WMx

Warwick Mathematics Exchange

CS259

Formal Languages

2024, February 19th

Desync, aka The Big Ree

Contents

Introduction

A *formal language* is a set of words with letters selected from a fixed alphabet, and formed according to a set of rules called a formal grammar. In computational complexity theory, formal languages can encode decision problems, and hence provide a way of comparing the relative strength of various models of computation by checking which languages they are able to parse. In logic, formal languages can be used to represent the syntax of axiomatic and deductive systems, and hence mathematics itself can be reduced to the manipulation of these formal languages.

Disclaimer: I make *absolutely no guarantee* that this document is complete nor without error. In particular, any content covered exclusively in lectures (if any) will not be recorded here. This document was written during the 2023 academic year, so any changes in the course since then may not be accurately reflected.

Notes on formatting

New terminology will be introduced in *italics* when used for the first time. Named theorems will also be introduced in *italics*. Important points will be **bold**. Common mistakes will be underlined. The latter two classifications are under my interpretation. YMMV.

> Content not taught in the course will be outlined in the margins like this. Anything outlined like this is not examinable, but has been included as it may be helpful to know alternative methods to solve problems.

The table of contents above, and any inline references are all hyperlinked for your convenience.

History

First Edition: 2024-02-16*
Current Edition: 2024-02-19

Authors

This document was written by R.J. Kit L., a maths student. I am not otherwise affiliated with the university, and cannot help you with related matters.

Please send me a PM on Discord @Desync#6290, a message in the WMX server, or an email to Warwick.Mathematics.Exchange@gmail.com for any corrections. (If this document somehow manages to persist for more than a few years, these contact details might be out of date, depending on the maintainers. Please check the most recently updated version you can find.)

If you found this guide helpful and want to support me, you can buy me a coffee!

(Direct link for if hyperlinks are not supported on your device/reader: ko-fi.com/desync.)

*Storing dates in big-endian format is clearly the superior option, as sorting dates lexicographically will also sort dates chronologically, which is a property that little and middle-endian date formats do not share. See ISO-8601 for more details. This footnote was made by the computer science gang.

1 Introduction

An *alphabet* is any non-empty set of symbols, often denoted by Σ. A *word* over an alphabet is a finite sequence of letters. Note that the empty string, denoted by ε, is a word.

The *Kleene star* $(-)^*$, is a unary operation on sets of symbols defined as follows. Given a set V, we define $V^0 = \{\varepsilon\}$, where ε is the *empty word* with *length* $|\varepsilon| = 0$, then recursively define

$$V^{i+1} = \{wv : w \in V^i, v \in V\}$$

for each $i > 0$. That is, V^i is the set of strings that can be formed by concatenating i strings in V together. Then, the Kleene star on V is given by

$$V^* = \bigcup_{i \geq 0} V^i$$

That is, V^* is the set of all possible words over V.

Note that if V is countable, then V^* is the countable union of countable sets and is hence countable. Also note that a set of strings has a monoidal structure under concatenation, so the Kleene star of a set V is exactly the free monoid on V.

Then, a *formal language* over an alphabet Σ is a set $L \subseteq \Sigma^*$. Note that there is no requirement that this set be non-empty, so $L = \emptyset \subseteq \Sigma^*$ is a language, called the *empty language*. Also, by definition, we have that the empty word ε is in Σ^* for any alphabet Σ. Note that $L' = \{\varepsilon\}$ is a non-empty language – it contains the empty word.

Given a language $L \subseteq \{0,1\}^*$, we may interpret it as a decision problem by deciding whether a given binary string belongs to L. Conversely, assuming a fixed efficient encoding, we can encode any decision problem as a formal language by taking all strings representing yes-instances of the decision problem to be in our language.

Theorem 1.1. *There are functions $f : \mathbb{N} \to \{0,1\}$ that are not computable by any algorithm.*

Proof. Algorithms are finite sequences of finite alphabets of possible instructions, so there are only countably many algorithms possible. Conversely, the set of functions $\mathbb{N} \to \{0,1\}$ has size $2^{\aleph_0} = \mathcal{P}(\mathbb{N}) = \mathfrak{c}$, which is uncountable. ■

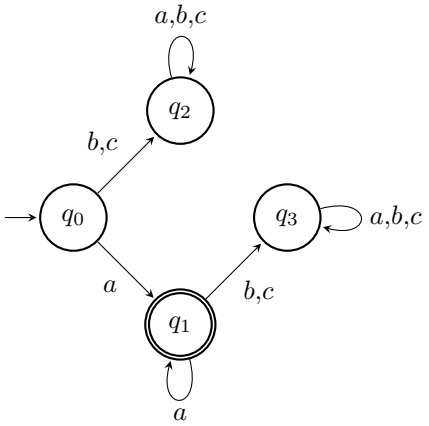
2 Regular Languages

2.1 Deterministic Finite Automata

A *deterministic finite automata* (DFA) is an abstract machine that either *accepts* or *rejects* a given word by reading through the symbols in the string and deterministically transitioning between different internal states depending on the current symbol and its current state. Formally, a DFA is given by 5-tuple (Q,Σ,q_0,F,δ), consisting of

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

We can visually represent a DFA as either a *state diagram*, or a *state transition table*:



	δ	a	b	c
$\rightarrow$	q_0	q_1	q_2	q_2
$*$	q_1	q_1	q_3	q_3
	q_2	q_2	q_2	q_2
	q_3	q_3	q_3	q_3

On the left, the transition function is given by the labelled arrows between states; the initial state is marked with a trailing arrow; and any final states are marked by two concentric circles.

On the left, the table on the right simply details the transition function, with the initial state marked with an arrow, and the accepting states marked with an asterisk.

In either case, we *run* the machine on a string by starting at the initial state; consuming the first character from the string; moving to the next state given by the transition function; then iterating this process until the string is empty. If the DFA is in an accepting state when the empty string is reached, then the word is *accepted*, and otherwise *rejected*.

Example. We run the string abc on the above DFA using the state diagram. We begin at q_0. The first character is a, so we proceed to q_1 with the remaining string bc. The next character is then b, so we move to q_3 with remaining string c. The next character is c, so we remain at q_3, and now the string is empty. q_3 is not an accepting state, so the string abc is rejected by this DFA. $\triangle$

Example. We list the outputs of some more strings:

Input	Output
a	Accept
aa	Accept
aab	Reject
b	Reject
c	Reject
bca	Reject

More concisely, this DFA accepts exactly the strings that consist solely of the character a. $\triangle$

Let $M = (Q,\Sigma,q_0,F,\delta)$ be a DFA, and let $s = s_1 s_2 \cdots s_n$ be a string, where $s_i \in \Sigma$ for each i. We define the *run* of M on s as follows:

- The run of M on ε is the state q_0.
- The run of M on the non-empty word s is the sequence of states $(r_i)_{i=0}^n$ given recursively by

$$r_i = \begin{cases} q_0 & i = 0 \\ \delta(r_{i-1},s_i) & i > 0 \end{cases}$$

The run of M on a word s is called an *accepting run* if the last state in the run is an accepting state of M, and we say that a word s is *accepted* or *recognised* by M if the run of M on s is an accepting run. The set of strings that M accepts forms a language over Σ called the language *accepted* or *recognised* by

M, denoted by $L(M)$.

$$L(M) \coloneqq \{s \in \Sigma^* : \text{the run of } M \text{ on } s \text{ is an accepting run}\}$$

Note that, for M to accept a language L', it must not only accept only the words in L', but also reject every word in $\Sigma^* \setminus L$.

The transition function $\delta : Q \times \Sigma \to Q$ of a DFA details the change in state upon reading a single symbol. We can expand this function to the *extended transition function* $\hat{\delta}$ that expresses the change in state upon reading an entire *string*.

Formally, we recursively define the extended transition function $\hat{\delta} : Q \times \Sigma^* \to Q$ as follows:

- For every state $q \in Q$, we have $\hat{\delta}(q,\varepsilon) = q$;
- For every state $q \in Q$ and word $s \in \Sigma^*$ with $s = wa$, $w \in \Sigma^*$, $a \in \Sigma$, we have $\hat{\delta}(q,s) = \delta\big(\hat{\delta}(q,w),a\big)$.

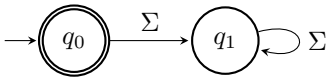
Using the extended transition function we can also write the language recognised by a DFA M as

$$L(M) = \big\{s \in \Sigma^* : \hat{\delta}(q_0,s) \in F\big\}$$

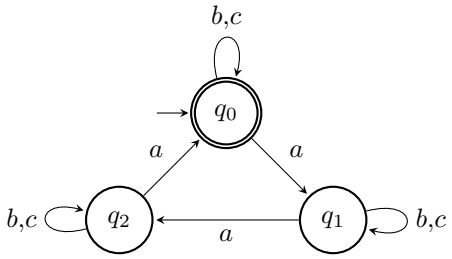
A language L is *regular* if it is accepted by some DFA.

Example.

- The empty language $L = \emptyset$ is regular; it is accepted by any DFA with $F = \emptyset$.
- The language $L = \Sigma^*$ is regular; it is accepted by any DFA with $F = Q$.
- The language $L = \{\varepsilon\}$ is regular; it is accepted by the DFA



- The language $L \subseteq \{a,b,c\}^*$ defined by $L = \{s \in \Sigma^* : \text{the number of } a\text{'s in } s \text{ is divisible by } 3\}$ is regular; it is accepted by the DFA



△

2.2 Closure Properties of Regular Languages

Because languages are sets, ordinary set operations also apply to languages. Regular languages are *closed* under certain operations, in the sense that the resulting language is also regular.

Regular languages are closed under

- **Complementation**:

  If L is regular, then $\overline{L} = \Sigma^* \setminus L$ is regular; if $L = L(M)$ is accepted by $M = (Q,\Sigma,q_0,F,\delta)$, then $\overline{L} = L(M')$ is accepted by $M' = (Q,\Sigma,q_0,Q \setminus F,\delta)$.

- **Intersection**:

  If L_1 and L_2 are regular, then $L_1 \cap L_2$ is regular. The idea here is to run the DFAs for L_1 and L_2 in parallel by using the Cartesian product of states and applying the transition functions pointwise, and accepting if and only if both original DFAs accept.

  If $L_1 = L(M_1)$ and $L_1 = L(M_1)$ with $M_1 = (Q_1,\Sigma,q_1,F_1,\delta_1)$, $M_2 = (Q_2,\Sigma,q_2,F_2,\delta_2)$, then $L_1 \cap L_2$ is accepted by the DFA

  $$M = \big(Q_1 \times Q_2,\Sigma,(q_1,q_2),F_1 \times F_2,\delta\big)$$

  with $\delta : (Q_1 \times Q_2) \times \Sigma \to Q_1 \times Q_2$ defined pointwise:

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

- **Union**:

  If L_1 and L_2 are regular, then $L_1 \cup L_2$ is regular. This follows from De Morgan's laws:

  $$L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$$

  and the closure properties of complementation and intersection, but we can also give an explicit DFA that recognises this union. As before, the idea is to run the DFAs for L_1 and L_2 in parallel, this time accepting if either of the original DFAs accept.

  If $L_1 = L(M_1)$ and $L_1 = L(M_1)$ with $M_1 = (Q_1,\Sigma,q_1,F_1,\delta_1)$, $M_2 = (Q_2,\Sigma,q_2,F_2,\delta_2)$, then $L_1 \cap L_2$ is accepted by the DFA

  $$M = \big(Q_1 \times Q_2,\Sigma,(q_1,q_2),(F_1 \times Q_2) \cup (Q_1 \times F_2),\delta\big)$$

  where δ is the same as for intersections.

- **Relative difference**:

  If L_1 and L_2 are regular, then $L_1 \setminus L_2$ is regular. This follows from closure under complementation and intersection, as,

  $$L_2 \setminus L_2 = L_1 \cap \overline{L_2}$$

- **Concatenation**:

  If L_1 and L_2 are regular, then $L_1 \cdot L_2 = \{wv : w \in L_1,v \in L_2\}$ is regular. Proof requires more machinery than we currently have.

- **Kleene star**:

  If L is regular, then L^* is regular. Follows from regularity of concatenation and unions:

  $$L^* = \{\varepsilon\} \cup L \cup (L \cdot L) \cup (L \cdot L \cdot L) \cup \cdots$$

Note that L_1 and $L_1 \setminus L_2$ being regular does not imply that L_2 is regular. For instance, if $L_1 = \emptyset$, then $L_1 \setminus L_2 = \emptyset$, regardless of the regularity of L_2.

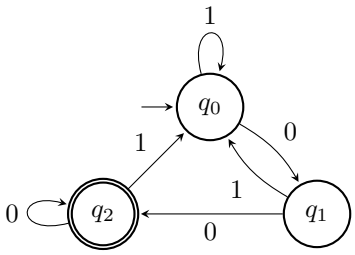
2.3 Non-Deterministic Finite Automata

Are regular languages closed under *reversal*? That is, if L is regular, then is the language

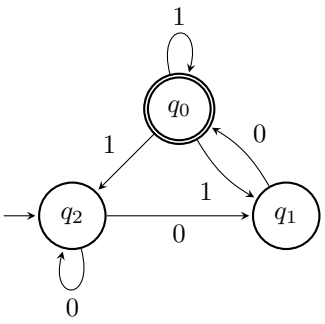
$$L^{\text{rev}} = \{w : w \text{ is the reverse of a string in } L\}$$

regular?

Consider the language $L = \{\text{binary strings ending with } 00\}$, accepted by the DFA



To build a DFA that accepts L^{rev}, we'd might think to reverse all arrows, then swap the start and accepting states:



However, this state diagram now has states with multiple exiting arrows labelled with the same symbol, and some states do not have an exiting arrow for every symbol in the alphabet. Moreover, if we had multiple accepting states, then we would also have multiple initial state in this reverse diagram. So, this state diagram does not describe a DFA. We instead extend the definition of an DFA to a *non-deterministic finite automata* (NFA).

A DFA must have exactly one transition out of a state for each symbol in the alphabet, so each word has a unique run. In contrast, an NFA may have multiple or no transitions out of a state for any given symbol, so an NFA may have multiple choices at each step, and the final state is not *determined* solely by the start state and input word, instead having a branching tree structure.

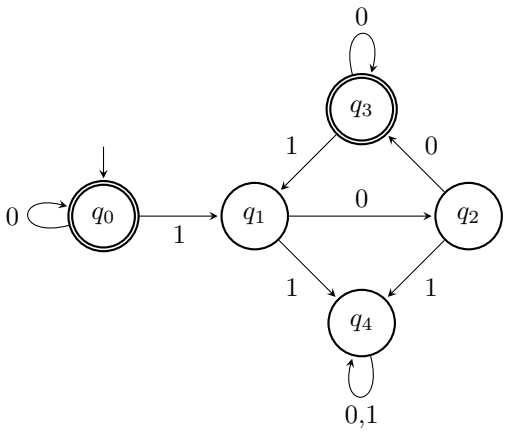
An NFA may also have *ε-transitions* – transitions that do not consume any input. This allows us to deal with multiple initial states by adding a new state to be initial, then adding ε-transitions from this state to all the previous initial states.

Formally, an NFA is a 5-tuple (Q,Σ,q_0,F,δ), consisting of

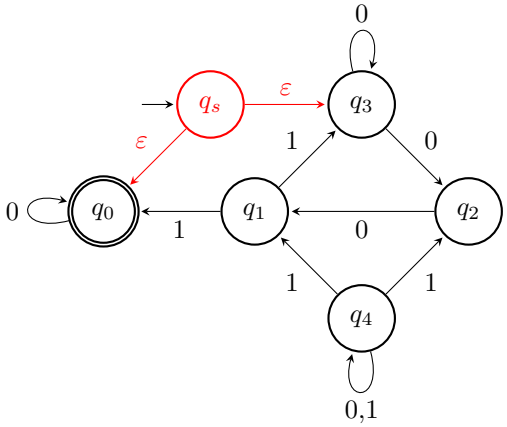
- a finite set Q of states;
- a finite alphabet Σ;
- an initial state $q_0 \in Q$ in which to start the computation;
- a set $F \subseteq Q$ of accepting or final states;
- and a transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$.

The first four entries are the same as for DFAs, but because an NFA may have multiple or no transitions out of a state for any given symbol, the transition function instead returns a set of states in $\mathcal{P}(Q)$, and we also include ε in the domain to account for ε-transitions, so the transition function is then a function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$. We also write Σ_ε to denote $\Sigma \cup \{\varepsilon\}$.

Example. Consider the language $L \subseteq \{0,1\}^*$ defined by $L = \{\text{every 1 is followed by 00}\}$, accepted by the DFA



By reversing the arrows and adding a new state equipped with ε-transitions to the two previous accepting states, we obtain a state diagram of an NFA that accepts $L^{\text{rev}} = \{\text{every 1 is preceded by 00}\}$:



We can also represent this NFA as a state transition table:

	δ	0	1	ε
$\rightarrow$	q_s	$\emptyset$	$\emptyset$	$\{q_0,q_3\}$
$*$	q_0	$\{q_0\}$	$\emptyset$	$\emptyset$
	q_1	$\emptyset$	$\{q_0,q_3\}$	$\emptyset$
	q_2	$\{q_1\}$	$\emptyset$	$\emptyset$
	q_3	$\{q_2,q_3\}$	$\emptyset$	$\emptyset$
	q_4	$\{q_4\}$	$\{q_1,q_2,q_4\}$	$\emptyset$

Note that every entry in the table is a set, unlike for a DFA.
We can also see that there is no way to enter state q_4, so we may remove it from the NFA and simplify the state diagram/transition table. $\triangle$

2.4 ε-Closure

What does the extended transition function of an NFA look like? The ordinary transition function already returns sets of states – unlike a DFA, which is *deterministic* and returns a single state – but we also have to deal with ε-transitions at every step in an NFA. For this, we define the ε-closure function,

$\textsc{eclose} : Q \to \mathcal{P}(Q)$.

Informally, given a state q, $\textsc{eclose}(q)$ is the set of states that can be reached from q by following ε-transitions alone (including taking no ε-transitions, so $q \in \textsc{eclose}(q)$). Formally, given a state q, $\textsc{eclose}(q)$ is the minimal set such that

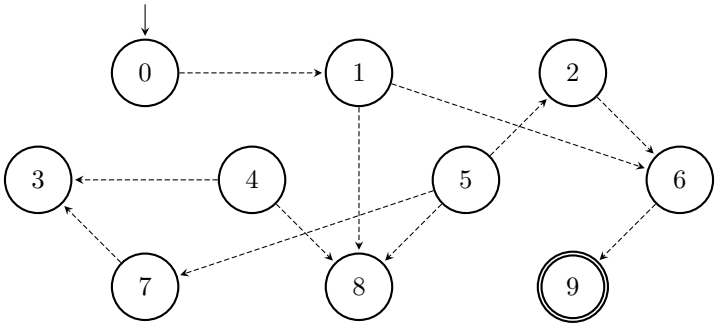
- $q \in \textsc{eclose}(q)$;
- $\forall p,r \in Q, \big(p \in \textsc{eclose}(q) \land r \in \delta(p,\varepsilon)\big) \to r \in \textsc{eclose}(q)$.

$\textsc{eclose}$ can be naturally extended to sets of states: given a set $X \subseteq Q$, we define

$$\textsc{eclose}(X) = \bigcup_{x \in X} \textsc{eclose}(x)$$

Note that the nullary union is empty, so $\textsc{eclose}(\emptyset) = \emptyset$.

Example. In the following, dashed arrows represent ε-transitions.



$\textsc{eclose}(0) = \{0,1,6,8,9\}$
$\textsc{eclose}(1) = \{1,6,8,9\}$
$\textsc{eclose}(2) = \{2,6,9\}$
$\textsc{eclose}(3) = \{3\}$
$\textsc{eclose}(4) = \{3,4,8\}$
$\textsc{eclose}(5) = \{2,3,5,6,7,8,9\}$
$\textsc{eclose}(6) = \{6,9\}$
$\textsc{eclose}(7) = \{3,7\}$
$\textsc{eclose}(8) = \{8\}$
$\textsc{eclose}(9) = \{9\}$

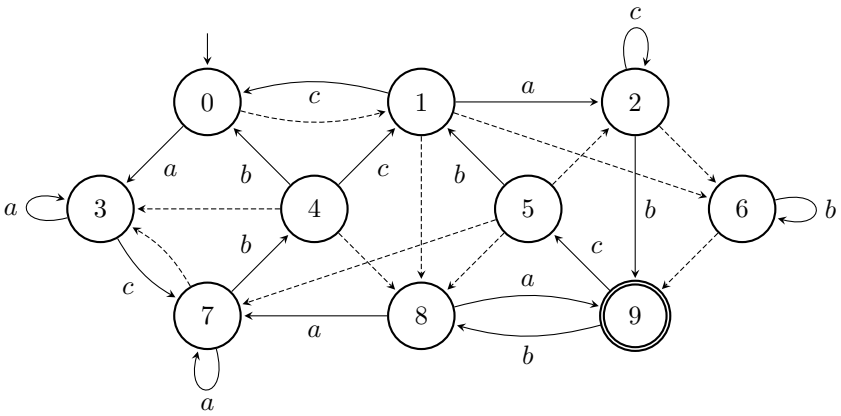
$\triangle$

The extended transition function $\hat{\delta}$ for an NFA $N = (Q,\Sigma,q_0,F,\delta)$ is then a function $\delta : Q \times \Sigma^* \to \mathcal{P}(Q)$ defined as follows:

- For every state $q \in Q$, we have $\hat{\delta}(q,\varepsilon) = \textsc{eclose}(q)$;
- For every state $q \in Q$ and word $s \in \Sigma^*$ with $s = wa$, $w \in \Sigma^*$, $a \in \Sigma$, we have

$$\hat{\delta}(q,s) = \textsc{eclose}\left(\bigcup_{p \in \hat{\delta}(q,w)} \delta(p,a)\right)$$

Example. The ε-transitions (and hence ECLOSE sets on singletons) are the same as in the previous diagram.



△

- $$\begin{aligned}\hat{\delta}(0,a) &= \text{ECLOSE} \bigcup_{p\in\hat{\delta}(0,\varepsilon)} \delta(p,a)\\ &= \text{ECLOSE} \bigcup_{p\in\text{ECLOSE}(0)} \delta(p,a)\\ &= \text{ECLOSE}\big(\delta(0,a)\cup\delta(1,a)\cup\delta(6,a)\cup\delta(8,a)\cup\delta(9,a)\big)\\ &= \text{ECLOSE}\big(\{3\}\cup\{2\}\cup\emptyset\cup\{7,9\}\cup\emptyset\big)\\ &= \text{ECLOSE}\big(\{2,3,7,9\}\big)\\ &= \bigcup_{z\in\{2,3,7,9\}} \text{ECLOSE}(x)\\ &= \{2,6,8,9\}\cup\{3\}\cup\{3,7\}\cup\{9\}\\ &= \{2,3,6,7,8,9\}\end{aligned}$$

- $$\begin{aligned}\hat{\delta}(0,aa) &= \text{ECLOSE} \bigcup_{p\in\hat{\delta}(0,a)} \delta(p,a)\\ &= \text{ECLOSE} \bigcup_{p\in\{2,3,6,7,8,9\}} \delta(p,a)\\ &= \text{ECLOSE}\big(\delta(2,a)\cup\delta(3,a)\cup\delta(6,a)\cup\delta(7,a)\cup\delta(8,a)\cup\delta(9,a)\big)\\ &= \text{ECLOSE}\big(\emptyset\cup\{3\}\cup\emptyset\cup\{7\}\cup\{7,9\}\cup\emptyset\big)\\ &= \text{ECLOSE}\big(\{3,7,9\}\big)\\ &= \bigcup_{z\in\{3,7,9\}} \text{ECLOSE}(x)\\ &= \{3\}\cup\{3,7\}\cup\{9\}\\ &= \{3,7,9\}\end{aligned}$$

- $\hat{\delta}(0,b) = \{6,8,9\}$ $\qquad$ $\hat{\delta}(1,a) = \{2,3,6,7,8,9\}$
  $\hat{\delta}(0,c) = \{0,1,6,8,9\}$ $\qquad$ $\hat{\delta}(2,b) = \{2,6,9\}$
  $\hat{\delta}(8,a) = \{3,7,9\}$ $\qquad$ $\hat{\delta}(3,c) = \emptyset$
  $\hat{\delta}(9,c) = \{2,3,5,6,7,8,9\}$ $\qquad$ $\hat{\delta}(4,a) = \{1,3,6,7,8,9\}$

2.5 Languages Recognised by NFA

Previously, we defined the language $L(M)$ recognised by a DFA M to be the set of words accepted by M. That is,

$$\begin{aligned} L(M) &\coloneqq \{s \in \Sigma^* : \text{the run of } M \text{ on } s \text{ is an accepting run}\} \\ &= \left\{s \in \Sigma^* : \hat{\delta}(q_0,s) \in F\right\} \end{aligned}$$

However, unlike a DFA, which is *deterministic* and always returns the same output, the run of an NFA on the same word may be different across several computations.

Let $N = (Q,\Sigma,q_0,F,\delta)$ be an NFA, and let s be a string. *A* run of N on s is a sequence of states $(r_i)_{i=1}^n$ such that

- $r_0 = q_0$;
- There exists a decomposition $s = s_1 s_2 \cdots s_n$, with $s_i \in \Sigma \cup \{\varepsilon\}$ for each i, such that for each $i > 0$, $r_i \in \delta(r_{i-1},s_i)$.

Then, an NFA N *accepts* or *recognises* a word s if there exists *some* accepting run.

Let $N = (Q,\Sigma,q_0,F,\delta)$ be an NFA. Then, the language $L(N)$ accepted or recognised by N is defined by

$$L(N) \coloneqq \{s \in \Sigma^* : \textit{some} \text{ run of } N \text{ on } s \text{ is an accepting run}\}$$

which we can again write in terms of the extended transition function:

$$= \left\{s \in \Sigma^* : \hat{\delta}(q_0,s) \cap F \neq \emptyset\right\}$$

Because the extended transition function returns the set of possible states after reading a word, we just check that it has non-empty intersection with the set of accepting states.

2.6 The Subset Construction

Are NFAs more powerful than DFAs?

Firstly, what do we even mean by "more powerful"? Intuitively, a computer is "more powerful" than a simple pocket calculator, but how do we formalise this notion? We might notice that a computer can have a calculator application within it – so a computer can do every task a calculator can. This shows that a computer is at least as powerful as a calculator. Importantly, to make this comparison strict, we note that there are tasks that a computer can do that a calculator cannot.

Given two computational models A and B, we say that A is *more powerful* or *expressive* than B if the class of languages recognised by A is a strict superset of the class of languages accepted by B. Note that it may be the case that two distinct computational models are incomparable under this relation if the class of languages they accept are not supersets of each other in either direction.

Clearly, every DFA is an NFA, as an NFA is a relaxation of the requirements of a DFA, so NFAs are at least as powerful as DFAs. However, are they strictly more powerful? It turns out that, for any NFA, we may *determinise* it and construct an equivalent DFA that recognises precisely the same language via the *subset* or *powerset construction*.

When a DFA is run on a word, we just keep track of a single state; that is, the state q_i that is reached upon reading a prefix of the input string, that can then be overwritten by the state that is reached upon reading the next symbol s.

In contrast, when running an NFA, we need to keep track of the set of all states that could be reached after seeing the same prefix, according to the non-deterministic choices made by the automaton. If, however, after a certain prefix has been read, a set S of states can be reached, then the set of symbols

reachable upon reading the next symbol s is a deterministic function of S and s. That is, while the states reached at each step in an individual run is non-deterministic, the set of states reachable at each step over all possible runs is fully deterministic, and as such, traversing sets of reachable states in this way describes the action of a DFA. This is the strategy of our construction.

Let $N = (Q,\Sigma,q_0,F,\delta)$ be the NFA to be determinised. Then, the DFA $M = (Q',\Sigma,q_0',F',\delta')$ defined by

- $Q' = \mathcal{P}(Q)$;
- $q_0' = \textsc{eclose}(q_0)$;
- $F' = \{X \subseteq Q : X \cap F \neq \emptyset\}$;
- $\delta'(X,a) = \bigcup_{x \in X} \textsc{eclose}\big(\delta(x,a)\big)$
  $= \Big\{z : \exists x \in X : z \in \textsc{eclose}\big(\delta(x,a)\big)\Big\}$

accepts the same language.

Theorem 2.1. *For all $s \in \Sigma^*$, $\hat{\delta}(q_0,s) = \hat{\delta}'(q_0',s)$.*

Proof. We induct on $|s|$. If $|s| = 0$, then $\hat{\delta}(q_0,s) = \hat{\delta}(q_0,\varepsilon) = \textsc{eclose}(q_0) = q_0' = \delta'(q_0',\varepsilon) = \hat{\delta}'(q_0',s)$. Otherwise, suppose $s = wa$ with $w \in \Sigma^*$, $a \in \Sigma$. Then,

$$\begin{aligned}
\hat{\delta}(q_0,s) &= \bigcup_{p \in \hat{\delta}(q_0,w)} \textsc{eclose}\big(\delta(p,a)\big) \\
&= \bigcup_{p \in \hat{\delta}'(q_0',w)} \textsc{eclose}\big(\delta(p,a)\big) \\
&= \delta'\big(\hat{\delta}(q_0',w),a\big) \\
&= \hat{\delta}(q_0',s)
\end{aligned}$$

■

2.7 Regular Expressions

A *regular expression*, *regex*, or a *pattern*, over an alphabet is a construction that specifies or *matches* a language over that alphabet. Regular expressions are defined recursively as follows:

Given an alphabet Σ, the following constants are *basic* regular expressions:

- (Empty set) $\emptyset$ is a valid regular expression, matching the empty language – $L(\emptyset) = \emptyset$;
- (Empty string) ε is a valid regular expression, matching the language containing the empty string – $L(\varepsilon) = \{\varepsilon\}$;
- (Literal character) $a \in \Sigma$ is a valid regular expression, matching the language containing only the character – $L(a) = \{a\}$;

and given two regular expressions R and S, we have:

- (Concatenation) $R \cdot S$, or RS, is a regular expression, matching the set of strings that can be obtained by concatenating a string accepted by R with a string accepted by S – $L(R \cdot S) = L(R) \cdot L(S)$;
- (Union) $R + S$, or $R|S$, is a regular expression, matching the union of the sets matched by R and S – $L(R + S) = L(R) \cup L(S)$;

- (Kleene Star) R^* is a regular expression, matching the smallest superset of the set matched by R that contains ε and is closed under concatenation – $L(R^*) = L(R)^*$;

In decreasing order, these operations have precedence *, $\cdot$, $+$.

Example. Let $\Sigma = \{a,b\}$ and $R = (a + b)^*$ be a regular expression over Σ. Intuitively, $(a + b)$ matches "a" or "b", so $L(R) = \Sigma^*$, but we can also unfold the definition algebraically:

$$\begin{aligned} L(R) &= L\big((a + b)^*\big) \\ &= L\big((a + b)\big)^* \\ &= \big(L(a) \cup L(b)\big)^* \\ &= \big(\{a\} \cup \{b\}\big)^* \\ &= \Sigma^* \end{aligned}$$

△

Example. If $\Sigma = \{a,b\}$, and $R = (a + b)^*(a + bb)$, then

$$\begin{aligned} L(R) &= L\big((a + b)^*(a + bb)\big) \\ &= L\big((a + b)^*\big)L\big((a + bb)\big) \\ &= \Sigma^* L\big((a + bb)\big) \\ &= \{\text{all strings over } \{a,b\} \text{ that end with } a \text{ or } bb\} \end{aligned}$$

△

Example. If $\Sigma = \{a,b\}$, and $R = (aa)^*(bb)^*b$, then

$$\begin{aligned} L(R) &= L\big((aa)^*(bb)^*b\big) \\ &= L\big((aa)\big)^* L\big((bb)\big)^* L\big(b\big) \\ &= \{\text{all strings over } \{a,b\} \text{ with an even number of } a\text{'s followed by an odd number of } b\text{'s}\} \end{aligned}$$

△

2.8 Generalised Non-Deterministic Finite Automata

Using regular expressions, we can define a *generalised non-deterministic finite automaton* (GNFA). A GNFA is a variation of a NFA where each transition may be any regular expression, and there may only be one transition between any two states, unlike a DFA or an NFA, which may have multiple such transitions. Furthermore, A GNFA must have exactly one initial state and one accepting state, and these states must be distinct.

A GNFA is a 5-tuple $(Q,\Sigma,q_{\text{start}},q_{\text{accept}},\delta)$, consisting of

- a finite set Q of states;
- a finite alphabet Σ;
- the start state, $q_{\text{start}} \in Q$;
- the accept state, $q_{\text{accept}} \in Q$;
- and a transition function $\delta : \big(Q \setminus \{q_{\text{accept}}\}\big) \times \big(Q \setminus \{q_{\text{start}}\}\big) \to \mathcal{R}$, where $\mathcal{R}$ is the set of all regular expressions over Σ.

2.9 Languages Recognised by Regular Expressions

As suggested by the name, regular expressions recognise precisely the class of regular languages, so NFAs, DFAs, and regular expressions are equally as expressive.

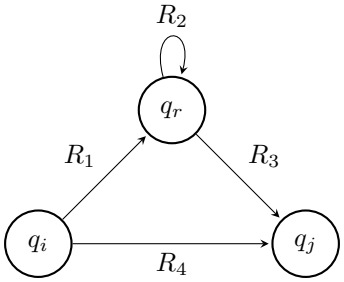
Theorem 2.2. *A language is regular if and only if it is described by a regular expression.*

Proof. Given a regular language $L = L(M)$ accepted by a DFA, we may convert this DFA into a GNFA as follows:

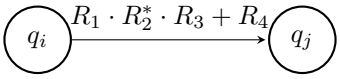
1. Add a new start state with an ε-transition to the previous start state.
2. Add a new accepting state with ε-transitions from the previous accepting states to this state.
3. Any transitions with multiple labels may be replaced with a transition labelled with the union of the previous labels.
4. For any ordered pair of states that do not end at the start state nor begin at the accept state and are disconnected, add a new transition between them labelled with $\emptyset$.

This GNFA may then be converted into a regular expression as follows:

1. If there are only two states, then we are done, as these must be the unique start and accepting states, and transition connecting them is a regular expression.
2. Otherwise, select some state $q_r \in Q \setminus \{q_{\text{start}},q_{\text{end}}\}$. Then, for all $(q_i,q_j) \in \big(Q \setminus \{q_{\text{start}},q_r\}\big) \times \big(Q \setminus \{q_{\text{end}},q_r\}\big)$, we may replace the transitions



by the single edge

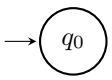


Once this has been done, we may remove q_r from the diagram, then pick a new state to be q_r, until the diagram has only the initial and accepting state remaining.

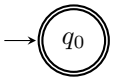
For the reverse implication, suppose we have a regular expression R that accepts $L(R)$. Then, we can construct a NFA N such that $L(N) = L(R)$.

We give an NFA for each of the basic regular expressions:

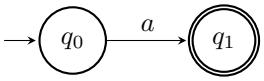
1. If $R = \emptyset$, then $L(R) = \emptyset$ is recognised by the NFA $N = \big(\{q_0\},\Sigma,q_0,\emptyset,\delta\big)$,



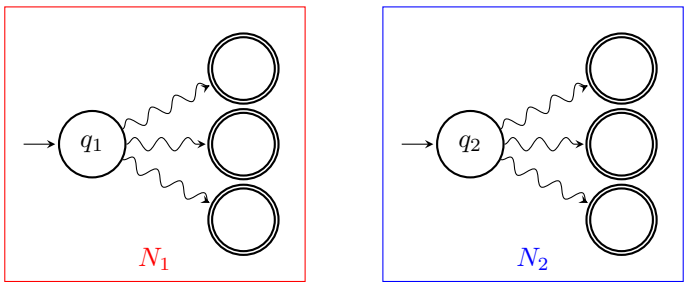
2. If $R = \varepsilon$, then $L(R) = \{\varepsilon\}$ is recognised by the NFA $N = \big(\{q_0\},\Sigma,q_0,\{q_0\},\delta\big)$,



3. If $R = a \in \Sigma$, then $L(R) = \{a\}$ is recognised by the NFA $N = \big(\{q_0,q_1\},\Sigma,q_0,\{q_1\},\delta\big)$,

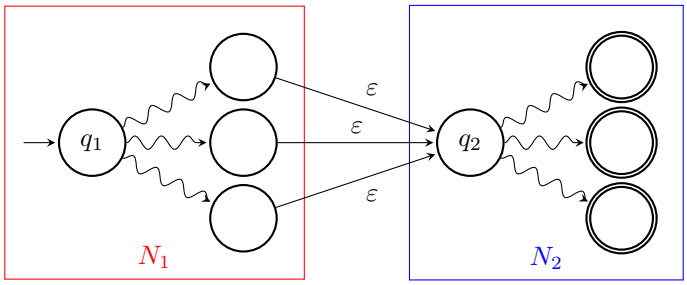


Then, given regular expressions R_1 and R_2 with NFAs $N_1 = (Q_1,\Sigma,q_1,F_1,\delta_1)$ and $N_2 = (Q_2,\Sigma,q_2,F_2,\delta_2)$

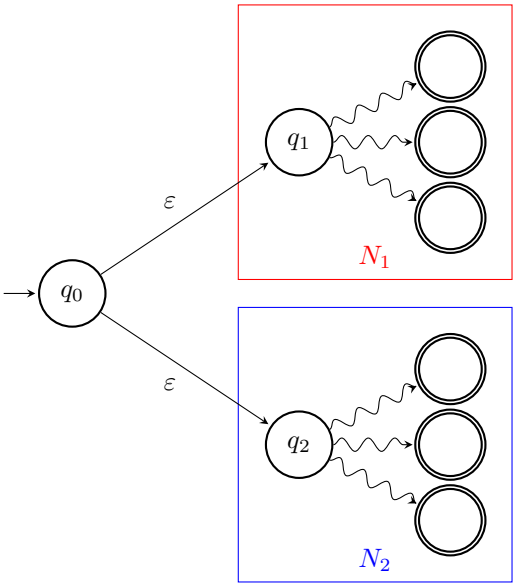


accepting $L(R_1)$ and $L(R_2)$, respectively,

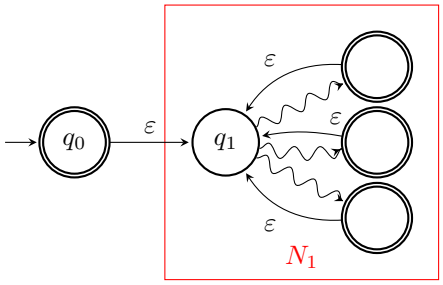
1. The language recognised by the concatenation $R_1 \cdot R_2$ is recognised by the NFA $N = \big(Q_1 \cup Q_2,\Sigma,q_1,F_2 \setminus F_1,\delta\big)$ formed by making the accepting states of R_1 no longer accepting, then attaching ε-transitions from the old accepting states to the initial state of R_2:



2. The language recognised by the union $R_1 + R_2$ is recognised by the NFA $N = \big(Q_1 \cup Q_2 \cup \{q_0\},\Sigma,q_1,F_2 \cup F_1,\delta\big)$ formed by adding a new starting state q_0 with ε-transitions to the previous start states:



3. The language recognised by the Kleene star R_1^* is recognised by the NFA $N = \big(Q_1 \cup \{q_0\}, \Sigma, q_1, F_1, \delta\big)$ formed by adding a new starting state q_0 that is also accepting – in order to accept the empty string – with an ε-transition to the previous start state, then adding ε-transitions from the previous accepting states to the start state – to allow for arbitrary concatenations of the R_1 NFA:



By induction, this construction extends to any regular expression. ■

3 Non-Regular Languages

3.1 The Myhill-Nerode Theorem

Because the change in state of a DFA is determined entirely by the current state and the next character, a DFA is effectively *memoryless*. That is, if two different strings converge to the same state, then the DFA will respond in precisely the same way to any further characters appended to them, so the two initial strings are, with respect to this DFA, identical. This motivates our next definition.

Two strings $x,y \in \Sigma^*$ are *distinguishable* by a language L if there exists a string $z \in \Sigma^*$ such that $x \cdot z \in L$ and $y \cdot z \notin L$, or vice versa, and we call z the *certificate* or *witness* of the distinguishability of x and y.

If two strings $x,y \in \Sigma^*$ are not distinguishable by L, then we say they are *indistinguishable*, and we write $x \equiv_L y$ to denote this relation. Note that this relation is on the *language* L, and is independent from any specific implementation in a particular DFA.

Furthermore, this relation forms an equivalence relation on L (i.e., it is transitive, reflexive, and symmetric), and we call the number of equivalence classes of L under $\equiv_L$ the *index* of $\equiv_L$.

Theorem (Myhill-Nerode)**.** *A language L is regular if and only if $\equiv_L$ has finite index.*

So, to prove a language is non-regular, we can find an infinite set of strings and show that they are pairwise distinguishable by L, and hence $\equiv_L$ has infinite index.

Example. Let $L = \{0^n 1^n : n \in \mathbb{N}\}$ be a language over $\Sigma = \{0,1\}$. Then, the set $\left\{0^i \in \Sigma^* : i \in \mathbb{N}\right\}$ is infinite, and two strings 0^i and 0^j with $i \neq j$ are distinguishable with 1^i as witness. It follows that $\equiv_L$ has infinite index, and hence L is non-regular by Myhill-Nerode. △

Example. Let $L = \{0^p : p \text{ prime}\}$ be a language over $\Sigma = \{0,1\}$.

Let $i \neq j$ and p be prime, and consider the sequence of integers

$$\begin{gathered} p + 0(j-i) \\ p + 1(j-i) \\ p + 2(j-i) \\ \vdots \\ p + p(j-i) \end{gathered}$$

Note that the first integer is $p+0(j-i) = p$, which is prime, and the final integer is $p+p(j-i) = p(1+j-i)$, which is composite. Let $1 \leq k \leq p$ be the least integer for which $p + k(j-i)$ is composite. Then, $1^{p+(k-1)(j-i)-i}$ is a certificate for the distinguishability of 1^i and 1^j:

$$\begin{aligned} 1^i \cdot 1^{p+(k-1)(j-i)-i} &= 1^{p+(k-1)(j-i)} && \in L \\ 1^j \cdot 1^{p+(k-1)(j-i)-i} &= 1^{p+k(j-i)} && \notin L \end{aligned}$$

So, every pair of elements of the infinite set $\left\{1^i \in \Sigma^* : i \in \mathbb{N}\right\}$ are distinguishable, so $\equiv_L$ has infinite index, and hence L is non-regular by Myhill-Nerode. △

Another way to show pairwise distinguishability is to order the infinite set of strings, $(s_i)_{i=1}^{\infty}$, then show that for all i, s^i is distinguishable from s^j for all $j > i$.

Example. Let $n_i(s)$ denote the number of occurrences of the character i in a string s, and let $L = \{s \in \Sigma^* : n_a(s) < n_b(s)\}$ be a language over $\Sigma = \{a,b\}$.

The set $\left\{a^i \in \Sigma^* : i \in \mathbb{N}\right\}$ is infinite, and two strings a^i and a^j with $i > j$ are distinguishable with b^{i+1} as witness, so L is non-regular. △

3.2 The Pumping Lemma for Regular Languages

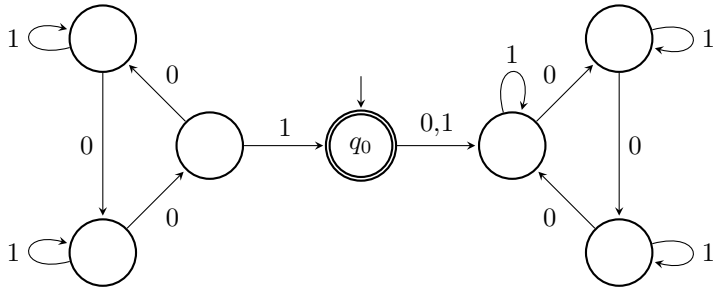
Let M be a DFA. If there is a cycle in the state diagram of M traversed by a string c, then we may traverse that cycle arbitrarily many times, and return to the same state. Again, because DFAs are memoryless, traversing the cycle once is the same as traversing it 2 times, or 3 times, or n times. If that cycle is reachable from the initial state, and can also reach an accepting state, this means that, given a word s accepted by M whose run intersects this cycle, we may add as many copies of c in the middle of s as we want, and the word will still be recognised by M.

That is, if there is a cycle reachable from the initial state that can also reach an accepting state in the state diagram of the DFA M, then the language $L(M)$ is infinite.

In fact, this is a complete characterisation of the DFAs that accept an infinite language: because DFAs must have finitely many states, this is the only way an infinite language can arise.

Theorem 3.1. *A language $L = L(M)$ is infinite if and only if there is a cycle reachable from the initial state that can also reach an accepting state in the state diagram of the DFA M.*

Note that it must be the same cycle that is reachable from the initial state and can reach an accepting state. For instance, the DFA M with state diagram



has both a cycle that is reachable from the start state, and a cycle that can reach an accepting state, but these cycles do not coincide, and $L(M) = \{\varepsilon\}$ is finite.

Lemma (Pumping Lemma)**.** *Let L be a regular language. Then, there exists an integer $p \geq 1$ (the pumping length), such that for every string $s \in L$ with length $|s| \geq p$, there exists a decomposition $s = x \cdot y \cdot z$ such that*

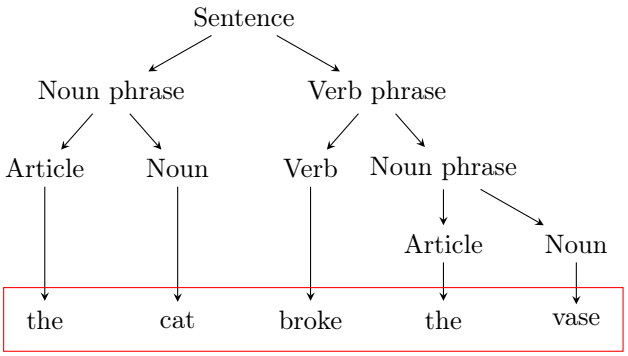
- $|y| \geq 1$*;*
- $|xy| \leq p$*;*
- *for all $n \geq 0$, $x \cdot y^n \cdot z \in L$.*

Example. Let $L = \{0^n 1^n : n \in \mathbb{N}\}$ be a language over $\Sigma = \{0,1\}$. Suppose there exists $p \geq 1$ such that the string $s = 0^p 1^p$ with length $|s| = 2p \geq p$ has a decomposition $s = x \cdot y \cdot z$ satisfying $|x \cdot y| \leq p$ and $|y| \geq 1$. From the former condition, y consists of only instances of 0; and from the latter, y contains at least one instance of 0. Pumping y to obtain $x \cdot y^2 \cdot z$ adds more 0s to the string without adding any 1s, so $x \cdot y^2 \cdot z \notin L$, contradicting the pumping lemma, and hence L is non-regular. △

Note that the pumping lemma is not biconditional; there exist non-regular languages that satisfy the pumping lemma, so the pumping lemma cannot be used to show that a language is regular.

4 Grammars

Ordinary languages not only contain words, but also have particular rules, called a *grammar*, that dictate how they can fit together. For instance, we could have a grammar fragment that says that sentences may be composed of a noun phrase and a verb phrase, which can each then be further decomposed:



so we have *derived* this sentence from the given grammar.

A *grammar* G is a 4-tuple (V,Σ,R,S), consisting of

- A finite set V of *variables* or *non-terminal* symbols;
- a finite set Σ, the alphabet, of *terminal* symbols;

- a finite set R of *substitution rules* or *productions*, where a substitution rule is a string of the form

$$\alpha \to \beta$$

  where $\alpha,\beta \in (V \cup \Sigma)^*$, and $\alpha \neq \varepsilon$.
- and an *initial variable* $S \in V$.

Example. In the above tree, "noun phrase" is a variable, while "the" is a terminal symbol, and "Article → the" is a substitution rule. △

If there are multiple substitution rules with the same term on the left, i.e. $\alpha \to \beta$ and $\alpha \to \gamma$, then we may abbreviate this by writing $\alpha \to \beta \mid \gamma$.

To generate a string from a given grammar G, we start with the initial variable, then, given a production $\alpha \to \beta$, replace an instance of α with β, and repeat, until there are only terminal symbols remaining.

The sequence of substitutions to generate a string from a grammar is then called a *derivation*.

Example. Let $G = \big(\{S,T\},\{0,1\},R,S\big)$ be a grammar, where

$$R = \begin{Bmatrix} S \to TT, \\ T \to 0T1 \mid \varepsilon \end{Bmatrix}$$

Some derivations are as follows:

$$\begin{aligned}
&S \Longrightarrow TT \Longrightarrow 0T1T \Longrightarrow 01T \quad\;\; \Longrightarrow 010T1 \Longrightarrow 0101 \\
&S \Longrightarrow TT \Longrightarrow 0T1T \Longrightarrow 00T11T \Longrightarrow 0011T \Longrightarrow 0011 \\
&S \Longrightarrow TT \Longrightarrow T\varepsilon \quad\;\; \Longrightarrow 0T1 \quad\;\; \Longrightarrow 01 \\
&S \Longrightarrow TT \Longrightarrow T\varepsilon \quad\;\; \Longrightarrow \varepsilon
\end{aligned}$$

△

A derivation is a *left-most derivation* if at each step, a production is applied to the left-most variable in the expression; *right-most derivations* are defined similarly.

Example. The first derivation in the previous example is a left-most derivation, and the last derivation is a right-most derivation. △

For any two strings $\alpha,\beta \in (V \cup \Sigma)^*$, we say that

- α *directly yields* β and write $\alpha \Longrightarrow \beta$ if α may be rewritten as β by applying a single production rule once;
- α *yields* β, or β is *derived from* α and write $\alpha \overset{*}{\Longrightarrow} \beta$ if α may be rewritten as β by applying a finite sequence of productions.

Given a grammar G, we then define the language $L(G)$ to be the set of all strings generated by G:

$$\begin{aligned}
L(G) &\coloneqq \{s \in \Sigma^* : s \text{ is derivable from } S \text{ using production rules in } G\} \\
&= \{s \in \Sigma^* : S \overset{*}{\Longrightarrow} s\}
\end{aligned}$$

Example. Let $G = (\{S\},\{0,1\},R,S)$ be a CFG, where

$$R = \{S \to S\}$$

Then, we have the (unique) production

$$S \Rightarrow S \Rightarrow S \Rightarrow S \Rightarrow \cdots$$

so G does not generate any strings, and hence $L(G) = \emptyset$. △

Example. Let $G = \big(\{S\},\{0,1\},R,S\big)$ be a grammar, where

$$R = \begin{Bmatrix} S \to 0S1, \\ S \to \varepsilon \end{Bmatrix}$$

or equivalently,

$$R = \{S \to 0S1 \mid \varepsilon\}$$

Then, we may generate the strings

$$\begin{aligned} &S \Rightarrow 0S1 \Rightarrow 01 \\ &S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011 \\ &S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111 \\ &\quad \vdots \\ &S \Rightarrow 0S1 \Rightarrow \cdots \Rightarrow 0^n S1^n \Rightarrow 0^n 1^n \end{aligned}$$

so $L(G) = \{0^n 1^n : n \in \mathbb{N}\}$. △

Evidently, grammars can generate non-regular languages.

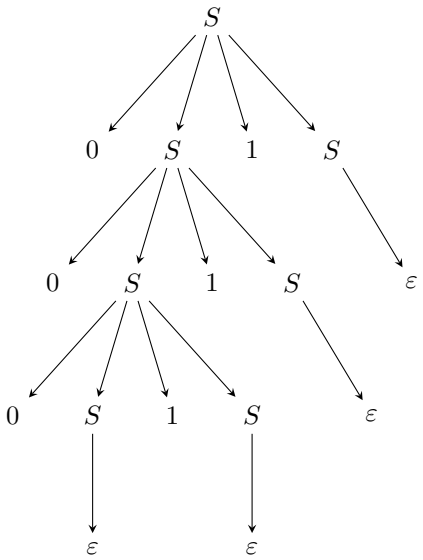
4.1 Parse Trees

We can represent a derivation with a *parse tree*, like the one at the beginning of this section.

Example. Let $G = \big(\{S\},\{0,1\},R,S\big)$ be a grammar, where

$$R = \{S \to 0S1S \mid \varepsilon\}$$

Then, the parse tree for one possible left-most derivation is:



Then, reading the terminals of the tree with an inorder depth first search, we obtain the string

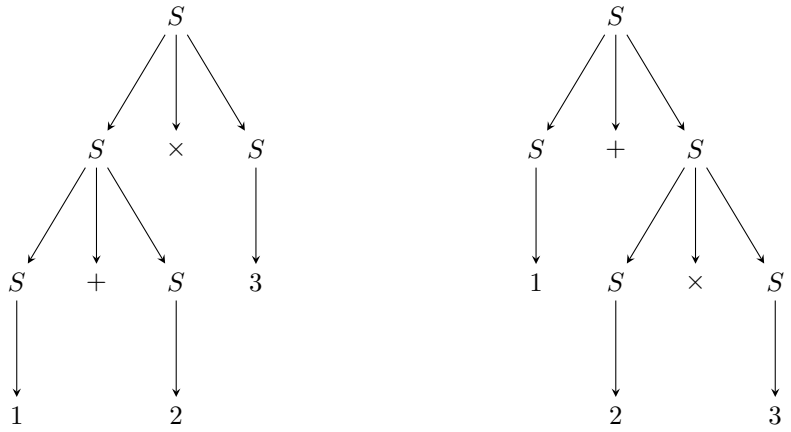
$$000\varepsilon1\varepsilon1\varepsilon1\varepsilon = 000111$$

also called the *yield* of the tree. △

Let $G = \big(\{S\},\{+,\times,0,1,\ldots,9\},R,S\big)$ be a CFG, where

$$R = \begin{Bmatrix} S \to S + S, \\ S \to S \times S, \\ S \to (S) \\ S \to 0 \mid 1 \mid \cdots \mid 9 \end{Bmatrix}$$

Consider the following pair of parse trees:



These trees both yield the string $1 + 2 \times 3$, so there isn't a unique parse tree for this string.

A grammar G is *ambiguous* if it can generate the same string with multiple parse trees, or equivalently, if the same string can be derived from two left-most derivations.

Some ambiguous grammars G may be rewritten as an equivalent unambiguous grammar H, with $L(G) = L(H)$. However, not all grammars admit an unambiguous equivalent. Such grammars are called *inherently ambiguous* grammars.

The problem of determining whether a grammar is ambiguous or not is undecidable.

4.2 Right/Left-Linear Grammars

A *linear grammar* is a grammar that has at most one variable in the right side of each substitution rule.

Example. The grammar $G = \big(V,\{0,1\},R,S\big)$ with rules

$$R = \begin{Bmatrix} S \to 0S1, \\ S \to \varepsilon \end{Bmatrix}$$

is linear, and generates the language $L(G) = \{0^n 1^n : n \in \mathbb{N}\}$. △

As demonstrated by this example, linear grammars may accept some non-regular languages. However, we may add a further restriction:

A *right-linear grammar* is a grammar $G = (V,\Sigma,R,S)$ where each rule is of the following form

- $A \to xB$;
- $A \to x$;

where $A,B \in V$ are variables and $x \in \Sigma^*$ is a string of terminals. A *left-linear grammar* is defined similarly, with the first production rule replaced by $A \to Bx$.

Any derivation of a word from a strict right-linear grammar is of the form

$$S \Longrightarrow a_1A_1 \Longrightarrow a_1a_2A_2 \Longrightarrow \cdots \Longrightarrow a_1a_2\cdots a_n$$

where $(A_i)_i \subseteq V$ and $(a_i)_i \subseteq \Sigma^*$; that is, strings grow towards the right as a derivation progresses.

It turns out that we can strengthen this restriction even further (and it will be convenient for us to do so) without changing the class of languages the grammar accepts:

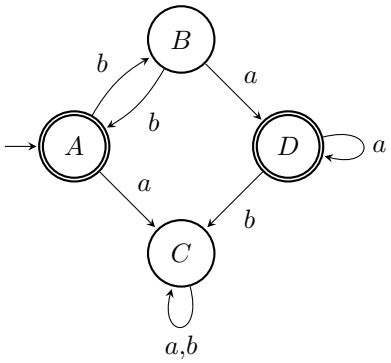
A *strictly right-linear grammar* is a grammar $G = (V,\Sigma,R,S)$ where each rule is of the following form

- $A \to xB$, where $A \in V$, $x \in \Sigma \cup \{\varepsilon\}$;
- $A \to x$.

Strings still grow to the right during a derivation, but the productions now only add a single symbol at a time.

Strightly (right/left)-linear grammars cannot accept all context-free languages. In fact, they accept precisely the regular languages.

Example. Consider the DFA



We construct the associated strightly right-linear grammar. We represent states as variables, with the starting variable representing the starting state, and the alphabet should be the same, so the grammar will be of the form $G = \big(\{A,B,C,D\},\{0,1\},R,A\big)$.

Then, productions should match up with the *outgoing* transitions, and whenever we have an accepting state, we allow a production of the empty string from the variable representing that state.

For instance, at the initial state A, we have transitions $\delta(A,0) = C$ and $\delta(A,1) = B$, so we have productions

$$A \to aC$$
$$A \to bB$$

A is also an accepting state, so we also have

$$A \to \varepsilon$$

For the other states, we have

$$B \to aD$$
$$B \to bA$$
$$C \to aC$$
$$C \to bC$$

$$D \to aD$$
$$D \to bC$$
$$D \to \varepsilon$$

Note that all the production rules are of the required form for this grammar to be right-linear.

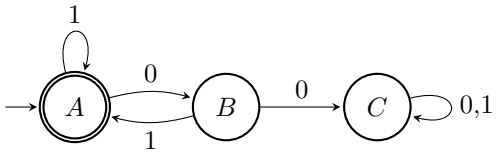
The only way to remove a variable is by replacing it with ε, but by construction, this happens only when the current state is an accepting state. Thus, this grammar generates precisely the language that the DFA recognises. △

We can also go the other way and construct a DFA given any strictly right-linear grammar.

Example. Consider the strictly right-linear grammar $G = \big(\{A,B,C\},\{0,1\},R,A\big)$, where

$$R = \left\{ \begin{aligned} A &\to 0B \mid 1A \mid \varepsilon, \\ B &\to 0C \mid 1A, \\ C &\to 0C \mid 1C \end{aligned} \right\}$$

The process is largely the same as the previous in reverse: we introduce a new state for each variable, add transitions $\delta(A,b) = C$ for each production $A \to bC$, and mark any variables with productions $A \to \varepsilon$ as accepting states:



△

Together, these constructions show more generally that:

Theorem 4.1. *A language L is accepted by a strict right-linear grammar if and only if L is regular.*

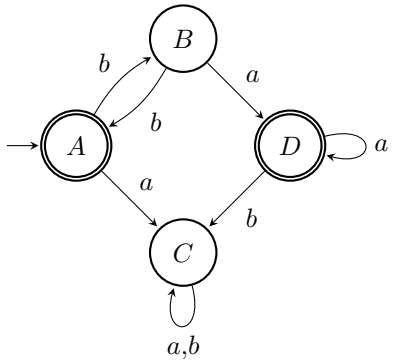
Proof. Let $L = L(M)$ for a DFA $M = (Q,\Sigma,q_0,F,\delta)$. Then, L is recognised by the right-linear grammar $G = (Q,\Sigma,R,q_0)$, where

$$R = \left\{ \begin{aligned} q &\to ap && \forall q,a \in Q : \delta(q,a) = p \\ q &\to \varepsilon && \forall q \in F \end{aligned} \right\}$$

The reverse construction is analogous: given a strictly right-linear grammar $G = (V,\Sigma,R,S)$, introduce a state for each variable $A \in V$; for each rule $A \to bC$ with $A,C \in V$, $b \in \Sigma$, define $\delta(A,b) = C$; and for each rule $A \to \varepsilon$, make the state A an accepting state. ■

We can also construct the strictly left-linear grammar with a similar modified method:

Example. Consider the same DFA as in the previous example:



To construct the associated strictly left-linear grammar, we proceed in much the same way, representing states as variables, but this time, we start at the *final* state, and represent *incoming* transitions with production rules, and finally add a production for the empty string only at the starting state.

The first problem is that there are multiple final states, but a grammar only allows one starting variable. We resolve this analogously to adding ε-transitions in an NFA: add a new state q^* and add productions of the form $q^* \to q$ for every final state q.

So, in this example, we have

$$q^* \to A$$
$$q^* \to D$$

(Note that this is of the required form, since $A = A\varepsilon$.)

Then, we add production rules corresponding to incoming transitions:

$$A \to \varepsilon \mid Bb$$
$$B \to Ab$$
$$C \to Aa \mid Ca \mid Cb \mid Db$$
$$D \to Ba \mid Da$$

This defines the required grammar $G = \big(\{q^*,A,B,C,D\},\{a,b\},R,q^*\big)$. △

The reverse construction is again similar. Thus, we have:

Theorem 4.2. *A language L is accepted by a strictly left-linear grammar if and only if L is regular.*

Proof. Let $L = L(M)$ for a DFA $M = (Q,\Sigma,q_0,F,\delta)$. Then, L is recognised by the strictly left-linear grammar $G = (Q \cup \{q^*\},\Sigma,R,q^*)$, where

$$R = \begin{Bmatrix} q_0 \to \varepsilon & \\ p \to qa & \forall q,a \in Q : \delta(q,a) = p \\ q^* \to q & \forall q \in F \end{Bmatrix}$$

Conversely, given a strictly left-linear grammar $G = (V,\Sigma,R,S)$, introduce a state for each variable $A \in V \setminus S$; for each rule $A \to Bc$ with $A,B \in V$ and $c \in \Sigma$, define $\delta(B,c) = A$; and for each rule $S \to A$, make the state A an accepting state. ■

4.3 Chomsky Hierarchy of Grammars

As we have seen previously, grammars can generate a wider class of languages than just regular languages. We can precisely classify when this happens in terms of constraints on what kind of productions are allowed in a grammar.

Let $G = (V,\Sigma,R,S)$ be a grammar, $A,B \in V$ be variables, $\alpha,\beta,\gamma,\delta \in (V \cup \Sigma)^*$ be strings of arbitrary symbols, and $x \in \Sigma^*$ be a string of terminals.

Grammar	Languages	Recognising Automata	Constraints
Type-3	Regular/(Right/Left)-Linear	Finite automata	$\left\{\begin{matrix} A \to x \\ A \to xB \end{matrix}\right\}$ (right regular) or $\left\{\begin{matrix} A \to x \\ A \to Bx \end{matrix}\right\}$ (left regular)
Type-2	Context-free	Non-deterministic pushdown automata	$A \to \alpha$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \to \alpha \gamma \beta$
Type-0	Recursively enumerable	Turing machine	$\alpha \to \beta$ $(\alpha \neq \varepsilon)$

Each type is a proper subset of the next, so there are recursively enumerable languages that are not context-sensitive, context-sensitive languages that are not context-free, and context-free languages that are not regular.

We call a type-2 grammar a *context-free grammar* (CFG), and the language generated by a CFG is called a *context-free language* (CFL).

5 Context-Free Languages

5.1 Pushdown Automata

A (*non-deterministic*) *pushdown automaton* (PDA) is a 6-tuple $P = (Q,\Sigma,\Gamma,\delta,q_0,F)$ consisting of

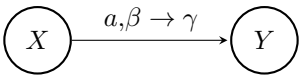
- a finite set Q of states;
- a finite set Σ, the input alphabet;
- a finite set Γ, the *stack symbol* alphabet;
- a transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$;
- an initial state $q_0 \in Q$;
- a set $F \subseteq Q$ of accepting states.

A PDA is effectively an NFA equipped with some limited memory in the form of a stack, to which we can push symbols from Γ and pop with the transition function. We will also usually assume that $\Gamma \supseteq \Sigma$ so we can store any read symbols on the stack.

As we will see, this additional memory allows us to recognise a strictly larger class of languages than NFA/DFA, such as the non-regular palindrome languages $L = \{ww^{\text{rev}} : w \in \Sigma^*\}$.

Also note that we are working with non-deterministic PDA. We will not discuss them in detail here, but unlike NFAs and DFAs which are equivalent, *deterministic* pushdown automata are provably less expressive than their non-deterministic counterparts.

As usual, we may represent a PDA as a state transition diagram; but now, the transitions are controlled not only by the currently read symbol, but also by the state of the stack. The labels in a state diagram are given in the form $a,\beta \to \gamma$, where $a \in \Sigma_\varepsilon$ and $\beta,\gamma \in \Gamma_\varepsilon$, where the arrow indicates the stack operation of popping β then pushing γ.



That is, when we are in the state S, this transition may only be traversed if the current read symbol is a and the top element of the stack is β.

In terms of the transition function, the label $a,\beta \to \gamma$ from a state X to a state Y represents the element $(Y,\gamma) \in \delta(X,a,\beta)$.

Any or all of a, β, and γ may be the empty string: if $a = \varepsilon$, then the transition consists only of the stack operation $\beta \to \gamma$, and it may be traversed without reading any symbols from the input string; if $\beta = \varepsilon$, then the stack operation just pushes γ to the stack, as popping the empty string effectively does not change the state of the stack, since we may assume infinitely many empty strings are on top of the stack; and if $\gamma = \varepsilon$, the stack operation just pops β from the stack, as pushing the empty string again does not change the state of the stack.

We also write $a,\beta \to \gamma_1 \dots \gamma_n$ to denote pushing multiple symbols onto the stack (note that γ_n is pushed first, and γ_1 last, in this notation). This can be converted into an ordinary transition via the provision of some new states such that the intermediary transitions only push one of the γ_i at a time.

Because the stack may always be regarded as having infinitely many empty strings on top, it is difficult to determine whether the stack is empty or not, so for convenience, we also often include the string \$ in Γ which we immediately push on to the stack at the beginning of a computation using the transition $\varepsilon,\varepsilon \to \$$. From then on, we can use the \$ symbol to detect when the stack is intended to be "empty", and we can use the transition $\varepsilon,\$ \to \varepsilon$ to fully empty the stack.

5.2 Languages Recognised by PDA

Given a PDA $P = (Q,\Sigma,\Gamma,\delta,q_0,F)$, we say that P *accepts* or recognises a string $s_1 \cdots s_k = s \in \Sigma^*$ if there exists a sequence $(r_i)_{i=1}^k \subseteq Q$ of states and a sequence $(\sigma_i)_{i=1}^k \subseteq \Gamma$ of stack symbols such that

- $r_0 = q_0$;
- $r_k \in F$;
- For all i, $(r_i,\beta) \in \delta(x_{i-1},s_i,\alpha)$ where $r_{i-1} = \alpha \cdot t$ and $r_i = \beta \cdot t$ for some $\alpha,\beta \in \Gamma_\varepsilon$, $t \in \Gamma^*$.

Example. Consider the CFG $G = \big(\{S\},\{0,1\},R,S\big)$ where

$$R = \begin{Bmatrix} S \to 0S1 \\ S \to \varepsilon \end{Bmatrix}$$

which generates the language

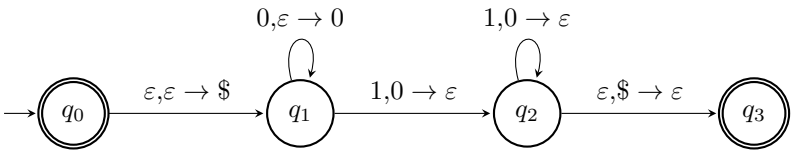
$$L(G) = \{0^n 1^n : n \geq 0\}$$

We construct a PDA that recognises this language as follows.

First, push the empty stack symbol \$ to the stack with $\varepsilon,\varepsilon \to \$$ from the initial state q_0 to a state q_1. Then, whenever we read a 0 from the string, push it on to the stack with $0,\varepsilon \to 0$, and we can repeat this arbitrarily many times, so this transition is a loop on q_1.

When we read a 1 from the string for the first time, we pop a 0 from the stack with $1,0 \to \varepsilon$ and move to a new state q_2, as we we will not allow any more 0s to be read from the input string. In this new state, we can then read 1s from the string and pop 0s from the stack.

If at any point, the stack is empty (i.e. we can read the \$ symbol), we can move to a final accepting state q_3.



△

We claim that PDA recognise precisely the class of context-free languages. To show this, we will show that every CFG can be converted into a PDA and vice versa.

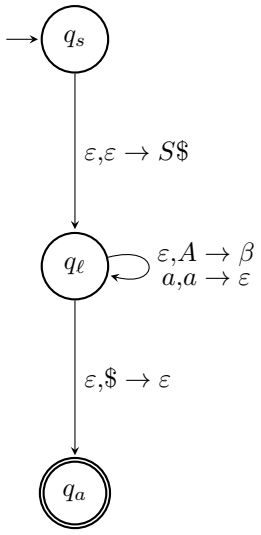
Lemma 5.1. *If a language L is context-free, then there exists a PDA that recognises L.*

Proof. Since L is context-free, there is a CFG $G = (V,\Sigma,R,S)$ such that $L(G) = L$.

We can decide whether a given string T is derivable from some fixed grammar using the following algorithm:

1. Push S\$ on to the stack.
2. While the symbol on the top of the stack is not \$:
   - If the top of the stack is a variable A, pop A and non-deterministically select a grammar rule for A from R, and push the production to the stack.
   - Otherwise, the top of the stack is a terminal a. Read the next input symbol in T and check if it equals a. If so, pop a from the stack and continue. Otherwise, reject T.
3. Once the top of the stack is \$, accept.

We implement this algorithm as a PDA as follows:



where the loop in the middle has a transition rule of the form $\varepsilon,A \to \beta$ for every terminal with production $A \to \beta$ in R; and a transition of the form $a,a \to \varepsilon$ for every terminal $a \in A$.

Note that we are implicitly using more than just 3 states when pushing multiple symbols to the stack. The vertical transitions represent steps 1 and 3 of the algorithm above, and the self-loops on q_ℓ simulating the grammar represents step 2.

The first kind of loops of the form $\varepsilon,A \to w$ correspond to a production being applied to the left-most variable A in a derivation. The second kind of loop $a,a \to \varepsilon$ correspond to matching the input T with the currently generated string. It is safe to do this incrementally, since a terminal is never replaced in a derivation, so if the first symbol is a terminal at any point, it will always be the same terminal at any point onwards in the derivation. ■

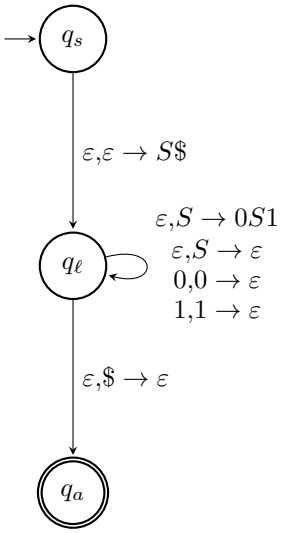
Example. Consider the CFG $G = \big(\{S\},\{0,1\},R,S\big)$ where

$$R = \begin{Bmatrix} S \to 0S1 \\ S \to \varepsilon \end{Bmatrix}$$

which generates the language

$$L(G) = \{0^n 1^n : n \geq 0\}$$

The corresponding PDA is given by



△

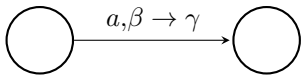
Before we prove the converse, we define a convenient normal form of PDA.

A *normalised PDA* is a PDA such that

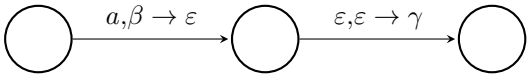
- N has a single accept state;
- N empties its stack before accepting;
- Each transition does exactly one of the following;
  - Push a symbol onto the stack;
  - Pop a symbol from the stack.

Every PDA can be converted into a normalised PDA:

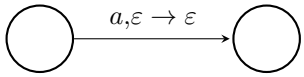
- Add a new accept state q_{accept} and add transitions of the form $\varepsilon,\$ \to \varepsilon$ from the old accept states to q_{accept};
- For every old accepting state and every stack symbol $\gamma \in \Gamma$, add a loop of the form $\varepsilon,\gamma \to \varepsilon$ to empty the stack;
- For every transition with both stack instructions,



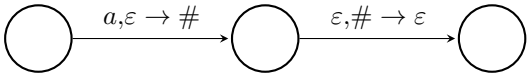
add a new intermediary state with transitions



For transitions with no stack instructions,



instead push and pop a new symbol # on the new transitions:



So, we may assume without loss of generality that all of our PDA are in normalised form.

Lemma 5.2. *If M is a PDA, then $L(M)$ is a context-free language.*

Proof. By the above, there exists a normalised PDA $N = (Q,\Sigma,\Gamma,\delta,q_0,\{q_a\})$ such that $L(N) = L(M)$. We construct a CFG $G = (V,\Sigma,R,S)$ as follows.

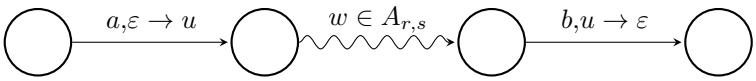
For each pair of states $p,q \in Q$, add a variable $A_{p,q}$ to V. If we can guarantee that $A_{p,q}$ generates precisely the set of strings that take us from p (starting with an empty stack) to q (ending with an empty stack), then we are done, as $S \coloneqq A_{q_0,q_a}$ would generate precisely the language accepted by N.

To achieve this, we define three types of rules.

1. For each state $p \in Q$, add the rule
$$A_{p,p} \to \varepsilon$$
since not reading any characters is a valid path from p to p with empty stacks.

2. For all states $p,q,r \in Q$, add the rule
$$A_{p,q} \to A_{p,r}A_{r,q}$$
since travelling from p to r with empty stacks, then r to q with empty stacks, is a valid path from p to q with empty stacks;

3. For all states $p,q,r,s \in Q$ and stack symbol $u \in \Gamma$ and (possibly empty) letters $a,b \in \Sigma_\varepsilon$, if $(r,u) \in \delta(p,a,\varepsilon)$ and $(q,\varepsilon) \in \delta(s,b,u)$, add the rule
$$A_{p,q} \to aA_{rs}b$$
since if r is reachable from p by pushing u to the stack, and q is reachable from s by popping u, then concatenating with any string w given by $A_{r,s}$ gives a path from p with empty stack to q with empty stack, since by construction, w leaves the stack unchanged, so the u is still available to be popped during the final transition.



■

The previous two results imply:

Theorem 5.3. *A language L is context-free if and only if there is a PDA recognising L.*

5.3 Chomsky Normal Form

A CFG $G = \{V,\Sigma,R,S\}$ is in *Chomsky Normal Form* (CNF) if every substitution rule in R is one of the following form:

- $S \to \varepsilon$;
- $A \to s$, where $A \in V$, $s \in \Sigma$;
- $A \to PQ$, where $A \in V$, $P,Q \in V \setminus \{S\}$.

Note that every parse tree generated by a CFG in CNF is a binary tree: every production either splits a variable into two variables, or replaces it with a terminal, creating a leaf node.

This also implies that every string of length n can be derived in exactly $2n-1$ production steps, since it requires exactly $n-1$ steps to produce n variables from S, and n more steps to replace the variables with terminals.

Theorem 5.4. *Every CFL is generated by a CFG in CNF.*

Proof. Given a CFG $G = \{V,\Sigma,R,S\}$, we convert G to CNF via the following procedure:

- **Eliminate the start symbol from right sides**:

  Add a new initial variable S_0 and a substitution rule $S_0 \to S$ to the old initial variable. This does not change the grammar's produced language, and the new initial variable S_0 will not occur on any rule's right side.

- **Eliminate rules with non-solitary terminals**

  To eliminate a rule where the right side is some combination of terminals $(a_i)_i$ and variables $(X_j)_j \neq \emptyset$,

  $$A \to (a_n \mid X_n)_n$$

  introduce for each such terminal a_i a new variable N_{a_i} and a new rule $N_{a_i} \to a_i$, then replace the rules

  $$A \to (a_n \mid X_n)_n$$

  with

  $$A \to (N_{a_n} \mid X_n)_n$$

  That is, replace any instance of a_i with N_{a_i}.

- **Eliminate rules with more than 2 variables**

  Replace each rule where the right side is some combination of $k \geq 3$ variables

  $$A \to X_1 \cdots X_k$$

  by introducing new variables $(A_i)_{i=1}^{n-2}$

  $$\begin{aligned} A &\to X_1 A_1 \\ A_1 &\to X_2 A_2 \\ A_2 &\to X_3 A_3 \\ &\vdots \\ A_{n-3} &\to X_{n-2} A_{n-2} \\ A_{n-2} &\to X_{n-1} X_n \end{aligned}$$

  (Similar to currying.)

- **Eliminate ε-rules**

  Remove any rules of the form $A \to \varepsilon$ for $A \neq S_0$. Then, for each rule containing n occurrences of A, add 2^n new copies of that rule with each possible combination of A replaced by ε (i.e. removed). Similar to inline expansion or β-reduction.

  For instance, if we had rules $A \to \varepsilon$ and $R \to aAbAcAd$, then we would remove this two rules and add in

  $$\begin{aligned} R &\to aAbAcAd \\ R &\to aAbAcd \\ R &\to aAbcAd \\ R &\to aAbcd \\ R &\to abAcAd \\ R &\to abAcd \\ R &\to abcAd \\ R &\to abcd \end{aligned}$$

  Directly remove any ε-rules introduced by this step.

- **Eliminate unit rules**

  A *unit rule* is a rule of the form

  $$A \to B$$

  for some variables A,$B \in V$.

  To remove a unit rule $A \to B$, for each rule

  $$B \to \Lambda_1 \cdots \Lambda_n$$

  where $(\Lambda_i)_{i=1}^n \subseteq V \cup \Sigma$ is some combination of variables and terminals, add a new rule

  $$A \to \Lambda_1 \cdots \Lambda_n$$

  unless this is a unit rule already removed.

■

Example. Consider a grammar with production rules

$$\begin{aligned} S &\to ASB \\ A &\to aAS \mid a \mid \varepsilon \\ B &\to SbS \mid A \mid bb \end{aligned}$$

We will convert this into CNF.

Start by eliminating the start variable S:

$$\begin{aligned} S_0 &\to S \\ S &\to ASB \\ A &\to aAS \mid a \mid \varepsilon \\ B &\to SbS \mid A \mid bb \end{aligned}$$

Then, we eliminate the terminals in $A \to aAS$, $B \to SbS$ and $B \to bb$:

$$\begin{aligned} S_0 &\to S \\ S &\to ASB \\ A &\to N_aAS \mid a \mid \varepsilon \\ B &\to SN_bS \mid A \mid N_bN_b \\ N_a &\to a \\ N_b &\to b \end{aligned}$$

Now, we eliminate the rules with 3 or more variables, $S \to ASB$, $A \to N_aAS$, and $B \to SN_bS$:

$$\begin{aligned} S_0 &\to S \\ S &\to AV_1 \\ A &\to N_aV_2 \mid a \mid \varepsilon \\ B &\to SV_3 \mid A \mid N_bN_b \\ N_a &\to a \\ N_b &\to b \\ V_1 &\to SB \\ V_2 &\to AS \\ V_3 &\to N_bS \end{aligned}$$

Now, we eliminate the ε-rule $A \to \varepsilon$

$$\begin{aligned} S_0 &\to S \\ S &\to AV_1 \mid V_1 \\ A &\to N_aV_2 \mid a \\ B &\to SV_3 \mid A \mid N_bN_b \mid \not{\varepsilon} \\ N_a &\to a \\ N_b &\to b \\ V_1 &\to SB \\ V_2 &\to AS \mid S \\ V_3 &\to N_bS \end{aligned}$$

Now, we eliminate the unit rules $S \to V_1$, $B \to A$, and $V_2 \to S$:

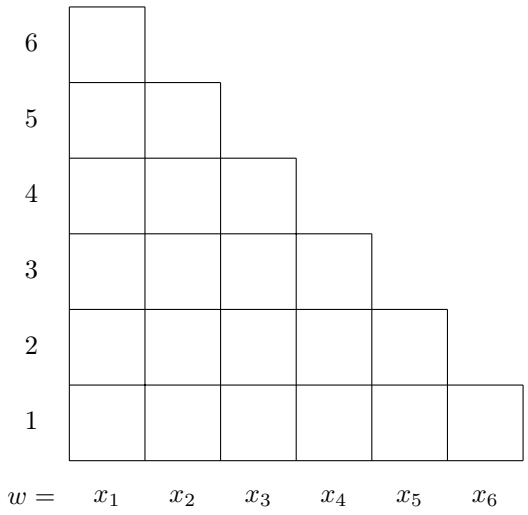
$$\begin{aligned} S_0 &\to S \\ S &\to AV_1 \mid SB \\ A &\to N_aV_2 \mid a \\ B &\to SV_3 \mid N_bN_b \mid N_aV_2 \mid a \\ N_a &\to a \\ N_b &\to b \\ V_1 &\to SB \\ V_2 &\to AS \mid AB_1 \mid SB \\ V_3 &\to N_bS \end{aligned}$$

△

5.4 Cocke–Younger–Kasami (CYK) Parsing

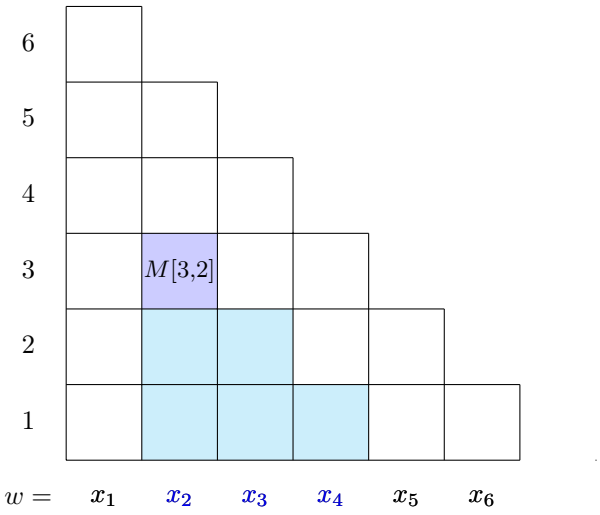
The CYK algorithm is a $\Theta(n^3 \cdot |G|)$ time dynamic programming algorithm for the bottom-up parsing of a CFG in CNF.

We build up a lower triangular table with width and height equal to the length of the word w we are parsing. For instance, if $w = x_1x_2x_3x_4x_5x_6$, then the table is initialised as:



Each entry $M[i,j]$ will contain the set of variables that can generate the substring $x_jx_{j+1}\cdots x_{i+j-1}$ of w. Note that the row number corresponds to the length of the substring.

For instance, $M[3,2]$ contains the set of variables that can generate the substring $x_2x_3x_4$. Visually, this is the substring in the triangular "cone" under the entry:



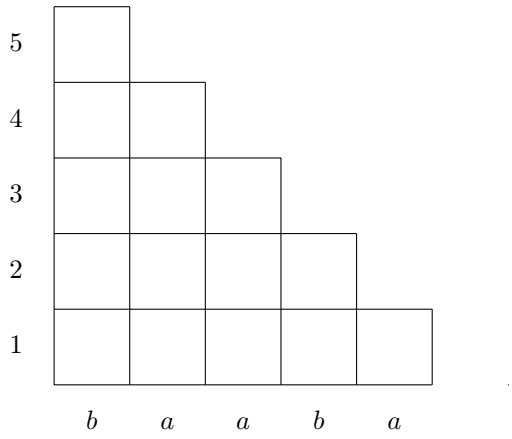
We start from the bottom row, and recursively fill in the table by considering how each substring can be constructed by concatenating previously constructed strings. Then, the unique cell in the top row will contain the starting variable S if and only if w can be derived from the grammar.

Example. Consider the CFG $G = \big(\{S,A,B,C\},\{a,b\},R,S\big)$ where

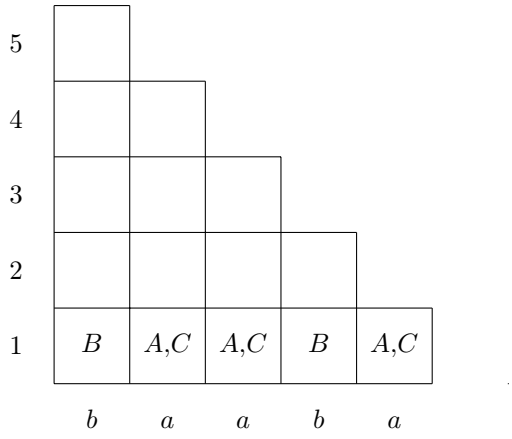
$$R = \begin{Bmatrix} S \to AB \mid BC, \\ A \to BA \mid a, \\ B \to CC \mid b, \\ C \to AB \mid a \end{Bmatrix}$$

Note that G is in CNF, as required.

We will parse the string $w = baaba$. The table is initialised as:



The first row is simple to fill in, as the substrings consist of single terminals:



Now, consider the first cell on the second row, $M[2,1]$. This corresponds to the substring ba.

This substring can be obtained by concatenating any variable that produce b followed by any variable that produces a, which we have already computed in the cells contained in the cone below this cell as $\{B\}$ and $\{A,C\}$.

We have the possible concatenations BA and BC. BA is produced by A, and BC is produced by S, so $M[2,1]$ contains S,A.

We continue similarly for the rest of the row. For the second cell, we have concatenations AA, AC, CA, and CC. CC can be produced by B, and the others cannot be produced, so this cell contains B. The third cell has concatenations AB and CB, which are produced by C and S, respectively. The fourth cell has concatenations BA and BC, which are produced by A and S, respectively.

5					
4					
3					
2	S,A	B	S,C	S,A	
1	B	A,C	A,C	B	A,C
	b	a	a	b	a

Moving onto the third row, the first cell $M[3,1]$ corresponds to the substring baa.

This can be produced by concatenating b with aa, or ba with a. From the previously computed cells in the cone below $M[3,1]$, we already know how to produce these substrings:

- We can produce b with B ($M[1,1]$) and aa with B ($M[2,2]$), so the concatenations are BB;
- We can produce ba with S,A ($M[2,1]$) and a with A,C ($M[1,3]$), so the concatenations are SA, SC, AA, and AC.

No productions generate any of these, so $M[3,1]$ is empty.

The next cell corresponds to the substring aab, which can be produced by concatenating a with ab, or aa with b.

- We can produce a with A,C and ab with S,C, so the concatenations are AS, AC, CS, and CC;
- We can produce aa with B and b with B, so the concatenations are BB.

Only CC can be produced (by B), so $M[3,2]$ contains B.

The next cell corresponds to the substring aba, which can be produced by concatenating a with ba, or ab with a.

- We can produce a with A,C and ba with S,A, so the concatenations are AS, AA, CS, and CA;
- We can produce ab with S,C and a with A,C, so the concatenations are SA, SC, CA, and CC.

Only CC can be produced (by B), so $M[3,3]$ contains B.

5					
4					
3	$\varnothing$	B	B		
2	S,A	B	S,C	S,A	
1	B	A,C	A,C	B	A,C
	b	a	a	b	a

On the fourth row, the first cell represents the substring $baab$, which can be constructed as:

- $b + aab$, produced by B and B, so the concatenations are BB;
- $ba + ab$, produced by S,A and S,C, so the concatenations are SS, SC, AS, AC;
- $baa + b$, produced by $\varnothing$ and B, so there are no concatenations.

(The pattern in the table should now be more apparent: we use the entries on each "leg" of the cone, starting from the bottom of one, and the top of the other.)

None of these concatenations are producible, so $M[4,1]$ is empty.

The second cell represents the substring $aaba$, which can be constructed as

- $a + aba$, produced by A,C and B, so the concatenations are AB, CB;
- $aa + ba$, produced by B and S,A, so the concatenations are BS, BA;
- $aab + a$, produced by B and A,C, so the concatenations are BA, BC.

AB can be produced by S and C, BA by A, and BC by S, so $M[4,2]$ contains S, A, and C:

5					
4	$\varnothing$	S,A,C			
3	$\varnothing$	B	B		
2	S,A	B	S,C	S,A	
1	B	A,C	A,C	B	A,C
	b	a	a	b	a

Finally, the unique cell at the top represents the entire string, which can be constructed as

- $b + aaba$, produced by B and S,A,C, giving BS, BA, BC;
- $ba + aba$, produced by S,A and B, giving SB, AB;
- $baa + ba$, produced by $\emptyset$ and S,A, so no concatenations;
- $baab + a$, produced by $\emptyset$ and A,C, so no concatenations.

BA can be produced by A, BC by S, and AB by S and C, so $M[5,1]$ contains S, A, and C:

5	S,A,C				
4	$\varnothing$	S,A,C			
3	$\varnothing$	B	B		
2	S,A	B	S,C	S,A	
1	B	A,C	A,C	B	A,C
	b	a	a	b	a

S is in the cell on the top row, so $w = baaba$ can be generated by this grammar. △

We can also recover the parse tree from the diagram by connecting non-empty entries along the legs of the cones.

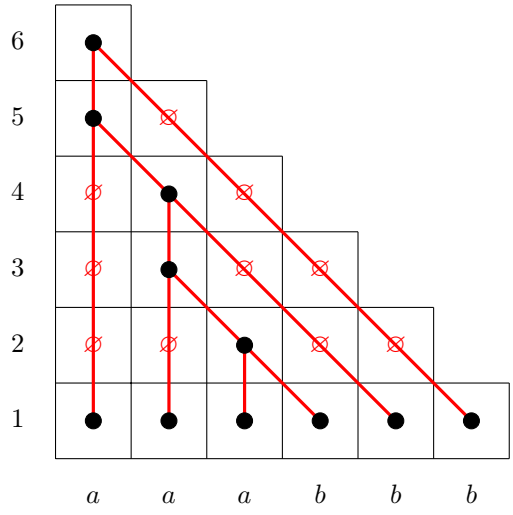
Example. Consider the CFG $G = \big(\{S,T,X,A,B\},\{a,b\},R,S\big)$ where

$$R = \left\{ \begin{array}{l} S \to AB \mid XB \mid \varepsilon, \\ T \to AB \mid XB, \\ X \to AT, \\ A \to a, \\ B \to b \end{array} \right\}$$

with the following CYM table parsing the word $aaabbb$:

6	S					
5	X	$\varnothing$				
4	$\varnothing$	T	$\varnothing$			
3	$\varnothing$	X	$\varnothing$	$\varnothing$		
2	$\varnothing$	$\varnothing$	T	$\varnothing$	$\varnothing$	
1	A	A	A	B	B	B
	a	a	a	b	b	b

Then, the parse tree is given by:



△

6 Non-Context-Free Languages

6.1 The Pumping Lemma for Context-Free Languages

Recall the pumping lemma for regular languages:

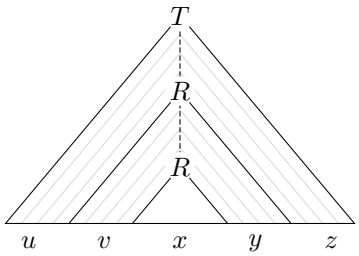
Lemma (Pumping Lemma)**.** *Let L be a regular language. Then, there exists an integer $p \geq 1$ (the pumping length), such that for every string $s \in L$ with length $|s| \geq p$, there exists a decomposition $s = x \cdot y \cdot z$ such that*

- $|y| \geq 1$;
- $|xy| \leq p$;
- *for all $n \geq 0$, $x \cdot y^n \cdot z \in L$.*

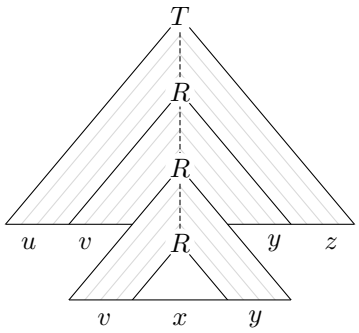
The intuition was that if an input string is too long, then it must loop somewhere inside the DFA.

A similar result holds for context-free languages, in that if a derived string is too long, it must repeat a variable somewhere in its parse tree. Similarly to before we can generate an infinite set of strings in the language by pumping this repeated variable:

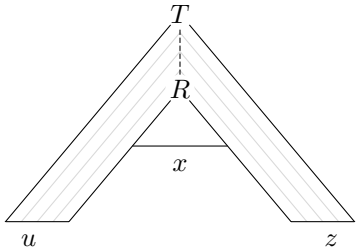
Suppose we repeat a variable R, and that the derivation between the first and second R yields a string that starts with a string v and ends with a string y.



Then, if we repeat R again, the derivation will repeat v and y:



We can also remove the repeated R:



Lemma 6.1 (Pumping Lemma)**.** *Let L be a context free language. Then, there exists an integer $p \geq 1$ (the pumping length) such that for every string $s \in L$ with length $|s| \geq p$, there exists a decomposition $s = u \cdot v \cdot x \cdot y \cdot z$ such that*

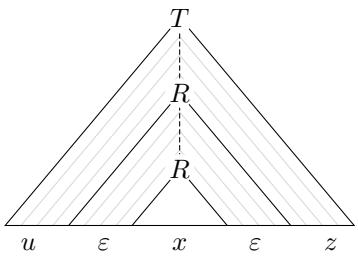
- $|vy| \geq 1$*;*
- $|vxy| \leq m$*;*
- *for all $n \geq 0$, $u \cdot v^n \cdot x \cdot y^n \cdot z \in L$.*

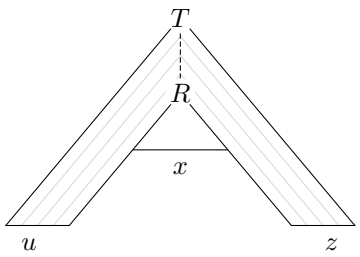
Proof. Let $G = (V,\Sigma,R,S)$ be a CFG, and suppose that the length of the longest string in the right side of a production rule in R is b. Then, any node in a parse tree yielding a string in $L(G)$ has at most b children. If the height of the tree is h, then it has at most b^h leaves, as each layer can only have b times as many nodes as the previous layer.

How long must a string be such that a variable is repeated on some root to leaf path of any parse tree of this string?

We claim that $p \coloneqq b^{|V|+1}$ is sufficient. Indeed, the parse tree of such a word must have $b^{|V|+1}$ leaves, so the tree must have height at least $|V| + 1$. But, there are only $|V|$ variables, so one must repeat.

Take the smallest parse tree for the string $w = uvxyz$. If $|vy| \not\geq 1$, then $v = y = \varepsilon$, and the section of the parse tree between the two Rs don't contribute anything, so $w = uxz$, and we have an even smaller parse tree for w, contradicting the minimality of the first tree.





If $|vxy| > p$, then the subtree rooted at the first R is itself long enough to have a repeated variable on a root to leaf path. So, we can split x into $x = u'x'y'$, so $w = (uv)v'x'y'(yz)$, which is still of the required form, but the middle substring has now decreased in length. We can then repeat this process until the middle substring has length at most p. ■

Example. Consider the language

$$L = \{a^n b^n c^n : n \geq 0\}$$

Suppose that L is context-free, and let p be the pumping length given by the pumping lemma. Take the string $w = a^p b^p c^p \in L$. Clearly, $|w| \geq m$.

Take an arbitrary decomposition $w = uvxyz$ with $|vxy| \leq p$ and $|vy| \geq 1$.

Since $|vy| \geq 1$, v and y contain at least 1 symbol from $\{a,b,c\}$. Because the middle of w contains p many bs and $|vxy| \leq p$, vxy contains at most 2 of the symbols. So, if we pump vxy 0 times, uv^0xy^0z will lose some number of only 2 of the characters. Hence $uv^0xy^0z \notin L$, and L is not context-free. △

Example. Consider the language

$$L = \left\{xx : x \in \{0,1\}^*\right\}$$

Suppose that L is context-free, and let p be the pumping length given by the pumping lemma. Take the string $w = 0^{p+1}1^{p+1}0^{p+1}1^{p+1} \in L$. Clearly, $|w| \geq p$.

Take an arbitrary decomposition $w = uvxyz$ with $|vxy| \leq p$ and $|vy| \geq 1$.

If we pump w 0 times, we obtain $uv^0xy^0z = 0^\alpha 1^\beta 0^\gamma 1^\delta$. Because $|vxy| \leq p$, vxy can intersect at most two adjacent sections of continguous 1s or 0s in w. If α and β or γ and δ are changed, then only one half of $w = \omega\omega$ has changed, so $uv^0xy^0z \notin L$. If β and γ are changed, then we have changed the number of 1s in the first ω, and/or the number of 0s in the second. In any case, $uv^0xy^0z \notin L$, so L is not context-free. △

6.2 Finiteness of Context-Free Languages

If L is a context-free language that has a string longer than the pumping length p, then the pumping lemma allows us to pump this string to generate infinitely many strings.

The converse also holds: if L contains no strings of length longer than p, then L is finite. In particular, it is also regular.

We also have that if L contains even one string of length longer than p, then we also have a bound on the maximal length of a shortest string as follows. Given any arbitrary string w, the pumping lemma gives a decomposition $w = uvxyz$ with $|vxy| \leq p$ and $|vy| \geq 1$. In particular, $|vy|$ is at most p when $x = \varepsilon$. So, pumping w down reduces its length by p. Repeating this process will eventually return a string with length between p and $2p - 1$.

6.3 Closure Properties of Context-Free Languages

Recall (§2.2) that regular languages are closed under

- Intersection;
- Union;
- Concatenation;
- Complementation;
- Kleene star.

The proof for intersections was to construct a DFA that simulated the two original DFAs in parallel, and accepting if both original DFAs accepted.

However, simulating a pair of PDAs using one PDA would require us to simulate two stacks with just one stack. This is provably impossible, and context-free languages are not closed under intersection.

Example. Consider the following CFLs:

$$L_1 = \{a^i b^i c^j : i,j, \geq 0\}$$
$$L_1 = \{a^i b^j c^j : i,j, \geq 0\}$$

Then, $L_1 \cap L_2 = \{a^n b^n c^n : n > 0\}$ is non-context-free. △

However, the intersection of a context-free language and a regular language *is* context-free, since we then only have one stack to simulate (i.e. just use the stack).

In contrast, for unions, we can use non-determinism to our advantage. Unlike for intersections, we only need to simulate one stack at a time in a union – not both simultaneously. We add a new start state for the PDA and an ε-transition to the start states of the previous PDAs. Then, this new PDA will accept if either of the previous PDAs accept.

Other similar construction works for grammars:

- **Union**:

  If $L_1 = L(G_1)$ and $L_2 = L(G_2)$ with $G_1 = (V_1,\Sigma_1,R_1,S_1)$ and $G_2 = (V_2,\Sigma_1,R_2,S_2)$, then $L_1 \cup L_2$ is generated by the grammar

  $$G = \big(V_1 \sqcup V_2, \quad \Sigma_1 \cup \Sigma_2, \quad R_1 \cup R_2 \cup \{S \to S_1 \mid S_2\}, \quad S\big)$$

- **Concatenation**:

  If $L_1 = L(G_1)$ and $L_2 = L(G_2)$ with $G_1 = (V_1,\Sigma_1,R_1,S_1)$ and $G_2 = (V_2,\Sigma_1,R_2,S_2)$, then $L_1 \cdot L_2$ is generated by the grammar

  $$G = \big(V_1 \sqcup V_2, \quad \Sigma_1 \cup \Sigma_2, \quad R_1 \cup R_2 \cup \{S \to S_1 S_2\}, \quad S\big)$$

- **Kleene star**:

  If $L = L(G_1)$ with $G_1 = (V_1,\Sigma_1,R_1,S_1)$, then L^* is generated by the grammar

  $$G = \big(V_1, \quad \Sigma_1, \quad R \cup \{S \to \varepsilon \mid S_1 S\}, \quad S\big)$$

Context-free languages are also not closed under complementation: using De Morgan's laws, we can write an intersection as:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Since context-free languages are closed under union, if they were also closed under complementation, they would be closed under intersection. But this cannot be the case.

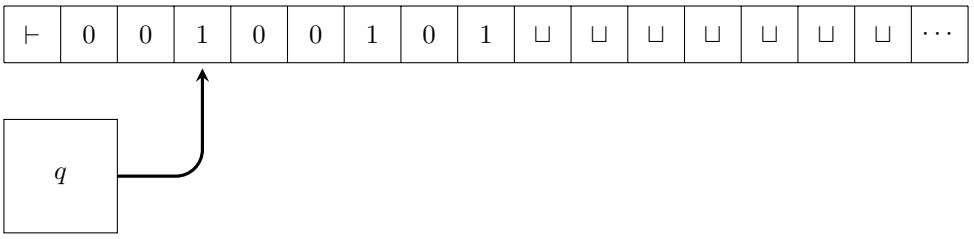
7 Recursively Enumerable Languages

A *Turing machine* is a 7-tuple $(Q,\Sigma,\Gamma,\delta,q_0,q_{\text{accept}},q_{\text{reject}})$, consisting of

- a finite set Q of states;
- a finite set Σ, the input alphabet;

- a finite set Γ, the *tape alphabet* not containing the *blank symbol* $\sqcup$;
- a transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L,R\}$;
- an initial state $q_0 \in Q$;
- an accepting state $q_{\text{accept}} \in Q$;
- an rejecting state $q_{\text{reject}} \in Q$, where $q_{\text{reject}} \neq q_{\text{accept}}$.

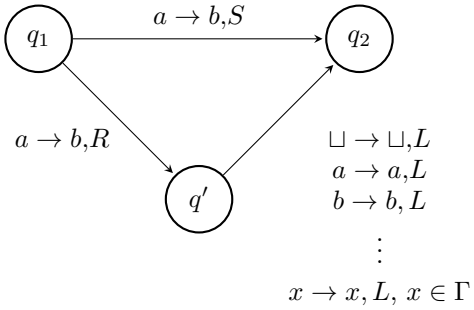
Instead of a stack, a Turing machine is equipped with a *tape* of memory – an array of memory that extends infinitely in one direction – and a *read/write head* pointing to one of the cells on the tape. The $\{L,R\}$ in the transition function indicates which way the read/write head should move after each instruction.



We mark the first cell with the reserved symbol $\vdash$, and unless otherwise specified, the other infinitely many cells with the reserved blank symbol $\sqcup$.

We can, as usual, represent a Turing machine as a state transition diagram. The labels in a state transition diagram are given in the form $a \to b,m$, where $m \in \{L,R\}$ is a movement instruction, and $a,b \in \Gamma$ are tape symbols. This time, the arrow indicates reading an a at the current tape position, writing a b, then moving the read/write head the specified direction.

Note that by this definition, the read/write head *must* move after every instruction. However, we can perform a memory operation $a \to b$ without any net movement, which we denote by $a \to b, S$, by performing the memory operation, moving right, reading/writing the same symbol back into this cell, then moving back left:



A *configuration* of a Turing machine $M = (Q,\Sigma,\Gamma,\delta,q_0,q_{\text{accept}},q_{\text{reject}})$ is a triple (u,q,v), where

- $q \in Q$ is the current state of the machine;
- The string on the tape is $u \circ v$, and the read/write head is on the first symbol of v.

Example. The configuration displayed on the tape above is

$$\big(\{0,0\},q,\{1,0,0,1,0,1\}\big)$$

△

Example.

- $(\vdash, q_0, w)$ is the *start* configuration;
- $(u, q_{\text{accept}}, v)$ is the *accepting* configuration;
- $(u, q_{\text{reject}}, v)$ is the *rejecting* configuration;

△

We say that a configuration $X = (u,q,v)$ *yields* another configuration $Y = (u',p,v')$ if the Turing machine can go from X to Y with a single state transition.

The *run* of a Turing machine $M = (Q,\Sigma,\Gamma,\delta,q_0,q_{\text{accept}},q_{\text{reject}})$ on a word w is a sequence of configurations $C_1,\ldots,C_r$ such that $C_1 = (\vdash, q_0, w)$ is the start configuration (i.e., the word w is written on the tape), and such that each C_i yields C_{i+1}.

The run is *accepting* if C_r is the accepting configuration, and *rejecting* if C_r is the rejecting configuration. Note that, unlike for the weaker automata we have seen, we have a new possible end state, because the run of a Turing machine on any given word may not necessarily be finite. If the run is infinite, then we say that the Turing machine does not *halt*.

We define the language $L(M)$ to be the set of words w such that M *accepts*:

$$L(M) \coloneqq \{s \in \Sigma : \text{the run of } M \text{ on } w \text{ is accepting}\}$$

A language L is *Turing-recognisable* or *recursively enumerable* (RE) if there exists a Turing machine M which accepts precisely the strings in L. We do not require that M explicitly rejects strings outside of L (since in this case, it may not halt), only that it does not accept any strings outside of L.

In contrast, a language L is *Turing-decidable* or *recursive* if there exists a Turing machine M which accepts precisely the strings in L, and always halts. That is, it must also explicitly reject strings outside of L.

If a Turing machine always halts, then we call it a *decider* or a *total Turing machine*.

Turing-decidable languages are precisely those for which an algorithm to determine membership exists.

Example. Is there a decider D, which, when given a Turing machine M, decides whether M:

(*i*) has more than 847 states?

(*ii*) takes more than 847 steps on the input ε?

(*iii*) takes more than 847 steps on *some* input?

(*iv*) takes more than 847 steps on *all* inputs?

(*v*) ever moves the read/write head more than 847 tape cells away from the endmarker on input ε?

(*vi*) accepts ε?

(*vii*) accepts any string at all?

(*viii*) accepts every string?

(*i*) Yes, just count the states until we reach 847. If we reach 847, accept M. Otherwise, reject M. As we have bounded the maximum count, this procedure will always halt, so this describes a decider.

(*ii*) Yes, just simulate the run of M on ε for 847 steps, or until M halts, whichever happens earlier. If M halts before 847 steps, reject M. Otherwise, accept M.

As we have bounded the number of steps, this will always halt, so this describes a decider.

(*iii*) Yes. Simulate the run of M on all possible inputs of size at most 847 for 847 steps, or until M halts, whichever happens earlier.

There are $(\Sigma + 1)^{847}$ many such inputs, which is finite, so the program will eventually halt.

It is also sufficient to only check these inputs, since if M runs for 847 steps on a word p of length greater than 847, then M also runs for at least 847 steps when the input is only the first 847 symbols of p, since M can only possibly access the first 847 symbols of p in 847 steps.

(*iv*) Yes. The program for the previous case also decides this problem.

(*v*) Yes. If the read/write head of M never goes past the 847th cell, then there are only finitely many possible configurations that M could be in, since a Turing machine only has finitely many states and tape symbols. Namely, there are

$$t \coloneqq 847 \cdot |Q| \cdot |\Gamma|^{847}$$

possible configurations. So, we simulate the run of M on ε for $t + 1$ steps, or until it halts, whichever happens earlier. If M ever moves the read/write head past the 847th cell, stop and accept M. Otherwise, reject M.

(*vi*) No.

(*vii*) No.

(*viii*) No.

We will prove these last three cases later as a special case of the *Membership Problem.* △

7.1 Modifications of Turing Machines

There are many useful ways in which we might modify a Turing machine. However, this model of computation is very robust, and many of these modifications end up being equivalent to an ordinary Turing machine:

Example. What if we equipped a Turing machine with three tapes instead of one?

It turns out that we can simulate a Turing machine M_3 with three tapes with a Turing machine M using just one.

We make the tape alphabet of M a tuple to store three symbols in each cell. Then, to mark the position of the read/write head, for each symbol a in the tape alphabets of M_3, add a marked copy $\hat{a}$ to M. Then, for each instruction moving a read/write head of M_3, we instead replace the marked symbols. △

Using this, we can indeed simulate a Turing machine M using another Turing machine, as we have been claiming in the previous example, by implementing two tapes and writing the source code of M on the first tape and the input to be run on the second.

Example. What if the tape were infinite in both directions?

We can "fold" the tape somewhere, forming two tapes that are infinite in one direction only. As seen by the previous example, we can simulate this on a Turing machine. △

An *enumeration machine* or *enumerator* is a modification of a Turing machine equipped with two tapes,

- an ordinary read/write tape, assumed to always start blank;
- an write-only *output tape*;

and a special *enumeration* state. When the machine enters the enumeration state, we say that it has *enumerated* whatever word is on the output tape. Then, the machine erases the output tape and sends the write-only head to the beginning of the output tape, before continuing.

Given an enumerator E, we define the language $L(E)$ of E to be the set of words enumerated by E.

$$L(E) \coloneqq \{s : E \text{ enumerates } s\}$$

Theorem 7.1. *A language L is Turing-recognisable if and only if there exists an enumerator E such that $L = L(E)$.*

Proof. Suppose $L = L(E)$. From E, construct a Turing machine M that operates on an input w as follows:

- Run E on w;
- Each time E enumerates a word u, compare it to w;
- If $w = u$, accept. Otherwise, continue.

If L accepts w, then it will eventually be enumerated by E. If L does not accept E, then M never halts. So, $L(M) = L(E)$.

Now, suppose that L is Turing-recognisable, and let M recognise L. We construct an enumerator E that operates on an input w as follows.

- Order all possible strings (i.e. via lexicographical order) as s_1,s_2,...
- For each $i = 1,2,3,\ldots$, repeat the following:
  - Run M for i steps on $s_1,\ldots,s_i$.
  - If any computations accept, print out the corresponding s_j.

■

A *Universal Turing Machine* (UTM) U takes an encoding of a Turing machine M and a word w, written as $\text{Enc}(M)\#w$ or just $\langle M,w\rangle$, and simulates the run of M on w.

Theorem 7.2. *A language L is Turing-recognisable if and only if there exists an enumerator E such that $L = L(E)$.*

Proof. Suppose $L = L(E)$. From E, construct a Turing machine M that operates on an input w as follows:

- Run E on w;
- Each time E enumerates a word u, compare it to w;
- If $w = u$, accept. Otherwise, continue.

If L accepts w, then it will eventually be enumerated by E. If L does not accept E, then M never halts. So, $L(M) = L(E)$.

Now, suppose that L is Turing-recognisable, and let M recognise L. We construct an enumerator E that operates on an input w as follows.

- Order all possible strings (i.e. via lexicographical order) as s_1,s_2,...
- For each $i = 1,2,3,\ldots$, repeat the following:
  - Run M for i steps on $s_1,\ldots,s_i$.
  - If any computations accept, print out the corresponding s_j.

■

7.2 Undecidability

7.2.1 The Halting Problem

Consider the language

$$\mathbf{HP} \coloneqq \{\langle M,x\rangle : M \text{ halts on } x\}$$

This language is Turing-recognisable by a universal Turing machine as we can simply simulate the run of M on x.

If M halts on x, then U accepts $\langle M,x\rangle$. Otherwise, U also fails to halt, which is allowed, as we only prohibit explicit acceptance in Turing-recognisability. So, **HP** is Turing-recognisable.

However, is it Turing-*decidable*?

That is, given an instance $\langle M,x\rangle$ of the halting problem, can we decide with a definite yes-or-no answer whether or not M will halt on any given string x?

Theorem 7.3. HP *is undecidable.*

Proof. Observe that any Turing machine M consists of only a finite amount of data. As such, this information can be encoded in binary. With such an encoding, we can inversely interpret every binary string b as a Turing machine M_b.

Suppose there exists a Turing machine U that decides the halting problem instance $\langle M_b,x\rangle$, given an encoding b of a Turing machine and an input x. Construct a table of its outputs on all possible strings:

	ε	0	1	00	01	10	11	000	001	010	$\cdots$
M_ε	H	L	L	H	L	H	H	H	L	L	$\cdots$
M_0	L	H	H	H	L	L	L	L	H	H	$\cdots$
M_1	L	H	L	L	H	L	L	H	H	H	$\cdots$
M_{01}	H	H	H	H	L	H	H	L	L	L	$\cdots$
M_{10}	H	L	H	L	L	L	L	L	H	L	$\cdots$
M_{11}	H	H	H	L	H	H	L	L	L	L	$\cdots$
M_{000}	L	L	L	H	H	H	H	L	H	H	$\cdots$
M_{001}	L	H	H	L	L	H	H	H	H	L	$\cdots$
M_{010}	H	L	L	H	L	L	H	H	L	L	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

We construct a Turing machine K as follows.

- Given an input string b, construct M_b.
- Simulate U on $\langle M_b,b\rangle$.
- If U halts, go into an infinite loop.
- If U accepts, halt.

That is, K halts on b if and only if M_b does not half on b, and vice versa, so its output is the reverse of the diagonal entries on the above table.

So, by construction, K disagrees with every Turing machine M_b on at least the input b, so K cannot be on the table, contradicting that this table contains every Turing machine. ■

7.2.2 The Membership Problem

Consider the language

$$\mathbf{MP} \coloneqq \{\langle M,x\rangle : x \in L(M)\}$$

Again, this language is Turing-recognisable by a universal Turing machine as we can simply simulate the run of M on x.

However, the *Membership Problem* is again undecidable:

Theorem 7.4. **MP** *is undecidable.*

Proof. Suppose U decides **MP**. From U, we will construct a Turing machine K that decides **HP** as follows:

- Given an input $\langle M,x\rangle$, K constructs a new Turing machine M' as follows:
  - Add a new accept state.
  - Redirect all incoming transitions to the old accept and reject states to the new accept state.
- Run U on M'.
- If U accepts M', accept M. Otherwise, reject M.

We have that K accepts M if and only if U accepts M' if and only if M accepts or rejects x. That is, if and only if M halts on x. So, K decides **HP**, which is undecidable. ■

7.3 Computability and Reductions

A function σ is *computable* if there is a decider such that, when run on input x, halts with $\sigma(x)$ on the output tape.

Given subsets $A \subseteq \Sigma^*$ and $B \subseteq \Delta^*$, a function $\sigma : \Sigma^* \to \Delta^*$ is a *mapping reduction* if

- For all $x \in \Sigma$, $x \in A$ if and only if $\sigma(x) \in B$;
- σ is computable;

and we write $A \leq_m B$ if such a reduction exists.

Theorem 7.5.

- *If $A \leq_m B$ and B is decidable, then A is decidable;*
- *If $A \leq_m B$ and A is undecidable, then B is undecidable.*

Proof. Let M decide B and let σ be a reduction from A to B. Then, we define a decider N for A as follows:

- Given an input w, compute $\sigma(w)$.
- Simulate the run of M on $\sigma(w)$, and return whatever M returns.

Because σ is a rediction from A to B, if $w \in A$, then $\sigma(w) \in B$. So, M accepts $\sigma(w)$ whenever $w \in A$, so N decides A.

The second claim is the contrapositive. ■

Here are some more languages:

$$\varepsilon\text{-}\textsc{Acceptance} \coloneqq \{\langle M\rangle : \varepsilon \in L(M)\}$$
$$\exists\text{-}\textsc{Acceptance} \coloneqq \{\langle M\rangle : L(M) \neq \emptyset\}$$

$$\forall\text{-}\textsc{Acceptance} \coloneqq \left\{\langle M\rangle : L(M) = \Sigma^*\right\}$$

Consider the following construction:

Let $\langle M,x\rangle$ be an instance of **HP**. Define the Turing machine M'_x as follows:

- Given an input y, ignore y and simulate the run of M on input x.
- If M halts, accept y. Otherwise, reject y.

By construction, M'_x accepts x if and only if M halts on x. So, $\langle M,x\rangle \in$ **HP** if and only if $\langle M'_x,x\rangle \in$ **MP**.

In fact, $\langle M,x\rangle \in$ **HP** if and only if:

(*i*) $\langle M'_x\rangle \in \varepsilon$-Acceptance;

(*ii*) $\langle M'_x\rangle \in \exists$-Acceptance;

(*iii*) $\langle M'_x\rangle \in \forall$-Acceptance.

Because M'_x ignores its input, it accepts any and all inputs if and only if M accepts x. That is, if and only if $\langle M,x\rangle \in$ **MP**. So none of these languages are decidable.

Example. Is the following language is decidable?

$$L \coloneqq \left\{\langle M_1,M_2\rangle : M_1 \text{ accepts a word that } M_2 \text{ does not.}\right\}$$

No, because we can decide ε-Acceptance given a decider M for L. We construct a decider U as follows:

- Given an input M_b, construct a Turing machine M' that accepts the string x if and only if $x \neq \varepsilon$ and M_b accepts x.
- Simulate the run of M on $\langle M_b,M'\rangle$.
- Accept M_b if M accepts. Otherwise, reject M_b.

By construction, M_b and M' accept the same non-empty words, but M' cannot accept the empty word. Thus, M_b accepts a word that M' does not if and only if M_b accepts ε. So, U decides ε-Acceptance. △

Theorem 7.6.

- *If $A \leq_m B$ and B is Turing-recognisable, then A is Turing-recognisable;*
- *If $A \leq_m B$ and A is not Turing-recognisable, then B is not Turing-recognisable.*

Proof. Identical to the previous, with recognisers replacing deciders. ■

Theorem 7.7. *A language L is decidable if and only if L and $\overline{L}$ are both Turing-recognisable.*

Proof. If L is decidable, then a decider for L also functions as a recogniser for L. For $\overline{L}$, use the same decider and complement the answer.

Conversely, if L and $\overline{L}$ are both Turing-recognisable with recognisers P and Q, then define the decider M as follows:

- Given an input x, simulate P and Q simultaneously with input x.
- If P accepts, halt and accept. If Q accepts, halt and reject.

■

7.4 Closure Properties of Turing-Recognisable and Turing-Decidable Languages

- **Complementation**:

  Decidable languages are closed under complementation, as we can simply invert the return value of a decider.

  However, Turing-recognisable languages are *not* closed under complementation.

- **Union**:

  Decidable and Turing-recognisable languages are both closed under union.

- **Intersection**:

  Decidable and Turing-recognisable languages are both closed under intersection.

- **Kleene star**:

  Decidable and Turing-recognisable languages are both closed under Kleene star.

- **Concatenation**:

  Decidable and Turing-recognisable languages are both closed under concatenation.

7.5 Pairwise Intersection Closures Properties

	Regular	CFL	Decidable	RE
Regular	Regular			
CFL	CFL	Decidable		
Decidable	Decidable	Decidable	Decidable	
RE	RE	RE	RE	RE