \{2, n-1\}$, using nondeterminism, then multiply i by j and compare the result with n; if any of the clones finds that $i \cdot j = n$, it halts (and accepts).

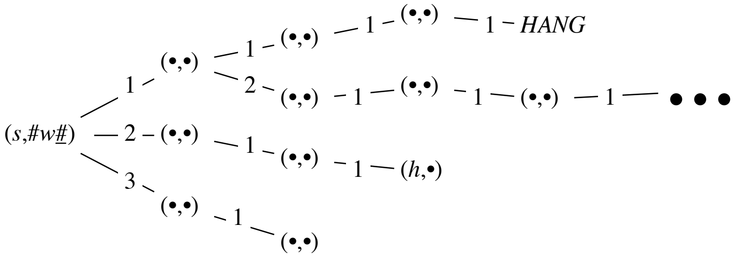
However, we are now going to show how nondeterminism at most allows us better speed, but not increased power:

Theorem Given a NDTM $M = (Q, \Sigma, \delta, s)$, there is a DTM $M' = (Q', \Sigma', \delta', s')$ such that M and M' accept the same language: $\forall w \in \Sigma^*,\ M \searrow w \ \Leftrightarrow\ M' \searrow w$.

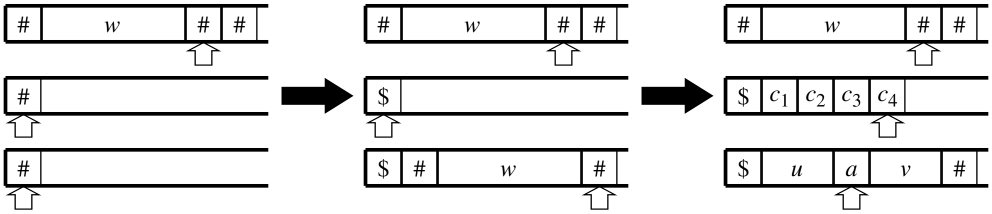
Proof M' must concurrently simulate all the computations of the clones of M. Unfortunately, the number of clones is not bounded. Fortunately, after n steps, the number of clones is finite. If r is the maximum number of clones generated at each step,

$$r = \max\{|\delta(q, a)| : q \in Q, a \in \Sigma\}$$

then there can be at most r^n clones after n steps. A DTM M' can simulate them "concurrently", but only through **dovetailing**. In other words, it must simulate up to move n all the possible clones (at most r^n of them) that can exist after n moves, for $n = 1$, then $n = 2$, etc. This corresponds to a BFS search of the tree describing the computations of the clones. If the DTM M' used a DFS, it might never come back from an infinite path, thus it might never halt even if one of the clones of M halts:



To keep track of these computations, M' can use a 2-D tape, or even simply three tapes:



The first tape simply keeps an original of the input string w at all times. The second tape is used as a stack to record the set of n nondeterministic choices corresponding to the clone currently being modeled ($c_i \in \{1, \ldots, r\}$), and the third tape has the same contents and head position as the tape of the clone currently being modeled. Every time M' completes the simulation of the first n moves of a clone:

- If the clone just halted, then M' halts as well.
- Otherwise, increment $c_1 c_2 \cdots c_n$ to the next configuration in **lexicographic order**. Note that some of these configurations might correspond to unavailable choices; for example, from $(s, \#w\underline{\#})$, fewer than r choices might be available; this is not a problem, simply define transitions to states that cause the simulation to hang in this case, or detect this case and skip to the next configuration.
- If $c_1 c_2 \cdots c_n = r^n$, then the next configuration is 1^{n+1}, that is, M' starts now to explore whether any clone can halt after $n + 1$ moves.
- In any case, M' can behave deterministically for as many moves as the height of the stack, choosing move c_i at the i-th step, and simulating the behavior of M on the third tape. □

□.

The proof we just completed uses two fundamental concepts that we will keep using: **dovetailing** and **enumeration in lexicographic order**.

We can then generalize the idea of simulation of an automaton by another one, and observe that these simulations are possible because we are not restraining the model of computation in any way except as follows:

> **Unbounded computational power is achievable only by applying a finite amount of resources for an unbounded amount of time.**

An example of computation that does not follow the above rule (hence it is not equivalent to TMs) is automata that can operate on real numbers (not floating point numbers, which are just a finite approximation of real numbers!) in a single step (e.g., $x \leftarrow y + z$).

4.5 Universal TM

So far, we have described TMs that can perform a specific task, such as deciding or accepting a language or computing a function. Today's computers are not built this way: they are general-purpose machines that receive in input a program P and some input w, and then run P on w.

We now define a TM that does the same, that is, a TM that receives in input the encoding of a TM M and of a string w, and simulates the computation of M on w.

Before doing so, however, we must agree on a way to describe a machine M and a string w in a standard format. For example, the set of possible input alphabets is infinite, even if the each alphabet is itself finite. However, thanks to the idea of homomorphism, all that really matters is the number of symbols in the alphabet (i.e., it makes no difference whether our alphabet is $\{a, b, c\}$ or $\{0, 1, 2\}$ when we talk about a given language).

We can define $\Sigma_\infty = \{a_1, a_2, a_3, \ldots\}$ as the union of all the alphabets we might be interested in. Note that Σ_∞ itself is not an alphabet, since it is an infinite set. Any set of interest to us will then be $\Sigma \subset \Sigma_\infty$ such that $|\Sigma| < \infty$.

Analogously, the names of the states of a TM is irrelevant, so we define the set $Q_\infty = \{q_1, q_2, q_3, \ldots\}$ and the set of states of any TM we will consider will simply be $Q \subset Q_\infty$ such that $|Q| < \infty$.

Then, we can define the encoding for any DTM $M = (Q, \Sigma, \delta, s)$, with $Q = \{q_1, \ldots q_k\}$ and $\Sigma = \{a_1, \ldots, a_l\}$:

- Define $e(L) = 1$, $e(R) = 11$, $\forall a_i \in \Sigma, e(a_i) = 1^{i+2}$.
- Define $e(h) = 1$, $\forall q_j \in Q, e(q_j) = 1^{j+1}$.
- Define the encoding of a single transition $\delta(q_i, a_j) = (p, b)$, with $q_i \in Q$, $a_j \in \Sigma$, $p \in Q \cup \{h\}$, $b \in \Sigma \cup \{L, R\}$ as
$$T_{i,j} = 0e(q_i)0e(a_j)0e(p)0e(b)0$$
- Define the encoding of M, as

$$\rho(M) = 0e(s)0 \quad \underbrace{T_{1,1} \cdots T_{1,l} T_{2,1} \cdots T_{k,l}}_{k \cdot l = |Q| \times |\Sigma| \text{ transitions}} \quad 0$$

- Define the encoding of $x = b_1 \cdots b_n \in \Sigma^*$, as

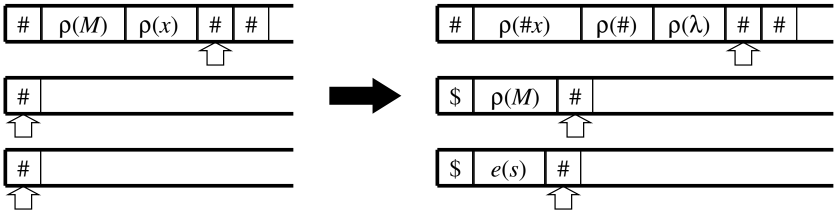
$$\rho(x) = 0e(b_1)0 \cdots 0e(b_n)0$$

Clearly, given a string $w \in \{0,1\}^*$, we can easily tell whether w is the encoding $\rho(M)$ of some DTM M; it is simply a syntax check. On the other hand, given a DTM M, there is a unique string $w \in \{0,1\}^*$ that encodes M.

Definition A universal TM (UTM) $U = (Q_U, \Sigma_U, \delta_U, s_U)$ is a DTM that receives as input a DTM M and a string x, both encoded, and simulates the behavior of M on x, that is:

$$\forall M = (Q, \Sigma, \delta, s), \forall x \in \Sigma^* : (s, \#x\underline{\#}) \overset{*}{\underset{M}{\vdash}} (h, u\underline{a}v) \;\Leftrightarrow\; (s_U, \#\rho(M)\rho(x)\underline{\#}) \overset{*}{\underset{U}{\vdash}} (h, \#\rho(u)\rho(a)\rho(v)\underline{\#})$$

We can use a 3-tape DTM to show how to build U:



From the initial configuration, U goes to the configuration shown on the right, where the first tape encodes the contents of the tape of M and the position of its head, the second tape contains the encoding of M, and the third tape contains the encoding of the current state of M. Then, it is a simple matter of pattern-matching: U must find on the second tape the move corresponding to the current state of M and the symbol under the head for M; once its finds it, it performs the required action on the first tape (move L, R, or change the encoding of the symbol under the head) and puts the encoding of the new state on the third tape. The simulation continues until the encoding of state h, i.e., a single 1, is put on the third tape, or it goes on forever, if this never happens (if M hangs, U can easily detect this and go into an infinite loop, or hang as well).

Before discussing the implications of the existence of a UTM, a few observations are in order:

- Recall that $\{0,1\}^*$ is countable.
- Hence, the set of all TM encodings, $\{w \in \{0,1\}^* : \exists M, w = \rho(M)\}$, is countable as well.
- Hence, there is a countable number of TMs, but the set $\mathcal{L}$ of languages over any alphabet Σ is uncountable, since $\mathcal{L} = 2^{\Sigma^*}$.
- Hence, we can only address only a small subset of languages even using the most powerful model of computation we know, TMs. Or, in other words, there is an uncountable number of languages that are not even Turing-acceptable.

We recall the proof that $\mathcal{L} = 2^{\Sigma^*}$ is uncountable, even when $\Sigma = \{1\}$. The proof is based on the idea of **diagonalization**. By contradiction, assume that $\mathcal{L}$ is countable. Then, we can say that

$\mathcal{L} = \{L_0, L_1, L_2, \ldots\}$, with $L_i \subseteq \{1\}^*$ and $L_i \neq L_j$ for $i \neq j$. We can then describe $\mathcal{L}$ using a boolean table with an infinite number of rows, corresponding to $\mathcal{L}$ and columns, corresponding to 1^*:

	ϵ	1	11	111	1111	$\cdots$	1^d
L_0							
L_1							
L_2							
L_3							
$\cdots$							
L_d							?

where the entry in position (L_i, w) is 1 if $w \in L_i$, 0 otherwise. Of course, we can do this only because we assume that the set $\mathcal{L}$ is countable. Then we can define the diagonal set

$$D = \{1^i : 1^i \notin L_i\}$$

Since $D \subseteq \{1\}^*$, $D \in \mathcal{L}$, hence there must be exactly one $d \in \mathbb{N}$ such that $D = L_d$. But this leads to a contradiction, because:

$$1^d \in D \iff 1^d \notin L_d \iff 1^d \notin D.$$

Hence $\mathcal{L}$ must be uncountable.

4.6 Decidable languages

Recall the definition of *TDL* and *TAL*. We have shown that $TDL \subseteq TAL$, but is the inclusion strict? In other words, is there a language L in *TAL* but not in *TDL*? To answer this question we need to prove a theorem and make use of the idea of **self-reference**, that is, define languages that "can talk about themselves". This is just another way of thinking about diagonalization.

Theorem If $L \in TAL$ and $\overline{L} \in TAL$, then $L \in TDL$.

Proof Simply dovetail M accepting L and $\overline{M}$ accepting $\overline{L}$. If M halts, we know that $\overline{M}$ will not, so stop and say Y. If $\overline{M}$ halts, we know that M will not, so stop and say N. No other possibility exists. □.

Corollary If $L \in TAL$ and $L \notin TDL$, then $\overline{L} \notin TAL$ (and, of course, $\overline{L} \notin TDL$).

Definition $K_0 = \{\rho(M)\rho(x) : M \searrow x\}$.

We can ask whether there is a TM M_0 that decides K_0. If there is, then $TAL = TDL$, that is, any acceptable language L is also decidable. This is because, if $L \in TAL$, we have a TM M that accepts L, and we can simply feed $\rho(M)\rho(x)$ to M_0. If M_0 halts and says Y, $x \in L$, if M_0 halts and says N, $x \notin L$.

Definition $K_1 = \{\rho(M) : M \searrow \rho(M)\}$.

This is self-reference! Clearly, $K_1 \in TAL$, since we can simply use the UTM to simulate M on $\rho(M)$. The UTM will halt iff M does. Also, if $K_0 \in TDL$, then $K_1 \in TDL$, because

$$\rho(M) \in K_1 \Leftrightarrow \rho(M)\rho(\rho(M)) \in K_0$$

But we now show that $K_1 \notin TDL$, which then implies also $K_0 \notin TDL$.

Theorem $K_1 \notin TDL$.

Proof By contradiction, if $K_1 \in TDL$, then its complement $\overline{K_1} \in TDL$ as well, which implies $\overline{K_1} \in TAL$, that is, there is a TM M^* that accepts $\overline{K_1}$. But $\overline{K_1}$ contains all the strings in $\{0,1\}^*$ that (1) either are not the encoding of a TM — these are easy to recognize — or (2) are the encoding $\rho(M)$ of a machine M that does not halt on its own encoding, $M \nearrow \rho(M)$. We can then ask whether $\rho(M^*)$ is in $\overline{K_1}$, and we reach a contradiction:

$$M^* \nearrow \rho(M^*) \;\Leftrightarrow\; \rho(M^*) \in \overline{K_1} \;\Leftrightarrow\; M^* \searrow \rho(M^*)$$

Hence, $\overline{K_1} \notin TAL \;\Rightarrow\; \overline{K_1} \notin TDL \;\Rightarrow\; K_1 \notin TDL \;\Rightarrow\; K_0 \notin TDL$. □.

Note The previous proof shows both examples of languages that are in TAL but not in TDL (e.g., K_0, K_1), and of languages that are not even in TAL (e.g., $\overline{K_0}$, $\overline{K_1}$).

4.7 Implications of the halting problem

The practical implications of $K_0 \notin TDL$, also called the **halting problem**, is that, just as there is no TM that can answer the question "does M halt on x", there is no general algorithm to determine whether a given program will stop (a necessary requirement for "working correctly") on a given input.

Definition We say that a yes-no problem P is semi-solvable iff there is no decision algorithm for it, but there is a **semi-algorithm** that will give an answer if the answer is Y, or will never terminate if the answer is N. Analogously, we say that a language L is semi-decidable if it is not decidable but it is acceptable.

Note Unfortunately, the words **unsolvable** and **undecidable**, which should perhaps be reserved to languages not even in TAL, such as $\overline{K_0}$, are often used instead of semi-solvable and semi-decidable.

Definition A set/language/problem A is **reducible** to B if we can exhibit an algorithm (a computable function) that transforms each instance of A into an instance of B.

The idea of reduction plays a fundamental role in proving that a certain set/language/problem has or does not have a certain property, such as being in TDL or TAL. To show that a set/language/problem A

- **has** property X, we can find a set/language/problem B known to have property X and reduce A to B.
- **does not have** property X, we can find a set/language/problem B known not to have property X and reduce B to A.

This is useful to classify problems into "degrees of difficulty": if I can reduce A to B, then B is at least as difficult as A, since knowing how to solve B immediately gives us a solution for A. Note that the converse is not true: A might be "easier" than B, but we can still solve A by using B, although this is like "using a bazooka to kill a mosquito".

Theorem The following problems are undecidable (let $L(M) = \{w : M \searrow w\}$):

1. Given inputs M and w: $M \searrow w$?
2. After fixing a given M, given input w: $M \searrow w$?
3. Given M: $M \searrow \epsilon$? Or in other words: $\epsilon \in L(M)$?
4. Given M: $\exists x, M \searrow x$? Or in other words: $L(M) \neq \emptyset$?
5. Given M: $\forall x, M \searrow x$? Or in other words: $L(M) = \Sigma^*$?
6. Given M_1 and M_2: $\forall x, M_1 \searrow x \Leftrightarrow M_2 \searrow x$? Or in other words: $L(M_1) = L(M_2)$?
7. Given M: is $L(M)$ regular? context-free? decidable?

Proof To show the undecidability of the above questions, we use reduction:

1. Proved in class, it is K_0.
2. Fixing M does not help, because we can choose to fix on an M that accepts a "difficult" language, consider for example M to be M_{K_0}, the TM that accepts K_0 (i.e., the UTM). Then,

$$M_{K_0} \searrow w \Leftrightarrow M \searrow x, \text{ where } w = \rho(M)\rho(x)$$

3. If M_ϵ existed that decides the language $\{\rho(M) : M \searrow \epsilon\}$, we could use it to decide K_0. To answer the question $M \searrow w$, we could simply build a TM $T_{M,w}$ that, if started on an empty string, writes w on its tape and begins simulating M on it. Then,

$$M_\epsilon \searrow \rho(T_{M,w}) \Leftrightarrow T_{M,w} \searrow \epsilon \Leftrightarrow M \searrow w$$

4. If $M_\exists$ existed that decides the language $\{\rho(M) : \exists x, M \searrow x\}$, we could use it to decide K_0. To answer the question $M \searrow w$, we could simply build a TM $Y_{M,w}$ that erases whatever input it is given, then writes w on its tape, and starts simulating M on it. Then,

$$M_\exists \searrow \rho(Y_{M,w}) \Leftrightarrow \exists x, Y_{M,w} \searrow x \Leftrightarrow M \searrow w$$

5. Of course, the above reduction works also in this case, since the input x is ignored, thus

$$\exists x, Y_{M,w} \searrow x \Leftrightarrow \forall x, Y_{M,w} \searrow x$$

6. If $M_=$ existed that decides the language $\{\rho(M_1)\rho(M_2) : L(M_1) = L(M_2)\}$, we could use it to decide the language $\{\rho(M) : L(M) = \Sigma^*\}$, which we just proved to be undecidable. We simply build a TM M_2 that halts as the first move. Hence, $L(M_2) = \Sigma^*$. Then,

$$M_= \searrow \rho(M)\rho(M_2) \Leftrightarrow L(M) = L(M_2) \Leftrightarrow L(M) = \Sigma^*$$

7. This is the most difficult one. Assume that exists a TM M_r that halts on $\rho(M)$ iff $L(M)$ is regular. We will show that the existence of M_r implies the decidability of $\{\rho(M') : M' \searrow \epsilon\}$. Given a specific machine M', build a machine M^* that first simulates M' on ϵ. If $M' \searrow \epsilon$, then

M^* starts behaving like M_{K_0}, the TM that accepts K_0. If $M' \nearrow \epsilon$, then, of course, M^* will simply run forever, so

$$L(M^*) = \begin{cases} \emptyset & \text{if } M' \nearrow \epsilon \\ K_0 & \text{if } M' \searrow \epsilon \end{cases}$$

and observe that $L(M^*)$ is regular (and context-free, and decidable) in the first case, but not in the second. Then,

$$M_r \searrow \rho(M^*) \;\Leftrightarrow\; L(M^*) \text{ is regular } \Leftrightarrow M' \nearrow \epsilon$$

Thus, if M_r existed, we could use it to accept $\overline{\{\rho(M) : M \searrow \epsilon\}}$, and, since its complement, $\{\rho(M) : M \searrow \epsilon\}$ is clearly acceptable as well, both $\overline{\{\rho(M) : M \searrow \epsilon\}}$ and $\{\rho(M) : M \searrow \epsilon\}$ should be decidable; however, we just showed that $\{\rho(M) : M \searrow \epsilon\}$ is not decidable. The same idea applies to "context-free" and "decidable", since $\emptyset$ is also context-free and decidable.

□.

4.8 Unrestricted grammars

We now consider the class of grammars that corresponds to the type of languages accepted by TMs. These are called **unrestricted grammars** or **rewriting systems**.

Definition An unrestricted grammar is a tuple $G = (V, T, S, P)$ where V, T, and S have the usual meaning, and P contains a finite set of productions with the only restriction that the left-hand side must contain at least one variable:

$$P \subseteq (V \cup T)^* V (V \cup T)^* \times (V \cup T)^* \quad \text{with} \quad |P| < \infty$$

and, as usual, the language generated by G is defined as $L(G) = \{w \in T^* : S \overset{*}{\underset{G}{\Longrightarrow}} w\}$.

Example A grammar to generate $\{w \in \{a,b,c\}^* : |w|_a = |w|_b = |w|_c\}$ is $G = (\{A, B, C, S\}, \{a, b, c\}, S, P)$, where P contains the following productions:

$S \to ABCS|\epsilon,\ AB \to BA,\ BA \to AB,\ AC \to CA,\ CA \to AC,\ BC \to CB,\ CB \to BC,\ A \to a,\ B \to b,\ C \to c$

Example A grammar to generate $\{a^{2^n} : n \in \mathbb{N}\}$ is $G = (\{[,], A, D, S\}, \{a\}, S, P)$, where P contains the following productions:

$$S \to [A],\ [\, \to [D,\ D] \to\,],\ DA \to AAD,\ [\, \to \epsilon,\ ] \to \epsilon,\ A \to a$$

We can think of the variable D as a "doubler". Every time a D is generated, on the left of the sentential form, it will be removed only by reaching the right, and, in doing so, it will double the number of A's it encounters. Note that, if we prematurely apply the production $] \to \epsilon$ (that is, before all D have been removed), we will simply not generate a string, since the remaining D's will be in the sentential form. So the sequences that generate strings are of the type

$$S \Longrightarrow [A] \overset{n}{\Longrightarrow} [D^n A] \overset{*}{\Longrightarrow} [A^{2^n} D^n] \overset{n}{\Longrightarrow} [A^{2^n}] \Longrightarrow A^{2^n}] \Longrightarrow A^{2^n} \overset{2^n}{\Longrightarrow} a^{2^n}$$

Example A grammar to generate $\{babaabaaab\cdots a^{n-1}ba^nb : n \in \mathbb{N}\}$ contains the following productions:

$$S\to AbS|X,\ Ab\to baA,\ Aa\to aA,\ AX\to X, X\to b$$

For example,

$$S \overset{3}{\Longrightarrow} (Ab)^3S \Longrightarrow (Ab)^3X \Longrightarrow AbAbbaAX \Longrightarrow AbAbbaX \Longrightarrow AbbaAbaX \Longrightarrow AbbabaAaX$$

$$\Longrightarrow AbbabaaAX \Longrightarrow AbbabaaX \overset{*}{\Longrightarrow} babaabaaaAX \Longrightarrow babaabaaaX \Longrightarrow babaabaaab$$

Note So far we have used grammars as language generators. However, we can also use them as function computers: $f : \Sigma_0^* \to \Sigma_1^*$ is **grammatically computable** iff there is a grammar $G = (V, T, S, P)$ and four strings $x, y, x', y' \in (V \cup T)^*$ such that

$$\forall u \in \Sigma_0^*,\ xuy \overset{*}{\underset{G}{\Longrightarrow}} x'f(u)y' \ \wedge\ \forall u' \in \Sigma_1^*,\ u' \neq f(u) \ \Rightarrow\ \neg(xuy \overset{*}{\underset{G}{\Longrightarrow}} x'u'y')$$

In other words, x, y, x', y' are just used to “bracket’ the input and the output, and $f(u)$ is the only string that can be generated within the brackets x' and y' starting from u bracketed by x and y.

Theorem Given a TM $M = (Q, \Sigma, \delta, s)$, there exists an unrestricted grammar $G = (V, \Sigma \cup \{[,]\}, S, P)$ such that

$$(q, u\underline{a}v) \overset{*}{\underset{M}{\vdash}} (q', u'\underline{a'}v') \quad \Leftrightarrow \quad [uqav] \overset{*}{\underset{G}{\Longrightarrow}} [u'q'a'v']$$

where $Q \cap \Sigma = \emptyset$ and $[,] \notin Q \cup \Sigma$.

Proof The grammar G is defined as follows:

- Let $V = Q \cup \{h, S\}$.
- For each TM move $\delta(q, a) = (p, b)$, $b \in \Sigma$, add $qa\to pb$ to P.
- For each TM move $\delta(q, a) = (p, R)$, add $qab\to apb$ to P, for all possible $b \in \Sigma$. Also add $qa]\to ap\#]$.
- For each TM move $\delta(q, a) = (p, L)$, add $bqac\to pbac$ to P, for all possible $b \in \Sigma$ and $c \in \Sigma \cup \{]\}$, as long as $a \neq \#$ or $c \neq]$. If $a = \#$, also add $bq\#]\to pb]$, for all possible $b \in \Sigma$.
- For each TM move $\delta(q, a) = (p, L)$, also add $[q\to[$, to model the fact that the Turing can HANG.

It is easy to see that this grammar simulates the TM. □.

Theorem Given an unrestricted grammar $G = (V, T, S, P)$, there is a TM $M = (Q, \Sigma, \delta, s)$, such that the language accepted by M is the same as the language generated by G:

$$S \overset{*}{\underset{G}{\Longrightarrow}} w \quad \Leftrightarrow \quad M \searrow w$$

Proof Of course, we could simply invoke Turing’s thesis and avoid having to describe M in detail. Alternatively, we can say that a TM M could simply generate all the strings over $V \cup T \cup \{\Longrightarrow\}$ in lexicographic order and, for each of them, check whether it represents a legal derivation of w. As soon as M finds such a derivation it can halt. Since lexicographic order is used, M is guaranteed to find such a derivation if one exists (i.e., we must use dovetailing!). □.

4.9 Context-sensitive languages

Definition An unrestricted grammar $G = (V, T, S, P)$ is a **context sensitive** grammar (CSG) iff all its productions are of the form $x \rightarrow y$ with $|x| \leq |y|$.

Note Of course this implies that CSGs cannot generate ϵ, but we can treat ϵ as a special case, as usual.

Definition A language L is a context-sensitive language (CSL) iff there is a CSG G such that $L \setminus \{\epsilon\} = L(G)$.

Theorem Given a CSL L there is a CSG $G = (V, T, S, P)$ such that $L \setminus \{\epsilon\} = L(G)$ and all the productions in P are of the form $xAy \rightarrow xvy$, where $A \in V$, $v \in (V \cup T)^+$, and $x, y \in (V \cup T)^*$.

Note This explain the name for these languages: x and y are the "context" in which it is legal to apply the (otherwise context-free) production $A \rightarrow v$.

Example We can show that $\{a^n b^n c^n : n \in \mathbb{N}\}$ is a CSL, since it is generated by the CSG with the following productions

$$S \rightarrow abc | aXbc, Xb \rightarrow bX, Xc \rightarrow Ybcc, bY \rightarrow Yb, aY \rightarrow aa | aaX$$

For example,

$$S \Longrightarrow aXbc \Longrightarrow abXc \Longrightarrow abYbcc \Longrightarrow aYbbcc \Longrightarrow aabbcc$$

Note that the productions are not in the "context-sensitive" format. However, we can transform them so that they are. For example, the effect of production $XB \rightarrow BX$ can be achieved by the following productions instead:

$$XB \rightarrow WB \quad WB \rightarrow WU \quad WU \rightarrow BU \quad BU \rightarrow BX$$

4.10 Linear-bounded automata

If we believe Church's thesis, we cannot extend the power of TMs in meaningful ways. We can then investigate what happens if we attempt to restrict TMs.

- If we allow only a fixed finite portion of the r/w tape, plus an unbounded read-only input tape, we obtain a NFA or DFA (depending on whether we allow nondeterministic moves or not). Recall that NFA and DFA have equivalent power.

- If we allow an unbounded r/w tape, but we force the TM to use it as a stack, we obtain a NPDA or a DPDA (depending on whether we allow nondeterministic moves or not). Recall that NPDA and DPDA are *not* equivalent.

- If we allow a **nondeterministic** TM to r/w only the portion of tape covered by the input, we obtain a **linear-bounded automaton** (LBA). One way to formalize this is to say that the input w is presented to the LBA enclosed in brackets "[" and "]", and that the transition function of the LBA satisfies $\forall p \in Q$, $\delta(p, [) \subseteq (Q \cup \{h\}) \times \{R\}$ and $\delta(p,]) \subseteq (Q \cup \{h\}) \times \{L\}$, where Q is the set of states of the LBA.

Definition A LBA $M = (Q, \Sigma \cup \{[,]\}, \delta, s)$ accepts the language $L(M) = \{w \in \Sigma : (s, [w\underline{]}) \overset{*}{\underset{M}{\vdash}} (h, [x\underline{a}y])\}$. Note that this implies $|w| = |xay|$.

Example The language $\{a^n b^n c^n : n \in \mathbb{N}\}$ is accepted by a LBA.

Example The language $\{a^{n!} : n \in \mathbb{N}\}$ is accepted by a LBA. We can simply attempt to divide the length of the string by 2, then the quotient by 3, and so on, until either we find that a division cannot be performed (we reject) or we find a quotient equal to one (and we accept).

Theorem L is a CSL iff there is a LBA that accepts L.

Proof Let's prove first that, if L is a CSL, then there is a LBA that accepts it. Observe that, given a CSG $G = (V, T, S, P)$ and $w \in L(G)$, every sentential form in the derivation of w has length at most $|w|$. Hence, by using a two-track tape LBA, we can store w on the first track and the "current" sentential form on the second track. Using nondeterminism, we can clone the LBA for every possible application of a production in the current sentential form, to obtain the "next" sentential form. If the sentential form should have length greater than $|w|$, the clone dies (i.e., it puts itself into an infinite loop) immediately. Every time a new sentential form is generated, it is compared with w on the first track; if they are equal the LBA halts. No clone will ever halt iff $w \notin L$.

For the reverse, we can observe that the same construction used to simulate a TM with an unrestricted grammar works, except that, since the LBA never goes beyond the "[" and "]" boundaries, there is no need for the corresponding productions that add or destroy blanks (the latter one would be illegal anyway because it violates the context-sensitive rule that the LHS must be no longer than the RHS in any production). □

Theorem If L is a CSL, then it is Turing-decidable.

Proof We simply need to observe that, given a CSG $G = (V, T, S, P)$ such that $L = L(G)$, the number of steps in a shortest derivation of $w \in L$ is bounded by a function of G and $|w|$. This is because, in

$$S = s_0 \Longrightarrow s_1 \Longrightarrow s_2 \Longrightarrow \cdots \Longrightarrow s_n = w$$

we know that $|s_i| \leq |s_{i+1}|$, and there are only $(|V \cup T|)^{|s|}$ different sentential forms of length $|s|$. Thus, the TM can find all the sentential forms of length 1 that can be derived from S, then all the sentential forms of length 2, and so on, until it generates all the sentential forms of length $|w|$ that can be derived from S and, if w is among them, halt with Y, or halt with N otherwise. In other words, the TM can detect cycles of the type

$$s_i \Longrightarrow s_{i+1} \Longrightarrow \cdots \Longrightarrow s_i,$$

because they are of bounded length, at most $(|V \cup T|)^{|s_i|}$. □

On the other hand, the following theorem shows that the reverse of the previous theorem is not true.

Theorem There exist $L \in \mathit{TDL}$ that is not a CSL.

Proof We use the idea of diagonalization. We can encode each CSG G in a string $\rho(G) \in \{0, 1\}^*$, using an encoding similar to that used as input for the UTM. Then, we can define a lexicographic order on $\{0, 1\}^+$ and define G_i to be the i-th CSG according to that order. The diagonal set is defined as

$$L = \{w_i : w_i \notin L(G_i)\}$$

	0	1	00	01	10	$\cdots$
G_0						
G_1						
G_2						
G_3						
$\cdots$						

Clearly, $L \in TDL$: given w, we compute its index i, that is, we find $w_i = w$. Then, we simply enumerate CSG encodings until we find the i-th one in lexicographic order, $\rho(G_i)$, and we test whether $w_i \notin L(G_i)$; all these tasks are algorithmic, thus they can be implemented so that they are guaranteed to complete. However, is we assume that L is a CSL, we reach a contradiction, because the CSG G^* that generates L cannot exist. If it existed, $G^* = G_d$ for some d, and

$$w_d \in L(G_d) \;\Leftrightarrow\; w_d \notin L \;\Leftrightarrow\; w_d \notin L(G_d) \qquad \square$$

Note: We defined LBA as nondeterministic automata. It is interesting to observe that it is not known whether the class of deterministic LBA is equivalent to (nondeterministic) LBA, or whether it is instead less powerful.

4.11 Recursive and recursively enumerable sets

Definition A language L (i.e., a countable set) is **recursively enumerable** iff there is an **enumeration** procedure for it, that is, a deterministic TM E that can produce the elements of L on the tape **in some order**, such that every element is eventually produced:

$$E = (Q, \Sigma, \delta, s) :\; \exists q_e \in Q \wedge \left(w \in L \;\Leftrightarrow (s, \#\underline{\#}) \overset{*}{\underset{E}{\vdash}} (q_e, \#w\underline{\#}x) \right)$$

Theorem If a language L is recursively enumerable, then it is recursively enumerable without repetition, that is, the TM does not produce the same string twice:

$$(s, \#\underline{\#}) \overset{i}{\underset{E}{\vdash}} (q_e, \#w\underline{\#}x) \;\wedge\; (s, \#\underline{\#}) \overset{j}{\underset{E}{\vdash}} (q_e, \#w\underline{\#}y) \;\Rightarrow\; i = j \quad \text{(and of course, } x = y\text{)}$$

Definition A language L (i.e., a countable set) is **recursive** iff there is an enumeration procedure for it, that is, a TM E that can produce the elements of L on the tape in lexicographic order:

$$E = (Q, \Sigma, \delta, s) :\; \exists q_e \in Q \;\wedge\; \left(w \in L \;\Leftrightarrow (s, \#\underline{\#}) \overset{*}{\underset{E}{\vdash}} (q_e, \#w\underline{\#}x) \right) \;\wedge$$

$$\left(\forall w_i, w_j \in L,\; (s, \#\underline{\#}) \overset{i}{\underset{E}{\vdash}} (q_e, \#w_i\underline{\#}x) \;\wedge\; (s, \#\underline{\#}) \overset{j}{\underset{E}{\vdash}} (q_e, \#w_j\underline{\#}x) \;\Rightarrow\; (i < j \;\Leftrightarrow\; w_i < w_j) \right)$$

where the comparison $w_i < w_j$ is performed according to lexicographic order.

Theorem L is recursively enumerable iff $L \in TAL$. L is recursive iff $L \in TDL$.

Proof TBD

4.12 The Post Correspondence problem

The Post Correspondence problem (PC) can be stated as follows. Given two sequences of n strings, $A = (w_1, \ldots, w_n)$ and $B = (v_1, \ldots, v_n)$, is there a $k \in \mathbb{N}^+$ and a sequence $(i_1, \ldots, i_k) \in \{1, \ldots, n\}^k$ such that

$$w_{i_1} \cdots w_{i_k} = v_{i_1} \cdots v_{i_k}$$

Example If $n = 4$, $w_1 = 11$, $w_2 = 111$, $w_3 = 100$, $w_4 = 0001$, $v_1 = 111$, $v_2 = 11$, $v_3 = 001$, and $v_4 = 111111$, the answer is "yes" because

$$w_1 w_3 w_2 = \underline{11}\ \underline{100}\ \underline{111} = v_1 v_3 v_2 = \underline{111}\ \underline{001}\ \underline{11}$$

PC is clearly semi-solvable, but we will show that it is not solvable. First we need to show that the Modified Post Correspondence problem (MPC) is not solvable, where MPC has the additional restriction that $i_1 = 1$, that is, the matching strings must start with w_1 and v_1, respectively.

To show that MPC is not solvable, we reduce the question $w \in L(G)$ for an unrestricted grammar G to it.

Given $G = (V, T, S, P)$ and $w \in T^+$, we create the sequences A and B as follows. Let $[,]$ be two new symbols not in $V \cup T$. Then,

	A	B
(w_1, v_1)	$[S \Longrightarrow$	$[$
$\forall a \in T$	a	a
$\forall A \in V$	A	A
	$\Longrightarrow$	$\Longrightarrow$
$\forall x \rightarrow y \in P$	y	x
(w_n, v_n)	$]$	$\Longrightarrow w]$

Example If $G = (\{S, A, B, C\}, \{a, b, c\}, S, \{S \rightarrow aABb \mid Bbb, Bb \rightarrow C, AC \rightarrow aac\})$ and $w = aaac$:

i	A	B
1	$[S \Longrightarrow$	$[$
2	a	a
3	b	b
4	c	c
5	A	A
6	B	B
7	C	C
8	S	S
9	$\Longrightarrow$	$\Longrightarrow$
10	$aABb$	S
11	Bbb	S
12	C	Bb
13	aac	AC
14	$]$	$\Longrightarrow aaac]$

and

$$\overbrace{\underbrace{[}_{v_1}\ \underbrace{S}_{v_{10}}\underbrace{\Longrightarrow}_{v_9}}^{w_1}\overbrace{\underbrace{a}_{v_2}\ \underbrace{A}_{v_5}\ \underbrace{Bb}_{v_{12}}}^{w_{10}}\overbrace{\underbrace{\Longrightarrow}_{v_9}}^{w_9}\overbrace{\underbrace{a}_{v_2}}^{w_2}\underbrace{\overbrace{A}^{w_5}\ \overbrace{C}^{w_{12}}}_{v_{13}}\underbrace{\overbrace{\Longrightarrow}^{w_9}\overbrace{a}^{w_2}\ \overbrace{aac}^{w_{13}}\ \overbrace{]}^{w_{14}}}_{v_{14}}$$

Clearly, a MPC built according to these rules has a solution iff $w \in L(G)$. Thus MPC is not decidable, since we have reduced the not decidable question "$w \in L(G)$?" to it.

We now show that PC is also not decidable by reducing MPC to PC. Note that reducing PC to MPC is trivial (we simply ask whether any of the n MPCs obtained by putting w_1 and v_1, then w_2 and v_2, up to w_n and v_n in first position and the others in any order has a solution), but does not prove anything.

Given an instance (A, B) of MPC with $A = (w_1, \dots, w_n)$ and $B = (v_1, \dots, v_n)$, define an instance (C, D) of PC as follows: $C = (y_0, \dots, y_{n+1})$ and $D = (z_0, \dots, z_{n+1})$ where

$$\forall i = 1, \dots, n,\ \ w_i = a_{i_1}a_{i_2}\cdots a_{i_{q_i}},\ \ v_i = a_{j_1}a_{j_2}\cdots a_{j_{r_i}},\ \ y_i = a_{i_1}\bigcirc a_{i_2}\bigcirc\cdots a_{i_{q_i}}\bigcirc,\ \ z_i = \bigcirc a_{j_1}\bigcirc a_{j_2}\cdots\bigcirc a_{j_{r_i}},$$

and

$$y_0 = \bigcirc y_1 \quad y_{n+1} = \triangle \quad z_0 = z_1 \quad z_{n+1} = \bigcirc\triangle$$

Clearly, because of the way we defined the PC, any solution of the PC must start with y_0 and z_0, since they are the only two strings such that one is a prefix of the other (provided w_1 is a prefix of v_1 or vice versa, which is of course a necessary condition for the MPC to have a solution). Hence, the PC has a solution iff the MPC has a solution.

4.13 Undecidable problems for CFLs

Using reduction from the PC problem, we now show that there are interesting problems for CFLs that are not decidable (they are clearly semi-decidable).

Theorem There is no algorithm to decide whether a generic CFG G is ambiguous.

Proof Remember that ambiguous means that there are two distinct derivation trees for a $W \in L(G)$. We reduce PC to this problem. Given $A = (w_1, \dots, w_n)$ and $B = (v_1, \dots, v_n)$ over an alphabet Σ, choose $b_1, \dots b_n \notin \Sigma$ and consider the languages

$$L_A = \{w_{i_1}\cdots w_{i_k}b_{i_k}\cdots b_{i_1} \ :\ k \in \mathbb{N}^+, i_1, \dots, i_k \in \{1, \dots, n\}\},$$

$$L_B = \{v_{i_1}\cdots v_{i_k}b_{i_k}\cdots b_{i_1} \ :\ k \in \mathbb{N}^+, i_1, \dots, i_k \in \{1, \dots, n\}\}.$$

Define the grammar

$$G = (\{S, S_A, S_B\}, \Sigma \cup \{b_1, \dots, b_n\}, P, S)$$

where P contains the productions

$$S \rightarrow S_A, \quad \forall i,\ S_A \rightarrow w_i S_A b_i | w_i b_i \quad S \rightarrow S_B, \quad \forall i,\ S_B \rightarrow v_i S_B b_i | v_i b_i$$

Clearly, $S_A \overset{*}{\Longrightarrow} x$ iff $x \in L_A$ and $S_B \overset{*}{\Longrightarrow} x$ iff $x \in L_B$ and $L(G) = L_A \cup L_B$.

But, if $S_A \overset{*}{\Longrightarrow} x$, there is only one way to do so, and if $S_B \overset{*}{\Longrightarrow} x$, there is only one way to do so. So, G is ambiguous iff there is an $x \in L(G)$ such that $S \Longrightarrow S_A \overset{*}{\Longrightarrow} x$ and $S \Longrightarrow S_B \overset{*}{\Longrightarrow} x$, that is, iff there is an $x \in L(G)$ such that

$$x = w_{i_1} \cdots w_{i_k} b_{i_k} \cdots b_{i_1} = v_{i_1} \cdots v_{i_k} b_{i_k} \cdots b_{i_1}$$

that is, iff

$$w_{i_1} \cdots w_{i_k} = v_{i_1} \cdots v_{i_k}$$

that is, iff the PC problem has a solution.

Since PC is not decidable, so is the determination of whether a CFG G is ambiguous. □

Theorem Given two generic CFGs G_1 and G_2, there is no algorithm to determine whether $L(G_1) \cap L(G_2) = \emptyset$.

Proof Again, we reduce PC to this problem. Given $A = (w_1, \ldots, w_n)$ and $B = (v_1, \ldots, v_n)$ over an alphabet Σ, choose $b_1, \ldots b_n \notin \Sigma$ and define two CFGs

$$G_A = (S_A, \Sigma \cup \{b_1, \ldots, b_n\}, P_A, S_A), \quad P_A = \{S_A \rightarrow w_i S_A b_i | w_i b_i \ : \ i = i, \ldots, n\},$$

$$G_B = (S_B, \Sigma \cup \{b_1, \ldots, b_n\}, P_B, S_B), \quad P_B = \{S_B \rightarrow v_i S_B b_i | v_i b_i \ : \ i = i, \ldots, n\}.$$

Clearly, $L(G_A) \cap L(G_B)$ contains an element iff the grammar G defined in the previous theorem is ambiguous, that is, iff the PC instance with $A = (w_1, \ldots, w_n)$ and $B = (v_1, \ldots, v_n)$ has a solution. □

Theorem $\overline{L(G_A)}$ and $\overline{L(G_B)}$ are also CFLs.

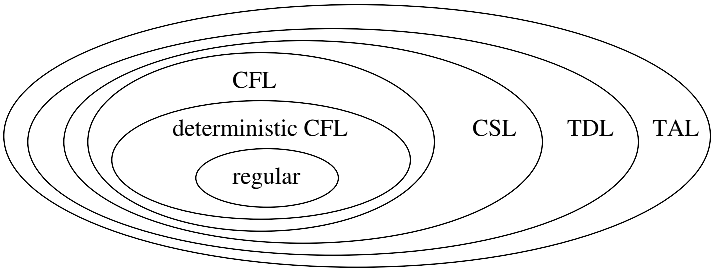
Proof We can build an NPDA, indeed even a deterministic PDA, that accepts them.

Theorem Given two generic CFGs G_1 and G_2, there is no algorithm to determine whether $L(G_1) = L(G_2)$.

Proof Let $L(G_1)$ be $\overline{L(G_A)} \cup \overline{L(G_B)}$, which is a CFL because $\overline{L(G_A)}$ and $\overline{L(G_B)}$ are CFLs and CFLs are closed under union. Let $L(G_2)$ be $(\Sigma \cup \{b_1, ..., b_n\})^*$, which is regular, thus a CFL. Then, $L(G_1) = \overline{L(G_A)} \cup \overline{L(G_B)} = \overline{L(G_A) \cap L(G_B)}$ is different from $L(G_2)$ iff the intersection of $L(G_A)$ and $L(G_B)$ contains an element, i.e., if the PC has a solution.

4.14 The language hierarchy

We can then observe that the following hierarchy holds:



Note: we do not know whether deterministic CSLs are strictly between CFLs and CSLs, or whether they coincide with the CSLs.