# Notes on Computation Theory

Konrad Slind
slind@cs.utah.edu

December 9, 2008

# Foreword

These notes are intended to support **cs3100**, an introduction to the theory of computation given at the University of Utah. *Very little of these notes are original with me.* Most of the examples and definitions may be found elsewhere, in the many good books on this topic. These notes can be taken to be an informal commentary and supplement to those texts. In fact, these notes are basically a direct transcription of my lectures. Any lacunae and mistakes are mine, and I would be glad to be informed of any that you might find.

The course is taught as a mathematics course aimed at computer science students. We assume that the students already have had a discrete mathematics course. However, that material is re-covered briefly at the start. We strive for a high degree of precision in definitions. The reason for this is that much of the difficulty students have in theory courses is in not 'getting' concepts. However, in our view, this should never happen in a theory course: the intent of providing mathematically precise definitions is to banish confusion. To that end, we provide formal definitions, and use subsequent examples to sort out pathological cases and provide motivation.

A minor novelty, not original with us, is that we proceed in the reverse of the standard sequence of topics. Thus we start with Turing machines and computability before going on to context-free languages and finally regular languages. The motivation for this is that not many topics are harmed in this approach (the pumping lemmas and non-determinism do become somewhat awkward) while the benefit is twofold: (1) the intellectually stimulating material on computability and undecidability can be treated early in the course, while (2) the immensely practical material dealing with finite state machines can be used to finish the course. So rather than getting more abstract, as is usual, the course actually gets more concrete and practical, which is often to the liking of the students.

The transition diagrams have been drawn with the wonderful *Vaucanson* LaTeX package developed by Sylvain Lombardy and Jacques Sakarovitch.

# Contents

# Chapter 1

# Introduction

This course is an introduction to the *Theory of Computation*. Computation is, of course, a vast subject and we will need to take a gradual approach to it in order to avoid being overwhelmed. First, we have to understand what we mean by the title of the course.

The word *Theory* implies that we study abstractions of computing systems. In an abstraction, irrelevant complications are dropped, in order to isolate the important concepts. Thus, studying the theory of subject $x$ means that simplified versions of $x$ are analyzed from various perspectives.

This brings us to *Computation*. The general approach of the course, as we will see, is to deal with very simple, deliberately restricted models of computers. We will study Turing machines and register machines, grammars, and automata. We devote some time to working *with* each model, in order to see what can be done with it. Also, we will prove more general results, which relate different models.

## 1.1 Why Study Theory?

A question commonly posed by practically minded students is

*Why study theory?*

Here are some answers. Some I like better than others!

1. It's a required course.

2. Theory gives exposure to ideas that permeate Computer Science: logic, sets, automata, grammars, recursion. Familiarity with these concepts will make you a better computer scientist.

3. Theory gives us mathematical (hence precise) descriptions of computational phenomena. This allows us to use mathematics, an intellectual inheritance with tools and techniques thousands of years old, to solve problems arising from computers.

4. It gives training in argumentation, which is a generally useful thing. As Lewis Carroll, author of *Alice in Wonderland*, dirty old man, and logician wrote:

   > *Once master the machinery of Symbolic Logic, and you have a mental occupation always at hand, of absorbing interest, and one that will be of real use to you in any subject you may take up. It will give you clearness of thought - the ability to see your way through a puzzle - the habit of arranging your ideas in an orderly and get-at-able form - and, more valuable than all, the power to detect fallacies, and to tear to pieces the flimsy illogical arguments, which you will so continually encounter in books, in newspapers, in speeches, and even in sermons, and which so easily delude those who have never taken the trouble to master this fascinating Art.*

5. It is required if you are interested in a research career in Computer Science.

6. A theory course distinguishes you from someone who has picked up programming at a 'job factory' technical school. (This is the *snob* argument, one which I don't personally believe in.)

7. Theory gives exposure to some of the absolute highpoints of human thought. For example, we will study the proof of the undecidability of the *halting problem*, a result due to Alan Turing in the 1930s. This theorem is one of the most profound intellectual developments—in any field—of the 20th Century. Pioneers like Turing have blazed trails deep into *terra incognita* and courses like **cs3100** allow us mere mortals to follow their footsteps.

8. Theory gives a nice setting for honing your problem solving skills. You probably haven't gotten smarter since you entered university but you have learned many subjects and—more importantly—you have been trained to solve problems. The belief is that improving your problem-solving ability through practice will help you in your career. Theory courses in general, and this one in particular, provide good exposure to a wide variety of problems, and the techniques you learn are widely applicable.

## 1.2 Overview

Although the subject matter of this course is *models of computation*, we need a framework—some support infrastructure—in which to work

**FRAMEWORK** is basic discrete mathematics, *i.e.*, some set theory, some logic, and some proof techniques.

**SUBJECT MATTER** is Automata, Grammars, and Computability

We will spend a little time recapitulating the framework, which you should have mastered in **cs2100**. You will not be tested on framework material, but you will get exactly nowhere if you don't know it.

Once the framework has been recalled, we will discuss the following subjects: Turing machines and computability; grammars and context-free languages; and finally, finite state automata and regular languages. The progression of topics will move from the abstract (computability) to the concrete (constructions on automata), and from fully expressive models of computation to more constrained models.

### 1.2.1 Computability

In this section, we will start by considering a classic model of computation—that of *Turing machines* (TMs). Unlike the other models we will study, a TM can do everything a modern computer can do (and more). The study of 'fully fledged', or unrestricted, models of computation, such as TMs, is known as *computability*.

We will see how to program TMs and, through experience, convince ourselves of their power, *i.e.*, that every algorithm can be programmed on a TM. We will also have a quick look at Register Machines, which are quite different from TMs, but of equivalent power. This leads to a discussion of 'what is an algorithm' and the Church-Turing thesis. Then we will see a limitative result: the undecidability of the halting problem. This states that it is not possible to mechanically determine whether or not an arbitrary program will halt on all inputs. At the time, this was a very surprising result. It has a profound influence on Computer Science since it can be leveraged to show that all manner of useful functionality that one might wish to have computers provide is, in fact, theoretically impossible.

### 1.2.2 Context-Free Grammars

Context-Free Grammars (CFGs) are a much more limited model of computation than Turing machines. Their prime application is that much, if not all, parsing of normal programming languages can be accomplished efficiently by parsers automatically generated from a CFG. *Parsing* is a stage in program compilation that maps from the *linear* strings of the program text into tree structures more easily dealt with by the later stages of compilation. This is the first—and probably most successful—application of generating programs from high-level specifications. CFGs are also useful in parsing human languages, although that is a far harder task to perform automatically. We will get a lot of experience with writing and analyzing grammars. The languages generated by context-free grammars are known as the *context-free* languages, and there is a class of machines used to process strings specified by CFGs, known as *push-down automata* (PDAs). We will also get some experience with constructing and analyzing PDAs.

**Pedantic Note**.The word is *grammar*, not *grammer*.
**Non-Pedantic Note**. The word *language* used here is special terminology and has little to do with the standard usage. Languages are set-theoretic entities and admit operations like union ($\cup$), intersection ($\cap$), concatenation, and replication (Kleene's 'star'). An important theme in the course is showing how simple operations on machines are reflected in these set-theoretic operations on languages.

### 1.2.3 Automata

Automata (singular: *automaton*) are a simple but very important class of computing devices. They are heavily used in compilers, text editors, VLSI circuits, Artificial Intelligence, databases, and embedded systems.

We will introduce and give a precise definition of *finite state automata* (FSAs) before investigating their extension to *non-deterministic* FSAs (NFAs). It turns out that FSAs are equivalent to NFAs, and we will prove this. We will discuss the languages recognized by FSAs, the so-called *regular* languages.

Automata are used to recognize, or accept, strings in a language. An alternative viewpoint is that of *regular expressions*, which *generate* strings. Regular expressions are equivalent to FSAs, and we will prove this.

Finally, we will prove the *pumping lemma* for regular languages. This, along with the undecidability of the halting problem, is another of what might be called *negative*, or limitative theorems, which show that there are some aspects of computation that are not captured by the model being considered. In other words, they show that the model is too weak to capture important notions.

**Historical Remark**. The history of the development of models of computation is a little bit odd, because the most powerful models were investigated first. The work of Turing (Turing machines), Church (lambda calculus), Post (Production Systems), and Goedel (recursive functions) on computability happened largely in the 1930's. These mathematicians were trying to nail down the notion of *algorithm*, and came up with quite different explanations. They were all right! Or at least that is the claim of the *Church-Turing Thesis*, an important philosophical statement, which we will discuss.

In the 1940's restricted notions of computability were studied, in order to give mathematical models of biological behaviour, such as the firing of neurons. These led to the development of automata theory. In the 1950's, formal grammars and the notion of context-free grammars (and much more) were invented by Noam Chomsky in his study of natural language.

# Chapter 2

# Background Mathematics

This should be review from **cs2100**, but we may be rusty after the summer layoff. We need some basic amounts of logic, set theory, and proof, as well as a smattering of other material.

## 2.1 Some Logic

Here is the syntax of the formulas of predicate logic. In this course we mainly use logical formulas to precisely express theorems.

| | |
|---|---|
| $A \wedge B$ | $conjunction$ |
| $A \vee B$ | $disjunction$ |
| $A \Rightarrow B$ | $(material) implication$ |
| $A$ iff $B$ | $equivalence$ |
| $\neg A$ | $negation$ |
| $\forall x.\ A$ | $universal\ quantification$ |
| $\exists x.\ A$ | $existential\ quantification$ |

After syntax we have semantics. The meaning of a formula is expressed in terms of *truth*.

- $A \wedge B$ is true iff $A$ is true and $B$ is true.
- $A \vee B$ is true iff $A$ is true or $B$ is true (or both are true).
- $A \Rightarrow B$ is true iff it is not the case that $A$ is true and $B$ is false.

- $A$ iff $B$ is true iff $A$ and $B$ have the same truth value.
- $\neg A$ is true iff $A$ is false
- $\forall x.\ A$ is true iff $A$ is true for all possible values of $x$.
- $\exists x.\ A$ is true iff $A$ is true for at least one value of $x$.

Note the recursion: the truth value of a formula depends on the truth values of its sub-formulas. This prevents the above definition from being circular. Also, note that the apparent circularity in defining iff by using 'iff' is only apparent—it would be avoided in a completely formal definition.

*Remark.* The definition of implication can be a little confusing. Implication is *not* 'if-then-else'. Instead, you should think of $A \Rightarrow B$ as meaning 'if $A$ is true, then $B$ must also be true. If $A$ is false, then it doesn't matter what $B$ is; the value of $A \Rightarrow B$ is true'.

Thus a statement such as $0 < x \Rightarrow x^2 \geq 1$ is true no matter what the value of $x$ is taken to be (supposing $x$ is an integer). This works well with universal quantification, allowing the statement $\forall x.\ 0 < x \Rightarrow x^2 \geq 1$ to be true. However, the price is that some plausibly false statements turn out to be true; for example: $0 < 0 \Rightarrow 1 < 0$. Basically, in an absurd setting, everything is held to be true.

**Example 1.** Suppose we want to write a logical formula that captures the following well-known saying:

> *You can fool all of the people some of the time, and you can fool some of the people all of the time, but you can't fool all of the people all of the time.*

We start by letting the atomic proposition $F(x, t)$ mean 'you can fool $x$ at time $t$'. Then the following formula

$$\begin{array}{l}(\forall x. \exists t.\ F(x,t)) \wedge \\ (\exists x. \forall t.\ F(x,t)) \wedge \\ \neg(\forall x. \forall t.\ F(x,t))\end{array}$$

precisely captures the statement. Notice that the first line asserts that each person could be fooled at a different time. If one wanted to express that there is a specific time at which everyone gets fooled, it would be

$$\exists t.\ \forall x.\ F(x,t)\ .$$

**Example 2.** What about

*Everybody loves my baby, but my baby don't love nobody but me.*

Let the atomic proposition $L(x, y)$ mean 'x loves y' and let $\mathbf{b}$ mean 'my baby' and let $\mathbf{me}$ stand for me. Then the following formula

$$(\forall x.\ L(x, \mathbf{b})) \land L(\mathbf{b}, \mathbf{me}) \land (\forall x.\ L(\mathbf{b}, x) \Rightarrow (x = \mathbf{me}))$$

precisely captures the statement. It is interesting to pursue what this means, since if everybody loves **b**, then **b** loves **b**. So I am my baby, which may be troubling for some.

**Example 3** (Lewis Carroll)**.** From the following assertions

1. There are no pencils of mine in this box.
2. No sugar-plums of mine are cigars.
3. The whole of my property, that is not in the box, consists of cigars.

we can conclude that no pencils of mine are sugar-plums. Transcribed to logic, the assertions are

$$\begin{array}{l}
\forall x.\ \mathit{inBox}(x) \Rightarrow \neg \mathit{Pencil}(x) \\
\forall x.\ \mathit{sugarPlum}(x) \land \mathit{Mine}(x) \Rightarrow \neg \mathit{Cigar}(x) \\
\forall x.\ \mathit{Mine}(x) \land \neg \mathit{inBox}(x) \Rightarrow \mathit{Cigar}(x)
\end{array}$$

From (1) and (3) we can conclude *All my pencils are cigars*. Now we can use this together with (2) to reach the conclusion

$$\forall x.\ \mathit{Pencil}(x) \land \mathit{Mine}(x) \Rightarrow \neg \mathit{sugarPlum}(x).$$

These examples feature somewhat whimsical subject matter. In the course we will be using symbolic logic when a high level of precision is needed.

## 2.2 Some Sets

A set is an collection of entities, often written with the syntax $\{e_1, e_2, \ldots, e_n\}$ when the set is finite. Making a set amounts to a decision to regard a collection of possibly disparate things as a single object. Here are some well-known mathematical sets:

- $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$. The booleans, also known as the bit values. In situations where no confusion with numbers is possible, one could have $\mathbb{B} = \{0, 1\}$.

- $\mathbb{N} = \{0, 1, 2, \ldots\}$. The *natural* numbers.

- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The integers.

- $\mathbb{Q} =$ the rational (fractional) numbers.

- $\mathbb{R} =$ the real numbers.

- $\mathbb{C} =$ the complex numbers.

**Note**. $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ will not be much used in the course, although $\mathbb{Q}$ and $\mathbb{R}$ will feature in one lecture.

**Note**. Some mathematicians think that $\mathbb{N}$ starts with 1. We will not adopt that approach in this course!

There is a rich collection of operations on sets. Interestingly, all these operations are ultimately built from *membership*.

**Membership of an element in a set** The notation $a \in S$ means that $a$ is a member, or element, of $S$. Similarly, $a \notin S$ means that $a$ is not an element of $S$.

**Equality of sets** Equality of sets $R$ and $S$ is defined $R = S$ iff $(\forall x.\ x \in R \text{ iff } x \in S)$. Thus two sets are equal just when they have the same elements. Note that sets have no intrinsic order. Thus $\{1, 2\} = \{2, 1\}$. Also, sets have no duplicates. Thus $\{1, 2, 1, 1\} = \{2, 1\}$.

**Subset** $R$ is a subset of $S$ if every element of $R$ is in $S$, but $S$ may have extras. Formally, we write $R \subseteq S$ iff $(\forall x.\ x \in R \Rightarrow x \in S)$. Having $\subseteq$ available allows an (equivalent) reformulation of set equality: $R = S$ iff $R \subseteq S \wedge S \subseteq R$.

A few more useful facts about $\subseteq$:

- $S \subseteq S$, for every set $S$.

- $P \subseteq Q \wedge Q \subseteq R \Rightarrow P \subseteq R$

There is also a useful notion of *proper* subset: $R \subset S$ means that all elements of $R$ are in $S$, but $S$ has one or more extras. Formally, $R \subset S$ iff $R \subseteq S \wedge R \neq S$.

It is a common error to confuse $\in$ and $\subseteq$. For example, $x \in \{x, y, z\}$, but that doesn't allow one to conclude $x \subseteq \{x, y, z\}$. However, it is true that $\{x\} \subseteq \{x, y, z\}$

**Union** The union of $R$ and $S$, $R \cup S$, is the set of elements occurring in $R$ or $S$ (or both). Formally, union is defined in terms of $\vee$: $x \in R \cup S$ iff $(x \in R \vee x \in S)$.

$$\{1, 2\} \cup \{4, 3, 2\} = \{1, 2, 3, 4\}$$

**Intersection** The intersection of $R$ and $S$, $R \cap S$, is the set of elements occurring in both $R$ and $S$. Formally, intersection is defined in terms of $\wedge$: $x \in R \cap S$ iff $(x \in R \wedge x \in S)$.

$$\{1, 2\} \cap \{4, 3, 2\} = \{2\}$$

**Singleton sets** A set with one element is called a *singleton*. Note well that a singleton set is not the same as its element: $\forall x.\ x \neq \{x\}$, even though $x \in \{x\}$, for any $x$.

**Set difference** $R - S$ is the set of elements that occur in $R$ but not in $S$. Thus, $x \in R - S$ iff $x \in R \wedge x \notin S$. Note that $S$ may have elements not in $R$. These are ignored. Thus

$$\{1, 2, 3\} - \{2, 4\} = \{1, 3\}.$$

**Universe and complement** Often we work in a setting where all sets are subsets of some fixed set $\mathcal{U}$ (sometimes called the *universe*). In that case we can write $\overline{S}$ to mean $\mathcal{U} - S$. For example, if our universe is $\mathbb{N}$, and $\mathit{Even}$ is the set of even numbers, then $\overline{\mathit{Even}}$ is the set of odd numbers.

**Example 4.** Let us take the Flintstone characters as our universe.

$$\begin{array}{rcl} F & = & \{\mathit{Fred}, \mathit{Wilma}, \mathit{Pebbles}, \mathit{Dino}\} \\ R & = & \{\mathit{Barney}, \mathit{Betty}, \mathit{BamBam}\} \\ \mathcal{U} & = & F \cup R \cup \{\mathit{Mr.\ Slate}\} \end{array}$$

Then we know

$$\emptyset = F \cap R$$

because the two families are disjoint. Also, we can see that

$$F - \{\mathit{Fred}, \mathit{Mr.\ Slate}\} = \{\mathit{Wilma}, \mathit{Pebbles}, \mathit{Dino}\}.$$

What about $\mathit{Fred} \subseteq F$ ? It makes no sense because $\mathit{Fred}$ is not a set. The subset operation requires two sets. However, $\{\mathit{Fred}\} \subseteq F$ is true; indeed $\{\mathit{Fred}\} \subset F$.

We also know

$$\{\mathit{Mr.\ Slate}, \mathit{Fred}\} \not\subseteq F$$

since Mr. Slate is not an element of $F$. Finally, we know that

$$\overline{F \cup R} = \{\mathit{Mr.\ Slate}\}$$

*Remark.* Set difference can be defined in terms of intersection and complement:

$$A - B = A \cap \overline{B}$$

**Empty set** The symbol $\emptyset$ stands for the empty set: the set with no elements. The notation $\{\}$ may also be used. The empty set acts as an algebraic identity for several operations:

$$\begin{array}{l} \emptyset \cup S = S \\ \emptyset \cap S = \emptyset \\ \emptyset \subseteq S \\ \emptyset - S = \emptyset \\ S - \emptyset = S \\ \overline{\emptyset} = \mathcal{U} \end{array}$$

**Set comprehension** This is also known as *set builder notatation*. The notation is

$$\{ \underbrace{\quad\quad}_{\text{template}} \mid \underbrace{\quad\quad}_{\text{condition}} \}.$$

This denotes the set of all items *matching* the template, which also meet the condition. This, combined with logic, gives a natural way to concisely describe sets:

$$\begin{array}{lcl}
\{x \mid x < 1\} & = & \{0\} \\
\{x \mid x > 1\} & = & \{2, 3, 4, 5, \ldots\} \\
\{x \mid x \in R \land x \in S\} & = & R \cap S \\
\{x \mid \exists y. x = 2y\} & = & \{0, 2, 4, 6, 8, \ldots\} \\
\{x \mid x \in \mathcal{U} \land x \text{ is male}\} & = & \{\mathit{Fred}, \mathit{Barney}, \mathit{BamBam}, \mathit{Mr.\ Slate}\}
\end{array}$$

The template can be a more complex expression, as we will see.

**Indexed union and intersection** It sometimes happens that one has a set of sets

$$\{ \underbrace{\{\ldots\}}_{S_1}, \ldots, \underbrace{\{\ldots\}}_{S_n} \}$$

and wants to 'union (or intersect) them all together', as in

$$\begin{array}{c}
S_1 \cup \ldots \cup S_n \\
S_1 \cap \ldots \cap S_n
\end{array}$$

These operations, known to some as *bigunion* and *bigintersection*, can be formally defined in terms of index sets:

$$\textstyle\bigcup_{i \in I} S_i = \{x \mid \exists i.\ i \in I \land x \in S_i\}$$

$$\textstyle\bigcap_{i \in I} S_i = \{x \mid \forall i.\ i \in I \Rightarrow x \in S_i\}$$

The generality obtained from using index sets allows one to take the bigunion of an infinite set of sets.

**Power set** The set of all subsets of a set $S$ is known as the *powerset* of $S$, written variously as $\mathcal{P}(\mathcal{S})$, $Pow(S)$, or $2^S$.

$$Pow(S) = \{s \mid s \subseteq S\}$$

For example,

$$Pow\{1,2,3\} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

If a finite set is of size $n$, the size of its powerset is $2^n$. A powerset is always larger than the set it is derived from, even if the orginal set is infinite.

**Product of sets** $R \times S$, the product of two sets $R$ and $S$, is made by pairing each element of $R$ with each element of $S$. Using set-builder notation, this can be concisely expressed:

$$R \times S = \{(x,y) \mid x \in R \wedge y \in S\}.$$

**Example 5.**

$$F \times R = \left\{ \begin{array}{l} (Fred, Barney),\ (Fred, Betty),\ (Fred, BamBam), \\ (Wilma, Barney),\ (Wilma, Betty),\ (Wilma, BamBam), \\ (Pebbles, Barney),\ (Pebbles, Betty),\ (Pebbles, BamBam), \\ (Dino, Barney),\ (Dino, Betty),\ (Dino, BamBam) \end{array} \right\}$$

In general, the size of the product of two sets will be the product of the sizes of the two sets.

**Iterated product** The iterated product of sets $S_1, S_2, \ldots, S_n$ is written $S_1 \times S_2 \times \ldots \times S_n$. It stands for the set of all $n$-tuples $(a_1, \ldots, a_n)$ such that $a_1 \in S_1, \ldots a_n \in S_n$. Slightly more formally, we could write

$$S_1 \times S_2 \times \ldots \times S_n = \{(a_1, \ldots a_n) \mid a_1 \in S_1 \wedge \ldots \wedge a_n \in S_n\}$$

An $n$-tuple $(a_1, \ldots a_n)$ is formally written as $(a_1, (a_2, \ldots, (a_{n-1}, a_n) \ldots))$, but, by convention, the parentheses are dropped. For example, $(a, b, c, d)$ is the conventional way of writing the 4-tuple $(a, (b, (c, d)))$. Unlike sets, tuples are *ordered*. Thus $a$ is the first element of the tuple, $b$ is the second, $c$ is the third, and $d$ is the fourth. Equality on $n$-tuples is captured by the following property:

$$(a_1, \ldots a_n) = (b_1, \ldots b_n) \text{ iff } a_1 = b_1 \wedge \ldots \wedge a_n = b_n.$$

It is important to remember that sets and tuples are different. For example,

$$(a, b, c) \neq \{a, b, c\}.$$

**Size of a set** The size of a set, also known as its *cardinality*, is just the number of elements in the set. It is common to write $|A|$ to denote the cardinality of set $A$.

$$|\ \{\mathsf{foo}, \mathsf{bar}, \mathsf{baz}\}\ | = 3$$

Cardinality for finite sets is straightforward; however it is worth noting that there is a definition of cardinality that applies to both finite and infinite sets: under that definition it can be proved that not all infinite sets have the same size! We will discuss this later in the course.

### Summary of useful properties of sets

Now we supply a few identities which are useful for manipulating expressions involving sets. The equalities can all be proved by expanding definitions. To begin with, we give a few simple facts about union, intersection, and the empty set.

$$\begin{array}{rcl}
A \cup B & = & B \cup A \\
A \cap B & = & B \cap A \\
A \cup A & = & A \\
A \cap A & = & A \\
A \cup \emptyset & = & A \\
A \cap \emptyset & = & \emptyset
\end{array}$$

The following identities are associative, distributive, and absorptive properties:

$$\begin{array}{rcl} A \cup (B \cup C) & = & (A \cup B) \cup C \\ A \cap (B \cap C) & = & (A \cap B) \cap C \\ \\ A \cap (B \cup C) & = & (A \cap B) \cup (A \cap C) \\ A \cup (B \cap C) & = & (A \cup B) \cap (A \cup C) \\ \\ A \cup (A \cap B) & = & A \\ A \cap (A \cup B) & = & A \end{array}$$

The following identities are the so-called *De Morgan* laws, plus a few others.

$$\begin{array}{rcl} \overline{A \cup B} & = & \overline{A} \cap \overline{B} \\ \overline{A \cap B} & = & \overline{A} \cup \overline{B} \\ \\ \overline{\overline{A}} & = & A \\ A \cap \overline{A} & = & \emptyset \end{array}$$

### 2.2.1 Functions

Informally, a function is a mechanism that takes an input and gives an output. One can also think of a function as a table, with the arguments down one column, and the results down another. In fact, if a function is finite, a table can be a good way to present it. Formally however, a *function* $f$ is a set of ordered pairs with the property

$$(a, b) \in f \land (a, c) \in f \Rightarrow b = c$$

This just says that a function is, in a sense, *univocal*, or *deterministic*: there is only one possible output for an input. Of course, the notation $f(a) = b$ is preferred over $(a, b) \in f$. The *domain* and *range* of a function $f$ are defined as follows:

$$\begin{array}{rcl} \mathsf{Dom}(f) & = & \{x \mid \exists y.\ f(x) = y\} \\ \mathsf{Rng}(f) & = & \{y \mid \exists x.\ f(x) = y\} \end{array}$$

**Example 6.** The notation $\{(n, n^2) \mid n \in \mathbb{N}\}$ specifies the function $f(n) = n^2$. Furthermore, $\mathsf{Dom}(f) = \mathbb{N}$ and $\mathsf{Rng}(f) = \{0, 1, 4, 9, 16, \ldots\}$.

A common notation for specifying that function $f$ has domain $A$ and range $B$ is the following:

$$f : A \rightarrow B$$

Another common usage is to say 'a function *over* (or *on*) a set'. This just means that the function takes its inputs from the specified set. As a trivial example, consider $f$, a function over $\mathbb{N}$, described by $f(x) = x + 2$.

**Partial functions** A function can be *total* or *partial*. Every element in the domain of a total function has a corresponding element in its range. In contrast, a partial function may not have any element in the range corresponding to some element of the domain.

**Example 7.** $f(n) = n^2$ is a total function on $\mathbb{N}$. On the other hand, suppose we want a function from Flintstones to the 'most-similar' Rubble:

| Flintstone | $\mapsto$ | Rubble |
|---|---|---|
| Fred | | Barney |
| Wilma | | Betty |
| Pebbles | | BamBam |
| Dino | | ?? |

This is best represented by a partial function that doesn't map Dino to anything.

**Subtle Point**. The notation $f : A \rightarrow B$ is usually taken to mean that the domain of $f$ is $A$, and the range is $B$. This will indeed be true when $f$ is a total function. However, if $f$ is partial, then the domain of $f$ can be a proper subset of $A$. For example, the specification $\mathit{Flintstone} \rightarrow \mathit{Rubble}$ could be used for the function presented above. □

Sometimes functions are specified (or implemented) by *algorithms*. We will study in detail how to define the general notion of what an algorithm is in the course, but for now, let's use our existing understanding. Using algorithms to define functions can lead to three kinds of partiality:

1. the algorithm hits an exception state, *e.g.*, attempting to divide by zero;

2. the algorithm goes into an infinite loop;

3. the algorithm runs for a very very very long time before returning an answer.

The second and third kind of partiality are similar but essentially different. Pragmatically, there is no difference between a program that will never return and one that will return after a trillion years. However, *theoretically* there is a huge difference: instances of the second kind are truly partial functions, while instances of the third are still total functions. A course in computational complexity explores the similarities and differences between the options.

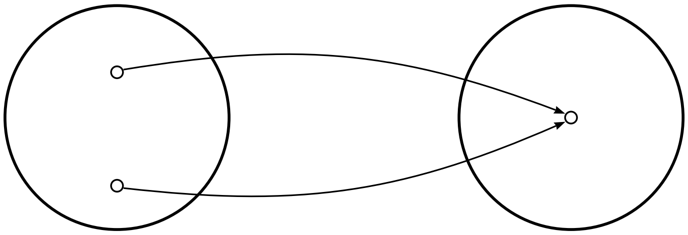
If a partial function $f$ is *defined* at an argument $a$, then we write $f(a) \downarrow$. Otherwise, $f(a)$ is undefined and we write $f(a) \uparrow$.

**Believe it or not**. $\emptyset$ is a function. It's the *nowhere defined* function.

**Injective and Surjective functions** An *injective*, or *one-to-one* function sends different elements of the domain to different elements of the range:

$$\textsf{Injective}(f) \text{ iff } \forall x\ y.\ x \in \textsf{Dom}(f) \wedge y \in \textsf{Dom}(f) \wedge x \neq y \Rightarrow f(x) \neq f(y).$$

Pictorially, the following situation is avoided by an injective function:



An important consequence : if there's an injection from $A$ to $B$, then $B$ is at least the size of $A$. For example, there is no injection from Flintstones to Rubbles, but there are injections in the other direction.

A *surjective*, or *onto* function is one in which every element of the range is produced by some application of the function:

$$\textsf{Surjective}(f : A \rightarrow B) \text{ iff } \forall y.\ y \in B \Rightarrow \exists x.\ x \in A \wedge f(x) = y$$

A *bijection* is a function that is both injective and surjective.

**Example 8** (Square root in $\mathbb{N}$)**.** Let $\sqrt{n}$ denote the number $x \in \mathbb{N}$ such that $x^2 \leq n$ and $(x+1)^2 > n$.

| $n$ | $\sqrt{n}$ |
|---|---|
| $0$ | $0$ |
| $1, 2, 3$ | $1$ |
| $4, 5, 6, 7, 8$ | $2$ |
| $9, 10, 11, 12, 13, 14, 15$ | $3$ |
| $16$ | $4$ |
| $\vdots$ | $\vdots$ |

This function is surjective, because all elements of $\mathbb{N}$ appear in the range; it is however, not injective, for multiple elements of the domain map to a single element of the range.

**Closure** Closure is a powerful idea, and it is used repeatedly in this course. Suppose $S$ is a set. If, for any $x \in S$, $f(x) \in S$, then $S$ is said to be *closed under* $f$.
For example, $\mathbb{N}$ is closed under squaring:

$$\forall n.\ n \in \mathbb{N} \Rightarrow n^2 \in \mathbb{N}.$$

A counter-example: $\mathbb{N}$ is not closed under subtraction: $2-3 \notin \mathbb{N}$ (unless subtraction is somehow re-defined so that $p - q = 0$ when $p < q$).

The 'closure' terminology can be used for functions taking more than one argument; thus, for example, $\mathbb{N}$ is closed under $+$.

## 2.3 Alphabets and Strings

An *alphabet* is a finite set of symbols, usually defined at the start of a problem statement. Commonly, $\Sigma$ is used to denote an alphabet.

**Examples**

$$\begin{array}{l} \Sigma = \{0, 1\} \\ \Sigma = \{a, b, c, d\} \\ \Sigma = \{foo, bar\} \end{array}$$

**Non-examples**

- $\mathbb{N}$ (or any infinite set)
- sets having symbols with shared substructure, *e.g.*, $\{foo, foobar\}$, since this can lead to nasty, horrible ambiguity.

### 2.3.1 Strings

A *string over an alphabet* $\Sigma$ is a finite sequence of symbols from $\Sigma$. For example, if $\Sigma = \{0, 1\}$, then $000$ and $0100001$ are strings over $\Sigma$. The strings provided in most programming languages are over the alphabet provided by the ASCII characters (and more extensive alphabets, such as Unicode, are common).

**NB**. Authors are sometimes casual about representing operations on strings: for example, string construction and string concatenation are both written by adjoining blocks of text. This is usually OK, but can be ambiguous: if $\Sigma$ = $\{o, f, a, b, r\}$ we could write the string $foobar$, or $f \cdot o \cdot o \cdot b \cdot a \cdot r$ (to be really precise). Similarly, if $\Sigma = \{foo, bar\}$, then we could also write $foobar$, or $foo \cdot bar$.

**The empty string** There is a unique string $\varepsilon$ which is the *empty* string. There is an analogy between $\varepsilon$ for strings and 0 for $\mathbb{N}$. For example, both are very useful as identity elements.
**NB**. Some authors use $\Lambda$ to denote the empty string.
**NB**. The empty string is not a symbol, it's a string with no symbols in it. Therefore $\varepsilon$ can't appear in an alphabet.

**Length** The length of a string $s$, written $\mathsf{len}(s)$, is obtained by counting each symbol from the alphabet in it. Thus, if $\Sigma = \{f, o, b, a, r\}$, then

$$\begin{array}{l} \mathsf{len}(\varepsilon) = 0 \\ \mathsf{len}(foobar) = 6 \end{array}$$

but $\mathsf{len}(foobar) = 2$, if $\Sigma = \{foo, bar\}$.
**NB**. Unlike some programming languages, strings are not terminated with an invisible $\varepsilon$ symbol.

**Concatentation** The concatenation of two strings $x$ and $y$ just places them next to each other, giving the new string $xy$. If we needed to be precise, we could write $x \cdot y$. Some properties of concatenation:

$$\begin{array}{ll} x(yz) = (xy)z & \text{associativity} \\ x\varepsilon = \varepsilon x = x & \text{identity} \\ \mathsf{len}(xy) = \mathsf{len}(x) + \mathsf{len}(y) & \end{array}$$

The *iterated concatenation* $x^n$ of a string $x$ is the $n$-fold concatenation of $x$ with itself.

**Example 9.** Let $\Sigma = \{a, b\}$. Then

$$\begin{array}{lcl} (aab)^3 & = & aabaabaab \\ (aab)^1 & = & aab \\ (aab)^0 & = & \varepsilon \end{array}$$

The formal definition of $x^n$ is by recursion:

$$\begin{array}{lcl} x^0 & = & \varepsilon \\ x^{n+1} & = & x^n \cdot x \end{array}$$

**Notation**. Repeated elements in a string can be superscripted, for convenience: for example, $aab = a^2b$.

**Counting** If $x$ is a string over $\Sigma$ and $a \in \Sigma$, then $\mathsf{count}(a, x)$ gives the number of occurrences of $a$ in $x$:

$$\begin{array}{lcl} \mathsf{count}(0, 0010) & = & 3 \\ \mathsf{count}(1, 000) & = & 0 \\ \mathsf{count}(0, \varepsilon) & = & 0 \end{array}$$

The formal definition of $\mathsf{count}$ is by recursion:

$$\begin{array}{lcl} \mathsf{count}(a, \varepsilon) & = & 0 \\ \mathsf{count}(a, b \cdot t) & = & \texttt{if}\ a = b\ \texttt{then}\ \mathsf{count}(a, t) + 1\ \texttt{else}\ \mathsf{count}(a, t) \end{array}$$

In the second clause of this definition, the expression $(b \cdot t)$ should be understood to mean that $b$ is a symbol concatenated to string $t$.

**Prefix** A string $x$ is a *prefix* of string $y$ iff there exists $w$ such that $y = x \cdot w$. For example, $abaab$ is a prefix of $abaababa$. Some properties of prefix:

- $\varepsilon$ is a prefix to every string
- $x$ is a prefix to $x$, for any string $x$.

A string $x$ is a *proper prefix* of string $y$ if $x$ is a prefix of $y$ and $x \neq y$.

**Reversal** The reversal $x^{\mathcal{R}}$ of a string $x = x_1 \cdot \ldots \cdot x_n$ is the string $x_n \cdot \ldots \cdot x_1$.

**Pitfalls** Here are some common mistakes people make when first confronted with sets and strings. All the following are true, but surprise some students.

- **<u>sets</u>** $\{a, b\} = \{b, a\}$

  **<u>strings</u>** $ab \neq ba$
- **<u>sets</u>** $\{a, a, b\} = \{a, b\}$

  **<u>strings</u>** $aab \neq ab$
- $\underbrace{\emptyset}_{\text{empty set}} \neq \underbrace{\varepsilon}_{\text{empty string}} \neq \underbrace{\{\varepsilon\}}_{\text{singleton set holding empty string}}$

Also, sometimes people seem to reason as follows:

> *The empty set has no elements in it. The empty string has no characters in it. So . . . the empty set is the same as the empty string.*

The first two assertions are true; however, the conclusion is false. Although the length of $\varepsilon$ is 0, and the size of $\emptyset$ is also 0, they are two quite different things.

## 2.4 Languages

So much for strings. Now we discuss *sets of strings*, also called *languages*. Languages are one of the important themes of the course.

We will start our discussion with $\Sigma^*$, the set of all strings over alphabet $\Sigma$. The set $\Sigma^*$ contains all strings that can be generated by iteratively concatenating symbols from $\Sigma$, any number of times.

**Example 10.** If $\Sigma = \{a, b, c\}$,

$$\Sigma^* = \{\underbrace{\varepsilon}_{\text{NB}}, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, \ldots\}$$

*Question*: What if $\Sigma = \emptyset$? What is $\Sigma^*$ then?
*Answer*: $\emptyset^* = \{\varepsilon\}$. It may seen odd that you can proceed from the empty set to a non-empty set by iterated concatenation. There is a reason for this, but for the moment, please accept this as convention.

If $\Sigma$ is a non-empty set, then $\Sigma^*$ is an infinite set, where each element is a finite string.

**Convention**. Lower case letters at the end of the alphabet, *e.g.*, $u, v, w, x, y, z$, are used to represent strings. Capital letters from the beginning of the alphabet *e.g.*, $A, B, C, L$ are used to represent languages.

### Set operations on languages

Now we apply set operations to languages; this will therefore be nothing new, since we've seen set operations already. However, it's worth the repetition.

**Union**

$$\{a, b, ab\} \cup \{a, c, ba\} = \{a, b, ab, c, ba\}$$

**Intersection**

$$\{a, b, ab\} \cap \{a, c, ba\} = \{a\}$$

**Complement** Usually, $\Sigma^*$ is the universe that a complement is taken with respect to. Thus

$$\overline{A} = \{x \in \Sigma^* \mid x \notin A\}$$

For example

$$\overline{\{x \mid \mathsf{len}(x) \text{ is even}\}} = \{x \in \Sigma^* \mid \mathsf{len}(x) \text{ is odd}\}$$

Now we lift operations on strings to work over *sets of strings*.

**Language reversal** The reversal of language $A$ is written $A^{\mathcal{R}}$ and is defined (note the overloading)

$$A^{\mathcal{R}} = \{x^{\mathcal{R}} \mid x \in A\}$$

**Language concatenation** The concatenation of languages $A$ and $B$ is defined:

$$AB = \{xy \mid x \in A \wedge y \in B\}$$

or using the 'dot' notation to emphasize that we are concatenating (note the overloading of $\cdot$):

$$A \cdot B = \{x \cdot y \mid x \in A \wedge y \in B\}$$

**Example 11.** $\{a, ab\}\ \{b, ba\} = \{ab, abba, aba, abb\}$

**Example 12.** Two languages $L_1$ and $L_2$ such that $L_1 \cdot L_2 = L_2 \cdot L_1$ and $L_1$ is not a subset of $L_2$ and $L_2$ is not a subset of $L_1$ and neither language is $\{\varepsilon\}$ are the following:

$$L_1 = \{aa\} \qquad\qquad L_2 = \{aaa\}$$

**Notes**

- In general $AB \neq BA$. Example: $\{a\}\{b\} \neq \{b\}\{a\}$.
- $A \cdot \emptyset = \emptyset = \emptyset \cdot A$.
- $A \cdot \{\varepsilon\} = A = \{\varepsilon\} \cdot A$.
- $A \cdot \varepsilon$ is nonsense—it's syntactically malformed.

**Iterated language concatenation** Well, if we can concatenate two languages, then we can certainly repeat this to concatenate any number of languages. Or concatenate a language with itself any number of times. The operation $A^n$ denotes the concatenation of $A$ with itself $n$ times. The formal definition is

$$\begin{array}{lcl} A^0 & = & \{\varepsilon\} \\ A^{n+1} & = & A \cdot A^n \end{array}$$

Another way to characterize this is that a string is in $A^n$ if it can be split into $n$ pieces, each of which is in $A$:

$$x \in A^n \text{ iff } \exists w_1 \ldots w_n.\ w_1 \in A \wedge \ldots \wedge w_n \in A \wedge (x = w_1 \cdots w_n).$$

**Example 13.** Let $A = \{a, ab\}$. Thus $A^3 = A \cdot A \cdot A \cdot \{\varepsilon\}$, by unrolling the formal definition. To expand further:

$$\begin{array}{ll} A \cdot A \cdot A \cdot \{\varepsilon\} & = A \cdot A \cdot A \\ & = A \cdot \{aa, aab, aba, abab\} \\ & = \{a, ab\} \cdot \{aa, aab, aba, abab\} \\ & = \{aaa, aaba, abaa, ababa, aaab, aabab, abaab, ababab\} \end{array}$$

**Kleene's Star** It happens that$A^n$ is sometimest limited because each string in it has been built by exactly $n$ concatenations of strings from $A$. A more general operation, which addresses this shortcoming, is the so-called *Kleene Star* operation.[1]

$$\begin{array}{lcl} A^* & = & \bigcup_{n \in \mathbb{N}} A^n \\ & = & A^0 \cup A^1 \cup A^2 \cup \ldots \\ & = & \{x \mid \exists n.\ x \in A^n\} \\ & = & \{x \mid x \text{ is the concatenation of zero or more strings from } A\} \end{array}$$

Thus $A^*$ is the set of all strings derivable by any number of concatenations of strings in $A$. The notion of all strings obtainable by *one or more* concatenations of strings in $A$ is often used, and is defined $A^+ = A \cdot A^*$, *i.e.,*

[1] Named for its inventor, the famous American logician Stephen Kleene (pronounced 'Klee-knee').

$$A^+ = \bigcup_{n>0} A^n = A^1 \cup A^2 \cup A^3 \cup \ldots$$

**Example 14.**

$$\begin{array}{rcl}
A & = & \{a, ab\} \\
A^* & = & A^0 \cup A^1 \cup A^2 \cup \ldots \\
 & = & \{\varepsilon\} \cup \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots \\
A^+ & = & \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots
\end{array}$$

Some facts about Kleene star:

- The previously introduced definition of $\Sigma^*$ is an instance of Kleene star.
- $\varepsilon$ is in $A^*$, for every language $A$, including $\emptyset^* = \{\varepsilon\}$.
- $L \subseteq L^*$.

**Example 15.** An infinite language $L$ over $\{a, b\}$ for which $L \neq L^*$ is the following:

$$L = \{a^n \mid n \text{ is odd}\} .$$

□

A common situation when doing proofs about Kleene star is reasoning with formulas of the form $x \in L^*$, where $L$ is a perhaps complicated expression. A useful approach is to replace $x \in L^*$ by $\exists n. x \in L^n$ before proceeding.

**Example 16.** Prove $A \subseteq B \Rightarrow A^* \subseteq B^*$.

*Proof.* Assume $A \subseteq B$. Now suppose that $w \in A^*$. Therefore, there is an $n$ such that $w \in A^n$. That means $w = x_1 \cdot \ldots \cdot x_n$ where each $x_i \in A$. By the assumption, each $x_i \in B$, so $w \in B^n$, so $w \in B^*$.

□

That concludes the presentation of the basic mathematical objects we will be dealing with: strings and their operations; languages and their operations.

**Summary of useful properties of languages**

Since languages are just sets of strings, the identities from Section 2.2 may freely be applied to language expressions. Beyond those, there are a few others:

$$
\begin{array}{lcl}
A \cdot (B \cup C) & = & (A \cdot B) \cup (A \cdot C) \\
(B \cup C) \cdot A & = & (B \cdot A) \cup (C \cdot A) \\
A \cdot (B_0 \cup B_1 \cup B_2 \cup \ldots) & = & (A \cdot B_0) \cup (A \cdot B_1) \cup (A \cdot B_2) \ldots \\
(B_0 \cup B_1 \cup B_2 \cup \ldots) \cdot A & = & (B_0 \cdot A) \cup (B_1 \cdot A) \cup (B_2 \cdot A) \cup \ldots \\
 & & \\
A^{**} = (A^*)^* & = & A^* \\
A^* \cdot A^* & = & A^* \\
A^* & = & \{\varepsilon\} \cup A^+ \\
\emptyset^* & = & \{\varepsilon\}
\end{array}
$$

## 2.5 Proof

Now we discuss proof. In this course we will go through many proofs; indeed, in order to pass this course, you will have to write correct proofs of your own. This raises the weighty question

*What is a proof?*

which has attracted much philosophical discussion over the centuries. Here are some (only a few) answers:

- *A proof is a convincing argument (that an assertion is true).* This is on the right track, but such a broad encapsulation is too vague. For example, suppose the argument convinces some people and not others (some people are more gullible than others, for instance).

- *A proof is an argument that convinces everybody.* This is too strong, since some people aren't rational: such a person might never accept a perfectly good argument.

- *A proof is an argument that convinces every "sane" person.* This just pushes the problem off to defining sanity, which may be even harder!

- *A proof is an argument that convinces a machine.* If humans cause so much trouble, let's banish them in favour of machines! After all, machines have the advantage of being faster and more reliable than humans. In the late 19th and early 20th Centuries, philosophers and mathematicians developed the notion of a *formal proof*, one which is a chain of extremely simple reasoning steps expressed in a rigorously circumscribed language. After computers were invented, people realized that such proofs could be automatically processed: a computer program could analyze a purported proof and render a yes/no verdict, simply by checking all the reasoning steps.

  This approach is quite fruitful (it's my research area) but the proofs are far too detailed for humans to deal with: they can take megabytes for even very simple proofs. In this course, we are after proofs that are readable but still precise enough that mistakes can easily be caught.

- *A proof is an argument that convinces a skeptical but rational person who has knowledge of the subject matter.* Such as me and the TAs. This is the notion of proof adopted by professional mathematicians, and we will adopt it. One consequence of this definition is that there may be grounds for you to believe a proof and for us not to. In that case, dialogue is needed, and we encourage you to come to us when our proofs don't convince you (and when yours don't convince us).

### 2.5.1 Review of proof terminology

Following is some of the specialized vocabulary surrounding proofs:

**Definition** The introduction of a new concept, in terms of existing concepts. An example: a *prime* number is defined to be a number greater than 1 whose factors are just 1 and itself. Formally, this is expressed as a predicate on elements of $\mathbb{N}$, and it relies on a definition of when a number evenly divides another:

$$\begin{array}{l} \mathsf{divides}(x, y) = \exists z . x * z = y \\ \mathsf{prime}(n) = 1 < n \land \forall k.\ \mathsf{divides}(k, n) \Rightarrow k = 1 \lor k = n \end{array}$$

**Proposition** A statement thought to be true.

**Conjecture** An unproved proposition. A conjecture has the connotation that the author has attempted—but failed—to prove it.

**Theorem** A proved proposition/conjecture.

**Lemma** A theorem. A lemma is usually a stepping-stone to a more important theorem. However, some lemmas are quite famous on their own, *e.g.*, König's Lemma, since they are so often used.

**Corollary** A theorem; generally a simple consequence of another theorem.

### 2.5.2 Review of methods of proof

Often students are perplexed at how to write a proof down, even when they have a pretty good idea of why the assertion is true. Some of the following comments and suggestions could help; however, we warn you that knowing how to do proofs is a skill that is learned by practice. We will proceed syntactically.

**To prove** $A \Rightarrow B$**:** The standard way to prove this is to assume $A$, and use that extra ammunition to prove $B$. Another (equivalent) way that is sometimes convenient is the *contrapositive*: assume $\neg B$ and prove $\neg A$.

**To prove** $A$ iff $B$**:** There are three ways to deal with this (the first one is most common):

- prove $A \Rightarrow B$ and also $B \Rightarrow A$
- prove $A \Rightarrow B$ and also $\neg A \Rightarrow \neg B$
- find an intermediate formula $A'$ and prove
  - $A$ iff $A'$ and
  - $A'$ iff $B$.

  These can be strung together in an 'iff chain' of the form:

$$A \text{ iff } A' \text{ iff } A'' \text{ iff } \ldots \text{ iff } B.$$

**To prove** $A \wedge B$**:** Separately prove $A$ and $B$.

**To prove $A \vee B$:** Rarely happens. Select which-ever of $A$, $B$ seems to be true and prove it.

**To prove $\neg A$:** Assume $A$ and prove a contradiction. This will be discussed in more depth later.

**To prove $\forall x.\ A$:** In order to prove a universally quantified statement, we have to prove it taking an *arbitrary but fixed* element to stand for the quantified variable. As an example statement, let's take *"for all $n$, $n$ is less than 6 implies $n$ is less than 5"*. (Of course this isn't true, but nevermind.) More precisely, we'd write $\forall n.\ n < 6 \Rightarrow n < 5$. In other words, we'd have to show

$$u < 6 \Rightarrow u < 5$$

for an arbitrary $u$.

Not all universal statements are proved in this way. In particular, when the quantification is over numbers (or other structured data, such as strings), one often uses induction or case analysis. We will discuss these in more depth shortly.

**To prove $\exists x.\ A$:** Supply a *witness* for $x$ that will make $A$ true. For example, if we needed to show $\exists x.\ \mathsf{even}(x) \wedge \mathsf{prime}(x)$, we would give the witness 2 and continue on to prove $\mathsf{even}(2) \wedge \mathsf{prime}(2)$.

**Proof by contradiction** In a proof of proposition $P$ by contradiction, we begin by assuming that $P$ is false, *i.e.*, that $\neg P$ is true. Then we use this assumption to derive a contradiction, usually by proving that some already established fact $Q$ is false. But that can't be, since $Q$ has a proof. We have a contradiction. Then we reason that we must have been mistaken to assume $\neg P$, so therefore $\neg P$ is false. Hence $P$ is true after all.

It must be admitted that this is a more convoluted method of proof than the others, but it often allows very nice arguments to be given.

It's an amazing fact that proof by contradiction can be understood in terms of programming: the erasing of all the reasoning between the initial assumption of $\neg P$ and the discovery of the contradiction is similar to what happens if an exception is raised and caught when a program in Java or ML executes. This correspondence was recognized and made mathematically precise in the late 1980's by Tim Griffin, then a PhD student at Cornell.

### 2.5.3 Some simple proofs

We now consider some proofs about sets and languages. Proving equality between sets $P = Q$ can be reduced to proving $P \subseteq Q \wedge Q \subseteq P$, or, equivalently, by showing

$$\forall x.\ x \in P \text{ iff } x \in Q$$

We will use the latter in the following proof, which will exercise some basic definitions.

**Example 17** ($\overline{A \cap B} = \overline{A} \cup \overline{B}$)**.** This proposition is equivalent to $\forall x.\ x \in \overline{A \cap B}$ iff $x \in (\overline{A} \cup \overline{B})$. We will transform the lhs[2] into the rhs[3] by a sequence of iff steps, most of which involve expansion of definitions.

*Proof.*

$$\begin{array}{lll} x \in \overline{A \cap B} & \text{iff} & x \in (\mathcal{U} - (A \cap B)) \\ & \text{iff} & x \in \mathcal{U} \wedge x \notin (A \cap B) \\ & \text{iff} & x \in \mathcal{U} \wedge (x \notin A \vee x \notin B) \\ & \text{iff} & (x \in \mathcal{U} \wedge x \notin A) \vee (x \in \mathcal{U} \wedge x \notin B) \\ & \text{iff} & (x \in \mathcal{U} - A) \vee (x \in \mathcal{U} - B) \\ & \text{iff} & (x \in \overline{A}) \vee (x \in \overline{B}) \\ & \text{iff} & x \in (\overline{A} \cup \overline{B}) \end{array}$$

□

Such 'iff' chains can be quite a pleasant way to present a proof.

**Example 18** ($\varepsilon \in A$ iff $A^+ = A^*$)**.** Recall that $A^+ = A \cdot A^*$.

*Proof.* We'll proceed by cases on whether or not $\varepsilon \in A$.

[2] left hand side

[3] right hand side

$\underline{\varepsilon \in A}$. Then $A = \{\varepsilon\} \cup A$, so

$$\begin{array}{rcl}
A^+ & = & A^1 \cup A^2 \cup \ldots \\
 & = & (\{\varepsilon\} \cup A) \cup A^2 \cup \ldots \\
 & = & A^0 \cup A^1 \cup A^2 \cup \ldots \\
 & = & A^*
\end{array}$$

$\underline{\varepsilon \notin A}$. Then every string in $A$ has length greater than 0, so every string in $A^+$ has length greater than 0. But $\varepsilon$, which has length 0, is in $A^*$, so $A^* \neq A^+$. [Merely noting that $A \neq \{\varepsilon\} \cup A$ and concluding that $A^* \neq A^+$ isn't sufficient, because you have to make the argument that $\varepsilon$ doesn't somehow get added in the $A^2 \cup A^3 \cup \ldots$.]

□

**Example 19** (**$(A \cup B)^{\mathcal{R}} = A^{\mathcal{R}} \cup B^{\mathcal{R}}$**)**.** The proof will be an iff chain.

*Proof.*

$$\begin{array}{lll}
x \in (A \cup B)^{\mathcal{R}} & \text{iff} & x \in \{y^{\mathcal{R}} \mid y \in A \cup B\} \\
 & \text{iff} & x \in \{y^{\mathcal{R}} \mid y \in A \vee y \in B\} \\
 & \text{iff} & x \in (\{y^{\mathcal{R}} \mid y \in A\} \cup \{y^{\mathcal{R}} \mid y \in B\}) \\
 & \text{iff} & x \in (A^{\mathcal{R}} \cup B^{\mathcal{R}})
\end{array}$$

□

**Example 20.** Let $A = \{w \in \{0,1\}^* \mid w \text{ has an unequal number of 0s and 1s}\}$. Prove that $A^* = \{0,1\}^*$.

*Proof.* We show that $A^* \subseteq \{0,1\}^*$ and $\{0,1\}^* \subseteq A^*$. The first assertion is easy to see, since any set of binary strings is a subset of $\{0,1\}^*$. For the second assertion, the theorem in Example 16 lets us reduce the problem to showing that $\{0,1\} \subseteq A$, which is true, since $0 \in A$ and $1 \in A$. □

**Example 21.** Prove that $L^* = L^* \cdot L^*$.

*Proof.* Assume $x \in L^*$. We need to show that $x \in L^* \cdot L^*$, *i.e.*, that there exists $u, v$ such that $x = u \cdot v$ and $u \in L^*$ and $v \in L^*$. By taking $u = x$ and $v = \varepsilon$ we satisfy the requirements and so $x \in L^* \cdot L^*$, as required.

Contrarily, assume $x \in L^* \cdot L^*$. Thus there exists $u, v$ such that $x = uv$ and $u \in L^*$ and $v \in L^*$. Now, if $u \in L^*$, then there exists $i$ such that $u \in L^i$; similarly, there exists $j$ such that $v \in L^j$. Hence $uv \in L^{i+j}$. So there exists an $n$ (namely $i + j$) such that $x \in L^n$. So $x \in L^*$. □

Now we will move on to an example that uses proof by contradiction.

**Example 22** (Euclid)**.** The following famous theorem has an elegant proof that illustrates some of our techniques, proof by contradiction in particular. The English statement of the theorem is

*The prime numbers are an infinite set.*

Re-phrasing this as *For every prime, there is a larger one*, we obtain, in mathematical notation:

$$\forall m.\ \mathsf{prime}(m) \Rightarrow \exists n.\ m < n \wedge \mathsf{prime}(n)$$

Before we start, we will need the notion of the *factorial* of a number. The factorial of $n$ will be written $n!$. Informally $n! = 1 * 2 * 3 * \cdots * (n-1) * n$. Formally, we can define factorial by recursion:

$$\begin{array}{rcl} 0! & = & 1 \\ (n+1)! & = & (n+1) * n! \end{array}$$

*Proof.* Towards a contradiction, assume the contrary, *i.e.*, that there are only finitely many primes. That means there's a largest one, call it $p$. Consider the number $k = p! + 1$. Now, $k > p$ so $k$ is not prime, by our assumption. Since $k$ is not equal to 1, it has a prime factor. Formally,

$$\exists q.\ \mathsf{divides}(q, k) \wedge \mathsf{prime}(q)$$

In a complete proof, we'd prove this fact as a separate lemma, but here we will take it as given. Now, $q \leq p$ since $q$ is prime. Then $q$ divides $p!$, since $p! = 1 * \ldots * q * \ldots * p$. Thus we have established that $q$ divides both $p!$ and $p! + 1$. However, the only number that can evenly divide $n$ and $n + 1$ is 1. But 1 is not prime. <u>Contradiction.</u> All the intermediate steps after making our assumption were immaculate, so our assumption must have been faulty. Therefore, there are infinitely many primes. □

This is a beautiful example of a rigorous proof of the sort that we'll be reading and—we hope—writing. Euclid's proof is undoubtedly slick, and probably passed by in a blur, but that's OK: proofs are not things that can be skimmed; instead they must be painstakingly followed.

**Note**. The careful reader will notice that this theorem does not in fact show that there is even one prime, let alone an infinity of them. We must display a prime to start the sequence (the number 2 will do).

### 2.5.4 Induction

The previous methods we've seen are generally applicable. Induction, on the other hand, is a specialized proof technique that only applies to structured data such as numbers and strings. Induction is used to prove universal properties.

**Example 23.** Consider the statement $\forall n.\ 0 < n!$. This statement is easy to check, by calculation, for any particular number:

$$\begin{array}{c} 0 < 0! \\ 0 < 1! \\ 0 < 2! \\ \ldots \end{array}$$

but not for all of them (that would require an infinite number of cases to be calculated, and proofs can't be infinitely long). This is where induction comes in: induction "bridges the gap with infinity". How? In 2 steps:

**Base Case** Prove the property holds for 0: $0 < 0!$, *i.e.*, $0 < 1$.

**Step Case** Assume the proposition for an *arbitrary* number, say $k$, and then show the proposition holds for $k + 1$: thus we assume the *induction hypothesis* (IH) $0 < k!$. Now we need to show $0 < (k + 1)!$. By the definition of factorial, we need to show

$$\begin{array}{ll} 0 < (k + 1) * k! & \textit{i.e.,} \\ 0 < k * k! + k! & \text{(by the definition of factorial)} \end{array}$$

By the IH, this is true. And the proof is complete.

□

In your work, we will require that the base cases and steps cases be clearly labelled as such, and we will also need you to identify the IH in the step case. Finally, you will also need to show when you use the IH in the proof of the step case.

**Example 24.** Iterated sums, via the $\Sigma$ operator, yield many problems which can be tackled by induction. Informally, $\Sigma_{i=0}^{n} = 0 + 1 + \ldots + (n - 1) + n$. Let's prove

$$\forall n.\ \Sigma_{i=0}^{n}(2i + 1) = (n + 1)^2$$

*Proof.* By induction on $n$.

**Base Case.** We substitute 0 for $n$ everywhere in the statement to be proved.

$$\begin{array}{lrcl} & \Sigma_{i=0}^{0}(2i+1) & = & (0+1)^2 \\ \text{iff} & 2*0+1 & = & 1^2 \\ \text{iff} & 1 & = & 1 \end{array}$$

**Step Case.** Assume the IH : $\Sigma_{i=0}^{n}(2i+1) = (n+1)^2$. Now we need to show the statement with $n+1$ in place of $n$. Thus we want to show $\Sigma_{i=0}^{n+1}(2i+1) = ((n+1)+1)^2$, and proceed as follows:

$$\begin{array}{rcl} \Sigma_{i=0}^{n+1}(2i+1) & = & ((n+1)+1)^2 \\ & = & (n+2)^2 \\ & = & n^2+4n+4 \\ \underbrace{\Sigma_{i=0}^{n}(2i+1)}_{\text{use of IH}}+2(n+1)+1 & = & n^2+4n+4 \\ (n+1)^2+2(n+1)+1 & = & n^2+4n+4 \\ (n+1)^2+2(n+1)+1 & = & n^2+4n+4 \\ n^2+4n+4 & = & n^2+4n+4 \end{array}$$

□

**Example 25** ($A^{m+n} = A^m A^n$)**.** The proof is by induction on $m$.

*Proof.* **Base case.** $m = 0$, so we need to show: $A^{0+n} = A^0 A^n$, *i.e.,* that $A^n = \{\varepsilon\} \cdot A^n$, which is true.

**Step case.** Assume IH : $A^{m+n} = A^m A^n$.
We show: $A^{(m+1)+n} = A^{m+1} A^n$ as follows:

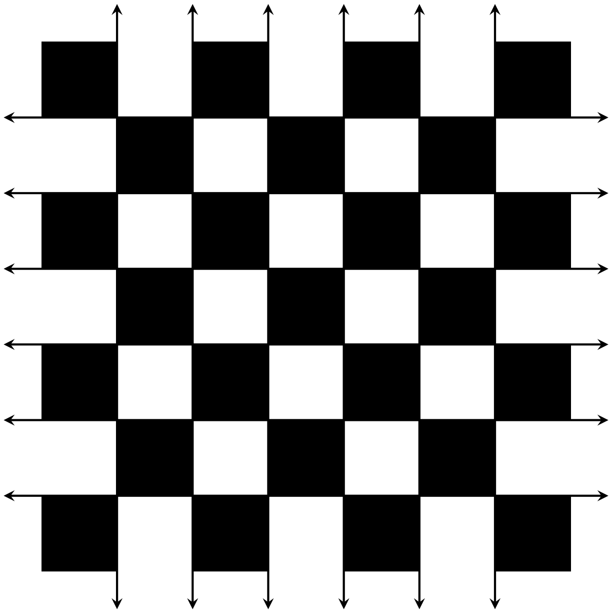
$$\begin{array}{llcl} & A^{(m+1)+n} & = & A^{m+1} \cdot A^n \\ \text{iff} & A^{1+(m+n)} & = & A \cdot \underbrace{A^m \cdot A^n}_{\text{IH}} \\ \text{iff} & A^{1+(m+n)} & = & A \cdot A^{m+n} \\ \text{iff} & A \cdot A^{m+n} & = & A \cdot A^{m+n} \end{array}$$

□

**Example 26.** Show that $L^{**} = L^*$.

*Proof.* The 'right-to-left' direction is easy since $A \subseteq A^*$, for all $A$. Thus it remains to show $L^{**} \subseteq L^*$. Assume $x \in L^{**}$. We wish to show $x \in L^*$. By the assumption there is an $n$ such that $x \in (L^*)^n$. We now induct on $n$.
Base case. $n = 0$, so $x \in (L^*)^0$, *i.e.*, $x \in \{\varepsilon\}$, *i.e.*, $x = \varepsilon$. This completes the base case, as $\varepsilon$ is certainly in $L^*$.
Step case. Let $IH = \forall x.\ x \in (L^*)^n \Rightarrow x \in L^*$. We want to show $x \in (L^*)^{n+1} \Rightarrow x \in L^*$. Thus, assume $x \in (L^*)^{n+1}$, *i.e.*, $x \in L^* \cdot (L^*)^n$. This implies that there exists $u, v$ such that $u \in L^*$ and $v \in (L^*)^n$. By the IH, we have $v \in L^*$. But then we have $x \in L^*$ because $A^* \cdot A^* = A^*$, for all $A$, as was shown in Example 21. □

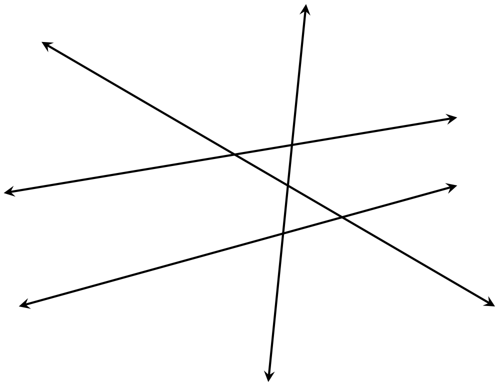
Here's a fun one. Suppose we have $n$ straight lines (infinite in both directions) drawn in the plane. Then it is always possible to assign 2 colors (say black and white) to the resulting regions so that adjacent regions have different colors. For example, if our lines have a grid shape, this is clearly possible:



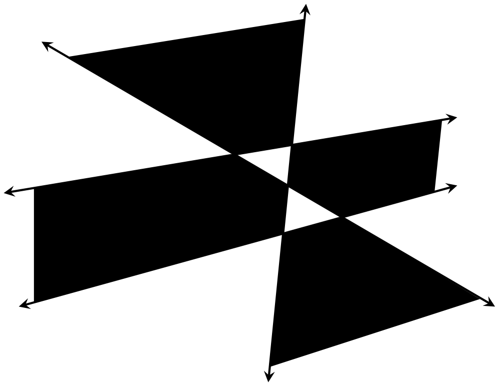
Is it also possible in general? Yes.

**Example 27** (Two-coloring regions in the plane)**.** Given $n$ lines drawn in the plane, it is possible to color the resulting regions black or white such that adjacent regions (those bordering the same line) have different colors.

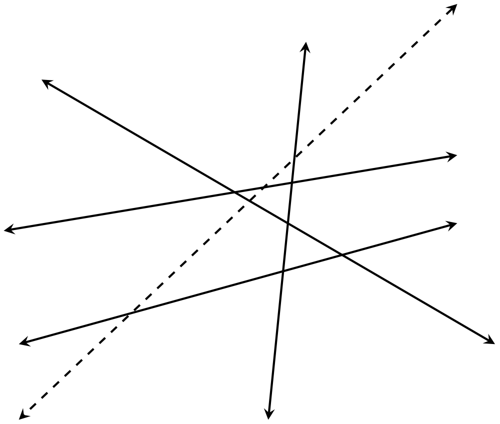
Let's look at a less regular example:



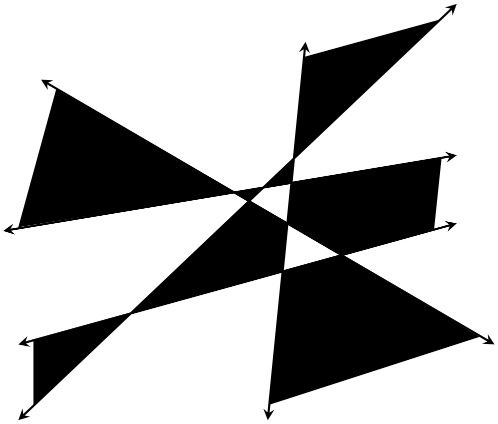
This can be 2-colored as follows:



There is a systematic way to achieve this, illustrated by what happens if we add a new line into the picture. Suppose we add a new (dashed) line to our example:



Now pick one side of the line (the left, say), and 'flip' the colors of the regions on that side. Leave the coloring on the right side alone. This gives us



which is again 2-colored. Now let's see how to prove that this works in general.

*Proof.* By induction on the number of lines on the plane.

**Base case.** If there are no lines on the plane, then pick a color and color the plane with it. Since there are no adjacent regions, the property holds.

**Step case.** Suppose the plane has $n$ lines on it. The IH says that adjacent regions have different colors. Now we add a line $\ell$ to the plane, and recolor regions on the left of $\ell$ as stipulated above. Now consider any two adjacent regions. There are three possible cases:

1. Both regions are on the non-flipped part of the plane. In that case, the IH says that the two regions have different colors.
2. Both regions are in the part of the plane that had its colors flipped. In that case, the IH says that the two regions had different colors. Flipping them results in the two regions again having different colors.
3. The two regions are separated by the newly drawn line, *i.e.*, $\ell$ divided a region into two. Now, the new sub-region on the right of $\ell$ stays the same color, while the new sub-region on the left is flipped. So the property is preserved.

□

**Example 28** (Incorrect use of induction)**.** Let's say that a set is *monochrome* if all elements in it are the same color. The following argument is flawed. Why?

1. Buy a pig; paint it yellow.

2. We prove that all finite sets of pigs are monochrome by induction on the size of the set:

   **Base case.** The only set of size zero is the empty set, and clearly the empty set of pigs is monochrome.

   **Step case.** The inductive hypothesis is that any set with $n$ pigs is monochrome. Now we show that any set $\{p_1, \ldots, p_{n+1}\}$ consisting of $n + 1$ pigs is also monochrome. By the IH, we know that $\{p_1, \ldots, p_n\}$ is monochrome. Similarly, we know that $\{p_2, \ldots, p_{n+1}\}$ is also monochrome. So $p_{n+1}$ is the same color as the pigs in $\{p_1, \ldots, p_n\}$. Therefore $\{p_1, \ldots, p_{n+1}\}$ is monochrome.

3. Since all finite sets of pigs are monochrome, the set of all pigs is monochrome. Since we just painted our pig yellow, it follows that all pigs are painted yellow.

[Flaw: We make two uses of the IH in the proof, and implicitly take two pigs out of $\{p_1, \ldots, p_{n+1}\}$. That means that $\{p_1, \ldots, p_{n+1}\}$ has to be of size at least two. Suppose it is of size 2, *i.e.*, consider some two-element set of pigs $\{p_1, p_2\}$. Now $\{p_1\}$ is monochrome and so is $\{p_2\}$. But the argument in the proof doesn't force every pig in $\{p_1, p_2\}$ to be the same color.]

### Strong Induction

Occasionally, one needs to use a special kind of induction called *strong*, or *complete*, induction to make a proof work. The difference between this kind of induction and ordinary induction is the following: in ordinary induction, the induction step is just that we assume the property $P$ holds for $n$ and use that as a tool to show that $P$ holds for $n + 1$; in strong induction, the induction hypothesis is that $P$ holds for *all* $m$ strictly smaller than $n$ and the goal is to show that $P$ holds for $n$.

Specified formally, we have

**Mathematical induction**

$$\forall P.\ P(0) \land (\forall m.\ P(m) \Rightarrow P(m+1)) \Rightarrow \forall n.\ P(n).$$

**Strong induction**

$$\forall P.\ (\forall n.\ (\forall m.\ m < n \Rightarrow P(m)) \Rightarrow P(n)) \Rightarrow \forall k.\ P(k).$$

Some remarks:

- Sometimes students are puzzled because strong induction doesn't have a base case. That is true, but often proofs using strong induction require a case split on whether a number is zero or not, and this is effectively considering a base case.
- Since strong induction allows the IH to be assumed for any smaller number, it may seem that more things can be proved with strong induction, *i.e.*, that strong induction is more powerful than mathematical induction. This is not so: mathematical induction and strong induction are inter-derivable (you can prove each from the other), and consequently, any proof using strong induction can be changed to use mathematical induction. However, this fact is mainly of theoretical interest: you should feel free to use whatever kind of induction works.

The following theorem about languages is interesting and useful; moreover, its proof uses both strong induction and ordinary induction. Warning: the statement and proof of this theorem are significantly more difficult than the material we have encountered so far!

**Example 29** (Arden's Lemma)**.** Assume that $A$ and $B$ are two languages with $\varepsilon \notin A$. Also assume that $X$ is a language having the property $X = (A \cdot X) \cup B$. Then $X = A^* \cdot B$.

*Proof.* Showing $X \subseteq A^* \cdot B$ and $A^* \cdot B \subseteq X$ will prove the theorem.

1. Suppose $w \in X$; we want to show $w \in A^* \cdot B$. We proceed by complete induction on the length of $w$. Thus the IH is that $y \in A^* \cdot B$, for any $y$ strictly shorter than $w$. We now consider cases on $w$:

- $w = \varepsilon$, *i.e.*, $\varepsilon \in X$, or $\varepsilon \in (A \cdot X) \cup B$. But note that $\varepsilon \notin (A \cdot X)$, by the assumption $\varepsilon \notin A$. Thus we have $\varepsilon \in B$, and so $\varepsilon \in A^* \cdot B$, as desired.
- $w \neq \varepsilon$. Since $w \in (A \cdot X) \cup B$, we consider the following cases:
  (a) $w \in (A \cdot X)$. Since $\varepsilon \notin A$, there exist $u, v$ such that $w = uv$, $u \in A$, $v \in X$, and $\mathsf{len}(v) < \mathsf{len}(w)$. By the IH, we have $v \in A^* \cdot B$ hence, by the semantics of Kleene star, we have $uv \in A^* \cdot B$, as required.
  (b) $w \in B$. Then $w \in A^* \cdot B$, since $\varepsilon \in A^*$.

2. We wish to prove $A^* \cdot B \subseteq X$, which can be reduced to the task of showing $\forall n.\ A^n \cdot B \subseteq X$. The proof proceeds by ordinary induction:
   (a) Base case. $A^0 \cdot B = B \subseteq (A \cdot X) \cup B$.
   (b) Step case. Assume the IH: $A^n \cdot B \subseteq X$. From this we obtain $A \cdot A^n \cdot B \subseteq A \cdot X$, *i.e.*, $A^{n+1} \cdot B \subseteq A \cdot X$. Hence $A^{n+1} \cdot B \subseteq X$.

□

# Chapter 3

# Models of Computation

Now we start the course for real. The questions we address in this part of the course have to deal with models for sequential computation[1] in a setting where there are no resource limits (time and space). Here are a few of the questions that arise:

- Is 'C' as powerful as Java? More powerful? What about Lisp, or ML, or Perl?
- What does it mean to be more powerful anyway?
- What about assembly language (say for the x86). How does it compare? Are low-level languages more powerful than high-level languages?
- Can programming languages be compared by expressiveness, somehow?
- What is an algorithm?
- What can computers do *in principle*, *i.e.*, without restriction on time, space, and money?
- What *can't* computers do? For example, are there some optimizations that a compiler can't make? Are there purported programming tasks that can't be implemented, no matter how clever the programmer(s)?

[1] We won't consider models for concurrency, for example.

This is undoubtedly a collection of serious questions, and we should say how we go about investigating them. First, we are not going to use any particular real-world programming language: they tend to be too big.[2] Instead, we will deal with relatively simple machines. Our approach will be to convince ourselves that the machines are powerful enough to compute whatever general-purpose computers can, and then to go on to consider the other questions.

## 3.1 Turing Machines

Turing created his machine[3] to answer a specific question of interest in the 1930's: what is an algorithm? His analysis of computation—well before computers were invented—was based on considering the essence of what a human doing a mechanical calculation would need, provided the job needed *no* creativity. He reasoned as follows:

- No creativity implies that each step in the calculation must be fully spelled out.
- The list of instructions followed must be finite, *i.e.*, programs must be finite objects.
- Each individual step in the calculation must take a finite amount of time to complete.
- Intermediate results may need to be calculated, so a scratch-pad area is needed.
- There has to be a way to keep track of the current step of the calculation.
- There has to be a way to view the complete current state of the calculation.

Turing's idea was *machine*-based. A Turing machine (TM) is a machine with a finite number of control states and an infinite tape, bounded at the

[2] However, some models-of-computation courses have used Scheme.

[3] For Turing's original 1936 paper, see "On Computable Numbers, with an Application to the Entscheidungsproblem", on the class webpage.

left and stretching off to the right. The tape is divided into cells, each of which can hold one symbol. The input of the machine is a string $w = a_1 \cdot a_2 \cdot \ldots \cdot a_n$ initially written on the leftmost portion of the tape, followed by an infinite sequence of blanks (␣):

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $a_1$ | $a_2$ | $\cdots$ | $a_{n-1}$ | $a_n$ | ␣ | ␣ | $\cdots$ |

The machine is able to move a read/write head left and right over the tape as it performs its computation. It can read and write symbols on the tape as it pleases. These considerations led Turing to the following formal definition.

**Definition 1** (Turing Machine)**.** A Turing machine is a 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$$

where

- $Q$ is a finite set of states.
- $\Sigma$ is the input alphabet, which never includes blanks.
- $\Gamma$ is the tape alphabet, which always includes blanks. Moreover, every input symbol is in $\Gamma$.
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function, where $L$ and $R$ are directions, telling the machine head which direction to go in a step.
- $q_0 \in Q$ is the start state
- $q_A \in Q$ is the accept state
- $q_R \in Q$ is the reject state. $q_A \neq q_R$.

The program that a Turing machine executes is embodied by the transition function $\delta$. Conceptually, the following happens when a transition

$$\delta(q_i, a) = (q_j, b, d)$$

is made by a Turing machine $M$:

- $M$ writes $b$ to the current tape cell, overwriting $a$.

- The current state changes from $q_i$ to $q_j$.
- The tape head moves to the left or right by one cell, depending on whether $d$ is $L$ or $R$.

**Example 30.** We'll build a TM that merely moves all the way to the end of its input and stops. The states of the machine will just be $\{q_0, q_A, q_R\}$. (We have to include $q_R$ as a state, even though it will never be entered.) The input alphabet $\Sigma = \{0, 1\}$, for simplicity. The tape alphabet $\Gamma = \Sigma \cup \{\textvisiblespace\}$ includes blanks, but is otherwise the same as the input alphabet. All that is left to specify is the transition function. The machine simply moves right along the tape until it hits a blank, then halts. Thus, at each step, it just writes back the current symbol, remains in $q_0$, and moves right one cell:

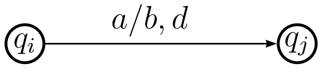
$$\begin{array}{c} \delta(q_0, 0) = (q_0, 0, R) \\ \delta(q_0, 1) = (q_0, 1, R) \end{array}$$

Once the machine hits a blank, it moves one cell to the left and stops:

$$\delta(q_0, \textvisiblespace) = (q_A, \textvisiblespace, L)$$

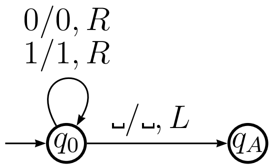
Notice that if the input string is $\varepsilon$, the first step the machine makes is moving left from the leftmost cell: it can't do that, so the tape head just stays in the leftmost cell.

Turing machines can also be represented by *transition diagrams*. A transition $\delta(q_i, a) = (q_j, b, d)$ between state $q_i$ and $q_j$ can be drawn as



and means that if the machine is in state $q_i$ and the current cell has an $a$ symbol, then the current cell is updated to have a $b$ symbol, the tape head moves one cell to the left or right (according to whether $d = L$ or $d = R$), and the current state becomes $q_j$.

For the current example, the state diagram is quite simple:



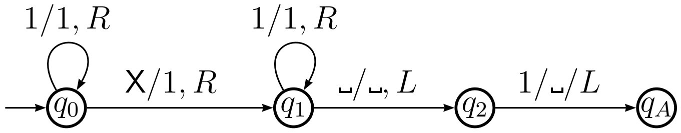
**Example 31** (Unary addition)**.** (Worked out in class.) Although Turing machines manipulate *symbols* and not numbers, they are quite often used to compute numerical functions such as addition, subtraction, multiplication, *etc*. To take a very simple example, suppose we want to add two numbers given in unary, *i.e.*, as strings over $\Sigma = \{1\}$. In this representation, for example, 3 is represented by 111 and 0 is represented by $\varepsilon$. The two strings to be added will be separated by a marker symbol **X**. Thus, if we wanted to add 3 and 2, the input would be

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | X | 1 | 1 | ␣ | ··· |

and the output should be

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | ␣ | ··· |

Here is the desired machine. It traverses the first number, then replaces the **X** with 1, then copies the second number, then erases the last 1 before accepting.

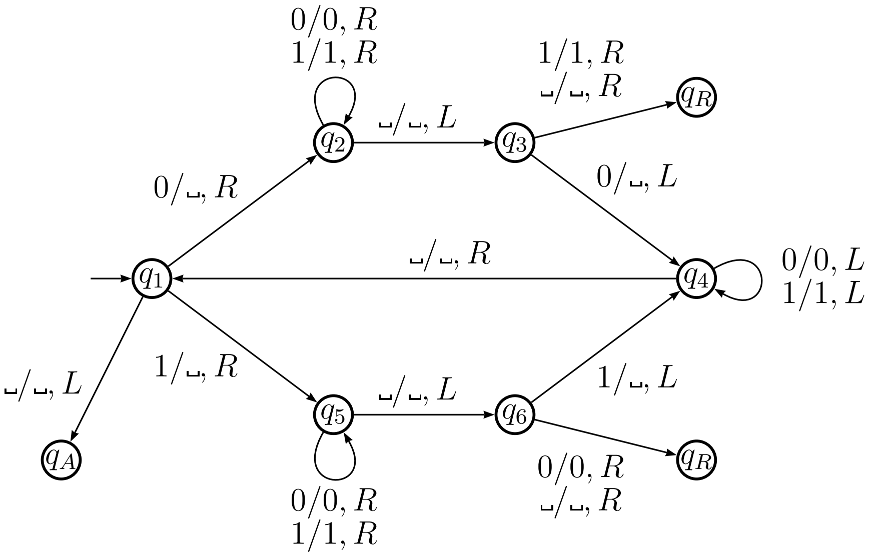


**Note**. Normally, Turing machines are used to accept or reject strings. In this example, the machine computes the unary addition of the two inputs and then always halts in the accept state. This convention is typically used when Turing machines are used to compute functions rather than just saying "yes" or "no" to input strings.

**Example 32.** We now give a transition diagram for a Turing machine $M$ that recognizes

$$\{w \cdot w^{\mathcal{R}} \mid w \in \{0,1\}^*\}.$$

Example strings in this language are $\varepsilon, 00, 11, 1001, 0110, 10100101, \ldots$.



The general idea is to go from the 'outside-in' on the input string, cancelling off equal symbols at each end. The loop $q_1 \to q_2 \to q_3 \to q_4 \to q_1$ replaces the leading symbol (a 0) with a blank, then moves to the rightmost uncancelled symbol, checks that it is a 0, overwrites it with a blank, then moves to the leftmost uncancelled symbol. If there isn't one, then the machine accepts. The lower loop $q_1 \to q_5 \to q_6 \to q_4 \to q_1$ is essentially the same as the upper loop, except that it cancels off a matching pair of 1s from each end. If the sought-for 0 (or 1) is not found at the rightmost uncancelled symbol, then the machine rejects (from $q_3$ or $q_6$).

Now back to some more definitions. A *configuration* is a snapshot of the complete state of the machine.

**Definition 2** (Configuration)**.** A Turing machine configuration is a triple $\langle \ell, q, r \rangle$, where $\ell$ is a string denoting the tape contents to the left of the tape head and $r$ is a string representing the tape to the right of the tape head. Since the tape is infinite, there is a point past which the tape is nothing but blanks. By convention, these are not included in $r$.[4] The leftmost symbol of $r$ is the current tape cell. The state $q$ is the current state of the machine.

A Turing machine starts in the configuration $\langle \varepsilon, q_0, w \rangle$ and repeatedly makes transitions until it ends up in $q_A$ or $q_R$. Note that a machine may

[4] However, this is not completely correct, since the machine may, for example, be given the empty string as input, in which case $r$ must have at least one blank.

never end up in $q_A$ or $q_R$, in which case it is said to be *looping* or *diverging*. After all, we would certainly want to model programs that never stop: in many cases such programs are useless, but they are undeniably part of what we understand by computation.

**Transitions and configurations** How do the transitions of a machine affect the configuration? There are two cases.

1. If we are moving right, there is always room to keep going right. On transition $\delta(q_i, a) = (q_j, b, R)$ the configuration change is

$$\langle u, q_i, a \cdot w \rangle \longrightarrow \langle u \cdot b, q_j, w \rangle.$$

   If the rightward portion $w$ of the tape is $\varepsilon$, blanks are added as needed.

2. If we are moving left by $\delta(q_i, a) = (q_j, b, L)$ then the configuration changes as follows:

   - When there is room to move left: $\langle u \cdot c, q_i, a \cdot w \rangle \longrightarrow \langle u, q_j, c \cdot b \cdot w \rangle$.
   - Moving left but up against left end of tape: $\langle \varepsilon, q_i, a \cdot w \rangle \longrightarrow \langle \varepsilon, q_j, b \cdot w \rangle$.

A sequence of linked transitions starting from the initial configuration is said to be an *execution*.

**Definition 3** (Execution)**.** An *execution* of Turing machine $M$ on input $w$ is a possibly infinite sequence

$$\langle \varepsilon, q_0, w \rangle \longrightarrow \ldots \longrightarrow \langle u, q_i, w \rangle \longrightarrow \langle u', q_j, w' \rangle \longrightarrow \ldots$$

of configurations, starting with the configuration $\langle \varepsilon, q_0, w \rangle$, where the configuration at step $i + 1$ is derived by making a transition from the configuration at $i$.

A *terminating* execution is one which ends in an accepting configuration $\langle u, q_A, w \rangle$ or a rejecting configuration $\langle u, q_R, w \rangle$, for some $u, w$.

*Remark.* The following distinctions are important:

- *$M$ accepts $w$* iff the execution of $M$ on $w$ is terminating and ends in the accept state:

$$\langle \varepsilon, q_o, w \rangle \xrightarrow{*} \langle \ell, q_A, r \rangle$$

- *$M$ rejects $w$* iff the execution of $M$ on $w$ is terminating and ends in the reject state:

$$\langle \varepsilon, q_o, w\rangle \xrightarrow{*} \langle \ell, q_R, r\rangle$$

- *$M$ does not accept $w$* iff $M$ rejects $w$ or $M$ loops on $w$

Now we make a definition that relates Turing machines to sets of strings.

**Definition 4** (Language of a Turing machine)**.** The language of a Turing machine $M$, $\mathcal{L}(M)$, is the set of strings accepted by $M$.

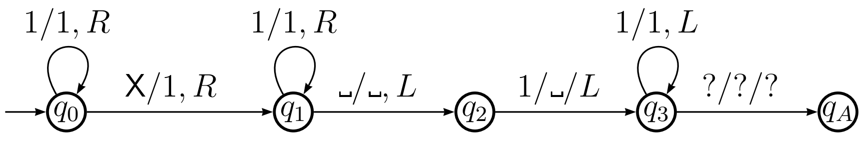
$$\mathcal{L}(M) = \{x \mid M \text{ halts and accepts } x\}.$$

**Definition 5** (Computation of function by Turing machine)**.** Turing machines are defined so that they can only accept or reject input (or loop). But most programs compute functions, *i.e.*, deliver *answers* beyond just 'yes' or 'no'. To achieve this is simple. A TM $M$ is said to *compute* a function $f$ if, when given input $w$ in the domain of $f$, the machine halts in its accept state with $f(w)$ written (leftmost) on the tape.

### 3.1.1 Example Turing Machines

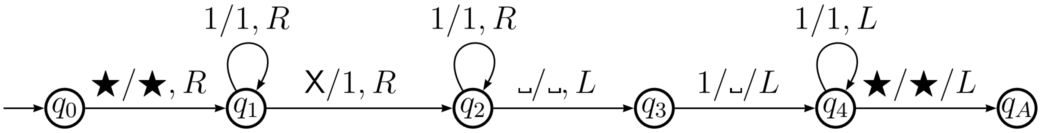
Now we look at a few more examples.

**Example 33** (Left edge detection)**.** Let's revisit Example 31, which implemented addition of numbers in unary representation. There, we made a pass over the input strings, then scrubbed off a trailing 1, then halted, leaving the tape head at the end of the unary sum. Instead, we want a machine that moves the head all the way back to the leftmost cell after performing the addition. And this reveals a problem. Here is a diagram of an incorrect machine, formed by adapting that of Example 31.

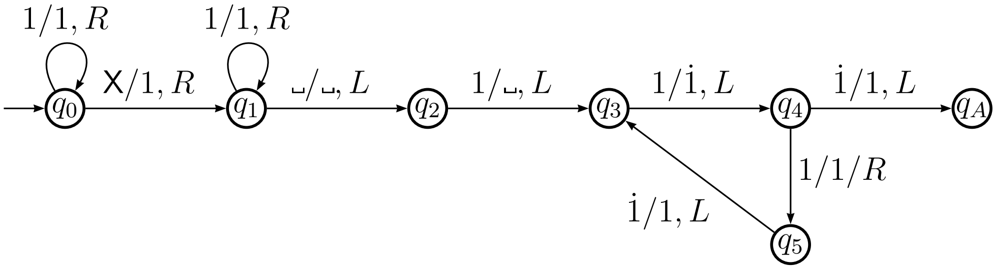


Once the machine enters state $q_3$, it has performed the addition and now uses a loop to move the tape head leftmost. But when the machine is moving left in a loop, such as in state $q_3$, there is a difficulty: the machine should leave the loop once it bumps into the left edge of the tape. But once the tape head reaches the leftmost cell, the machine will repeatedly try to move left on a '1', unaware that it is overwriting the same '1' eternally. There are two ways to deal with this problem:

- Before starting the computation, attach a special marker (*e.g.*, ★) to the front of the input string. Thus, if the input to the machine is $w$, change it so that the machine starts up with ★$w$ as the initial contents of the tape. Provided that ★ is never written anywhere else on the tape, the machine can easily detect ★ and break out of its 'move-left' behaviour. One can either require that ★ be attached at the beginning of the input from the beginning, or take the input, shift it all right by one cell and put ★ in the first cell. Assuming the former, the following machine results.



- When making a looping scan to the leftmost cell, add some special-purpose code to detect the left edge. We know that when the machine bumps into the left edge, it writes the new character on top of the old and then can't move the tape head. The idea is to write a 'marked' version of the symbol on the tape and attempt to move left. In the next step, if the marked symbol is seen, then the machine must be at the left edge and the loop can be exited. If the marked symbol is not seen, then the machine *has* been able to move left, and we go back and 'erase' the mark from the symbol before continuing. For the current example this yields the following machine, where the leftward loop in state $q_3$ has been replaced by the loop $q_3 \to q_4 \to q_5 \to q_3$.



Here is an execution of the second machine on the input $111X11$:

$$\begin{array}{l}
\langle \varepsilon, q_0, 111X11 \rangle \\
\langle 1, q_0, 11X11 \rangle \\
\langle 11, q_0, 1X11 \rangle \\
\langle 111, q_0, X11 \rangle \\
\langle 1111, q_1, 11 \rangle \\
\langle 11111, q_1, 1 \rangle \\
\langle 111111, q_1, \sqcup \rangle \\
\langle 11111, q_2, 1\sqcup \rangle \\
\langle 1111, q_3, 1\sqcup\sqcup \rangle \\
\langle 111, q_4, 1\dot{1}\sqcup\sqcup \rangle \\
\langle 1111, q_5, \dot{1}\sqcup\sqcup \rangle \\
\langle 111, q_3, 11\sqcup\sqcup \rangle \\
\langle 11, q_4, 1\dot{1}1\sqcup\sqcup \rangle \\
\langle 111, q_5, \dot{1}1\sqcup\sqcup \rangle \\
\langle 11, q_3, 111\sqcup\sqcup \rangle \\
\langle 1, q_4, 1\dot{1}11\sqcup\sqcup \rangle \\
\langle 11, q_5, \dot{1}11\sqcup\sqcup \rangle \\
\langle 1, q_3, 1111\sqcup\sqcup \rangle \\
\langle \varepsilon, q_4, 1\dot{1}111\sqcup\sqcup \rangle \\
\langle 1, q_5, \dot{1}111\sqcup\sqcup \rangle \\
\langle \varepsilon, q_3, 11111\sqcup\sqcup \rangle \\
\langle \varepsilon, q_4, \dot{1}1111\sqcup\sqcup \rangle \\
\langle \varepsilon, q_A, 11111\sqcup\sqcup \rangle
\end{array}$$

In the next example, we will deal with strings of balanced parentheses. We give the name *BAL* to the set of all such strings. The following strings

$$\varepsilon, (), ((())), (())()()$$

are members of *BAL*.

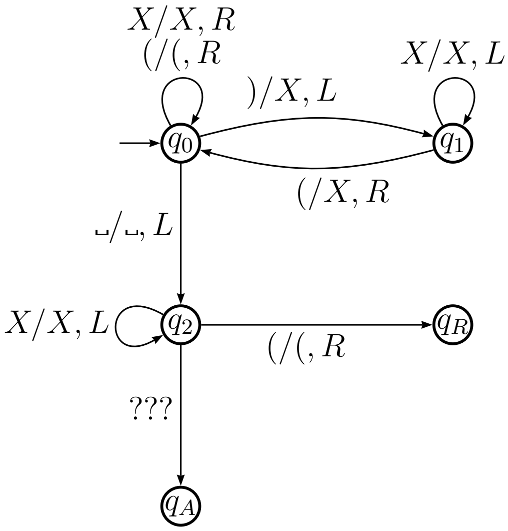
**Example 34** (BAL—first try)**.** Give a transition diagram for a Turing machine that accepts only the strings in $BAL$. First, of course, we need to have a high-level algorithm in mind: the following is a reasonable start:

1. Search right for a ')'.
2. If one is not found, scan left for a '('. If found, reject, else accept.

3. Otherwise, overwrite the ')' with an '$X$' and scan left for a '('.

4. If one is found, overwrite it with '$X$' and go to 1. Otherwise, reject.

The following diagram captures this algorithm. It is not quite right, because of left-edge detection problems, but it is close.



In state $q_0$, we scan right, skipping open parens and X's, looking for a closing parenthesis, and transition to state $q_1$ when one is found. If one is not found, we must hit blanks, in which case we transition to state $q_2$.

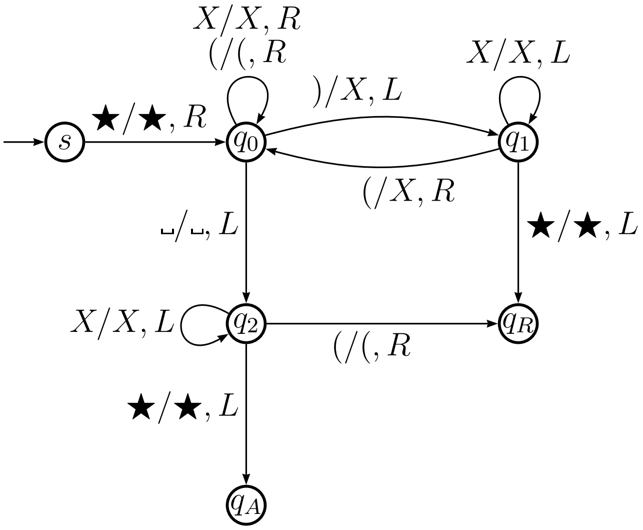
$q_1$ If we find ourselves in $q_1$, we've found the first ')' and replaced it with an 'X'. Now we have to scan left and find the matching open parenthesis, skipping over any 'X' symbols. (Caution: left edge detection needed!) Once the first open paren to the left is found, we over-write it with an 'X' and go to state $q_0$. Thus we have successfully cancelled off one pair of matching parens, and can go to the beginning of the loop, *i.e.*, $q_0$, to look for another pair.

$q_2$ If we find ourselves in $q_2$, we have unsuccessfully searched to the right looking for a closing paren. That means that every closing paren has been paired up with an open paren. However, we must still deal with the possibility that there are more open parens than closing parens in the input, in which case we should reject. So we search back left looking for a remaining open paren. If none exist, we accept; otherwise we reject.

Thus, in state $q_2$ we scan to the left, skipping over 'X' symbols. If we encounter an open paren, we transition to state $q_R$ and reject. If we don't, then we ought to accept.

Now we will re-do the $BAL$ example properly, using both ways of detecting the left edge.

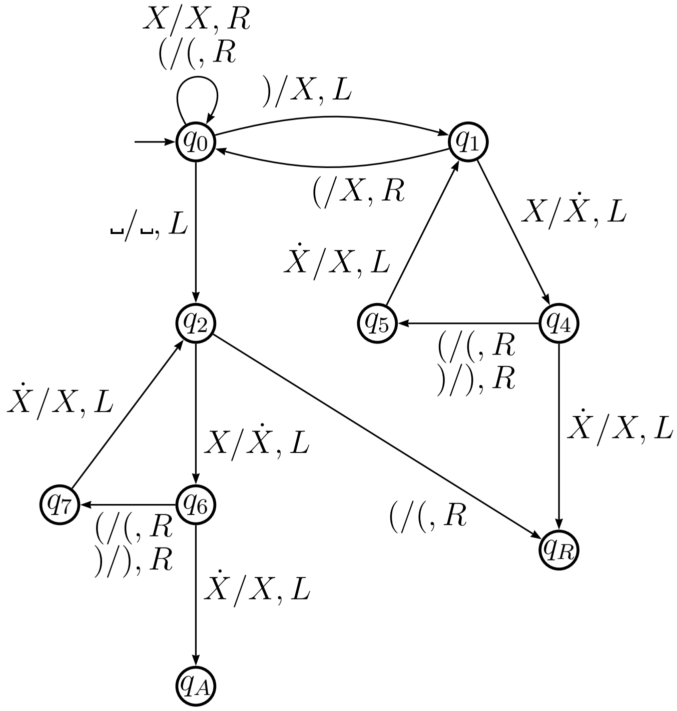
**Example 35** (BAL done right (1))**.** We expect a ★ in the first cell, followed by the real input.



The language recognized by this machine is $\{\bigstar\} \cdot BAL$.

**Example 36** (BAL done right (2))**.** Each loop implementing a leftward scan is augmented with extra states. The naive (incorrect) loop implementing the left scan at $q_1$ is replaced by a loop $q_1 \to q_4 \to q_5 \to q_1$, which is exited either by encountering an open paren (transition to $q_0$) or by bumping against the left edge (no corresponding open paren to a close paren, so transition to reject state $q_R$).

Similarly, the incorrect loop implementing the left scan at $q_2$ is replaced by a loop $q_2 \to q_6 \to q_7 \to q_2$, which is exited either by encountering an open paren (open paren with no corresponding close paren, so transition to $q_R$) or by bumping against the left edge (no unclosed open parens, so transition to accept state $q_A$).



**Example 37.** Let's try to build a TM that accepts the language

$$\{w \cdot w \mid w \in \{a, b\}^*\}$$

We first need a high-level outline of our algorithm. The following steps are needed:

1. Locate the middle of the string. With an ordinary programming language, one would use some kind of arithmetic calculation to find the middle index. However, TMs don't have arithmetic operations built in, and it can be somewhat arduous to provide them.

   Instead, we adopt the following approach: mark the leftmost symbol, then go to the end of the string and mark the last symbol. Then go all the way to the left and mark the leftmost unmarked symbol. Then go all the way to the right and mark the rightmost unmarked symbol. Repeat until there are no unmarked symbols. Because we have worked 'outside-in' this phase of processing should end up with the tape head on the first symbol of the second half of the string.

If the string is not of even length then, at some step, the leftmost symbol will get marked, but there will be no corresponding rightmost unmarked symbol.

2. Now we check that the two halves are equal. Starting from the first character of the right half of the string, call it $\dot{c}$, we remove the mark and move left until the leftmost marked symbol is detected. We will have to detect the left edge in this step! If the leftmost marked symbol is indeed $\dot{c}$, then we unmark it (otherwise we reject). Then we scan right over (a) remaining marked symbols in the left half of the string and then (b) unmarked symbols in the first part of the right half of the string. We then either find a marked symbol, or we hit the blanks.

3. Repeat for the second, third, *etc* characters. Finally, the rightward scan for a marked symbol on the rhs doesn't find anything and ends in the blanks. And then we can accept.

Now that we have a good idea of how the algorithm should work, we will go ahead and design the TM in detail. (But note that often this higher level of description suffices to convince people that a proposed algorithm is implementable on a TM, and actually providing the full TM description is not necessary.)

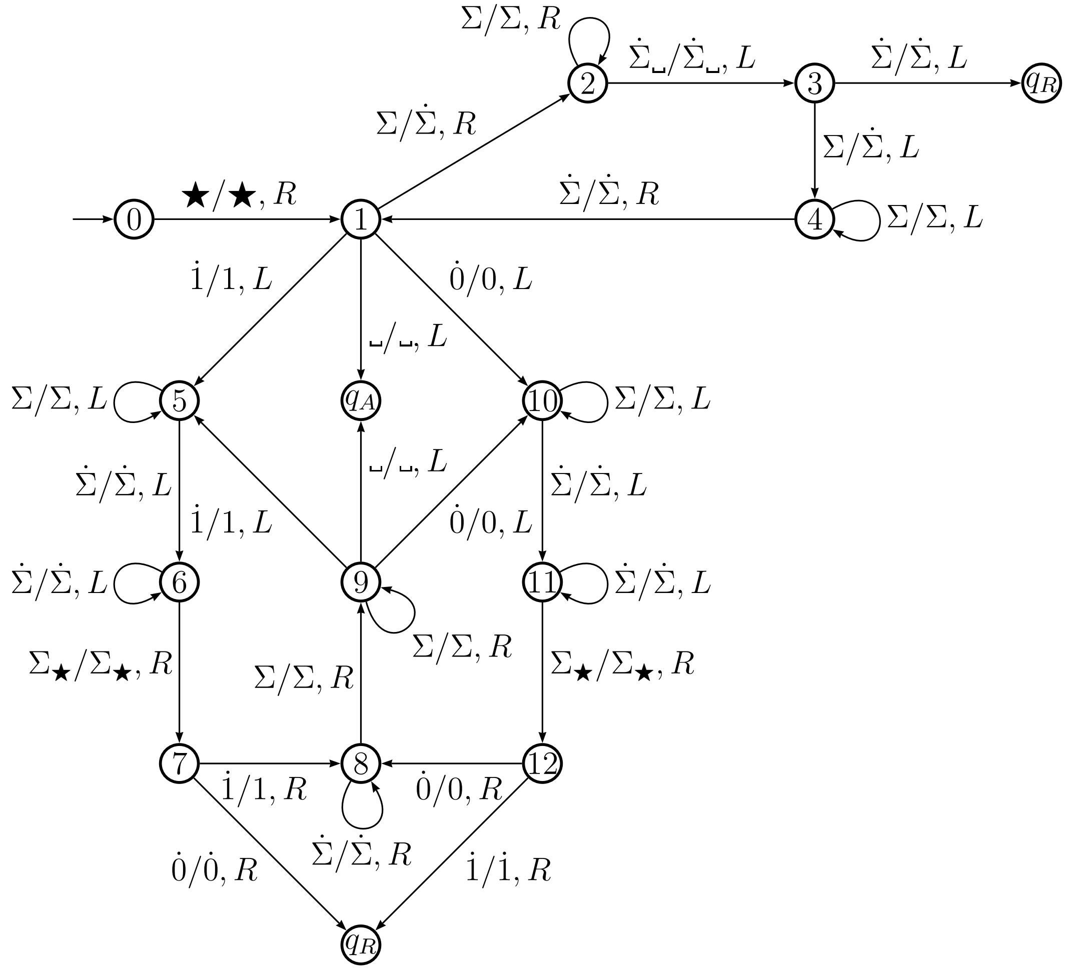
In the transition diagram, we use several shorthand notations:

- $\Sigma/\Sigma, L$ says that the transition replaces any symbol (so either 0 or 1) by itself and moves left. Thus $\Sigma$ is being used to represent any particular symbol in $\Sigma$, saving us from writing out two transitions.

- $\dot{\Sigma}$ represents any marked symbol in the alphabet.

- $\Sigma_{\bigstar} = \Sigma \cup \{\bigstar\}$.

- $\Sigma_{\sqcup} = \Sigma \cup \{\sqcup\}$.



We assume that the input is prefixed with ★, thus the transition from state 1 to 2 just hops over the ★. If the input is not prefixed with ★, there is a transition to $q_R$ (not included in the diagram). Having got to state 2, the first pass of processing proceeds in the loop of states $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$. In state 2 the leftmost unmarked character is marked and then there is a sweep over unmarked characters until either a marked character or the blanks are encountered (state 2). Then the tape head is moved one cell to the left. (Note that $\Sigma = \{0, 1\}$ in this example.) In state 3, we should be at the rightmost unmarked symbol on the tape. If it is however, a marked symbol, that means that the leftmost unmarked symbol has no corresponding rightmost unmarked symbol, so we reject. Otherwise, we loop left over the unmarked symbols until we hit a marked symbol, then move right.

We are then either at an unmarked symbol, in which case we go through the $1 \to 2 \to 3 \to 4 \to 1$ loop again, or else we are at a marked symbol. In fact, this will be the first symbol in the second half of the string, and we move to the second phase of processing. This phase features two nearly identical loops. If the marked symbol is a $\dot{1}$, then the left loop $5 \to 6 \to 7 \to 8 \to 9$ is taken; otherwise, if the marked symbol is $\dot{0}$,the right loop $10 \to 11 \to 12 \to 8 \to 9$ is taken.

We now describe the left loop. In state 1 the leftmost marked cell in the second half of the string is a $\dot{1}$; now we traverse over the prefix of unmarked cells in the second half (state 5); then we traverse over the suffix of marked cells in the first half of the string (state 6). Thus we arrive either at ★ or at the rightmost unmarked cell in the first half of the string, and move right into state 7. This leaves us looking at a marked cell. We expect the matching $\dot{1}$ to the one seen in state 1, which takes us to state 8 (after unmarking it); if we see a $\dot{0}$, then we reject. So if we are in state 8, we have located the matching symbol in the first half of the string, and unmarked it. Now we move right to the next element to consider in the second half of the string. This involves skipping over the remaining marked symbols in the first half of the string (state 9), then the prefix of unmarked symbols in the second half of the string (state 10).

Then we are either looking at a marked symbol, in which case we go around the loop again (either to state 5 if it is a $\dot{1}$ or to state 10 if it is a $\dot{0}$). Or else we are looking at the blanks, which means that there are no more symbols to unmark, and we can accept.

We now trace the execution of $M$ on the string $010010$, by giving a sequence of machine configurations. In several steps (17, 20, 24, 31, and 34) we use ellipsis to abbreviate a sequence of steps. Hopefully, these will be easy to fill in!

| Step | Config | Step | Config |
|---|---|---|---|
| 1 | $(\varepsilon, q_0, \bigstar 010010)$ | 26 | $(\bigstar\dot{0}\dot{1}, q_{10}, \dot{0}0\dot{1}\dot{0})$ |
| 2 | $(\varepsilon, q_1, 010010)$ | 27 | $(\bigstar\dot{0}, q_{11}, \dot{1}\dot{0}0\dot{1}\dot{0})$ |
| 3 | $(\bigstar, q_1, 010010)$ | 28 | $(\bigstar, q_{11}, \dot{0}\dot{1}\dot{0}0\dot{1}\dot{0})$ |
| 4 | $(\bigstar\dot{0}, q_2, 10010)$ | 29 | $(\varepsilon, q_{11}, \bigstar\dot{0}\dot{1}\dot{0}0\dot{1}\dot{0})$ |
| 5 | $(\bigstar\dot{0}1, q_2, 0010)$ | 30 | $(\bigstar, q_{12}, \dot{0}\dot{1}\dot{0}0\dot{1}\dot{0})$ |
| 6 | $(\bigstar\dot{0}10, q_2, 010)$ | 31 | $(\bigstar 0, q_8, \dot{1}\dot{0}0\dot{1}\dot{0})$ $\ldots$ |
| 7 | $(\bigstar\dot{0}100, q_2, 10)$ | 32 | $(\bigstar 0\dot{1}\dot{0}, q_8, 0\dot{1}\dot{0})$ |
| 8 | $(\bigstar\dot{0}1001, q_2, 0)$ | 33 | $(\bigstar 0\dot{1}\dot{0}0, q_9, \dot{1}\dot{0})$ |
| 9 | $(\bigstar\dot{0}10010, q_2, \sqcup)$ | 34 | $(\bigstar 0\dot{1}\dot{0}, q_5, 01\dot{0})$ $\ldots$ |
| 10 | $(\bigstar\dot{0}1001, q_3, 0)$ | 35 | $(\bigstar, q_6, 0\dot{1}\dot{0}01\dot{0})$ |
| 11 | $(\bigstar\dot{0}100, q_4, 1\dot{0})$ | 36 | $(\bigstar 0, q_7, \dot{1}\dot{0}01\dot{0})$ |
| 12 | $(\bigstar\dot{0}10, q_4, 01\dot{0})$ | 37 | $(\bigstar 01, q_8, \dot{0}01\dot{0})$ |
| 13 | $(\bigstar\dot{0}1, q_4, 001\dot{0})$ | 38 | $(\bigstar 01\dot{0}, q_8, 01\dot{0})$ |
| 14 | $(\bigstar\dot{0}, q_4, 1001\dot{0})$ | 39 | $(\bigstar 01\dot{0}0, q_9, 1\dot{0})$ |
| 15 | $(\bigstar, q_4, \dot{0}1001\dot{0})$ | 40 | $(\bigstar 01\dot{0}01, q_9, \dot{0})$ |
| 16 | $(\bigstar\dot{0}, q_1, 1001\dot{0})$ | 41 | $(\bigstar 01\dot{0}0, q_{10}, 10)$ |
| 17 | $(\bigstar\dot{0}\dot{1}, q_2, 001\dot{0})$ $\ldots$ | 42 | $(\bigstar 01\dot{0}, q_{10}, 010)$ |
| 18 | $(\bigstar\dot{0}\dot{1}001, q_2, \dot{0})$ | 43 | $(\bigstar 01, q_{10}, \dot{0}010)$ |
| 19 | $(\bigstar\dot{0}\dot{1}00, q_3, 1\dot{0})$ | 44 | $(\bigstar 0, q_{11}, 1\dot{0}010)$ |
| 20 | $(\bigstar\dot{0}\dot{1}00, q_4, \dot{1}\dot{0})$ $\ldots$ | 45 | $(\bigstar 01, q_{12}, \dot{0}010)$ |
| 21 | $(\bigstar\dot{0}\dot{1}, q_4, 00\dot{1}\dot{0})$ | 46 | $(\bigstar 010, q_8, 010)$ |
| 22 | $(\bigstar\dot{0}, q_1, \dot{1}00\dot{1}\dot{0})$ | 47 | $(\bigstar 0100, q_9, 10)$ |
| 23 | $(\bigstar\dot{0}\dot{1}, q_1, 00\dot{1}\dot{0})$ | 48 | $(\bigstar 01001, q_9, 0)$ |
| 24 | $(\bigstar\dot{0}\dot{1}, q_1, 00\dot{1}\dot{0})$ $\ldots$ | 49 | $(\bigstar 010010, q_9, \sqcup)$ |
| 25 | $(\bigstar\dot{0}\dot{1}\dot{0}, q_1, \dot{0}\dot{1}\dot{0})$ | 50 | $(\bigstar 01001, q_A, 0)$ |

**End of example**

How TMs work should be quickly absorbed by computer scientists. After only a few detailed examples, such as the above, it becomes clear that Turing machine programs are very much like assembly-language programs, probably even worse in the level of detail. However, you should also become convinced that any program that could be coded in a high-level language like Java or ML could also be coded in a Turing machine, given enough effort. For example, it is tedious but not conceptually difficult to model the essential aspects of a microprocessor as a Turing machine: the ALU operations (addition, multiplication, *etc.*) can be implemented by the standard grade-school algorithms, the registers of the machine can be placed at certain specified sections of the tape, and the random-access memory can also be modelled by the tape. And so on.

**Ways of specifying Turing machines** There are three ways to specify a Turing machine. Each is appropriate at different times.

- Low level: the transition diagram is given explicitly. This level is only for true pedants! We sometimes ask for this, but it is often too detailed.

- Medium level: the operations of the algorithm are phrased in higher-level terms, but still in terms of the Turing machine model. Thus algorithmic steps are expressed in terms of how the tape head moves back and forth, *e.g., move the tape head all the way to the right, marking each character until the blanks are hit*. Also data layout conventions are discussed.

- High level: pseudo-code for the algorithm is given, in the standard manner that algorithms are discussed in Computer Science. The Church-Turing Thesis (to be discussed) will give us confidence that such a high-level description is implementable on a Turing machine.

### 3.1.2 Extensions

On top of the basic Turing machine model, more convenient models can be built. These new models still recognize the same set of languages, however.

#### Multiple Tapes

It can be very convenient to use Turing machines with multiple (unbounded) tapes. For example, if asked to implement addition of binary numbers on a Turing machine, it would be quite useful to have five tapes: a tape for each number being added, one to hold the sum, one for the carry being propagated, and one for holding the two original inputs. Such requirements can be easily implemented by an ordinary Turing machine: for $n$ tapes,

| |
|---|
| $tape_n$ |
| $tape_{n-1}$ |
| $\vdots$ |
| $tape_1$ |

we simply divide the single tape into $n$ distinct regions, separated by special markers, *e.g.*, **X** in the following:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $tape_1$ | **X** | $\cdots$ | **X** | $tape_{n-1}$ | **X** | $tape_n$ | **X** | ␣ | $\cdots$ |

Although it is easy to see—conceptually, at least—how to organize the $n$ tapes, including dealing with the situation when a tape 'outgrows its boundaries' and has to be resized, it is more difficult to see how to achieve the control of the machine. The transition function for $n$ tapes conceptually has to simultaneously look at $n$ current cells, write $n$ new contents, and make $n$ different moves of the $n$ independent tape heads. Thus the transition function will have the following specification:

$$\delta : Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{L, R\}^n$$

Each of the $n$ sub-tapes will have one cell deemed to be the current cell. We will use a system of markings to implement this idea. Since cells can't be marked, we will mark the contents of the current cell. This means that the tape alphabet $\Gamma$ will double in size: each symbol $a_i \in \Gamma$ will have a marked analogue $\dot{a}_i$, and by convention, there will be exactly one marked symbol per sub-tape. With this support, a move in the $n$-tape machine will consist of (1) a left-to-right sweep wherein the steps prescribed by $\delta$ are taken at each marked symbol, followed by (2) a right-to-left sweep to reset the tape head to the left-most cell on the underlying tape.

### Two-way infinite tape

A different extension would be to support a machine that has its tape extending infinitely in both directions.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $\cdots$ | ␣ | $w_1$ | $w_2$ | $\cdots$ | $w_{n-1}$ | $w_n$ | ␣ | $\cdots$ |

By convention, a computation would start with the tape head on the left-most symbol of the input string. This machine model is relatively easy (although detailed) to implement using an ordinary Turing machine. The main technique is to 'fold' the doubly infinite tape in two and merge the cells so that alternating cells on the resulting singly infinite tape belong to the two halves of the doubly-infinite tape. It helps to think of the tape elements as being labelled with integers. Thus if the original tape was labelled as

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $-3$ | $-2$ | $-1$ | $0$ | $1$ | $2$ | $3$ | $\cdots$ |

the single-tape version would be laid out as

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ★ | $0$ | $1$ | $-1$ | $2$ | $-2$ | $3$ | $-3$ | $\cdots$ |

Again, the details of how the control of the machine is achieved with an ordinary Turing machine are relatively detailed.

A further extension—non-determinism—can be built on top of ordinary Turing machines, and is discussed later.

### 3.1.3 Coding and Decoding

As we have seen, TMs are very basic machines, but you should be convinced that they allow all kinds of computation. They are similar, in a way, to machine-level instructions on an x86 or ARM chip. By using (in effect) `goto` statements, loops can be implemented, and the basic word and arithmetic operations can be modelled.

A separate aspect of the modelling of more conventional computation is *how to deal with high-level data*? For example, the only data accepted by a TM are strings built from $\Sigma$. How to deal with computations involving numbers, pairs, lists, finite sets, trees, graphs, and even Turing machines themselves? The answer is *coding*: complex datastructures will be represented by using a convention for reading and writing them as strings.

Bits can be represented by $0$ and $1$, of course, and we all know how numbers in $\mathbb{N}$ and $\mathbb{Z}$ are encoded in binary. A pair of elements $(a, b)$ may be encoded as follows. Suppose that object $a$ encodes to a string $w_a$ and object $b$ encodes to $w_b$. Then one simple way to encode $(a, b)$ is as

$$\blacktriangleleft w_a \ddagger w_b \blacktriangleright$$

where $\blacktriangleleft$, $\blacktriangleright$, and $\ddagger$ are symbols not occurring in the encoding of $a$ and $b$ . The encoding of a list of objects $[a_1, \cdots, a_n]$ can be implemented by iterating the encoding of pairs, *i.e.*, as

$$\blacktriangleleft a_1 \ddagger \blacktriangleleft a_2 \ddagger \blacktriangleleft \cdots \ddagger \blacktriangleleft a_{n-1} \ddagger a_n \blacktriangleright \cdots \blacktriangleright\blacktriangleright\blacktriangleright$$

Finite sets can be encoded in the same way as lists. Arbitrary trees can be represented by binary trees, and binary trees can be encoded, again, as nested pairs. In effect, we are re-creating Lisp s-expressions. A graph is usually represented as $(V, E)$ where $V$ is a finite set of nodes and $E$ is a finite set of edges, *i.e.*, a set of pairs of nodes. Again, this format is easy to encode and decode.

The art of coding and decoding data pervades Computer Science. There is even a related research area known as *Coding Theory*, but the subject of efficient algorithms for coding and decoding is really orthogonal to our purposes in this course.

We shall henceforth assume that encoding and decoding of any desired high-level data can be accomplished. We will assume that, for each particular problem intended to be solved by a TM, the input is given in an encoding that the machine can decode correctly. Similarly, if a machine produces output, it will likewise be decodable. For this, we will adopt the notation $\langle A \rangle$, to represent an object $A$ which has been encoded to a string, and which will be decodable by the TM to a correct representation of $A$. A tuple of objects in the input will be written as $\langle A_1, \ldots, A_k \rangle$.

**Encoding Turing machines**

The encoding approach outlined above is quite general, but just to reinforce the ideas we will discuss a specific encoding for Turing machines. This format will be expected by Turing machines that take encoded Turing machines as input and calculate facts about those machines. This ability will let us formulate questions about the theoretical properties of programs that manipulate and analyze programs.

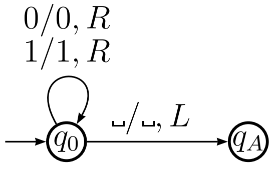
Recall that a Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$. The simplest possible way to represent this tuple on a Turing machine tape is to write out the elements of the tuple from left to right on the tape, using a delimiter to separate the components. Something like the following

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w_Q$ | X | $w_\Sigma$ | X | $w_\Gamma$ | X | $w_\delta$ | X | $w_{q_0}$ | X | $w_{q_A}$ | X | $w_{q_R}$ | X | ␣ | $\cdots$ |

where $w_Q, w_\Sigma, w_\Gamma, w_\delta, w_{q_0}, w_{q_A}$ and $w_{q_R}$ are strings representing the components of the machine. In detail

- $Q$ is a finite set of states. We could explicitly list out the state elements, but will instead just write out the *number* of states in unary notation.
- $\Sigma$ is the input alphabet, a finite set of symbols. Our format will list the symbols out, in no particular order, separated by blanks. Recall that the blank is not itself an input symbol.
- $\Gamma$ is the tape alphabet, a finite set of symbols with the property that $\Sigma \subset \Gamma$. In our format, we will just list the extra symbols not already in $\Sigma$. Blank is a tape symbol.
- $\delta$ is a function and one might think that there would be a problem with representing it, especially since functions can have infinite domains. Fortunately, $\delta$ has a finite domain (since $Q$ is finite and $\Gamma$ is finite, $Q \times \Gamma$ is finite). Therefore $\delta$ can be listed out. Each individual transition $\delta(p, a) = (q, b, d)$ can be represented as a 5-tuple $(p, a, q, b, d)$. On the tape each of these tuples will look like *p␣a␣q␣b␣d*. (If $a$ or $b$ happen to be the blank symbol ␣ no ambiguity should result.) Each 5-tuple will be separated from the others by a XX.
- $q_0, q_A, q_R$ will be represented by numbers in unary notation.

**Example 38.** The following simple machine



will be represented on tape by

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 111 | X | 0␣1 | X | ␣ | X | $\delta$ | X | 1 | X | 11 | X | 111 | X | ␣ | ··· |

where the $\delta$ portion is (only the last transition is written one symbol per cell and $q_0, q_A$ are encoded by $1, 11$):

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1␣0␣1␣0␣11 | XX | 1␣1␣1␣1␣11 | XX | 1 | ␣ | ␣ | ␣ | 11 | ␣ | ␣ | ␣ | 1 |

Note that the direction $L$ is encoded by $1$ and $R$ is encoded by $11$.

**Example 39.** A TM $M$ that takes an arbitrary TM description and tests whether it has an even number of states can be programmed as follows: on input $\langle Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R \rangle$, $M$ checks that the input is in fact a representation of a TM and then checks the number of states in $Q$. If that number is even, then $M$ clears the tape and transitions into its accept state (which is different than the $q_A$ of the input machine) and halts; otherwise, it clears the tape and rejects.

### 3.1.4 Universal Turing machines

The following question naturally arises:

> *Can a single Turing machine be written to simulate the execution of arbitrary Turing machines?*

The answer is yes. A so-called *universal Turing machine* $\mathcal{U}$ can be written that expects as input $\langle M, w \rangle$, where $M$ is a Turing machine description, encoded as outlined above, for example, and $w$ an input string for $M$. $\mathcal{U}$ simulates the execution of $M$ on $w$. The simulation is completely unintelligent: it simply transcribes the way that $M$ would behave when started with $w$ on its input tape. Thus given input in the following format

| | | | | | |
|---|---|---|---|---|---|
| $\mathit{TM\ description}$ | X | $w$ | X | ␣ | $\cdots$ |

an execution of $\mathcal{U}$ sequentially lists out the configurations of $M$ as it evaluates on $w$. At each 'step' of $M$, $\mathcal{U}$ goes to the end of the tape, looks at the current configuration of $M$, extracts $(p, a)$, the current state and cell value from it, then looks up the corresponding $(q, b, d)$ (next state, cell value, and direction) from the description of $M$, and uses that information to go to the end of the tape and append the new configuration for the execution of $M$ on $w$. As it runs, the execution of $\mathcal{U}$ will generate a tape that evolves as follows:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathit{TM\ description}$ | X | $w$ | X | ␣ $\cdots$ | | | | | | | |
| $\mathit{TM\ description}$ | X | $w$ | X | $\mathit{config}_0$ | X | ␣ $\cdots$ | | | | | |
| $\mathit{TM\ description}$ | X | $w$ | X | $\mathit{config}_0$ | X | $\mathit{config}_1$ | X | ␣ $\cdots$ | | | |
| $\mathit{TM\ description}$ | X | $w$ | X | $\mathit{config}_0$ | X | $\mathit{config}_1$ | X | $\mathit{config}_2$ | X | ␣ | $\cdots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | |

If $M$ eventually halts on $w$, then $\mathcal{U}$ will detect this (since the last configuration on the tape will be in state $q_A$ or $q_R$) and will then transition into the corresponding terminal state of $\mathcal{U}$. Thus, if $M$ halts when run on $w$, then the simulation of $M$ on $w$ by $\mathcal{U}$ will also halt. If $M$ loops on $w$, the simulation of $M$'s execution of $w$ will also diverge, endlessly appending new configurations to the end of the tape.

### Self application

A Turing machine $M$ that takes as input an (encoded) TM and performs some calculation using the description of the machine can, of course, be applied to all manner of Turing machines. Is there a problem with applying $M$ to its own description? After all, the notion of self-application can be extremely confusing, since infinite regress is a lurking possibility. But consider the 'even-number-of-states-tester' given above in Example 39. When applied to itself, *i.e.*, to its own description, it performs in a sensible manner since it just treats its input machine description as data. The following 'real-world' example similarly shows that the treatment of programs as (encoded) data allows some instances of self-reference to be straightforward.

**Example 40.** The Unix **wc** utility counts the number of characters in a file. It can be applied to itself

```
bash-2.05b$ /usr/bin/wc /usr/bin/wc
     58     480   22420 /usr/bin/wc
```

with no fear of infinite regress. Turing machines that treat their input TM descriptions simply as data typically don't raise any difficult conceptual issues.

## 3.2 Register Machines

Now that we have seen Turing machines, it is worth having a look at another model of computation. A *Register Machine* (RM) has a fixed number of registers, each of which can hold a natural number of unbounded size (no petty and humiliating 32- or 64-bit restrictions for theoreticians!).

A register machine program is a list of simple instructions. There are only 2 kinds of instruction:

- Inc $r\ i$. Add 1 to register $r$ and move to instruction $i$
- Test $r\ i\ j$. Test if register $r$ is equal to zero: if so, go to instruction $i$; else subtract 1 from register $r$ and go to instruction $j$.

Like TMs there is a notion of the current 'state' of a Register machine; this is just the current instruction that is to be executed. By convention, if the machine is asked to execute the 0-th instruction it will stop. An execution of a RM starts in state 1, with input $(n_1, \ldots, n_k)$ loaded into registers $R_1, \ldots, R_k$ and executes instructions from the program until instruction 0 is entered.

**Example 41** (Just stop)**.** The following RM program immediately halts, no matter what data is in the registers.

| | |
|---|---|
| 0 | HALT |
| 1 | Test $R_0$ $I_0$ $I_0$ |

Execution starts at instruction 1. The Test instruction checks if $R_0 = 0$ and, no matter what the result is, the next instruction is $I_0$, *i.e.*, the HALT instruction.

**Example 42** (Infinite loop)**.** The following RM program immediately goes into an infinite loop, no matter what data is in the registers.

| | |
|---|---|
| 0 | HALT |
| 1 | Test $R_0$ $I_1$ $I_1$ |

Execution starts at instruction 1. The Test instruction checks if $R_0 = 0$ and, no matter what the result is, the next instruction is $I_1$, *i.e.*, the same instruction. Thus an execution will step-by-step decrement $R_0$ down to 0 but will then keep going.

**Example 43.** The following RM program adds the contents of $R_0$ to $R_1$, destroying the contents of $R_0$.

| | |
|---|---|
| 0 | HALT |
| 1 | Test $R_0$ $I_0$ $I_2$ |
| 2 | Inc $R_1$ $I_1$ |

Execution starts at instruction 1. The Test instruction checks $R_0$, exiting if it holds 0. Otherwise, it decrements $R_0$ and transfers control to instruction 2. This then adds 1 to $R_1$ and transfers control back to instruction 1.

The following table shows how the execution evolves, step by step when $R_0$ has been loaded with 3 and $R_1$ with 19.

| Step | $R_0$ | $R_1$ | Instr |
|---|---|---|---|
| 0 | 3 | 19 | 1 |
| 1 | 2 | 19 | 2 |
| 2 | 2 | 20 | 1 |
| 3 | 1 | 20 | 2 |
| 4 | 1 | 21 | 1 |
| 5 | 0 | 21 | 2 |
| 6 | 0 | 22 | 1 |
| 7 | 0 | 22 | HALT |

In the beginning (step 0), the machine is loaded with its input numbers, and is at instruction 1. At step 1 $R_0$ is decremented and the machine moves to instruction 2. And so on.

Notice that we could also represent the execution by the following sequence of triples $(R_0, R_1, \mathit{Instr})$:

$$(3, 19, 1), (2, 19, 2), (2, 20, 1), (1, 20, 2), (1, 21, 1), (0, 21, 2), (0, 22, 1), (0, 22, 0).$$

OK, one more example. How about adding $R_0$ and $R_1$, putting the result in $R_2$, *leaving $R_0$ and $R_1$ unchanged?*

**Example 44.** As always, it is worth thinking about this at a high level before diving in and writing out the exact instructions. The best approach I could think of uses five registers $R_0, R_1, R_2, R_3, R_4$. We use $R_3$ and $R_4$ to store the original values of $R_0$ and $R_1$. The program first (instructions 1–3) repeatedly decrements $R_0$ and adds 1 to both $R_2$ and $R_3$. At the end of this phase, $R_0$ is 0 and both $R_2$ and $R_3$ will hold the original contents of $R_0$.

Next (instructions 4–6) the program repeatedly decrements $R_1$ and adds 1 to both $R_2$ and $R_4$. At the end of this phase, $R_1$ is 0, $R_2$ holds the sum of the original values of $R_0$ and $R_1$, and $R_4$ holds the original contents of $R_1$.

Finally, a couple of loops (instructions 7–8 and 9–10) move the contents of $R_3$ and $R_4$ back to $R_0$ and $R_1$. Here is the program:

| | |
|---|---|
| 0 | HALT |
| 1 | Test $R_0$ $I_4$ $I_2$ |
| 2 | Inc $R_2$ $I_3$ |
| 3 | Inc $R_3$ $I_1$ |
| 4 | Test $R_1$ $I_7$ $I_5$ |
| 5 | Inc $R_2$ $I_6$ |
| 6 | Inc $R_4$ $I_4$ |
| 7 | Test $R_3$ $I_9$ $I_8$ |
| 8 | Inc $R_0$ $I_7$ |
| 9 | Test $R_4$ HALT $I_{10}$ |
| 10 | Inc $R_1$ $I_9$ |

For the intrepid, here's the execution of the machine when $R_0 = 2$ and $R_1 = 3$.

| Step | $R_0$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ | Instr |
|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 0 | 0 | 0 | 1 |
| 1 | 1 | 3 | 0 | 0 | 0 | 2 |
| 2 | 1 | 3 | 1 | 0 | 0 | 3 |
| 3 | 1 | 3 | 1 | 1 | 0 | 1 |
| 4 | 0 | 3 | 1 | 1 | 0 | 2 |
| 5 | 0 | 3 | 2 | 1 | 0 | 3 |
| 6 | 0 | 3 | 2 | 2 | 0 | 1 |
| 7 | 0 | 3 | 2 | 2 | 0 | 4 |
| 8 | 0 | 2 | 2 | 2 | 0 | 5 |
| 9 | 0 | 2 | 3 | 2 | 0 | 6 |
| 10 | 0 | 2 | 3 | 2 | 1 | 4 |
| 11 | 0 | 1 | 3 | 2 | 1 | 5 |
| 12 | 0 | 1 | 4 | 2 | 1 | 6 |
| 13 | 0 | 1 | 4 | 2 | 2 | 4 |
| 14 | 0 | 0 | 4 | 2 | 2 | 5 |
| 15 | 0 | 0 | 5 | 2 | 2 | 6 |
| 16 | 0 | 0 | 5 | 2 | 3 | 4 |
| 17 | 0 | 0 | 5 | 2 | 3 | 7 |
| 18 | 0 | 0 | 5 | 1 | 3 | 8 |
| 19 | 1 | 0 | 5 | 1 | 3 | 7 |
| 20 | 1 | 0 | 5 | 0 | 3 | 8 |
| 21 | 2 | 0 | 5 | 0 | 3 | 7 |
| 22 | 2 | 0 | 5 | 0 | 3 | 9 |
| 23 | 2 | 0 | 5 | 0 | 2 | 10 |
| 24 | 2 | 1 | 5 | 0 | 2 | 9 |
| 25 | 2 | 1 | 5 | 0 | 1 | 10 |
| 26 | 2 | 2 | 5 | 0 | 1 | 9 |
| 27 | 2 | 2 | 5 | 0 | 0 | 10 |
| 28 | 2 | 3 | 5 | 0 | 0 | 9 |
| 29 | 2 | 3 | 5 | 0 | 0 | HALT |

Now, as we have seen, the state of a Register machine computation that uses $n$ registers is $(R_0, \ldots, R_{n-1}, I)$, where $I$ holds the index to the current instruction to be executed.

## 3.3 The Church-Turing Thesis

Computability theory developed in the 1930's in an amazing burst of creativity by logicians. We have seen two *fully-featured* models of computation—Turing machines and Register machines—but there are many more, for example $\lambda$-calculus (due to Alonzo Church), combinators (due to Haskell Curry), Post systems (due to Emil Post), Term Rewriting systems, unrestricted grammars, cellular automata, FRACTRAN, *etc*.

These models have all turned out to be equivalent, in that each allows the same set of functions to be computed. Before we give an indication of what such an equivalence proof looks like in the case of Turing machines and Register machines, we can make some general remarks.

A full model of computation can be seen as a setting forth of a general way to do sequential computation, *i.e.*, to deterministically compute the values of functions, over some kind of data (often strings or numbers). The requirement on an author of such a model is to show how all tasks we regard as being 'computable' by a real mechanical device, or solvable by an algorithm, may be realized in the model. Typically, this splits into showing

- How all manner of data, *e.g.*, trees, graphs, arrays, *etc*, can be encoded to, and decoded from, the data representation used by the model.

- How all manner of algorithmic steps and data manipulation may be reduced to steps in the proposed model.

Put this way, it seems possible that a chaotic situation could have developed, where multiple competing notions of computability struggled for supremacy. But that hasn't happened. Instead, the proofs of equivalence among all the different models mean that people can use whatever model they prefer, secure in the knowledge that, were it necessary or convenient, they could have worked in any of the other models. This is the *Church-Turing Thesis*: that any model of computation is as powerful as any other, and that any fresh one proposed is anticipated to be equivalent to all the others. This is what gives people confidence that any algorithm coded as a 'C' program can also be coded up in Java, or Perl, or any other general purpose programming language. Note well, however, that all considerations

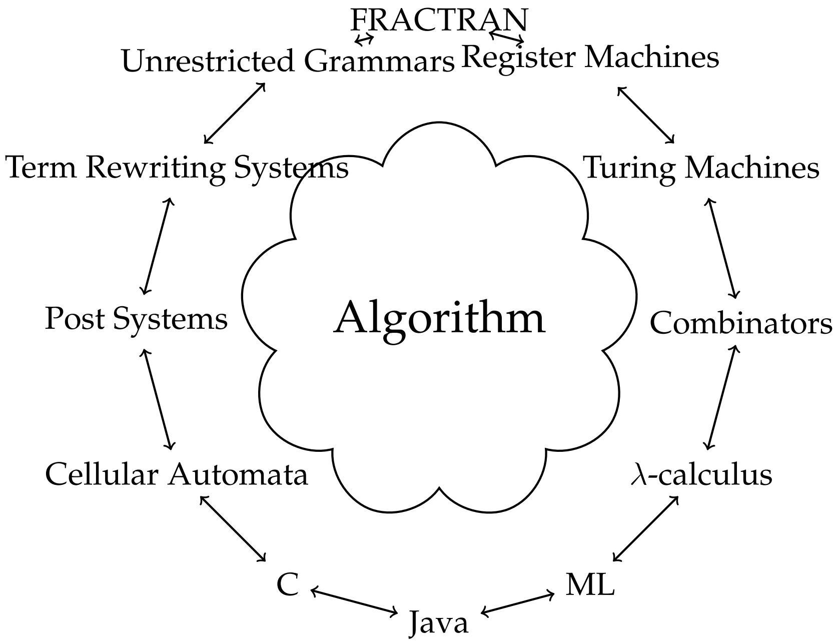


Figure 3.1: A portion of the Turing tarpit

of efficiency have been discarded; we are in this course only concerned with what can be done in principle.

This is a *thesis* and not a *theorem* because it is not provable, only refutable. A genuinely new model of computation, not equivalent to Turing machine computation, may be proposed tomorrow and then the CT-Thesis will go the way of the dodo. All that would be necessary would be to demonstrate a program in the new computational model that was not computable on a sufficiently programmed Turing machine. However, given the wide array of apparently very different models, which are all nonetheless equivalent, most experts feel that the CT-Thesis is secure.

The CT-Thesis says that no model of computation exists having greater power than Turing machines, or Register machines, or $\lambda$-calculus, or any of the others we have mentioned. As a consequence, no model of computation can lay claim to being *the unique* embodiment of algorithms. An algorithm, therefore, is an abstract idea, which can have concrete realizations in particular models of computation, but also of course on real computers and in real programming languages. However, we (as yet) have no abstract definition of the term *algorithm*, of which the models of computation are instances. Thus the search for an algorithm implementing a requirement has to be met by supplying a program. If such a program exists in one model of general computation or programming language, then it can be translated to any other model of general computation or programming language. Contrarily, if a requirement *cannot* be implemented in a particular model of computation, then it also cannot be implemented in any other.

As a result of adopting the CT-Thesis, we can use abstract methods, *e.g.*, notation from high-level programming languages or even mathematics, to describe algorithmic behaviour, and know that the algorithm can be implemented on a Turing Machine. Thus we may shift our attention from painstakingly implementing algorithms in horrific detail on simple models of computation. We will now assume programs exist to implement the desired algorithms. Of course, we may be challenged to show that a purported algorithm is indeed implementable; then we may choose whatever Turing-equivalent model we wish in order to write the program.

Finally, the scope of the CT-Thesis must be understood. The models of computation are intended to capture computation of mathematical functions. In other words, whenever a TM $M$ is applied to an input string $w$, it will always return the same answer. Dealing with interactive, randomized, or distributed computation requires extensions which have been the source of much debate. They are however, beyond the scope of this course.

### 3.3.1 Equivalence of Turing and Register machines

The usual way that equivalence between models of computation $A$ and $B$ is demonstrated is to show two things:

- How an arbitrary $A$ program $p_A$ can be translated to a corresponding $B$ program $p_B$ such that running $p_A$ on an input $i_A$ will yield an identical answer to running $p_B$ on input $i_B$ (which is $i_A$ translated into the format required by $B$ programs).

- And *vice versa*.

Thus, if $A$-programs can be simulated by $B$-programs and $B$-programs can be simulated by $A$-programs, every algorithm that can be written in $A$

can also be written in $B$, and *vice versa*. The equivalence can be captured in the following diagram:

$$\begin{tikzcd}
(p_A, i_A) \arrow[r, "\mathsf{run}_A"] \arrow[d, "\mathsf{toB}"'] & \mathit{result}_A \\
(p_B, i_B) \arrow[r, "\mathsf{run}_B"'] & \mathit{result}_B \arrow[u, "\mathsf{toA}"']
\end{tikzcd}$$

which expresses the following equation:

$$\mathsf{run}_A(p_A, i_A) = \mathsf{toA}(\mathsf{run}_B(\mathsf{toB}_{\mathit{prog}}(p_A), \mathsf{toB}_{\mathit{inputs}}(i_A)) \ .$$

### Simulating a Register machine on a Turing machine

OK, let's consider how we could mimic the execution of a Register machine on a Turing machine. Well, the first task is to use the data representation of Turing machines (tape cells) to represent the data representation of Register machines (a tuple of numbers). This is quite easy, since we have seen it already in mimicking multiple tapes with a single tape. If the given RM has $n$ registers containing $n$ values, the tape is divided into $n$ separate sections, each which holds a string (say in unary representation) representing the corresponding number.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| ★ | $\langle R_0 \rangle$ | X | $\langle R_1 \rangle$ | X | $\ldots$ | X | $\langle R_{n-1} \rangle$ | X | ␣ | $\cdots$ |

Now how do we represent a RM program (list of instructions) as a TM program? The rough idea is to represent each RM instruction by a sequence of TM transitions. Fortunately there are only two instructions to consider:

Inc $R_i$ $I_k$ would get translated to a sequence of TM operations that would (assuming tape head in leftmost cell):

- Move left-to-right until it found the portion of tape corresponding to $R_i$.

- Increment that register (which is of course represented by a bitstring). Also note that this operation could require resizing the portion of tape for $R_i$.
- Move tape head all the way left.
- Move to state that represents the beginning of the TM instruction sequence for instruction $k$.

Test $R_i\ I_j\ I_k$ again translates to a sequence of TM operations. Again, assume tape head in leftmost cell:

- Find $R_i$ portion of the tape.
- Check if that portion is empty ($\varepsilon$ corresponds to the number 0).
- If it is empty move to the TM state representing the start of the sequence of TM operations corresponding to the instruction $I_j$.
- If not, decrement the $R_i$ portion of the tape (again via bitstring operations).
- And then go to the TM state corresponding to the RM instruction $k$.

### Simulating a Turing machine on a Register machine

What about modelling TMs by RMs? That is somewhat more convoluted. One might think that each cell of the TM tape could be represented by a corresponding register. But this has a problem, namely that the number of registers in a register machine must be fixed in advance, while a TM tape can grow and shrink as the computation progresses. The correct modelling of the tape by registers depends on an interesting technique known as *Goedel numbering*.[5]

**Definition 6** (Goedel Numbering)**.** Goedel needed to encode a string $w = a_1a_2a_3 \ldots a_n$ over alphabet $\Sigma$ as a single number $n \in \mathbb{N}$. Recall that the primes are infinite, *i.e.*, form an infinite sequence $p_1, p_2, p_3, \ldots$. Also recall—from high school math—that every number has a unique *prime factorization*.[6] Let $\mathsf{C} : \Sigma \to \mathbb{N}$ be a bijective coding of alphabet symbols by natural

[5] Named after the logician Kurt Goedel, who used it in his famouse proof of the incompleteness of Peano Arithmetic.

[6] This fact is known as the Fundamential Theorem of Arithmetic.

numbers. Goedel's idea was to basically treat $w$ as a representation of the prime factorization of some number.

$$\mathsf{GoedelNum}(a_1 a_2 a_3 \ldots a_n) = p_1^{\mathsf{C}(a_1)} \times p_2^{\mathsf{C}(a_2)} \times p_3^{\mathsf{C}(a_3)} \times \ldots \times p_n^{\mathsf{C}(a_n)}\ .$$

**Example 45.** Let's use the string `foobar` as an example. Taking ASCII as the coding system, we have: $\mathsf{C}(\mathtt{f}) = 102$, $\mathsf{C}(\mathtt{o}) = 111$, $\mathsf{C}(\mathtt{b}) = 98$, $\mathsf{C}(\mathtt{a}) = 97$, and $\mathsf{C}(\mathtt{r}) = 114$. The Goedel number of `foobar` is calculated as follows:

$$\begin{aligned}
&\mathsf{GoedelNum}(\mathtt{foobar}) = 2^{102} \times 3^{111} \times 5^{111} \times 7^{98} \times 11^{97} \times 13^{114} \\
&\quad = 1189679170019703840793287149012377883426615618297637319510394 \\
&\qquad 7574598871098617271030519293961835685919348532420791233827072 \\
&\qquad 8538012810574164152676463935146039040414969145473843310178957 \\
&\qquad 3364837167118857151641747100976606825914043115103029097422785 \\
&\qquad 5974811557927500973868941709639836429302128569769744552203051 \\
&\qquad 4615853924974445020975274707809005698958254207677547656309707 \\
&\qquad 0312500000000000000000000000000000000000000000000000000000000 \\
&\qquad 0000000000000000000000000000000000000000000000
\end{aligned}$$

The importance of Goedel numbers is that, given a number that we know to be the result of a call to GoedelNum, we can take it apart to get the original sequence, or parts of the original sequence. Thus we can use GoedelNum to encode the contents of a Turing machine tape as a single (very large) number, which can be held in a register of the Register machine.

In fact, we won't hold the entire tape in one register, but will keep the current configuration $\langle \ell, q, r \rangle$ in registers $R_1, R_2, R_3$. In $R_0$ we will put the Goedel number of the transition function $\delta$ of the Turing machine, *i.e.*, the Goedel number of its tape format (see Section 3.1.3 for details of the format). The RM program simulating the execution of the TM will examine $R_2$ and $R_3$ to find $(p, a)$, the current state and tape cell contents. Then $R_0$ can be manipulated to give a number representing $(q, b, d)$, which are then used to update $R_1, R_2, R_3$. That finishes the current step of the TM; the RM program then checks to see if a final state of the TM has been entered. If so, the RM stops executing; if not, the next TM step is simulated, continuing until the simulated TM enters a final state.

## 3.4 Recognizabilty and Decidability

Now that we have become familiar with Turing machines and Register machines, we should ask what they are good for. For example, Turing machines definitely aren't good for programming, so why do we study and use them?

- People agree that they capture the notion of a sequential algorithm. Although there are many other formalisms that also capture this notion, Turing machines were historically the first convincing model that researchers of the time agreed upon.
- Robustness. The class of functions that TMs capture doesn't change when the Turing machine model is tinkered with, *e.g.*, by adding multiple tapes, non-determinism, *etc*.
- Turing machines are easy to analyze with respect to their time and space usage, unlike many other models of computation. This supports the field of computational complexity, in which classifications according to the resource usage of computations is studied.
- They can be used to prove decidability results and, more interestingly, *undecidability* results. In fact, Turing's invention of Turing machines led him to prove the undecidability of the halting problem. This is what we turn to now.

We know that a Turing machine takes an input and either accepts it, rejects it, or loops. In order to tell if a Turing machine works properly, one needs to specify what answers it should compute. The usual way to do this is to specify the set of strings the TM must accept, *i.e.*, its language. We now make a crucial (as it turns out) distinction between TMs that reject strings outside their language and TMs that either reject or loop when given a string outside their language.

**Definition 7** (Turing-recognizable, Recursively enumerable)**.** A language $L$ is *Turing recognizable*, or *recursively enumerable*, if there is some Turing machine $M$ (called a *recognizer*) that accepts each string in $L$:

$$L = \mathcal{L}(M) = \{w \mid M \text{ halts and accepts } w\} .$$

But what about strings not in $L$? Notice that $M$ only has to halt in the accept state when run on strings in $L$ to be a recognizer; when run on strings outside of $L$, $M$ is allowed to halt in the reject state or loop.

The following definition captures the subset of recognizers that never loop.

**Definition 8** (Decidable, Recursive)**.** A language $L$ is *decidable*, or *recursive*, if there exists a Turing machine $M$, called a *decider*, that recognizes $L$ and always halts.

Thus a decider always says 'yes' given a string in the specified language (call it $L$), and always says 'no' given a string outside the specified language, *i.e.*, in $\Sigma^* - L$. Obviously, it is better to have a decider than a recognizer, since a decider always gives a verdict in a finite time. Moreover, a decider is automatically a recognizer.

A restatement of these definitions is the following.

**Definition 9** (Decision problem)**.** The *decision problem* for a language $L$ is just the question: is there a decider for $L$? Equivalently, one asks if $L$ is decidable. Similarly, the *recognition problem* for a language $L$ is the question: is $L$ recognizable (recursively enumerable).

*Remark.* There is a lot of overlapping terminology. In particular, the notions of *language* and *problem* are essentially the same thing: a set of strings over an alphabet. A problem often has the implication that an encoding has been used to translate higher-level data structures into strings.

In general, to show that a problem is decidable, it suffices to (a) exhibit an algorithm for solving the problem and (b) to show that the algorithm terminates on all inputs. To show that a language is recognizable only the first requirement has to be met.

Some examples of decidable languages:

- binary strings
- natural numbers
- twos complement numbers
- sorted lists of numbers
- balanced binary trees

- Well-formed Turing machines
- Well-formed C programs
- $\{\langle i, j, k\rangle \mid i + j = k\}$
- $\{\langle \ell, \ell'\rangle \mid \ell' \text{ is a sorted permutation of } \ell\}$

In a better world than ours, one would expect that more informative properties such as the following would also be decidable:

- Turing machines that halt no matter what input is given them.
- C programs that never crash.
- Given a program in a favourite programming language, and a program location, is the location reachable?
- Given programs $p_1$ and $p_2$ does $p_1$ behave just like $p_2$? In other words is it the case that $p_1(x) = p_2(x)$, for any input $x$?
- Is a program the most efficient possible?

Unfortunately, such is not the case. And we can prove it. In fact, in order to make such claims, a clever proof is necessary since we are asserting that various problems are *unsolvable*, *i.e.*, that *no* program can be constructed to solve the problem.

But first we are going to look at a set of decision problems about TMs that can be solved. If Turing machines seem too unworldly for you, the problems are easily restated to apply to your favorite programming language or microprocessor.

### 3.4.1 Decidable problems about Turing machines

Here is a list of decidable problems about TMs. We are given TM $M$. The number 35 is not important in the following.

1. Does $M$ have at least 35 states?
   Decidable (obviously).

2. Does $M$ take more than 35 steps on input $x$?
   Decidable.
   The decider will use $\mathcal{U}$ to run $M$ on $x$ for 35 steps. If $M$ has not entered $q_A$ or $q_R$ by then, accept, else reject.

3. Does $M$ take more than 35 steps on some input?
   Decidable.
   The decider will simulate $M$ on all strings of length $\leq 35$, for 35 steps. If $M$ has not entered $q_A$ or $q_R$ by then, for at least one string, accept, else reject. We need only consider strings of length $\leq 35$: longer strings will take more than 35 steps to read.

4. Does $M$ take more than 35 steps on all inputs?
   Decidable.
   Similar to the previous, except that we require that $M$ take $\geq 35$ steps for each string.

5. Does $M$ ever move the tape head more than 35 cells away from the starting position?
   Decidable.
   $M$ will either loop infinitely within the 35 cells, stop within the 35 cells, or break out of the 35 cells. We can detect the infinite loop by keeping track of the configurations that $M$ can get into: for a fixed tape size (35 in this problem), this is a finite number of configurations. In particular, if $M$ has $n$ states and $k$ tape symbols, then the number of configurations it can get into is $36 * n * k^{35}$. If $M$ hasn't re-entered a previous state or halted in that number of moves, the tape head must have moved more than 35 cells away from its initial position.

### 3.4.2 Recognizable problems about Turing Machines

**Example 46.** Suppose our task is to take an arbitrary Turing machine description $M$ and an input $w$ and figure out whether $M$ accepts $w$. This decision problem can be formally stated as

$$L = \{\langle M, w\rangle \mid M \text{ accepts } w\}$$

This problem is recognizable, since we can use the universal TM $\mathcal{U}$ in the following way: $\mathcal{U}$ is invoked on $\langle M, w\rangle$ and keeps track of the state of

$M$ as the execution of $M$ on $w$ unfolds. If $M$ halts, then we look to see what final state it was in: if it was its accept state, then the result is to accept. Otherwise the result is to reject the input. Now, the other possibility is that $M$ doesn't halt on $w$. What then? Since $\mathcal{U}$ simply follows the execution of $M$ step by step, then if $M$ loops on $w$, the simulation will never stop. The problem of getting a decider would require being able to calculate from the given description $\langle M, w\rangle$ whether the ensuing computation would stop or not. And we can't do that.

The following example illustrates an important technique for building deciders and recognizers.

**Example 47** (Dovetailing)**.** Suppose our task is to take an arbitrary Turing machine description $M$ and tell whether there is any string that it accepts. This decision problem can be formally stated as

$$L = \{\langle M\rangle \mid \exists w.\ M \text{ accepts } w\}$$

A naive stab at an answer would say that a recognizer for $L$ is readily implemented by generating strings in $\Sigma^*$ in increasing order, one at a time, and running $M$ on them. If a $w$ is generated so that a simulated execution of $M$ accepts $w$, then our recognizer halts and accepts. However, it may be the case that $M$ accepts nothing, in which case this program will loop forever. Thus, on the face of it, this is a recognizer but not a decider.

However, this is not even a recognizer! What if $M$ loops on some $w$, but would accept some (longer) string $w'$? Blind simulation will loop on $w$ and $M$ will never get invoked on $w'$.

This problem can be solved, in roughly the same way as the same problem is solved in time-shared operating systems running processes: some form of fair scheduling. This can be implemented by interleaving the generation of strings with applying $M$ to each of them for a limited number of steps. The algorithm goes round-by-round. In the first round of the algorithm, $M$ is simulated for one step on $\varepsilon$. In the second round of the algorithm, $M$ is simulated for one more step on $\varepsilon$, and $M$ is simulated for one step on $0$. In the third round of the algorithm, $M$ is simulated for one more step on $\varepsilon$ and $0$, and $M$ is simulated for one step on $1$. In the fourth round, $M$ is simulated for one more step on $\varepsilon$, $0$, $1$, and is simulated for one step on $00$. Computation proceeds, where in each round all existing sub-computations advance by one step, and a new sub-computation on the next string in increasing order is started. This proceeds until in some sub-computation $M$ enters the accept state. If it enters a reject state, that sub-computation is dropped. The process just outlined is often called *dovetailing*, because of the fan shape that the computation takes.

Clearly, if some string $w$ is accepted by $M$, it will start being processed in some round, and eventually accepted, possibly much later. So the language $L$ is recursively enumerable (recognizable).

One way of showing a language $L$ is decidable, is to show that there are recognizers for $L$ and $\overline{L}$.

**Theorem 1.** *If $L$ is recognizable and $\overline{L}$ is recognizable then $L$ is decidable.*

*Proof.* Let $M_1$ be a recognizer $L$ and $M_2$ a recognizer for $\overline{L}$. For any $x \in \Sigma^*$, $x \in L$ or $x \in \overline{L}$. A decider for $L$ can thus be implemented that, on input $x$, dovetails execution of $M_1$ and $M_2$ on $x$. Eventually, one of the machines must halt, with an accept or reject verdict. If $M_1$ halts first, the decider returns the verdict of $M_1$; if $M_2$ halts first, the decider returns the opposite of the verdict of $M_2$. □

### 3.4.3 Closure Properties

A *closure property* has the format

> If $L_1$ and $L_2$ have property $P$ then the language resulting from applying some operation to $L_1$ and $L_2$ also has property $P$.

The decidable languages are closed under union, intersection, concatenation, complement, and Kleene star. The recognizable languages are closed under all the above but complement and set difference.

**Example 48.** The decidable languages are closed under union. Formally, this is expressed as

$$\textsf{Decidable}\ L_1 \wedge \textsf{Decidable}\ L_2 \Rightarrow \textsf{Decidable}\ (L_1 \cup L_2)$$

*Proof.* Suppose $L_1$ and $L_2$ are decidable. Then there exists a decider $M_1$ for $L_1$ and a decider $M_2$ for $L_2$. Now we claim there is a decider $M$ for $L_1 \cup L_2$. Let $M$ be the following machine:

> "On input $w$, invoke $M_1$ on $w$. If it accepts, then accept. Otherwise, $M_1$ halts and rejects $w$ (because it is a decider), so we invoke $M_2$ on $w$. If it accepts, $M$ accepts, otherwise it rejects."

That $M$ is a decider is clear since both $M_1$ and $M_2$ halt on all inputs. That $M$ accepts the union of $L_1$ and $L_2$ is also clear, since $M$ accepts $x$ if and only if one or both of $M_1, M_2$ accept $x$. □

When seeking to establish a closure property for recognizable languages, we have to guard against the fact that the recognizers may not terminate on objects outside of their language.

**Example 49.** The recognizable languages are closed under union. Formally, this is expressed as

$$\textsf{Recognizable}\ L_1 \wedge \textsf{Recognizable}\ L_2 \Rightarrow \textsf{Recognizable}\ (L_1 \cup L_2)$$

Suppose $L_1$ and $L_2$ are recognizable. Then there exist a recognizer $M_1$ for $L_1$ and a recognizer $M_2$ for $L_2$. Now we claim there is a recognizer $M$ for $L_1 \cup L_2$.

*Proof.* The following machine seems natural, but doesn't solve the problem.

> "On input $w$, invoke $M_1$ on $w$. If it accepts, then accept. Otherwise, invoke $M_2$ on $w$. If it accepts, $M$ accepts, otherwise it rejects."

The problem with this purported solution is that $M_1$ may loop on $w$ while $M_2$ would accept it. In that case, $M$ won't accept $w$ when it should, because $M_2$ never gets a chance to run on $w$. So we have to use dovetailing again. A suitable solution works as follows:

> "On input $w$, invoke $M_1$ and $M_2$ on $w$ in the following fashion: execute $M_1$ for one step on $w$, then execute $M_2$ for one step on $w$. Repeat in this step-wise manner until either $M_1$ or $M_2$ halts. (It may be the case that neither halts, of course, since both are recognizers.) If the halting machine is in the accept state, then accept. Otherwise, run the other machine until it halts. If it accepts, then accept. If it rejects, then reject. Otherwise, the second machine is in a loop, so $M$ loops."

This machine accepts $w$ whenever $M_1$ or $M_2$ will, and loops otherwise, hence is a recognizer for $L_1 \cup L_2$. □

## 3.5 Undecidability

Many problems are not decidable. This is easy to see by a cardinality argument: there are simply far more languages (uncountable) than there are algorithms (countable). So some languages—almost all in fact—have no decider (or recognizer) However, that is an abstract argument; what we want to know is whether any specific problem is decidable or recognizable.

To show that a problem can not be solved requires extreme cleverness, since it is required to show that no solution could ever be found! It is sometimes said that one can't prove a negative, but that is not true in mathematics.

In this section we will show the undecidability of the *Halting Problem*, a result due to Alan Turing. The importance of this theorem is twofold: first, it is the first and arguably most fundamental result about the limits of computation (namely, some problems can not be algorithmically solved); second, many other undecidability results stem from it. The proof uses a cool technique, which we pause to introduce here.

### 3.5.1 Diagonalization

The *diagonalization* proof technique works as follows: assume that you have a complete listing of objects, and then construct a new object that should be in the list but can't be since it differs, in at least one place, with every object in the listing. A contradiction thereby ensues. This technique was invented by Georg Cantor to show that $\mathbb{R}$ is not countable, *i.e.*, that there are far more real numbers than natural numbers. This shows that there is more than one size of infinity.

The existence of a bijection between two sets is used to implement the notion of the sets 'being the same size', or *equinumerous*. When the sets are finite, we just count their elements and compare the resulting numbers. However, equinumerosity is unusual when the sets are of infinite size. For example, the set of even numbers is equinumerous with $\mathbb{N}$ even though it is a proper subset of $\mathbb{N}$!

**Theorem 2.** *$\mathbb{R}$ can not be put into one-to-one correspondence with $\mathbb{N}$. More formally, there is no bijection between $\mathbb{R}$ and $\mathbb{N}$.*

*Proof.* We are in fact going to show that the set of real numbers between $0$ and $1$ are not countable, which implies that $\mathbb{R}$ is not countable. This we show by illustrating that there is no surjection from $\mathbb{N}$ to $\{r \in \mathbb{R} \mid 0 \leq r < 1\}$, *i.e.*, for every mapping from $\mathbb{N}$ to $\mathbb{R}$, some real numbers will be left out.

Towards a contradiction, suppose that there is such a surjection, *i.e.*, the real numbers between $0$ and $1$ can be arranged in a complete listing indexed by natural numbers. This gives us a table, infinite in both directions. Each row of the table represents one real number, and all real numbers are in the table.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0.5 | 3 | 1 | 1 | 7 | 8 | 2 | ⋯ |
| 0.4 | 3 | 0 | 0 | 1 | 2 | 9 | ⋯ |
| 0.7 | 7 | 6 | 5 | 1 | 0 | 2 | ⋯ |
| 0.0 | 1 | 0 | 0 | 0 | 0 | 0 | ⋯ |
| 0.9 | 0 | 3 | 2 | 6 | 8 | 4 | ⋯ |
| 0.0 | 0 | 0 | 1 | 1 | 1 | 0 | ⋯ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋱ |

Note that the arrangement of the numbers in the table doesn't matter; what is important is that the listing is complete and that each row is indexed by a natural number. Now we build an infinite sequence of digits $D$ by traversing the diagonal in the listing and *changing* each digit of the diagonal. For example, we could build

$$D = 0.647172\ldots$$

(There are of course many other possible choices of $D$, such as $425858\ldots$: what is important is that $D$ differs from the diagonal at each digit.) Because $D$ is an infinite string of digits, it must lie in the table. On the other hand, we have intentionally constructed $D$ so that it differs with the first number in the table at the first place, with the second number at the second place, with the third number at the third place, *etc*. Thus $D$ cannot be the first number, the second number, the third number, *etc*. So $D$ is not in the table. Contradiction. Conclusion: $\mathbb{R}$ is too big to be put in one-to-one correspondence with $\mathbb{N}$. □

The reason why this result is surprising is that the rationals $\mathbb{Q}$ (fractions with numerator and denominator in $\mathbb{N}$) do have a one-to-one correspondence with $\mathbb{N}$, even though between every two elements of $\mathbb{N}$ there are an infinite number of elements of $\mathbb{Q}$!

**Note.** Pedants may enjoy pointing out that there is a difficulty with infinitely repeatedly digits since, *e.g.*, $0.19999\ldots = 0.2000\ldots$. If $D$ ended with an infinite repetition of 0s or 9s the argument wouldn't work (because, *e.g.*, $0.199999\ldots$ differs from $0.2000\ldots$ at each digit, but they are equal numbers). We therefore exclude $0$ and $9$ from being used in the construction of $D$.

**Note.**

Diagonalization can be used to show that there is no surjection from a set to its power set; hence there is no bijection, hence the sets are not equinumerous. The proof is almost identical to the one we've just seen.

### 3.5.2 Existence of Undecidable Problems

At least one problem is undecidable. As we will see later, this result can be used as a fulcrum with which to show that many other problems are undecidable.

**Theorem 3** (Halting Problem is undecidable)**.**
*Let the alphabet be* $\{0, 1\}$. *The language*

$$HP = \{\langle M, w\rangle \mid M \text{ is a Turing machine and } M \text{ halts when run on } w\}$$

*is not decidable.*

*Proof.* First we consider a two-dimensional table, infinite in both directions. Down one side we list all possible machines, along the other axis we list all inputs that a machine could take.

|  | $\varepsilon$ | $0$ | $1$ | $00$ | $01$ | $10$ | $11$ | $000$ | $\cdots$ |
|---|---|---|---|---|---|---|---|---|---|
| $M_\varepsilon$ |  |  |  |  |  |  |  |  |  |
| $M_0$ |  |  |  |  |  |  |  |  |  |
| $M_1$ |  |  |  |  |  |  |  |  |  |
| $M_{00}$ |  |  |  |  |  |  |  |  |  |
| $M_{01}$ |  |  |  |  |  |  |  |  |  |
| $M_{10}$ |  |  |  |  |  |  |  |  |  |
| $M_{11}$ |  |  |  |  |  |  |  |  |  |
| $M_{000}$ |  |  |  |  |  |  |  |  |  |
| $\vdots$ |  |  |  |  |  |  |  |  |  |

There is a tight connection between binary string $w$ and machine $M_w$. If $w$ is a valid encoding of a machine, we use $M_w$ to stand for that machine. If $w$ is not a valid encoding of a machine, it will denote a *dummy* machine that simply halts in the accept state as soon as it starts to run. Thus each $M_w$ is a valid Turing machine, and every TM may be found at least once in the list. This bears repeating: this list of TMs is *complete*, every possible TM is in it.

Thus an entry of the table indexed by $(i, j)$ represents machine $M_i$ with input being the binary string corresponding to $j$.

Now let's get started on the argument. Towards a contradiction, we suppose that there is a decider $H$ for the halting problem. Thus $H$ is a TM having the property that, when given input $\langle w, u\rangle$, for any $w$ and $u$, it correctly determines if $M_w$ halts when given input $u$. Being a decider, $H$ must itself always finish its computation in a finite time.

Therefore, we could use $H$ to fill in each cell in the table with T (signifying that the program halted on the input) or F (the program goes into an infinite loop when given the particular input). Notice that $H$ can not simply blindly execute $M_w$ on $u$ and return T when $M_w$ on $u$ terminates: what if $M_w$ on $u$ didn't terminate?

Now consider the following TM $N$ which calls $H$ as a sub-routine.

```
$N =$ "On input $x$, perform the following steps :
          1. Write $\langle x, x\rangle$ to the input tape
          2. Invoke $H$
          3. If $H\langle x, x\rangle = \mathsf{T}$ then loop() else halt with accept."
```

So $N$ goes into a infinite loop on input $x$ just when $M_x$ halts on $x$. Perverse!

What $N$ does is go down the diagonal of the table, reversing the verdict of $H$ at each point. This finally leads us to our contradiction:

- Given that we have assumed that $H$ is a TM, $N$ is a valid TM, therefore is in the (complete) listing of machines.

- $N$ behaves differently from every machine in the list, for at least one argument (it loops on $x$ iff $M_x$ halts on $x$). So $N$ can't possibly be on the list.

The resolution of this contradiction is to conclude that the assumption that $H$ exists is false. There is no halting detector.

**Alternate proof** A slightly different alternative proof—one which makes use of self-reference—proceeds as follows: we construct $N$ as before, but then ask the question: how does $N$ behave when applied to itself, *i.e.*, what is the value of $N(N)$? By instantiating the definition of $N$, we have

$$N(N) = \texttt{if } H(N,N) = \mathsf{T} \texttt{ then } \mathsf{loop()} \texttt{ else } \mathrm{accept}$$

Now we do a case analysis on whether $N$ halts on $N$: if it does, then $N(N)$ goes into an infinite loop; contrarily, if $N$ loops on $N$, then $N(N)$ halts. Conclusion: $N$ halts on $N$ iff $N$ does not halt on $N$. Contradiction.

□

This result lets us conclude that there can not be a procedure that will tell if arbitrary programs halt on all possible inputs. In other words, the halting problem is *algorithmically unsolvable*. (Note the use of the CT-Thesis here: we have moved from a result about Turing machines to a claim about all algorithms.)

To be clear: certain syntactically recognizable classes of programs, *e.g.*, those in which the only looping construct is bounded `for`-loops, always halt. However, it's impossible to write a halting detector that will work correctly for all programs.

### 3.5.3 Other undecidable problems

All the undecidable problems mentioned above are shown to be undecidable by (essentially) showing how the halting problem can be 'coded up' in them. For example, checking whether $M$ accepts $\varepsilon$. That this is undecidable is seen by the fact that $M$ could write out a machine description

$N$ and an input string $x$ and then act as a universal TM, simulating $N$ on $x$, accepting $\varepsilon$ if $N$ does halt on $x$. But the halting problem is undecidable, hence so is this problem. Similar constructions are used to show that the other problems are undecidable.

The following example shows that some languages are not even recognizable!

**Example 50.** The recognizable languages are not closed under complement.

Consider the complement of the halting problem ($\overline{HP}$). If this problem was recursively enumerable, then we could get a decision procedure for the halting problem. How? Since HP is recursively enumerable and (by assumption) $\overline{HP}$ is recursively enumerable, a decision procedure can be built that works as follows: on input $\langle M, w\rangle$, incrementally execute both recognizers for the two languages. Since $HP \cup \overline{HP} = \Sigma^*$, one of the recognizers will eventually accept $\langle M, w\rangle$. So in finite time, the halting (or not) of $M$ on $w$ will be detected, for any $M$ and $w$. But this can't be because we have already shown that *HP* is undecidable. Thus $\overline{HP}$ can't be recursively enumerable and thus must be a member of a class of languages properly larger than the recursively enumerable languages.

Thus the set of all halting programs is recognizable, but the set of all programs that do not halt can't be recognizable for otherwise the set of halting programs would be decidable, and we already know that such is not the case.

# Chapter 4

# Context-Free Grammars

Context Free Grammars (CFGs) first arose in the late 1950s as part of Noam Chomsky's work on the formal analysis of natural language. CFGs can capture some of the syntax of natural languages, such as English, and also of computer programming languages. Thus CFGs are of major importance in Artifical Intelligence and the study of compilers.

Compilers use both automata and CFGs. Usually the lexical structure of a programming language is given by a collection of regular expressions which define the identifiers, keywords, literals, and comments of the language. These regular expressions can be translated into an automaton, usually called the *lexer*, which recognizes the basic lexical elements (*lexemes*) of programs. A *parser* for the programming language will take a stream of lexemes coming from a lexer and build a *parse tree* (also known as an *abstract syntax tree* or AST) by using a CFG. Thus parsing takes the linear string of symbols given by a program text and produces a tree structure which is more suitable for later phases of compilation such as semantic analysis, optimization, and code generation. This is illustrated in Figure 4.1 This is a naive picture; many compilers use more than one kind of abstract syntax tree in their work. The main point is that tree structures are far easier to work with than linear strings.

**Example 51.** A fragment of English can be captured with the following grammar which is presented in a style very much like *Backus-Naur Form* (BNF) in which grammar *variables* are in upper case and enclosed by $\langle - \rangle$, while terminals, or literals, are in lower case.

$$\begin{array}{lcl}
\langle\text{SENTENCE}\rangle & \longrightarrow & \langle\text{NP}\rangle\ \langle\text{VP}\rangle \\
\langle\text{NP}\rangle & \longrightarrow & \langle\text{CNOUN}\rangle \mid \langle\text{CNOUN}\rangle\ \langle\text{PP}\rangle \\
\langle\text{VP}\rangle & \longrightarrow & \langle\text{CVERB}\rangle \mid \langle\text{CVERB}\rangle\ \langle\text{PP}\rangle \\
\langle\text{PP}\rangle & \longrightarrow & \langle\text{PREP}\rangle\ \langle\text{CNOUN}\rangle \\
\langle\text{CNOUN}\rangle & \longrightarrow & \langle\text{ARTICLE}\rangle\ \langle\text{NOUN}\rangle \\
\langle\text{CVERB}\rangle & \longrightarrow & \langle\text{VERB}\rangle \mid \langle\text{VERB}\rangle\ \langle\text{NP}\rangle \\
\langle\text{ARTICLE}\rangle & \longrightarrow & \text{a} \mid \text{the} \\
\langle\text{NOUN}\rangle & \longrightarrow & \text{boy} \mid \text{girl} \mid \text{flower} \\
\langle\text{VERB}\rangle & \longrightarrow & \text{touches} \mid \text{likes} \mid \text{sees} \\
\langle\text{PREP}\rangle & \longrightarrow & \text{with}
\end{array}$$

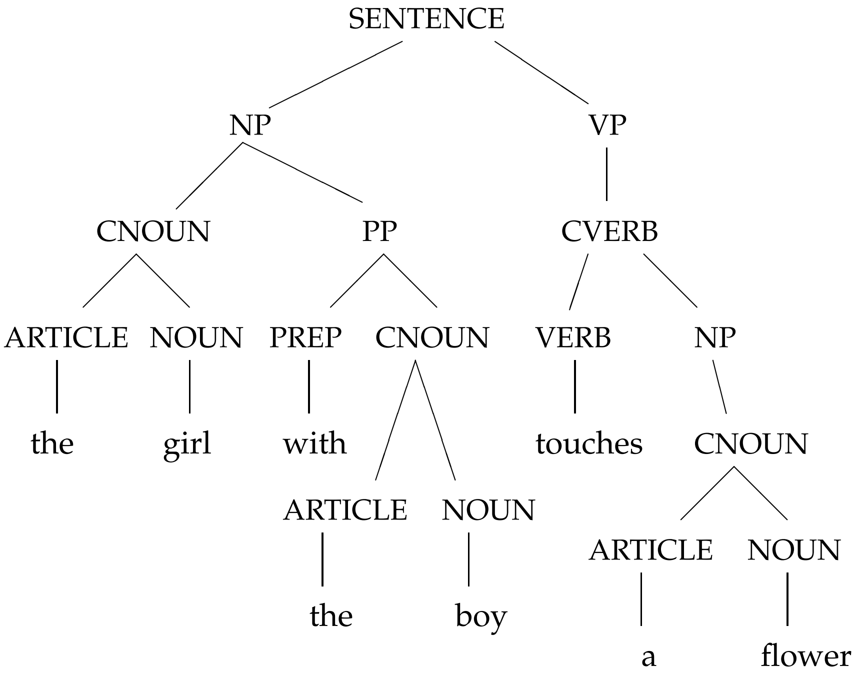
A sentence that can be generated from the grammar is

$$\underbrace{\textit{the girl with the boy}}_{\text{NP}}\underbrace{\textit{touches a flower}}_{\text{VP}}$$

This can be pictured with a so-called *parse tree*, which summarizes the ways in which the sentence may be produced.

$$\begin{array}{lcl}
\text{program text} \xrightarrow{lexing} \text{lexeme stream} \xrightarrow{parsing} \text{AST} & \xrightarrow{semantic\ analysis} & \text{AST} \\
 & \xrightarrow{optimization} & \text{AST} \\
 & \xrightarrow{code\ generation} & \text{executable}
\end{array}$$

Figure 4.1: Stages in compilation



Reading the leaves of the parse tree from left to right yields the original string. The parse tree represents the possible derivations of the sentence.

**Definition 10** (Context-free grammar)**.** A context-free grammar is a 4-tuple $(V, \Sigma, R, S)$, where

- $V$ is a finite set of variables
- $\Sigma$ is a finite set of terminals
- $R$ is a finite set of rules, each of which has the form

  $$A \longrightarrow w$$

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

**Note.** $V \cap \Sigma = \emptyset$. This helps us keep our sanity, because variables and terminals can't be confused. In general, our convention will be that variables are upper-case while terminals are in lower case.

A CFG is a device for generating strings. The way a string is generated is by starting with the start variable $S$ and performing *replacements for variables*, according to the rules.

A *sentential form* is a string in $(V \cup \Sigma)^*$. A *sentence* is a string in $\Sigma^*$. Thus every sentence is a sentential form, but in general a sentential form might not be a sentence, in particular when it has variables occurring in it.

**Example 52.** If $\Sigma = \{0, 1\}$ and $V = \{U, W\}$, then $00101$ is a sentence and therefore a sentential form. On the other hand, $WW$ and $W01U$ are sentential forms that are not sentences.

To rephrase our earlier point: a CFG is a device for generating, ultimately, sentences. However, at intermediate points, the generation process will produce sentential forms.

**Definition 11** (One step replacement)**.** Let $u, v, w \in (V \cup \Sigma)^*$. Let $A \in V$. We write

$$u\underline{A}v \overset{G}{\Rightarrow} u\underline{w}v$$

to stand for the replacement of variable $A$ by $w$ at the underlined location. This replacement is only allowed if there is a rule $A \longrightarrow w$ in $R$. When it is clear which grammar is being referred to, the $G$ in $\overset{G}{\Rightarrow}$ will be omitted.

Thus we can replace *any* variable $A$ in a sentential form by its 'right hand side' $w$. Note that there may be more than one occurrence of $A$ in the sentential form; in that case, only one occurrence may be replaced in a step. Also, there may be more than one variable possible to replace in the sentential form. In that case, it is arbitrary which variable gets replaced.

**Example 53.** Suppose that we have the grammar $(V, \Sigma, R, S)$ where $V = \{S, U\}$ and $\Sigma = \{a, b\}$ and $R$ is given by

$$\begin{array}{rcl} S & \longrightarrow & UaUbS \\ U & \longrightarrow & a \\ U & \longrightarrow & b \end{array}$$

Then we can write $S \Rightarrow UaUbS$. Now consider $UaUbS$. There are 3 locations of variables that could be replaced (two $U$s and one $S$). In one step we can get to the following sentential forms:

- $UaUb\underline{S} \Rightarrow UaUb\underline{UaUbS}$ (Replacing $S$)
- $\underline{U}aUbS \Rightarrow \underline{a}aUbS$ (Applying $U \longrightarrow a$ at the first location of $U$)
- $\underline{U}aUbS \Rightarrow \underline{b}aUbS$ (Applying $U \longrightarrow b$ at the first location of $U$)
- $Ua\underline{U}bS \Rightarrow Ua\underline{a}bS$ (Applying $U \longrightarrow a$ at the second location of $U$)
- $Ua\underline{U}bS \Rightarrow Ua\underline{b}bS$ (Applying $U \longrightarrow b$ at the second location of $U$)

**Definition 12** (Multiple steps of replacement)**.** Let $u, w \in (V \cup \Sigma)^*$. The notation

$$u \overset{*}{\Rightarrow}_G \; w$$

asserts that there exists a finite sequence

$$u \overset{G}{\Rightarrow} u_1 \ldots \overset{G}{\Rightarrow} u_n \overset{G}{\Rightarrow} w$$

of one-step replacements, using the rules in $G$, leading from $u$ to $w$.

This definition is a stepping stone to a more important one:

**Definition 13** (Derivation)**.** Suppose we are given a grammar $G$ with start variable $S$. If $S \overset{*}{\Rightarrow}_G \; w$ and $w \in \Sigma^*$, then we say $G$ *generates* $w$. Similarly,

$$S \overset{G}{\Rightarrow} u_1 \ldots \overset{G}{\Rightarrow} u_n \overset{G}{\Rightarrow} w$$

is said to be a *derivation* of $w$.

Now we can define the set of strings derivable from a grammar, *i.e.*, the language of the grammar: it is the set of sentences, *i.e.*, strings lacking variables, generated by $G$.

**Definition 14** (Language of a grammar)**.** The language $\mathcal{L}(G)$ of a grammar $G$ is defined by

$$\mathcal{L}(G) = \{x \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G \; x\}$$

**Definition 15** (Context-free language)**.** $L$ is a context-free language if there is a CFG $G$ such that $\mathcal{L}(G) = L$.

One question that is often asked is *Why Context-Free?*; in other words, what aspect of CFGs is 'free of context' (whatever that means)? The answer comes from examining the allowed structure of a rule. A rule in a context-free grammar may only have the form $V \longrightarrow w$. When making a replacement for $V$ in a derivation, the symbols surrounding $V$ in the sentential form do not affect whether the replacement can take place or not. Hence context-free. In contrast, there is a class of grammars called *context-sensitive grammars*, in which the left hand side of a rule can be an arbitrary sentential form; such a rule could look like $abVc \longrightarrow abwc$, and a replacement for $V$ would only be allowed in a sentential form where $V$ occurred in the 'context' $abVc$. Context-sensitive grammars, and phrase-structure grammars are more powerful formalisms than CFGs, and we won't be discussing them in the course.

**Example 54.** Let $G$ be given by the following grammar:

$$(\underbrace{\{S\}}_{Variables}, \underbrace{\{0,1\}}_{\Sigma}, \underbrace{\{S \longrightarrow \varepsilon, S \longrightarrow 0S1\}}_{Rules}, \underbrace{S}_{Start\ variable})$$

The following are some derivations using $G$:

- $S \Rightarrow \varepsilon$
- $S \Rightarrow 0S1 \Rightarrow 0\varepsilon 1 \Rightarrow 01$
- $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 00\varepsilon 11 \Rightarrow 0011$

We "see" that $\mathcal{L}(G) = \{0^n1^n \mid n \geq 0\}$. A rigorous proof of this would require proving the statement

$$\forall w \in \Sigma^*.\ w \in \mathcal{L}(G) \text{ iff } \exists n.\ w = 0^n1^n$$

and the proof would proceed by induction on the length of the derivation.

**Example 55.** Give a CFG for the language $L = \{0^n1^{2n} \mid n \geq 0\}$.

The answer to this can be obtained by a simple adaptation of the grammar in the previous example:

$$\begin{array}{lll} S & \longrightarrow & \epsilon \\ S & \longrightarrow & 0S11 \end{array}$$

*Convention.* We will usually be satisfied to give a CFG by giving its rules. Usually, the start state will be named $S$, and the variables will be written in upper case, while members of $\Sigma$ will be written in lower case. Furthermore, multiple rules with the same left-hand side will be collapsed into a single rule, where the right-hand sides are separated by a $|$. Thus, the previous grammar could be completely and unambiguously given as $S \longrightarrow \varepsilon \mid 0S11$.

**Example 56** (Palindromes)**.** Give a grammar for generating palindromes over $\{0, 1\}^*$. Recall that the palindromes over alphabet $\Sigma$ can be defined as $PAL = \{w \in \Sigma^* \mid w = w^{\mathcal{R}}\}$. Some examples are $101$ and $0110$ for binary strings. For ASCII, there are some famous palindromes:[1]

- *madamImadam*, the first thing said in the Garden of Eden.
- *ablewasIereIsawelba*, attributed to Napoleon.
- *amanaplanacanalpanama*
- *Wonder if Sununu's fired now*

Now, considerable ingenuity is sometimes needed when constructing a grammar for a language. One way to start is to enumerate the first few strings in the language and see if any regularities are apparent. For $PAL$, we know that $\varepsilon \in PAL$, $0 \in PAL$, and $1 \in PAL$. Then we might ask the question *Suppose $w$ is in $PAL$. How can I then make other strings in $PAL$?* In our example, there are two ways:

- $0w0 \in PAL$
- $1w1 \in PAL$

A little thought doesn't reveal any other ways of building elements of $PAL$, so our final grammar is

$$S \longrightarrow \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1.$$

**Example 57.** Give a grammar for generating the language of balanced parentheses. The following strings are in this language: $\varepsilon$, (), (()), ()(), (()(())), *etc.* Well, clearly, we will need to generate $\varepsilon$, so we will have a rule

[1] These and many more can be found at `http://www.palindromes.org`.

$S \longrightarrow \varepsilon$. Now we assume that we have a string $w$ with balanced parentheses, and want to generate a new string in the language from $w$. There are two ways of doing this:

- $(w)$
- $ww$

So the grammar can be given by

$$S \longrightarrow \varepsilon \mid (S) \mid SS.$$

**Example 58.** Give a grammar that generates

$$L = \{x \in \{0,1\}^* \mid \mathsf{count}(x,0) = \mathsf{count}(x,1)\}$$

the set of binary strings with an equal numbers of 0s and 1s.

Now, clearly $\varepsilon \in L$. Were we to proceed as in the previous example, we'd suppose that $w \in L$ and try to see what strings in $L$ we could build from $w$. You might think that $0w1$ and $1w0$ would do it, but not so! Why? Because we need to be able to generate strings like $0110$ where the endpoints are the same. So the attempt

$$S \longrightarrow \varepsilon \mid 0S1 \mid 1S0$$

doesn't work. Also, we couldn't just add $0w0$ and $1w1$ in an effort to repair this shortcoming, because then we could generate strings not in $L$, such as $00$.

We want to think of $S$ as denoting all strings with an equal number of 0s and 1s. The previous attempts have the right idea—take a balanced string $w$ and make another balanced string from it—but only add the 0s and 1s at the outer edges of the string. Instead, we want to add them at internal locations as well. The following grammar supports this:

$$S \longrightarrow \varepsilon \mid S0S1S \mid S1S0S \tag{4.1}$$

Another grammar that works:

$$S \longrightarrow \varepsilon \mid S0S1 \mid S1S0$$

And another (this is perhaps the most elegant):

$$S \longrightarrow \varepsilon \mid 0S1 \mid 1S0 \mid SS$$

Here's a derivation of the string $0^3 1^6 0^3$ using grammar (4.1).

$$\begin{array}{rcl}
S & \Rightarrow & S1\underline{S}0S \\
 & \Rightarrow & S1S1\underline{S}0S0S \\
 & \Rightarrow & S1S1S1S0S0S0S \\
 & \overset{*}{\Rightarrow} & \underline{S}1\varepsilon 1\varepsilon 1\varepsilon 0\varepsilon 0\varepsilon 0\varepsilon \\
 & \Rightarrow & S0\underline{S}1S1\varepsilon 1\varepsilon 1\varepsilon 0\varepsilon 0\varepsilon 0\varepsilon \\
 & \Rightarrow & S0S0\underline{S}1S1S1\varepsilon 1\varepsilon 1\varepsilon 0\varepsilon 0\varepsilon 0\varepsilon \\
 & \Rightarrow & S0S0S0S1S1S1S1\varepsilon 1\varepsilon 1\varepsilon 0\varepsilon 0\varepsilon 0\varepsilon \\
 & \overset{*}{\Rightarrow} & \varepsilon 0\varepsilon 0\varepsilon 0\varepsilon 1\varepsilon 1\varepsilon 1\varepsilon 1\varepsilon 1\varepsilon 1\varepsilon 0\varepsilon 0\varepsilon 0\varepsilon \\
 & = & 000111111000 \\
 & = & 0^3 1^6 0^3.
\end{array}$$

Note that we used $\overset{*}{\Rightarrow}$ to abbreviate multiple steps.

## 4.1 Aspects of grammar design

There are several strategies commonly used to build grammars. The main one we want to focus on now is the use of *closure properties*.

The context-free languages enjoy several important closure properties. Recall that a closure property asserts something of the form

*If $L$ has property $P$ then $f(L)$ also has property $P$.*

The proofs of the closure properties involve constructions. The constructions for decidable and recognizable languages were phrased in terms of machines; the constructions for CFLs are on grammars, although they may be done on machines as well.

**Theorem 4** (Closure properties of CFLs)**.** *If $L, L_1, L_2$ are context-free languages, then so are $L_1 \cup L_2$, $L_1 \cdot L_2$, and $L^*$.*

*Proof.* Let $L, L_1$ and $L_2$ be context-free languages. Then there are grammars $G, G_1, G_2$ such that $\mathcal{L}(G) = L$, $\mathcal{L}(G_1) = L_1$, and $\mathcal{L}(G_2) = L_2$. Let

$$\begin{array}{rcl}
G & = & (V, \Sigma, R, S) \\
G_1 & = & (V_1, \Sigma_1, R_1, S_1) \\
G_2 & = & (V_2, \Sigma_2, R_2, S_2)
\end{array}$$

Assume $V_1 \cap V_2 = \emptyset$. Let $S_0, S_3$ and $S_4$ be variables not occurring in $V \cup V_1 \cup V_2$. These assumptions are intended to avoid confusion when making the constructions.

- $L_1 \cup L_2$ is generated by the grammar $(V_1 \cup V_2 \cup \{S_3\}, \Sigma_1 \cup \Sigma_2, R_3, S_3)$ where
$$R_3 = R_1 \cup R_2 \cup \{S_3 \longrightarrow S_1 \mid S_2\} .$$
In other words, to get a grammar that recognizes the union of $L_1$ and $L_2$, we build a combined grammar and add a new rule saying that a string is in $L_1 \cup L_2$ if it can be derived by either $G_1$ or by $G_2$.

- $L_1 \cdot L_2$ is generated by the grammar $(V_1 \cup V_2 \cup \{S_4\}, \Sigma_1 \cup \Sigma_2, R_4, S_4)$ where
$$R_4 = R_1 \cup R_2 \cup \{S_4 \longrightarrow S_1 S_2\} .$$
In other words, to get a grammar that recognizes the concatenation of $L_1$ and $L_2$, we build a combined grammar and add a new rule saying that a string $w$ is in $L_1 \cdot L_2$ if there is a first part $x$ and a second part $y$ such that $w = xy$ and $G_1$ derives $x$ and $G_2$ derives $y$.

- $L^*$ is generated by the grammar $(V \cup \{S_0\}, \Sigma, R_5, S_0)$ where
$$R_5 = R \cup \{S_0 \longrightarrow S_0 S \mid \varepsilon\} .$$
In other words, to get a grammar that recognizes the Kleene star of $L$, we add a new rule saying that a string is in $L^*$ if it can can be derived by generating some number of strings by $G$ and then concatentating them together. The empty string is explicitly tossed in via the rule $S \longrightarrow \varepsilon$.

□

*Remark.* One might ask: *what about closure under intersection and complement?* It happens that the CFLs are not closed under intersection and we can see this by the following counterexample.

**Example 59.** Let grammar $G_1$ be given by the following rules:

$$\begin{array}{rcl} A & \longrightarrow & PQ \\ P & \longrightarrow & aPb \mid \varepsilon \\ Q & \longrightarrow & cQ \mid \varepsilon \end{array}$$

Then $\mathcal{L}(G_1) = \{a^i b^i c^j \mid i, j \geq 0\}$. Let grammar $G_2$ be given by

$$\begin{array}{lcl} B & \longrightarrow & RT \\ R & \longrightarrow & aR \mid \varepsilon \\ T & \longrightarrow & bTc \mid \varepsilon \end{array}$$

Then $\mathcal{L}(G_2) = \{a^i b^j c^j \mid i, j \geq 0\}$. Thus $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \{a^i b^i c^i \mid i, j > 0\}$. But this is not a context-free language, as we shall see after discussing the pumping lemma for CFLs.

**Example 60.** Construct a CFG for

$$L = \{0^m 1^n \mid m \neq n\} .$$

A solution to this problem comes from realizing that $L = L_1 \cup L_2$, where

$$\begin{array}{l} L_1 = \{0^m 1^n \mid m < n\} \\ L_2 = \{0^m 1^n \mid m > n\} \end{array}$$

$L_1$ can be generated with the CFG given by the following rules $R_1$:

$$S_1 \longrightarrow 1 \mid S_1 1 \mid 0 S_1 1$$

and $L_2$ can be generated by the following rules $R_2$

$$S_2 \longrightarrow 0 \mid 0 S_2 \mid 0 S_2 1 .$$

We obtain $L$ by adding the rule $S \longrightarrow S_1 \mid S_2$ to $R_1 \cup R_2$.

**Example 61.** Give a CFG for $L = \{x\#y \mid x^{\mathcal{R}} \text{ is a substring of } y\}$. Note that $x$ and $y$ are elements of $\{0,1\}^*$ and that # is another symbol, not equal to 0 or 1.
The key point is that the notion *is a substring of* can be expressed by concatentation.

$$\begin{array}{ll} L & = \{x\#y \mid x^{\mathcal{R}} \text{ is a substring of } y\} \\ & = \{x\#ux^{\mathcal{R}}v \mid x, u, v \in \{0,1\}^*\} \\ & = \underbrace{\{x\#ux^{\mathcal{R}} \mid x, u \in \{0,1\}^*\}}_{L_1} \cdot \underbrace{\{v \mid v \in \{0,1\}^*\}}_{L_2} \end{array}$$

A grammar for $L_2$ is then

$$S_2 \longrightarrow \varepsilon \mid 0 S_2 \mid 1 S_2$$

A grammar for $L_1$:

$$S_1 \longrightarrow 0S_10 \mid 1S_11 \mid \#S_2$$

Thus, the final grammar is

$$\begin{array}{l} S \longrightarrow S_1S_2 \\ S_1 \longrightarrow 0S_10 \mid 1S_11 \mid \#S_2 \\ S_2 \longrightarrow \varepsilon \mid 0S_2 \mid 1S_2 \end{array}$$

**Example 62.** Give a CFG for $L = \{0^i1^j \mid i < 2j\}$.

This problem is difficult to solve directly, but one can split it into the union of two easier languages:

$$L = \{0^i1^j \mid i < j\} \cup \{0^i1^j \mid j \leq i < 2j\}$$

The grammar for the first language is

$$\begin{array}{l} S_1 \longrightarrow AB \\ A \longrightarrow 0A1 \mid \varepsilon \\ B \longrightarrow 1B \mid 1 \end{array}$$

The second language can be rephrased as $\{0^{j+k}1^j \mid k < j\}$ and *that* can be rephrased in terms of $k$ (letting $j = k + \ell + 1$, for some $\ell$):

$$\{0^{k+\ell+1+k}1^{k+\ell+1} \mid k, \ell \geq 0\} = \{00^{2k+\ell}1^{k+\ell}1 \mid k, \ell \geq 0\}$$

and from this we have the grammar for the second language

$$\begin{array}{l} S_2 \longrightarrow 0X1 \\ X \longrightarrow 00X1 \mid Y \\ Y \longrightarrow 0Y1 \mid \varepsilon \end{array}$$

Putting it all together gives

$$\begin{array}{l} S \longrightarrow S_1 \mid S_2 \\ \\ S_1 \longrightarrow AB \\ A \longrightarrow 0A1 \mid \varepsilon \\ B \longrightarrow 1B \mid 1 \\ \\ S_2 \longrightarrow 0X1 \\ X \longrightarrow 00X1 \mid A \end{array}$$

**Example 63.** Give a CFG for $L = \{a^ib^jc^k \mid i = j + k\}$.

If we note that

$$\begin{array}{ll} L & = \{a^ja^kb^jc^k \mid j, k \geq 0\} \\ & = \{a^ka^jb^jc^k \mid j, k \geq 0\} \end{array}$$

we quickly get the grammar

$$\begin{array}{ll} S & \longrightarrow aSc \mid A \\ A & \longrightarrow aAb \mid \varepsilon \end{array}$$

**Example 64.** Give a CFG for $L = \{a^ib^jc^k \mid i \neq j + k\}$.

The solution begins by splitting the language into two pieces:

$$\begin{array}{ll} L & = \{a^ib^jc^k \mid i \neq j + k\} \\ & = \underbrace{\{a^ib^jc^k \mid i < j + k\}}_{L_1} \cup \underbrace{\{a^ib^jc^k \mid j + k < i\}}_{L_2} \end{array}$$

In $L_1$, there are more $b$s and $c$s, in total, than $a$s. We again start by attempting to scrub off equal numbers of $a$s and $c$s. At the end of that phase, there may be more $a$s left, in which case the $c$s are gone, or, there may be more $c$s left, in which case the $a$s are gone.

$$\begin{array}{ll} S_1 & \longrightarrow aS_1c \mid A \mid B \\ A & \longrightarrow aAb \mid C \\ B & \longrightarrow bD \mid Dc \end{array}$$

The rule for $A$ scrubs off any remaining $a$s, leaving a non-empty string of $b$s. The rule for $B$ deals with a (non-empty) string $b^ic^j$. Thus we add the rules

$$\begin{array}{ll} C & \longrightarrow b \mid bC \\ D & \longrightarrow EF \\ E & \longrightarrow \varepsilon \mid bE \\ F & \longrightarrow \varepsilon \mid cF \end{array}$$

To obtain a grammar for $L_2$ is easier:

$$\begin{array}{ll} S_2 & \longrightarrow aS_2c \mid B_2 \\ B_2 & \longrightarrow aB_2b \mid C_2 \\ C_2 & \longrightarrow aC_2 \mid a \end{array}$$

And finally we complete the grammar with

$$S \longrightarrow S_1 \mid S_2$$

**Example 65.** Give a CFG for

$$L = \{a^m b^n c^p d^q \mid m + n = p + q\} .$$

This example takes some thought. At its core, the problem is (essentially) a perverse elaboration of the language $\{0^n 1^n \mid n \geq 0\}$ (which is generated by the rules $S \longrightarrow \varepsilon \mid 0S1$). Now, strings in $L$ have the form

$$\begin{array}{|c|c||c|c|}\hline a^m & b^n & c^p & d^q \\ \hline\end{array}$$

where $m + n = p + q$ and the double line marks the midpoint in the string.

We will build the grammar in stages. We first construct a rule that will 'cancel off' $a$ and $d$ symbols from the outside-in :

$$S \longrightarrow aSd$$

In fact, $\mathsf{min}(m, q)$ symbols get cancelled. After this step, there are two cases to consider:

1. $m \leq q$, *i.e.*, all the leading $a$ symbols have been removed, leaving the remaining string $b^n c^p d^i$, where $i = q - m$.
2. $q \leq m$, *i.e.*, all the trailing $d$ symbols have been removed, leaving the remaining string $a^j b^n c^p$, where $j = m - q$.

In the first case, the situation looks like

$$\begin{array}{|c||c|c|}\hline b^n & c^p & d^i \\ \hline\end{array}$$

We now cancel off $b$ and $d$ symbols from the outside-in (if possible—it could be that $i = 0$) using the following rule:

$$A \longrightarrow bAd$$

After this rule finishes, all trailing $d$ symbols have been trimmed and the situation looks like

$$\begin{array}{|c||c|}\hline b^{n-i} & c^p \\ \hline\end{array}$$

Now we can use the rule

$$C \longrightarrow bCc \mid \varepsilon$$

to trim all the matching $b$ and $c$ symbols that remain (there must be an equal number of them). Thus, for this case, we have constructed the grammar

$$\begin{array}{lcl} S & \longrightarrow & aSd \mid A \\ A & \longrightarrow & bAd \mid C \\ C & \longrightarrow & bCc \mid \varepsilon \end{array}$$

The second case, $q \leq m$, is completely similar: the situation looks like

$$\begin{array}{|c||c|c|} \hline a^j & b^n & c^p \\ \hline \end{array}$$

We now cancel off $a$ and $c$ symbols from the outside-in (if possible—it could be that $i = 0$) using the following rule:

$$B \longrightarrow aBc$$

After this rule finishes, all trailing $c$ symbols have been trimmed and the situation looks like

$$\begin{array}{|c||c|} \hline b^n & c^{p-j} \\ \hline \end{array}$$

Now we can re-use the rule

$$C \longrightarrow bCc \mid \varepsilon$$

to trim all the matching $b$ and $c$ symbols that remain. Thus, to handle the case $q \leq m$ we have to add the rule $B \longrightarrow aBc$ to the grammar, resulting in the final grammar

$$\begin{array}{lcl} S & \longrightarrow & aSd \mid A \mid B \\ A & \longrightarrow & bAd \mid C \\ B & \longrightarrow & aBc \mid C \\ C & \longrightarrow & bCc \mid \varepsilon \end{array}$$

Now we examine a few problems about the language generated by a grammar.

**Example 66.** What is the language generated by the grammar given by the following rules?

$$\begin{array}{lcl} S & \longrightarrow & ABA \\ A & \longrightarrow & a \mid bb \\ B & \longrightarrow & bB \mid \varepsilon \end{array}$$

The answer is easy: $(a + bb)b^*(a + bb)$. The reason why it is easy is that an $A$ leads in one step to terminals (either $a$ or $bb$); also, $B$ expands to an arbitrary number of $b$s.

Now for a similar grammar which is harder to understand:

**Example 67.** What is the language generated by the grammar given by the following rules?

$$\begin{array}{rcl} S & \longrightarrow & ABA \\ A & \longrightarrow & a \mid bb \\ B & \longrightarrow & bS \mid \varepsilon \end{array}$$

We see that the grammar is nearly identical to the previous, except for recursion on the start variable: a $B$ can expand to $bS$, which means that another trip through the grammar will be required. Let's generate some sentential forms to get a feel for the language (it will be useful to refrain from substituting for $A$):

$$S \Rightarrow A\underline{B}A \Rightarrow Ab\underline{S}A \Rightarrow AbA\underline{B}AA \Rightarrow AbAb\underline{S}AA \Rightarrow AbAbABAAA \Rightarrow \ldots$$

What's the pattern here? It helps to focus on $B$ alone. Let's expand $B$ out and try to not include $S$ in the sentential forms:

$$\begin{array}{l} B \Rightarrow \varepsilon \\ B \Rightarrow bABA \Rightarrow bA\varepsilon A = bAA \\ B \Rightarrow bABA \Rightarrow bAbABAA \Rightarrow (bA)^2\varepsilon A^2 = (bA)^2A^2 \\ B \Rightarrow bABA \Rightarrow bAbABAA \Rightarrow (bA)^2bABAA^2 \Rightarrow (bA)^3\varepsilon A^3 = (bA)^3A^3 \\ \quad\vdots \end{array}$$

By scrutiny, we can see that $B \overset{*}{\Rightarrow} w$ implies $w = (bA)^nA^n$, for all $n \geq 0$. Since $S \Rightarrow ABA$, we have

$$\mathcal{L}(G) = \underbrace{(a + bb)}_{A}\,\underbrace{(b(a + bb))^n}_{(bA)^n}\,\underbrace{(a + bb)^n}_{A^n}\,\underbrace{(a + bb)}_{A}$$

Simplified a bit, we obtain:

$$\mathcal{L}(G) = \{(a + bb)(ba + b^3)^n(a + bb)^{n+1} \mid n \geq 0\}$$

### 4.1.1 Proving properties of grammars

In order to prove properties of grammars, we typically use induction. Sometimes, this is induction on the length of derivations, and sometimes on the length of strings. It is a matter of experience which kind of induction to do!

**Example 68.** Prove that every string produced by

$$G = S \longrightarrow 0 \mid S0 \mid 0S \mid 1SS \mid SS1 \mid S1S$$

has more 0's than 1's.

*Proof.* Let $P(x)$ mean $\mathsf{count}(1, x) < \mathsf{count}(0, x)$. We wish to show

$$S \overset{*}{\Rightarrow} w \Rightarrow P(w).$$

Consider a derivation of an arbitrary string $w$. Since we don't know anything about $w$, we don't know anything about the length of its derivation. Let's say that the derivation takes $n$ steps. We are going to proceed by induction on $n$. We can assume that $0 < n$, because derivations need to have at least one step. Now let's take for our inductive hypothesis the following statement: $P(x)$ *holds for any string* $x$ *derived in fewer than* $n$ *steps*. The first step in the derivation must be an application of one of the six rules in the grammar:

$S \longrightarrow 0$. Then the length of the derivation is 1, so $w = 0$, so $P(w)$ holds.

$S \longrightarrow S0$. In this case, there must be a derivation $S \overset{n-1}{\Rightarrow} u$ such that $w = u0$. By the IH, $P(u)$ holds, so $P(w)$ holds, since we've added another 0.

$S \longrightarrow 0S$. Similar to previous.

$S \longrightarrow 1SS$. In this case, there must be derivations $S \overset{k}{\Rightarrow} u$ and $S \overset{\ell}{\Rightarrow} v$ such that $w = 1uv$. Now $k < n$ and $\ell < n$ so, by the IH, $P(u)$ and $P(v)$ both hold, so $P(w)$ holds, since

$$\begin{array}{rcl} \mathsf{count}(1, 1uv) & = & 1 + \mathsf{count}(1, u) + \mathsf{count}(1, v) \\ & < & \mathsf{count}(0, u) + \mathsf{count}(0, v). \end{array}$$

$S \longrightarrow SS1$. Similar to previous.

$S \longrightarrow S1S$. Similar to previous.

□

**Note.** We are using a version of induction called *strong induction*; when trying to prove $P$ holds for derivations of length $n$, strong induction allows us to assume $P$ holds for all derivations of length $m$, provided $m < n$.

## 4.2 Ambiguity

It is well known that natural languages such as English allow ambiguous sentences: ones that can be understood in more than one way. At times ambiguity arises from differences in the semantics of words, *e.g.*, a word may have more than one meaning. One favourite example is the word *livid*, which can mean 'ashen' or 'pallid' but could also mean 'black-and-blue'. So when one is *livid with rage*, is their face white or purple?

Ambiguity of a different sort is found in the following sentences: compare *Fruit flies like a banana* with *Time flies like an arrow*. The structure of the parse trees for the two sentences are completely different. In natural languages, ambiguity is a good thing, allowing much richness of expression, including puns. On the other hand, ambiguity is a *terrible* thing for computer languages. If a grammar for a programming language allowed some inputs to be parsed in two different ways, then different compilers could compile a source file differently, which leads to much unhappiness.

In order to deal with ambiguity formally, we have to make a few definitions. To assert that a grammar is ambiguous, we really mean to say that some string has more than one parse tree. But we want to avoid formalizing what parse trees are. Instead, we'd like to formalize the notion in terms of derivations. However, we can't simply say that a grammar is ambiguous if there is some string having more than one derivation. That doesn't work, since there can be many 'essentially similar' derivations of a string. (In fact, this is exactly what a parse tree captures.) The following notion forces some amount of determinism on all derivations of a string.

**Definition 16** (Leftmost derivation)**.** A *leftmost* derivation is one in which, at each step, the leftmost variable in the sentential form is replaced.

But that can't take care of it all. The choice of variable to be replaced in a leftmost derivation might be fixed, but there could be multiple right hand sides for that variable. This is what leads to different parse trees.

**Definition 17** (Ambiguity)**.** A grammar $G$ is *ambiguous* if there is a string $w \in \mathcal{L}(G)$ that has more than one *leftmost* derivation.

Now let's look at an ambiguous grammar for arithmetical expressions.

**Example 69.** Let $G$ be

$$\begin{array}{rcl} E & \longrightarrow & E+E \\ & | & E-E \\ & | & E*E \\ & | & E/E \\ & | & -E \\ & | & C \\ & | & V \\ & | & (E) \\ C & \longrightarrow & 0 \mid 1 \\ V & \longrightarrow & x \mid y \mid z \end{array}$$

That $G$ is ambiguous is easy to see: consider the expression $x + y * z$. By expanding the '$+$' rule first, we have a derivation that starts $E \Rightarrow E+E \Rightarrow \cdots$ and the expression would be parsed as $x + (y * z)$. By expanding the '$*$' rule first, we have a derivation that starts $E \Rightarrow E * E \Rightarrow \cdots$ and the expression would be parsed as $(x + y) * z$.

Now some hard facts about ambiguity:

- Some CFLs can only be generated by ambiguous grammars. These are called *inherently ambiguous* languages.

  **Example 70.** The language $\{a^i b^j c^k \mid (i = j) \vee (j = k)\}$ is inherently ambiguous.

- The decision problem of checking whether a grammar is ambiguous or not is not solvable.

However, let's not be depressed by this. There's a common technique that often allows us to change an ambiguous grammar into an equivalent unambiguous grammar, based on *precedence*. In the standard way of reading arithmetic expressions, $*$ binds more tightly than $+$, so we would tend to read—courtesy of the indoctrination we received in grade school—$x+y*z$ as standing for $x+(y*z)$. Happily, we can transform our grammar to reflect this, and get rid of ambiguity. In setting up precedences, we will use a notion of *level*. All operators at the same level have the same binding strength relative to operators at other levels, but will have 'internal' precedences among themselves as well. Thus, both $+$ and $-$ bind less tightly than $*$, but also $-$ binds tighter than $+$. We can summarize this for arithmetic operations as follows:

$$\begin{array}{ll} \{-,+\} & \text{bind less tightly than} \\ \{/,*\} & \text{bind less tightly than} \\ \{-(\text{unary negation})\} & \text{bind less tightly than} \\ \{V, C, (-)\} & \end{array}$$

From this classification we can write the grammar directly out. We have to split $E$ into new variables which reflect the precedence levels.

$$\begin{array}{ll} E \longrightarrow & E+T \mid E-T \mid T \\ T \longrightarrow & T*U \mid T/U \mid U \\ U \longrightarrow & -U \mid F \\ F \longrightarrow & C \mid V \mid (E) \\ C \longrightarrow & 0 \mid 1 \\ V \longrightarrow & x \mid y \mid z \end{array}$$

Now if we want to generate a leftmost derivation, there are no choices. Let's try it on the input $x - y * z + x$:

$$\begin{aligned} E &\Rightarrow E+T \Rightarrow (T-T)+T \Rightarrow (U-T)+T \\ &\Rightarrow (F-T)+T \Rightarrow (V-T)+T \Rightarrow (x-T)+T \\ &\Rightarrow (x-(T*U))+T \Rightarrow (x-(U*U))+T \Rightarrow (x-(F*U))+T \\ &\Rightarrow (x-(y*U))+T \overset{*}{\Rightarrow} (x-(y*z))+T \\ &\overset{*}{\Rightarrow} (x-(y*z))+x \end{aligned}$$

**Note**. We are forced to expand the rule $E \longrightarrow E+T$ first; otherwise, had we started with $E \longrightarrow E-T$ then we would have to generate $y*z+x$ from $T$. This can only be accomplished by expanding $T$ to $T*U$, but then there's no way to derive $z+x$ from $U$.
**Note**. This becomes more complex when right and left associativity are to be supported.

**Example 71** (Dangling **else**). Suppose we wish to support not only

`if` *test* `then` *action* `else` *action*

statements in a programming language, but also *one-armed* `if` statements of the form `if` *test* `then` *action*. This leads to a form of ambiguity known as the *dangling else*. A skeletal grammar including both forms is

$$\begin{array}{rcl} S & \longrightarrow & \texttt{if}\ B\ \texttt{then}\ A \mid A \\ A & \longrightarrow & \texttt{if}\ B\ \texttt{then}\ A\ \texttt{else}\ S \mid C \\ B & \longrightarrow & b_1 \mid b_2 \mid b_3 \\ C & \longrightarrow & a_1 \mid a_2 \mid a_3 \end{array}$$

Then the sentence

$$\texttt{if}\ b_1\ \texttt{then}\ \texttt{if}\ b_2\ \texttt{then}\ a_1\ \texttt{else}\ \texttt{if}\ b_3\ \texttt{then}\ a_2\ \texttt{else}\ a_3$$

can be parsed as

$$\texttt{if}\ b_1\ \texttt{then}\ (\texttt{if}\ b_2\ \texttt{then}\ a_1)\ \texttt{else}\ \texttt{if}\ b_3\ \texttt{then}\ a_2\ \texttt{else}\ a_3$$

or as

$$\texttt{if}\ b_1\ \texttt{then}\ (\texttt{if}\ b_2\ \texttt{then}\ a_1\ \texttt{else}\ \texttt{if}\ b_3\ \texttt{then}\ a_2)\ \texttt{else}\ a_3$$

How can this be repaired?

## 4.3 Algorithms on CFGs

We now consider a few algorithms that can be applied to context free grammars. In some cases, these are intended to compute various interesting properties of the grammars, or of variables in the grammars, and in some cases, they can be used to simplify a grammar into a form more suitable for machine processing.

**Definition 18** (Live variables)**.** A variable $A$ in a context-free grammar $G = (V, \Sigma, R, S)$ is said to be *live* if $A \overset{*}{\Rightarrow} x$ for some $x \in \Sigma^*$.

To compute the live variables, we proceed in a bottom-up fashion. Rules are processed right to left. Thus, we begin by marking every variable $V$ where there is a rule $V \longrightarrow rhs$ such that $rhs \in \Sigma^*$. Then we repeatedly do the following: mark variable $U$ when there is a rule $U \longrightarrow rhs$ such that every variable in $rhs$ is marked. This continues until no unmarked variable gets marked. The live variables are those that are marked.

**Definition 19** (Reachable variables)**.** A variable $A$ in a context-free grammar $G = (V, \Sigma, R, S)$ is said to be *reachable* if $S \overset{*}{\Rightarrow} \alpha A \beta$ for some $\alpha, \beta$ in $(\Sigma \cup V)^*$.

The previous algorithm propagates markings from right to left in rules. To compute the reachable variables, we do the opposite: processing proceeds top down and from left to right. Thus, we begin by marking the start variable. Then we look at the rhs of every production of the form $S \longrightarrow \mathit{rhs}$ and mark every unmarked variable in $\mathit{rhs}$. We continue in this way until no unmarked variables become marked. The reachable variables are those that are marked.

**Definition 20** (Useful variables)**.** A variable $A$ in a context-free grammar $G = (V, \Sigma, R, S)$ is said to be *useful* if for some string $x \in \Sigma^*$ there is a derivation of $x$ that takes the form $S \overset{*}{\Rightarrow} \alpha A \beta \overset{*}{\Rightarrow} x$. A variable that is not useful is said to be *useless*. If a variable is not live or is not reachable then it is clearly useless.

**Example 72.** Find a grammar having no useless variables which is equivalent to the following grammar

$$\begin{array}{l} S \longrightarrow ABC \mid BaB \\ A \longrightarrow aA \mid BaC \mid aaa \\ B \longrightarrow bBb \mid a \\ C \longrightarrow CA \mid AC \end{array}$$

The reachable variables of this grammar are $\{S, A, B, C\}$ and the live variables are $\{A, B, S\}$. Since $C$ is not live, $\mathcal{L}(C) = \emptyset$, hence $\mathcal{L}(ABC) = \emptyset$ and also $\mathcal{L}(BaC) = \emptyset$, so we can delete the rules $S \longrightarrow ABC$ and $A \longrightarrow BaC$ to obtain the new, equivalent, grammar

$$\begin{array}{l} S \longrightarrow BaB \\ A \longrightarrow aA \mid aaa \\ B \longrightarrow bBb \mid a \end{array}$$

In this grammar, $A$ is not reachable, so any rules with $A$ on the lhs can be dropped. This leaves

$$\begin{array}{l} S \longrightarrow BaB \\ B \longrightarrow bBb \mid a \end{array}$$

### 4.3.1 Chomsky Normal Form

It is sometimes helpful to eliminate various forms of redundancy in a grammar. For example, a grammar with a rule such as $V \longrightarrow \varepsilon$ occurring in it might be thought to be simpler if all occurrences of $V$ in the right hand side of a rule were eliminated. Similarly, a rule such as $P \longrightarrow Q$ is an indirection that can seemingly be eliminated. A grammar in *Chomsky Normal Form*[2] is one in which these redundancies do not occur. However, the simplification steps are somewhat technical, so we will have to take some care in their application.

**Definition 21** (Chomsky Normal Form)**.** A grammar is in Chomsky Normal Form if every rule has one of the following forms:

- $A \longrightarrow BC$
- $A \longrightarrow a$

where $A, B, C$ are variables, and $a$ is a terminal. Furthermore, in all rules $A \longrightarrow BC$, we require that neither $B$ nor $C$ are the start variable for the grammar. Notice that the above restrictions do not allow a rule of the form $A \longrightarrow \varepsilon$; however, this will disallow some grammars. Therefore, we allow the rule $S \longrightarrow \varepsilon$, where $S$ is the start variable.

The following algorithm translates grammar $G = (V, \Sigma, R, S)$ to Chomsky Normal Form:

1. Create a new start variable $S_0$ and add the rule $S_0 \longrightarrow S$. Now the start variable is not on the right hand side of any rule.

2. Eliminate all rules of the form $A \longrightarrow \varepsilon$. For each rule of the form $V \longrightarrow uAw$, where $u, w \in (V \cup \Sigma)^*$, we add the rule $V \longrightarrow uw$. It is important to notice that we must do this for every occurrence of $A$ in the right hand side of the rule. Thus the rule

   $$V \longrightarrow uAwAv$$

   yields the new rules

   $$\begin{array}{lcl} V & \longrightarrow & uwAv \\ V & \longrightarrow & uAwv \\ V & \longrightarrow & uwv \end{array}$$

   If we had the rule $V \longrightarrow A$, we add $V \longrightarrow \varepsilon$. This will get eliminated in later steps.

[2] In theoretical computer science, a *normal form* of an expression $x$ is an equivalent expression $y$ which is in reduced form, *i.e.*, $y$ cannot be further simplified.

3. Eliminate all rules which merely replace one variable by another, *e.g.*, $V \longrightarrow W$. These are sometimes called *unit* rules. Thus, for each rule $W \longrightarrow u$ where $u \in (V \cup \Sigma)^*$, we add $V \longrightarrow u$.

4. Map rules into binary. A rule $A \longrightarrow u_1 u_2 \ldots u_n$ where $n \geq 3$ and each $u_i$ is either a symbol from $\Sigma$ or a variable in $V$, is replaced by the collection of rules
$$\begin{array}{lcl} A & \longrightarrow & u_1 A_1 \\ A_1 & \longrightarrow & u_2 A_2 \\ A_2 & \longrightarrow & u_3 A_3 \\ & \vdots & \\ A_{n-2} & \longrightarrow & u_{n-1} u_n \end{array}$$
where $A_1, \ldots, A_{n-2}$ are new variables. Each of the $u_i$ must be a variable. If it is not, then add a rule $U_i \longrightarrow u_i$, and replace $u_i$ everywhere in the rule set with $U_i$.

**Theorem 5.** *Every grammar $G$ has a Chomsky Normal Form $G'$, and $\mathcal{L}(G) = \mathcal{L}(G')$.*

**Example 73.** Let $G$ be given by the following grammar:

$$\begin{array}{lcl} S & \longrightarrow & ASA \mid aB \\ A & \longrightarrow & B \mid S \\ B & \longrightarrow & b \mid \varepsilon \end{array}$$

We will convert this to Chomsky Normal Form by following the steps in the algorithm.

1. Add new start variable. This is accomplished by adding the new rule $S_0 \longrightarrow S$.

2. Now we eliminate the rule $B \longrightarrow \varepsilon$. We must make a *copy* of each rule where $B$ occurs on the right hand side (underlined below). Therefore the grammar
$$\begin{array}{lcl} S_0 & \longrightarrow & S \\ S & \longrightarrow & ASA \mid a\underline{B} \\ A & \longrightarrow & \underline{B} \mid S \\ B & \longrightarrow & b \mid \varepsilon \end{array}$$

is transformed to

$$\begin{array}{lcl} S_0 & \longrightarrow & S \\ S & \longrightarrow & ASA \mid aB \mid a \\ A & \longrightarrow & B \mid S \mid \varepsilon \\ B & \longrightarrow & b \end{array}$$

Notice that, for example, we don't drop $A \longrightarrow B$; instead we keep it and add $A \longrightarrow \varepsilon$. So we've dropped one $\varepsilon$-rule and added another.

3. Eliminate $A \longrightarrow \varepsilon$. This yields the following grammar:

$$\begin{array}{lcl} S_0 & \longrightarrow & S \\ S & \longrightarrow & ASA \mid AS \mid SA \mid S \mid aB \mid a \\ A & \longrightarrow & B \mid S \\ B & \longrightarrow & b \end{array}$$

   We have now finished eliminating $\varepsilon$-rules and can move to eliminating unit rules.

4. Eliminate $S \longrightarrow S$. This illustrates a special case: when asked to eliminate a rule $V \longrightarrow V$, the rule may simply be dropped without any more thought. Thus we have the grammar

$$\begin{array}{lcl} S_0 & \longrightarrow & S \\ S & \longrightarrow & ASA \mid AS \mid SA \mid aB \mid a \\ A & \longrightarrow & B \mid S \\ B & \longrightarrow & b \end{array}$$

5. Eliminate $S_0 \longrightarrow S$. In this case, that means that wherever there is a rule $S \longrightarrow w$, we will add $S_0 \longrightarrow w$. Thus we have

$$\begin{array}{lcl} S_0 & \longrightarrow & ASA \mid AS \mid SA \mid aB \mid a \\ S & \longrightarrow & ASA \mid AS \mid SA \mid aB \mid a \\ A & \longrightarrow & B \mid S \\ B & \longrightarrow & b \end{array}$$

6. Eliminate $A \longrightarrow B$. In this case, that means that wherever there is a rule $B \longrightarrow w$, we will add $A \longrightarrow w$. Thus we have

$$\begin{array}{lcl} S_0 & \longrightarrow & ASA \mid AS \mid SA \mid aB \mid a \\ S & \longrightarrow & ASA \mid AS \mid SA \mid aB \mid a \\ A & \longrightarrow & S \mid b \\ B & \longrightarrow & b \end{array}$$

7. Eliminate $A \longrightarrow S$. In this case, that means that wherever there is a rule $S \longrightarrow w$, we will add $A \longrightarrow w$. Thus we have

   $$\begin{array}{lcl} S_0 & \longrightarrow & ASA \mid AS \mid SA \mid aB \mid a \\ S & \longrightarrow & ASA \mid AS \mid SA \mid aB \mid a \\ A & \longrightarrow & ASA \mid AS \mid SA \mid aB \mid a \mid b \\ B & \longrightarrow & b \end{array}$$

   That finishes the elimination of unit rules. Now we map the grammar to binary form.

8. The rule $S \longrightarrow ASA$ needs to be split, which is accomplished by adding a new rule $A_1 \longrightarrow SA$, and replacing all occurrences of $ASA$ by $AA_1$:

   $$\begin{array}{lcl} S_0 & \longrightarrow & AA_1 \mid AS \mid SA \mid aB \mid a \\ S & \longrightarrow & AA_1 \mid AS \mid SA \mid aB \mid a \\ A & \longrightarrow & AA_1 \mid AS \mid SA \mid aB \mid a \mid b \\ A_1 & \longrightarrow & SA \\ B & \longrightarrow & b \end{array}$$

9. The grammar is still not in final form: right-hand sides such as $aB$ are not in the correct format. This is taken care of by adding a new rule $U \longrightarrow a$ and propagating its definition to all binary rules with the terminal $a$ on the right hand side. This gives us the final grammar in Chomsky Normal Form:

   $$\begin{array}{lcl} S_0 & \longrightarrow & AA_1 \mid AS \mid SA \mid UB \mid a \\ S & \longrightarrow & AA_1 \mid AS \mid SA \mid UB \mid a \\ A & \longrightarrow & AA_1 \mid AS \mid SA \mid UB \mid a \mid b \\ A_1 & \longrightarrow & SA \\ B & \longrightarrow & b \\ U & \longrightarrow & a \end{array}$$

As we can see, conversion to Chomsky Normal Form (CNF) can lead to bulky and awkward grammars. However a grammar $G$ in CNF has various advantages. One of them is that every step in a derivation using $G$ makes *demonstrable* progress towards the final string because either

- the sentential form gets strictly longer (by 1); or

- a new terminal symbol appears.

**Theorem 6.** *If $G$ is in Chomsky Normal Form, then for any string $w \in \mathcal{L}(G)$ of length $n \geq 1$, exactly $2n-1$ steps are required in a derivation of $w$.*

*Proof.* Let $S \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n$ be a derivation of $w$ using $G$, which is in CNF. We first note that, if $n = 1$, then $w = \varepsilon$ (non-terminals can't be derived in one step from $S$).

Since $w \in \Sigma^*$, we know that $w$ is made up of $n$ terminals. In order to produce those $n$ terminals with a CNF grammar, we would need $n$ applications of rules of the form $V \longrightarrow a$. In order for there to have been $n$ such applications, there must have been $n$ variables 'added' in the derivation. Notice that the only way to add a variable into the sentential form is to apply a rule of the form $A \longrightarrow BC$, which replaces one variable by two. Thus we require *at least* $n-1$ applications of rules of the form $A \longrightarrow BC$ to produce enough variables to replace by our $n$ terminals. Thus we require at least $2n-1$ steps in the derivation of $w$.

Could the derivation of $w$ be longer than $2n-1$ steps? No. If that was so, there would have to be more than $n-1$ steps of the form $A \longrightarrow BC$ in the derivation, and then we would have some uneliminated variables. □

## 4.4 Context-Free Parsing

A natural question to ask is the following: given context-free grammar $G$, and a string $w$, is $w$ in the language generated by $G$, or equivalently, $G \overset{*}{\Rightarrow} w$? This can be phrased as the decision problem inCFL:

$$\mathsf{inCFL} = \{\langle G, w\rangle \mid G \overset{*}{\Rightarrow} w\}.$$

Note well that the decision problem is to decide, given an arbitrary grammar and an arbitrary string, whether that string can be generated by that grammar.

One possible approach would be to enumerate derivations in increasing length, attempting to see if $w$ eventually gets derived, but of course this is hopelessly inefficient, and moreover won't terminate if $w \notin \mathcal{L}(G)$. A better approach would be to translate the grammar to Chomsky Normal Form and then enumerate all derivations of length $2n-1$ (there are a finite number of these) checking each to see if $w$ is derived. If it is, then acceptance, otherwise no derivation of $w$ of length $2n-1$ exists, so no derivation of $w$ exists at all, so rejection. Again, this is quite inefficient.

Fortunately, there are general algorithms for context-free parsing that run relatively efficiently. We are going to look at one, known as the CKY algorithm[3], which is directly based on grammars in Chomsky Normal Form. If $G$ is in CNF then it has only rules of the form

$$S \longrightarrow V_1 V_2 \mid \ldots$$

(For the moment, we'll ignore the fact that a rule $S \longrightarrow \varepsilon$ may be allowed. Also we will ignore rules of the form $V \longrightarrow a$.) Suppose that we want to parse the string

$$w = w_1 w_2 \ldots w_n$$

Now, $S \overset{*}{\Rightarrow} w$ if

$$\begin{array}{l} S \Rightarrow V_1 V_2 \quad \text{and} \\ V_1 \overset{*}{\Rightarrow} w_1 \ldots w_i \quad \text{and} \\ V_2 \overset{*}{\Rightarrow} w_{i+1} \ldots w_n \end{array}$$

for some splitting of $w$ at index $i$. This recursive splitting process proceeds until the problem size becomes 1, *i.e.*, the problem becomes one of finding a rule $V \longrightarrow w_i$ that generates a single terminal.

Now, of course, the problem is that there are $n-1$ ways to split a string of length $n$ in two pieces having at least one symbol each. The algorithm considers all of the splits, but in a clever way. The processing goes bottom-up, dealing with shorter strings before longer ones. In this way, solutions to smaller problems can be re-used when dealing with larger problems. Thus this algorithm is an instance of the technique known as *dynamic programming*.

The main notion in the algorithm is

$$N[i, i+j]$$

which denotes the set of variables in $G$ that can derive the substring $w_i \ldots w_{i+j-1}$. Thus $N[i, i+1]$ refers to the variables that can derive the single symbol $w_i$. If we can properly implement this abstraction, then all we have to do to decide if $S \overset{*}{\Rightarrow} w$, roughly speaking, is compute $N[1, n+1]$ and check whether

[3] After the co-inventors Cocke, Kasami, and Younger.

$S$ is in the resulting set. (Note: we will index strings starting at 1 in this section.)

Thus we will systematically compute the following, moving from a step-size of 1, to one of $n$, where $n$ is the length of $w$:

| *Step size* | |
|---|---|
| 1 | $N[1,2], N[2,3], \ldots N[n, n+1]$ |
| 2 | $N[1,3], N[2,4], \ldots N[n-1, n+1]$ |
| $\vdots$ | $\vdots$ |
| $n$ | $N[1, n+1]$ |

In the algorithm, $N[i,j]$ is represented by a two-dimensional array $N$, where the contents of location $N[i,j]$ is the set of variables that generate $w_i \ldots w_j$. We will only need to consider a triangular sub-array.

The ideas are best introduced by example.

**Example 74.** Consider the language $BAL$ of balanced parentheses, generated by the grammar

$$S \longrightarrow \varepsilon \mid (S) \mid SS$$

This grammar is not in Chomsky Normal Form, but the following steps will achieve that:

- New start symbol

$$\begin{array}{rcl} S_0 & \longrightarrow & S \\ S & \longrightarrow & \varepsilon \mid (S) \mid SS \end{array}$$

- Eliminate $S \longrightarrow \varepsilon$

$$\begin{array}{rcl} S_0 & \longrightarrow & S \mid \varepsilon \\ S & \longrightarrow & (S) \mid () \mid S \mid SS \end{array}$$

- Drop $S \longrightarrow S$

$$\begin{array}{rcl} S_0 & \longrightarrow & S \mid \varepsilon \\ S & \longrightarrow & (S) \mid () \mid SS \end{array}$$

- Eliminate $S_0 \longrightarrow S$

$$\begin{array}{rcl} S_0 & \longrightarrow & \varepsilon \mid (S) \mid () \mid SS \\ S & \longrightarrow & (S) \mid () \mid SS \end{array}$$

- Put in binary rule format. We add two rules for deriving the opening and closing parentheses:

$$\begin{aligned} L &\longrightarrow ( \\ R &\longrightarrow ) \end{aligned}$$

and then the final grammar is

$$\begin{aligned} S_0 &\longrightarrow \varepsilon \mid LA \mid LR \mid SS \\ S &\longrightarrow LA \mid LR \mid SS \\ A &\longrightarrow SR \\ L &\longrightarrow ( \\ R &\longrightarrow ) \end{aligned}$$

Now, let's try the algorithm on parsing the string $(()(()))$ with this grammar. The length $n$ of this string is 8. We start by constructing an array $N$ with $n + 1 = 9$ rows and $n$ columns. Then we write the string to be parsed along the diagonal:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ( | | | | | | | |
| 2 | | ( | | | | | | |
| 3 | | | ) | | | | | |
| 4 | | | | ) | | | | |
| 5 | | | | | ( | | | |
| 6 | | | | | | ) | | |
| 7 | | | | | | | ) | |
| 8 | | | | | | | | ) |
| 9 | | | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Now we consider, for each substring of length 1 in the string, the variables that could derive it. For example, the element at $N[2, 3]$ will be $L$, since the rule $L \longrightarrow ($ can be used to generate a '(' symbol. In this way, each $N[i, i + 1]$, *i.e.*, just below the diagonal is filled in:

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| 1 | ( |  |  |  |  |  |  |  |
| 2 | $L$ | ( |  |  |  |  |  |  |
| 3 |  | $L$ | ) |  |  |  |  |  |
| 4 |  |  | $R$ | ( |  |  |  |  |
| 5 |  |  |  | $L$ | ( |  |  |  |
| 6 |  |  |  |  | $L$ | ) |  |  |
| 7 |  |  |  |  |  | $R$ | ) |  |
| 8 |  |  |  |  |  |  | $R$ | ) |
| 9 |  |  |  |  |  |  |  | $R$ |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Now we consider, for each substring of length 2 in the string, the variables that could derive it. Now here's where the cleverness of the algorithm manifests itself. All the information for $N[i, i+2]$ can be found by looking at $N[i, i+1]$ and $N[i+1, i+2]$. So we can re-use information already calculated and stored in $N$. For strings of length 2, it's particularly easy, since the relevant information is directly above and directly to the right. For example, the element at $N[1, 3]$ is calculated by asking "is there a rule of the form $V \longrightarrow LL$?" There is none, so $N[1, 3] = \emptyset$. Similarly, the entry at $N[2, 4] = S_0, S$ because of the rules $S_0 \longrightarrow LR$ and $S \longrightarrow LR$. Proceeding in this way, the next diagonal of the array is filled in as follows:

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| 1 | ( |  |  |  |  |  |  |  |
| 2 | $L$ | ( |  |  |  |  |  |  |
| 3 | $\emptyset$ | $L$ | ) |  |  |  |  |  |
| 4 |  | $S_0, S$ | $R$ | ( |  |  |  |  |
| 5 |  |  | $\emptyset$ | $L$ | ( |  |  |  |
| 6 |  |  |  | $\emptyset$ | $L$ | ) |  |  |
| 7 |  |  |  |  | $S_0, S$ | $R$ | ) |  |
| 8 |  |  |  |  |  | $\emptyset$ | $R$ | ) |
| 9 |  |  |  |  |  |  | $\emptyset$ | $R$ |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Now substrings of length 3 are addressed. It's important to note that all ways of dividing a string of length 3 into non-empty substrings has to be considered. Thus $N[i, i+3]$ is computed from $N[i, i+1]$ and $N[i+1, i+3]$ as well as $N[i, i+2]$ and $N[i+2, i+3]$. For example, let's calculate $N[1, 4]$

- $N[1,2] = L$ and $N[2,4] = S$, but there is no rule of the form $V \longrightarrow LS$, so this split produces no variables
- $N[1,3] = \emptyset$ and $N[3,4] = R$, so this split produces no variables also

Hence $N[1,4] = \emptyset$. By similar calculations, $N[2,5], N[3,6], N[4,7]$ are all $\emptyset$. In $N[5,8]$ though, we can use the rule $A \longrightarrow SR$ to derive $N[5,7]$ followed by $N[7,8]$. Thus the next diagonal is filled in:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ( | | | | | | | |
| 2 | $L$ | ( | | | | | | |
| 3 | $\emptyset$ | $L$ | ) | | | | | |
| 4 | $\emptyset$ | $S_0, S$ | $R$ | ( | | | | |
| 5 | | $\emptyset$ | $\emptyset$ | $L$ | ( | | | |
| 6 | | | $\emptyset$ | $\emptyset$ | $L$ | ) | | |
| 7 | | | | $\emptyset$ | $S_0, S$ | $R$ | ) | |
| 8 | | | | | $A$ | $\emptyset$ | $R$ | ) |
| 9 | | | | | | $\emptyset$ | $\emptyset$ | $R$ |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Filling in the rest of the diagonals yields

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | ( | | | | | | | |
| 2 | $L$ | ( | | | | | | |
| 3 | $\emptyset$ | $L$ | ) | | | | | |
| 4 | $\emptyset$ | $S_0, S$ | $R$ | ( | | | | |
| 5 | $\emptyset$ | $\emptyset$ | $\emptyset$ | $L$ | ( | | | |
| 6 | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $L$ | ) | | |
| 7 | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $S_0, S$ | $R$ | ) | |
| 8 | $\emptyset$ | $S_0, S$ | $\emptyset$ | $S_0, S$ | $A$ | $\emptyset$ | $R$ | ) |
| 9 | $S_0, S$ | $A$ | | $A$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $R$ |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Since $S_0 \in N[1,9]$, we have shown the existence of a parse tree for the string $(()(()))$.

An implementation of this algorithm can be coded in a concise triply-nested loop of the form:

```
For each substring length
    For each substring $u$ of that length
```

```
            For each split of $u$ into non-empty pieces
        ....
```

As a result, the running time of the algorithm is $O(n^3)$ in the length of the input string.

Other algorithms for context-free parsing are more popular than the CKY algorithm. In particular, a top-down CFL parser due to Earley is more efficient in many cases.

## 4.5 Grammar Decision Problems

We have seen that the decision problem inCFL

$$\text{inCFL} = \{\langle G, w\rangle \mid G \overset{*}{\Rightarrow} w\}.$$

is decidable. Here we list a number of other decision problems about grammars along with their decidability status:

**emptyCFL.** Does a CFG generate any strings at all?

$$\text{emptyCFL} = \{\langle G\rangle \mid \mathcal{L}(G) = \emptyset\}$$

Decidable. How?

**fullCFL.** Does a CFG generate all strings over the alphabet?

$$\text{fullCFL} = \{\langle G\rangle \mid \mathcal{L}(G) = \Sigma^*\}$$

Undecidable.

**subCFL.** Does one CFG generate a subset of the strings generated by another?

$$\text{subCFL} = \{\langle G_1, G_2\rangle \mid \mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)\}$$

Undecidable.

**sameCFL.** Do two CFGs generate the same language?

$$\text{sameCFL} = \{\langle G_1, G_2\rangle \mid \mathcal{L}(G_1) = \mathcal{L}(G_2)\}$$

Undecidable.

**ambigCFG.** Is a CFG ambiguous, *i.e.*, is there some string $w \in \mathcal{L}(G)$ with more than one leftmost derivation using $G$? Undecidable.

## 4.6 Push Down Automata

Push Down Automata (PDAs) are a machine counterpart to context-free grammars. PDAs *consume*, or process, strings, while CFGs generate strings. A PDA can be roughly characterized as follows:

$$PDA = TM - tape + stack$$

In other words, a PDA is a machine with a finite number of control states, like a Turing machine, but it can only access its data in a stack-like fashion as it operates.

*Remark.* Recall that a *stack* is a 'last-in-first-out' (LIFO) queue, with the following operations:

**Push** Add an element $x$ to the top of the stack.

**Pop** Remove the top element from the stack.

**Empty** Test the stack to see if it is empty. We won't use this feature in our work.

Only the top of the stack may be accessed in any one step; multiple pushes and pops can be used to access other elements of the stack.

Use of the stack puts an *explicit memory* at our disposal. Moreover, a stack can hold an unbounded amount of information. However, the constraint to access the stack in LIFO style means that use of memory is also constrained.

Here's the formal definition.

**Definition 22** (Push-Down Automaton)**.** A Push-Down Automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- $Q$ is a finite set of control states.
- $q_0$ is the start state.
- $F$ is a finite set of accepting states.
- $\Sigma$ is the input alphabet (finite set of symbols).

- $\Gamma$ is the stack alphabet (finite set of symbols). $\Sigma \subseteq \Gamma$. As for Turing machines, the need for $\Gamma$ being an extension of $\Sigma$ comes from the fact that it is sometimes convenient to use symbols other than those found in the input alphabet as special markers in the stack.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \longrightarrow 2^{Q \times (\Gamma \cup \{\varepsilon\})}$ is the transition function. Although $\delta$ seems daunting, it merely incorporates the use of the stack. When making a transition step, the machine uses the current input symbol *and* the top of the stack in order to decide what state to move to. However, that's not all, since the machine must also update the top of the stack at each step.

It is obviously complex to make a step of computation in a PDA. We have to deal with non-determinism and $\varepsilon$-transitions, but the stack must also be taken account of. Suppose $q \in Q$, $a \in \Sigma$, and $u \in \Gamma$. Then a computation step

$$\delta(q, a, u) = \{(q_1, v_1), \ldots, (q_n, v_n)\}$$

means that if the PDA is in state $q$, reading tape symbol $a$, and symbol $u$ is at the top of the stack, then there are $n$ possible outcomes. In outcome $(q_i, v_i)$, the machine has moved to state $q_i$, and $u$ at the top of the stack has been replaced by $v_i$.

Descriptions of how the top of stack changes in a computation step seem peculiar at first glance. We summarize the possibilities in the following table.

| operation | step | result |
|---|---|---|
| push $x$ | $\delta(q, a, \varepsilon)$ | $(q_i, x)$ |
| pop $x$ | $\delta(q, a, x)$ | $(q_i, \varepsilon)$ |
| skip | $\delta(q, a, \varepsilon)$ | $(q_i, \varepsilon)$ |
| replace$(c, d)$ | $\delta(q, a, c)$ | $(q_i, d)$ |

Note that we have to designate the element in a pop operation. This is unlike conventional stacks.

Let's try to explain this curious notation. It helps to explicitly include the input string and the stack. Thus a *configuration* of the machine is a triple $(q, \mathit{string}, \mathit{stack})$. We use the notation $c \cdot t$ to represent the stack (a string) where the top of the stack is $c$ and the rest of the stack is $t$. A confusing aspect of the notation is that a stack $t$ is sometimes regarded as a stack with the $\varepsilon$-symbol as the top element, so $t$ is the same as $\varepsilon \cdot t$.

- When pushing symbol $x$, the configuration changes from $(q, a\cdot w, \varepsilon\cdot t)$ to $(q_i, w, x \cdot t)$.

- When popping symbol $x$, the configuration changes from $(q, a\cdot w, x\cdot t)$ to $(q_i, w, t)$.

- When we don't wish the stack to change at all in a computation step, the machine moves from a configuration $(q, a \cdot w, \varepsilon \cdot t)$ to $(q_i, w, \varepsilon \cdot t)$.

- Finally, on the occasion that we actually do wish to change the symbol $c$ at the top of stack with symbol $d$, the configuration $(q, a \cdot w, c \cdot t)$ changes to $(q_i, w, d \cdot t)$.

Now that steps of computation are better understood, the notion of an execution is easy. An execution starts with the configuration $(q_0, s, \varepsilon)$, *i.e.*, the machine is in the start state, the input string $s$ is on the tape, and the stack is empty. A successful execution is one which finishes in a configuration where $s$ has been completely read, the final state of the machine is an accept state, and the stack is empty.

*Remark.* Notice that the notion of acceptance means that, even if the machine ends up in a final state after processing the input string, the string may still not be accepted; the stack must also be empty in order for the string to be accepted.

Acceptance can be formally defined as follows:

**Definition 23** (PDA execution and acceptance)**.** A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ *accepts* $w = w_1 \cdot w_2 \cdot \ldots \cdot w_n$, where each $w_i \in \Sigma$, just in case there exists

- a sequence of states $r_0, r_1, \ldots, r_m \in Q$;

- a sequence of stacks (strings in $\Gamma^*$) $s_0, s_1, \ldots s_m$

such that the following three conditions hold:

**1. Initial condition** $r_0 = q_0$ and $s_0 = \varepsilon$.

**2. Computation steps**

$$\forall i.\ 0 \leq i \leq m - 1 \Rightarrow (r_{i+1}, b) \in \delta(r_i, w_{i+1}, a) \wedge s_i = a \cdot t \wedge s_{i+1} = b \cdot t$$

where $a, b \in \Sigma \cup \{\varepsilon\}$ and $t \in \Gamma^*$.

**3. Final condition** $r_m \in F$ and $s_m = \varepsilon$.

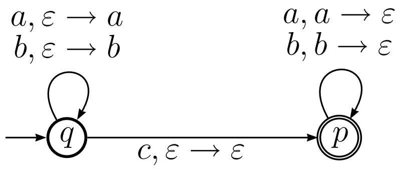
As usual, the language $\mathcal{L}(M)$ of a PDA $M$ is the set of strings accepted by $M$.

As for Turing machines, PDAs can be represented by state transition diagrams.

**Example 75.** Let $M = (Q, \Sigma, \Gamma, q_0, F)$ be a PDA where $Q = \{p, q\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $q_0 = q$, $F = \{p\}$, and $\delta$ is given as follows

$$\begin{array}{lcl}
\delta(q, a, \varepsilon) & = & \{(q, a)\} \\
\delta(q, b, \varepsilon) & = & \{(q, b)\} \\
\delta(q, c, \varepsilon) & = & \{(p, \varepsilon)\} \\
\delta(p, a, a) & = & \{(p, \varepsilon)\} \\
\delta(p, b, b) & = & \{(p, \varepsilon)\}
\end{array}$$

It is very hard to visualize what $M$ is supposed to do! A diagram helps. We'll use the notation $a, b \rightarrow c$ on a transition between states $q_i$ and $q_j$ to mean $(q_j, c) \in \delta(q_i, a, b)$.


$a, \varepsilon \rightarrow a$
$b, \varepsilon \rightarrow b$
$a, a \rightarrow \varepsilon$
$b, b \rightarrow \varepsilon$
$q$
$p$
$c, \varepsilon \rightarrow \varepsilon$

In words, $M$ performs as follows. It stays in state $q$ pushing input symbols on to the stack until it encounters a $c$. Then it moves to state $p$, in which it repeatedly pops $a$ and $b$ symbols off the stack provided the input symbol is identical to that on top of the stack. If at the end of the string, the stack is empty, then the machine accepts.

However, notice that failure in processing a string is not explicitly represented. For example, what if the input string has no $c$ in it? In that case, $M$ will never leave state $q$. Once the input is finished, we find that we are not in a final state and so can't accept the string. For another example, what if we try the string $ababcbab$? Then $M$ will enter state $p$ with the stack $baba$, *i.e.*, with the configuration $(p, bab, baba)$. Then the following configurations happen:

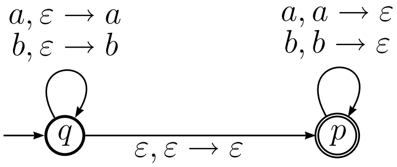
$$(p, bab, baba) \Rightarrow (p, ab, aba) \Rightarrow (p, b, ba) \Rightarrow (p, \varepsilon, a)$$

At this point the input string is exhausted and the computation stops. We cannot accept the original string—although we are in an accept state—because the stack is not empty. Thus we see that

$$\mathcal{L}(M) = \{wcw^{\mathcal{R}} \mid w \in \{a,b\}^*\} \ .$$

This example used $c$ as a marker telling the machine when to change states. It turns out that such an expedient is not needed because we have non-determinism at our disposal.

**Example 76.** Find a PDA for $L = \{ww^{\mathcal{R}} \mid w \in \{a,b\}^*\}$.
The PDA is just that of the previous example with the seemingly innocuous alteration of the transition from $p$ to $q$ to be an $\varepsilon$-transition.


$a, \varepsilon \rightarrow a$
$b, \varepsilon \rightarrow b$
$a, a \rightarrow \varepsilon$
$b, b \rightarrow \varepsilon$
$q$
$p$
$\varepsilon, \varepsilon \rightarrow \varepsilon$

The machine may non-deterministically move from $q$ to $p$ at any point in the execution. The only thing that matters for acceptance is that there is some point at which the move works, *i.e.*, results in an accepting computation.

For example, given input $abba$, how does the machine work? The following sequence of configurations shows an accepting computation path:

$$\begin{array}{rcl}
(q, abba, \varepsilon) & \rightarrow & (q, bba, a) \\
& \rightarrow & (q, ba, ba) \\
& \rightarrow & (p, ba, ba) \\
& \rightarrow & (p, a, a) \\
& \rightarrow & (p, \varepsilon, \varepsilon)
\end{array}$$

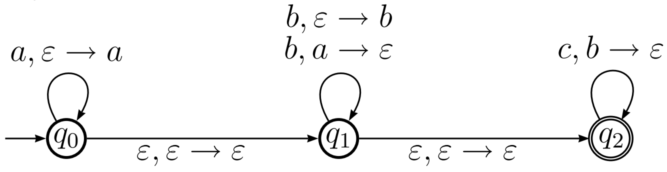
The following is an unsuccessful execution:

$$\begin{array}{rcl}
(q, abba, \varepsilon) & \rightarrow & (q, bba, a) \\
& \rightarrow & (q, ba, ba) \\
& \rightarrow & (p, a, bba) \\
& \rightarrow & (p, a, bba) \\
& \rightarrow & \text{blocked!}
\end{array}$$

**Example 77.** Build a PDA to recognize

$$L = \{a^i b^j c^k \mid i + k = j\}$$

The basic idea in finding a solution is to use *states* to enforce the order of occurrences of letters, and to use the stack to enforce the requirement that $i + k = j$.



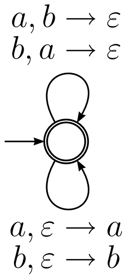
In the first state, $q_0$, we simply push $a$ symbols. Then we move to state $q_1$ where we either

- pop an $a$ when a $b$ is seen on the input, or
- push a $b$ for every $b$ seen.

If $i > j$ then there will be more $a$ symbols on the stack than consecutive $b$ symbols remaining in the input. In this case, some $a$ symbols will be left on the stack when leaving $q_1$. This situation is not catered for in state $q_2$, so the machine will block and the input will not be accepted. This is the correct behaviour.

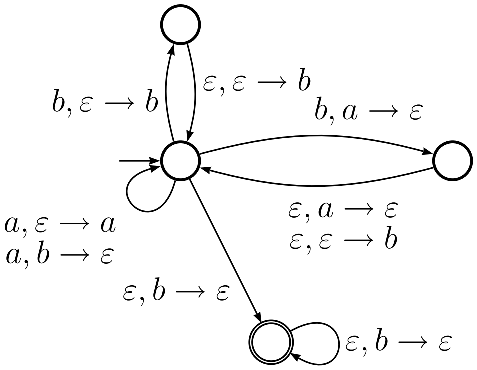
On the other hand, if $i \leq j$, there are more consecutive $b$ symbols than there are $a$ symbols on the stack, so in state $q_1$ the stack of $i$ $a$ symbols will become empty and then be filled with $j - i$ $b$ symbols. Entering state $q_2$ with such a stack will result in acceptance only if there are $j - i$ $c$ symbols left in the input.

**Example 78.** Give a PDA for $L = \{x \in \{a, b\}^* \mid \mathsf{count}(a, x) = \mathsf{count}(b, x)\}$.



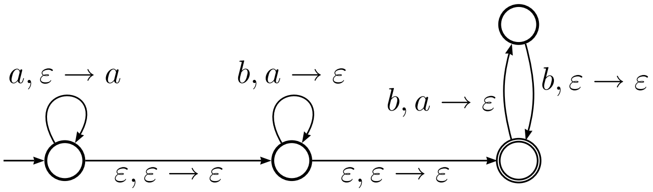
**Example 79.** Give a PDA for $L = \{x \in \{a,b\}^* \mid \mathsf{count}(a,x) < 2 * \mathsf{count}(b,x)\}$.

This problem can be rephrased as: *$x$ is in $L$ if, after doubling the number of $b$'s in $x$, we have more $b$'s than $a$'s.* We can build a machine to do this explicitly: every time it sees a $b$ in the input string, it will treat it as 2 consecutive $b$'s.



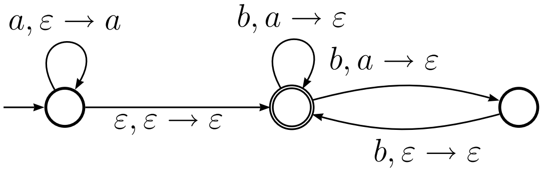
$b, \varepsilon \to b$
$\varepsilon, \varepsilon \to b$
$b, a \to \varepsilon$
$\varepsilon, a \to \varepsilon$
$\varepsilon, \varepsilon \to b$
$a, \varepsilon \to a$
$a, b \to \varepsilon$
$\varepsilon, b \to \varepsilon$
$\varepsilon, b \to \varepsilon$

**Example 80.** Give a PDA for $L = \{a^i b^j \mid i \le j \le 2i\}$. $L$ is equivalent to $\{a^i b^i b^k \mid k \le i\}$, which isequivalent to $\{a^k a^\ell b^\ell b^{2k} \mid k, \ell \ge 0\}$. A machine dealing with this language is easy to build, by putting different functionality in different states. Non-deterministic transitions take care of *guessing* the right time to make a transition. In the first state, we push $k + \ell$ $a$ symbols; in the second we cancel off $b^\ell$; and in the last we consume $b^{2k}$ while popping $k$ $a$ symbols.



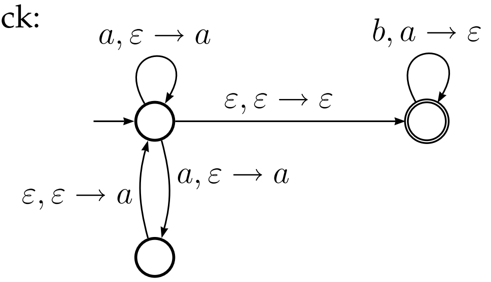
$a, \varepsilon \to a$
$b, a \to \varepsilon$
$b, a \to \varepsilon$
$b, \varepsilon \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$

Note that we could shrink this machine, by merging the last two states:



$a, \varepsilon \to a$
$b, a \to \varepsilon$
$b, a \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$
$b, \varepsilon \to \varepsilon$

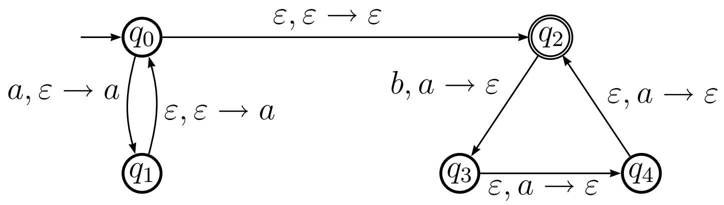
This machine non-deterministically chooses to cancel one or two $b$ symbols for each $a$ seen in the input. Note that we could also write an equivalent machine that non-deterministically chooses to push one or two $a$ symbols to the stack:


$a, \varepsilon \rightarrow a$
$b, a \rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow a$
$a, \varepsilon \rightarrow a$

**Example 81.** Build a PDA to recognize

$$L = \{a^i b^j \mid 2i = 3j\}$$

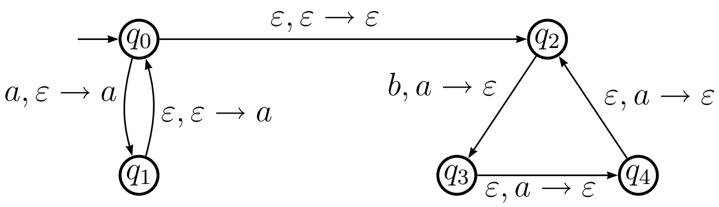
Thus $L = \{\varepsilon, a^3b^2, a^6b^4, a^9b^6, \ldots\}$. The idea behind a solution to this problem is that we wish to push and pop multiple $a$ symbols. On seeing an $a$ in the input, we want to push two $a$ symbols on the stack. When we see a $b$, we want to pop three $a$ symbols. The technical problem we face is that pushing and popping only deal with one symbol at a time. Thus in order to deal with multiple symbols, we will need to employ multiple states.


$q_0$
$q_1$
$q_2$
$q_3$
$q_4$
$\varepsilon, \varepsilon \rightarrow \varepsilon$
$a, \varepsilon \rightarrow a$
$\varepsilon, \varepsilon \rightarrow a$
$b, a \rightarrow \varepsilon$
$\varepsilon, a \rightarrow \varepsilon$
$\varepsilon, a \rightarrow \varepsilon$

**Example 82.** Build a PDA to recognize

$$L = \{a^i b^j \mid 2i \neq 3j\}$$

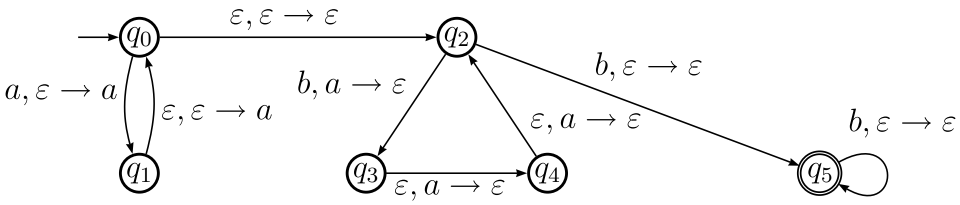
This is a more difficult problem. We will be able to re-use the basic idea of the previous example, but must now take extra cases into account. The success state $q_2$ of the previous example will now change into a reject state. But there is much more going on. We will build the solution incrementally. The basic skeleton of our answer is



$q_0$
$q_1$
$q_2$
$q_3$
$q_4$
$\varepsilon, \varepsilon \to \varepsilon$
$a, \varepsilon \to a$
$\varepsilon, \varepsilon \to a$
$b, a \to \varepsilon$
$\varepsilon, a \to \varepsilon$
$\varepsilon, a \to \varepsilon$

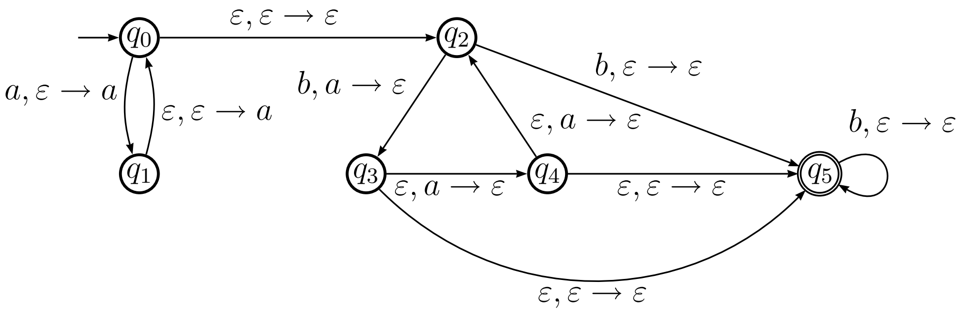
If we arrive in $q_2$ where the input has been exhausted and the stack is empty, we should reject, and that is what the above machine does. The other cases in $q_2$ are

- There is remaining input and the stack is not empty. This case is already covered: go to $q_3$.
- There is remaining input and the stack is empty. We can assume that the head of the remaining input is a $b$. (All the leading $a$ symbols have already been dealt with in the $q_0, q_1$ pair.) We need to transition to an accept state where we ensure that the rest of the input is all $b$ symbols. Thus we invent a new accept state $q_5$ where we discard the remaining $b$ symbols in the input.



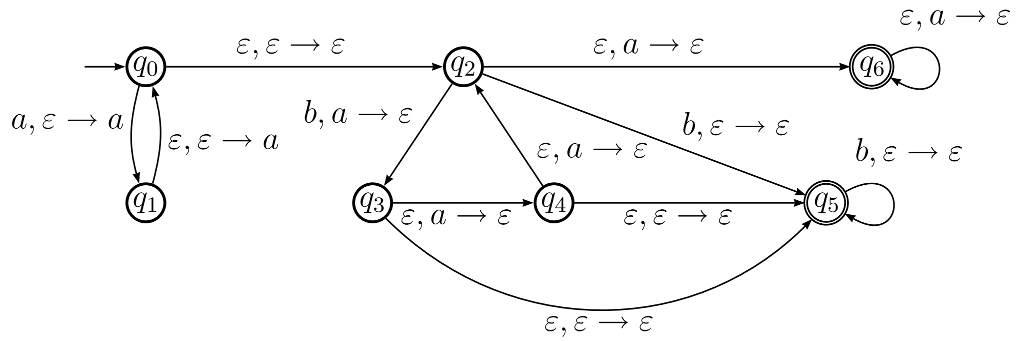
$q_0$
$q_1$
$q_2$
$q_3$
$q_4$
$q_5$
$\varepsilon, \varepsilon \to \varepsilon$
$a, \varepsilon \to a$
$\varepsilon, \varepsilon \to a$
$b, a \to \varepsilon$
$b, \varepsilon \to \varepsilon$
$\varepsilon, a \to \varepsilon$
$b, \varepsilon \to \varepsilon$
$\varepsilon, a \to \varepsilon$

We further notice that this situation can happen in $q_3$ and $q_4$, so we add $\varepsilon$-transitions from them to $q_5$:



$q_0$
$q_1$
$q_2$
$q_3$
$q_4$
$q_5$
$\varepsilon, \varepsilon \to \varepsilon$
$a, \varepsilon \to a$
$\varepsilon, \varepsilon \to a$
$b, a \to \varepsilon$
$b, \varepsilon \to \varepsilon$
$\varepsilon, a \to \varepsilon$
$b, \varepsilon \to \varepsilon$
$\varepsilon, a \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$

- The input is exhausted, but the stack is not empty. Thus we have excess $a$ symbols on the stack and we need to jettison them before accepting. This is handled in a new final state $q_6$:


$\varepsilon, a \rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow \varepsilon$
$q_0$
$q_2$
$\varepsilon, a \rightarrow \varepsilon$
$q_6$
$a, \varepsilon \rightarrow a$
$\varepsilon, \varepsilon \rightarrow a$
$b, a \rightarrow \varepsilon$
$b, \varepsilon \rightarrow \varepsilon$
$\varepsilon, a \rightarrow \varepsilon$
$b, \varepsilon \rightarrow \varepsilon$
$q_1$
$q_3$
$q_4$
$q_5$
$\varepsilon, a \rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow \varepsilon$

This is the final PDA.

## 4.7 Equivalence of PDAs and CFGs

The relationship between PDAs and CFGs is similar to that between DFAs and regular expressions; namely, the languages accepted by PDAs are just those that can be generated by CFGs.

**Theorem 7.** *Suppose $L$ is a context-free language. Then there is a PDA $M$ such that $\mathcal{L}(M) = L$.*

**Theorem 8.** *Suppose $M$ is a PDA. Then there is a grammar $G$ such that $\mathcal{L}(G) = \mathcal{L}(M)$, i.e., L(M) is context-free.*

The proofs of these theorems take a familiar approach: given an arbitrary grammar, we construct the corresponding PDA; and given an arbitrary PDA, we construct the corresponding grammar.

### 4.7.1 Converting a CFG to a PDA

The basic idea in the construction is to build $M$ so that it simulates the leftmost derivation of strings using $G$. The machine we construct uses the terminals and non-terminals of the grammar as stack symbols. What we conceptually want to do is to use the stack to hold the sentential form that evolves during a derivation. At each step, the topmost variable in the stack will get replaced by the rhs of some grammar rule. Of course, there are several problems with implementing this concept. For one, the PDA can only access the top of its stack: it can't find a variable below the top. For another, even if the PDA could find such a variable, it couldn't fit the rhs into a single stack slot. But these are not insurmountable. We simply have to arrange things so that the PDA always has the leftmost variable of the sentential form on top of the stack. If that can be set up, the PDA can use the technique of using extra states to push multiple symbols 'all at once'.

The other consideration is that we are constructing a PDA after all, so it needs to consume the input string and give a verdict. This fits in nicely with our other requirements. In brief, the PDA will use $\varepsilon$-transitions to push the rhs of rules into the stack, and will use 'normal' transitions to consume input. In consuming input, we will be able to remove non-variables from the top of the stack, always guaranteeing that a variable is at the top of the stack.

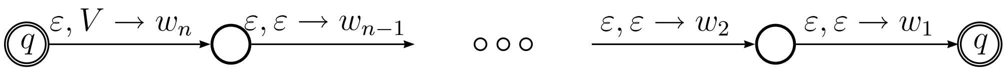
Here are the details. Let $G = (V, \Sigma, R, S)$. We will construct $M = (Q, \Sigma, \underbrace{V \cup \Sigma}_{\Gamma}, \delta, q_0, \{q\})$ where

- $Q = \{q_0, q\} \cup \mathit{RuleStates}$;
- $q_0$ is the start variable;
- $q$ is a *dispatch* state at the center of a loop. It is also the sole accept state for the PDA
- $\delta$ has rules for getting started, for consuming symbols from the input, and for pushing the rhs of rules onto the stack.

  **getting started.** $\delta(q_0, \varepsilon, \varepsilon) = \{(q, S)\}$. The start symbol is pushed on the stack and a transition is made to the loop state.

  **consuming symbols.** $\delta(q, a, a) = \{(q, \varepsilon)\}$, for every terminal $a \in \Sigma$.

  **pushing rhs of rules** For each rule $R_i = V \longrightarrow w_1 \cdot w_2 \cdot \cdots \cdot w_n$ in $R$, where each $w_i$ may be a terminal or non-terminal, add $n - 1$ states to $\mathit{RuleStates}$. Also add the loop (from $q$ to $q$)



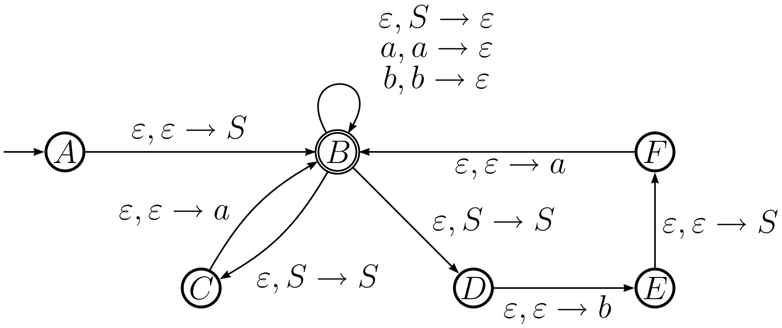
$q$
$\varepsilon, V \to w_n$
$\varepsilon, \varepsilon \to w_{n-1}$
$\varepsilon, \varepsilon \to w_2$
$\varepsilon, \varepsilon \to w_1$
$q$

which pushes the rhs of the rule, using the $n-1$ states. Note that the symbols in the rhs of the rule are pushed on the stack in *right-to-left* order.

**Example 83.** Let $G$ be given by the grammar

$$S \longrightarrow aS \mid aSbS \mid \varepsilon$$

The corresponding PDA is



$\varepsilon, S \to \varepsilon$
$a, a \to \varepsilon$
$b, b \to \varepsilon$
$\varepsilon, \varepsilon \to S$
$A$
$B$
$F$
$\varepsilon, \varepsilon \to a$
$\varepsilon, \varepsilon \to a$
$\varepsilon, S \to S$
$\varepsilon, \varepsilon \to S$
$C$
$\varepsilon, S \to S$
$D$
$E$
$\varepsilon, \varepsilon \to b$

Consider the input $aab$. A derivation using $G$ is

$$S \Rightarrow aS \Rightarrow aaSbS \Rightarrow aa\varepsilon bS \Rightarrow aa\varepsilon b\varepsilon = aab$$

As a sequence of machine configurations, this looks like

$$\begin{array}{rl}
(A, aab, \varepsilon) \longrightarrow & (B, aab, S) \\
\longrightarrow & (C, aab, S) \\
\longrightarrow & (B, aab, aS) \\
\longrightarrow & (B, ab, S) \\
\longrightarrow & (D, ab, S) \\
\longrightarrow & (E, ab, bS) \\
\longrightarrow & (F, ab, SbS) \\
\longrightarrow & (B, ab, aSbS) \\
\longrightarrow & (B, b, SbS) \\
\longrightarrow & (B, b, bS) \\
\longrightarrow & (B, \varepsilon, S) \\
\longrightarrow & (B, \varepsilon, \varepsilon)
\end{array}$$

And so the machine would accept $aab$.

### 4.7.2 Converting a PDA to a CFG

The previous construction, spelled out in full would look messy, but is in fact quite simple. Going in the reverse direction, *i.e.*, converting a PDA to a CFG, is more difficult. The basic idea is to consider any two states $p, q$ of PDA $M$ and think about what strings could be consumed in executing $M$ from $p$ to $q$. Those strings will be represented by a variable $V_{pq}$ in $G$, the grammar we are building. By design, the strings generated by $V_{pq}$ would be just those substrings consumed by $M$ in going from $p$ to $q$. Thus $S$, the start variable, will stand for all strings consumed in going from $q_0$ to an accept state. This is clear enough, but as always for PDAs, we must consider the stack, hence the story will be more involved; for example, we will use funky variables of the form $V_{pAq}$, where $A$ represents the top of the stack.

The construction goes as follows: given PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, we will construct a grammar $G$ such that $\mathcal{L}(G) = \mathcal{L}(M)$. Two main steps achieve this goal:

- $M$ will be modified to an equivalent $M'$, with a more desirable form.

  An important aspect of $M'$ is that it *always* looks at the top symbol on the stack in each move. Thus, no transitions of the form $a, \varepsilon \rightarrow b$

(pushing $b$ on the stack), $\varepsilon, \varepsilon \to \varepsilon$, or $a, \varepsilon \to \varepsilon$ (ignoring stack) are allowed. How can these be eliminated from $\delta$ without changing the behaviour? First we need to make sure that the stack is never empty, for if $M'$ is going to look at the top element of the stack, the stack had better never be empty. This can be ensured by starting the computation with a special token ($) in the stack and then maintaining an invariant that the stack never thereafter becomes empty. It will also be necessary to allow $M'$ to push two stack symbols in one move: since $M'$ always looks at the top stack symbol, we need to push two symbols in order to get the effect of a push operation on a stack. This can be implemented by using extra states, but we will simply assume that $M'$ has this extra convenience.

Furthermore, we are going to add a new start state $s$ and the transition $\varepsilon, \varepsilon \to \$$, which pushes $ on the stack when moving from the new start state $s$ to the original start state $q_0$. We also add a new final state $q_f$, with $\varepsilon, \$ \to \varepsilon$ transitions from all members of $F$ to $q_f$. Thus the machine $M'$ has a single start state and a single end state, always examines the top of its stack, and behaves the same as the machine $M$.

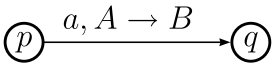
- Construct $G$ so that it simulates the working of $M'$. We first construct the set of variables of $G$.

  $$V = \{V_{pAq} \mid p, q \in Q \cup \{q_f\} \wedge A \in \Gamma \cup \{\$\}\}$$

  Thus we create a lot of new variables: one for each combination of states and possible stack elements. The intent is that each variable $V_{pAq}$ will generate the following strings:

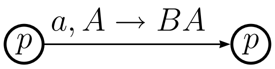
  $$\{x \in \Sigma^* \mid (p, a, A) \xrightarrow{*} (q, \varepsilon, \varepsilon)\}$$

  Now, because of the way we constructed $M'$, there are three kinds of transitions to deal with:

  1. 

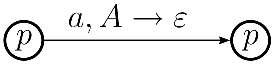
     Add the rule $V_{pAr} \longrightarrow aV_{qBr}$ for all $r \in Q \cup \{q_f\}$

2.



Add the rule $V_{pAr} \longrightarrow aV_{qBr'}V_{r'Ar}$ for all $r, r' \in Q \cup \{q_f\}$

3.



Add the rule $V_{pAq} \longrightarrow a$.

The above construction works because the theorem

$$V_{pAq} \overset{*}{\Rightarrow} w \text{ iff } (p, w, A) \xrightarrow{*} (q, \varepsilon, \varepsilon)$$

can be proved (it's a little complicated though). From this we can immediately get

$$V_{q_0\$q_f} \overset{*}{\Rightarrow} w \text{ iff } (q_0, w, \$) \xrightarrow{*} (q_f, \varepsilon, \varepsilon)$$

Thus by making $V_{q_0\$q_f}$ the start symbol of the grammar, we have achieved our goal.

## 4.8 Parsing

To be added ...

# Chapter 5

# Automata

Automata are a particularly simple, but useful, model of computation. They were initially proposed[1] as a simple model for the behaviour of neurons.

> *The concept of a finite automaton appears to have arisen in the 1943 paper "A logical calculus of the ideas immanent in nervous activity", by Warren McCullock and Walter Pitts. These neurobiologists set out to model the behaviour of neural nets, having noticed a relationship between neural nets and logic:*
>
> > *"The 'all-or-none' law of nervous activity is sufficient to ensure that the activity of any neuron may be represented as a proposition. ... To each reaction of any neuron there is a corresponding assertion of a simple proposition."*

In 1951 Kleene introduced regular expressions to describe the behaviour of finite automata. He also proved the important theorem saying that regular expressions exactly capture the behaviours of finite automata. In 1959, Dana Scott and Michael Rabin introduced *non-deterministic* automata and showed the surprising theorem that they are equivalent to deterministic automata. We will study these fundamental results. Since those early years, the study of automata has continued to grow, showing that they are indeed a fundamental idea in computing.

[1] This historical material is taken from an article by Bob Constable at *The Kleene Symposium*, an event held in 1980 to honour Stephen Kleene's contribution to logic and computer science.

We said that automata are a model of computation. That means that they are a simplified abstraction of 'the real thing'. So what gets abstracted away? One thing that disappears is any notion of hardware or software. We merely deal with states and transitions between states.

| We keep | We drop |
|---|---|
| some notion of state | notion of memory |
| stepping between states | variables, commands, expressions |
| start state | syntax |
| end states | |

The distinction between program and machine executing it disappears. One could say that an automaton *is* the machine and the program. This makes automata relatively easy to implement in either hardware or software.

From the point of view of *resource consumption*, the essence of a finite automaton is that it is a strictly *finite* model of computation. Everything in it is of a fixed, finite size and cannot be extended in the course of the computation.

## 5.1 Deterministic Finite State Automata

More precisely, a DFA (*Deterministic Finite State Automaton*) is a simple machine that reads an input string—one symbol at a time—and then, after the string has been completely read, decides whether to *accept* or *reject* the whole string. As the symbols are read, the automaton can change its state, to reflect how it reacts to what it has seen so far.

Thus, a DFA conceptually consists of 3 parts:

- A *tape* to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from $\Sigma$.
- A *tape head* for reading symbols from the tape
- A *control*, which itself consists of 3 things:
  - a finite number of states that the machine is allowed to be in
  - a current state, initially set to a *start state*
  - a state transition function for changing the current state

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

- The tape head reads the current tape cell and sends the symbol $s$ found there to the control. Then the tape head moves to the next cell. The tape head can only move forward.

- The control takes $s$ and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been traversed, the final state is examined. If it is an *accept* state, the input string is accepted; otherwise, the string is rejected. All the above can be summarized in the following formal definition:

**Definition 24** (Deterministic Finite State Automaton)**.** A *Deterministic Finite State Automaton* DFA is a 5-tuple:

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

where

- $Q$ is a finite set of states

- $\Sigma$ is a finite alphabet

- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function (which is *total*).

- $q_0 \in Q$ is the start state

- $F \subseteq Q$ is the set of accept states

A single computation step in a DFA is just the application of the transition function to the current state and the current symbol. Then an *execution* consists of a linked sequence of computation steps, stopping once all the symbols in the string have been processed.

**Definition 25** (Execution)**.** If $\delta$ is the transition function for machine $M$, then a step of computation is defined as

$$\mathsf{step}(M, q, a) = \delta(q, a)$$

A sequence of steps $\Delta$ is defined as

$$\begin{array}{l} \Delta(M, q, \epsilon) = q \\ \Delta(M, q, a \cdot x) = \Delta(M, \mathsf{step}(M, q, a), x) \end{array}$$

Finally, an *execution* of $M$ on string $x$ is a sequence of computation steps beginning in the start state $q_0$ of $M$:

$$\mathsf{execute}(M, x) = \Delta(M, q_0, x)$$

The *language recognized by a DFA $M$* is the set of all strings accepted by $M$, and is denoted by $\mathcal{L}(M)$. We will make these ideas precise in the next few pages.

### 5.1.1 Examples

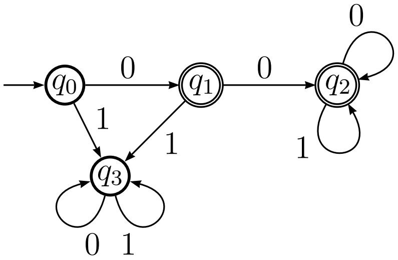
Now we shall review a collection of examples of DFAs.

**Example 84.** DFA $M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- The start state is $q_0$ (this will be our convention)
- $F = \{q_1, q_2\}$
- $\delta$ is defined by the following table:

|  | 0 | 1 |
|---|---|---|
| $q_0$ | $q_1$ | $q_3$ |
| $q_1$ | $q_2$ | $q_3$ |
| $q_2$ | $q_2$ | $q_2$ |
| $q_3$ | $q_3$ | $q_3$ |

This presentation is nicely formal, but very hard to comprehend. The following *state transition diagram* is far easier on the brain:



Notice that the start state is designated by an arrow with no source. Final states are marked by double circles. The strings accepted by $M$ are:

$$\{0, 00, 000, 001, 000, 0010, 0011, 0001, \ldots\}$$

**NB.** The transition function is *total*, so every possible combination of states and input symbols must be dealt with. Also, for every $(q, a) \in Q \times \Sigma$, there is exactly one next state (which is why these are *deterministic* automata). Thus, given any string $x$ over $\Sigma$, there is only one *path* starting from $q_0$, the labels of which form $x$.

A state $q$ in which every next state is $q$ is a *black hole* state since the current state will never change until the string is completely processed. If $q$ is an accept state, then we call $q$ a *success* state. If not, it's called a *failure* state. In our example, $q_3$ is a failure state and $q_2$ is a success state.

*Question* : What is the language accepted by $M$, *i.e.*, what is $\mathcal{L}(M)$?

*Answer* : $\mathcal{L}(M)$ consists of 0 and all binary strings starting with 00. Formally we could write

$$\mathcal{L}(M) = \{0\} \cup \{00x \mid x \in \Sigma^*\}$$

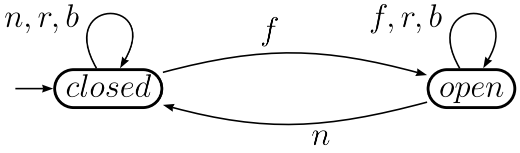
**Example 85.** Now we will show how to design an FSA for an automatic door controller. The controller has to open the door for incoming customers, and not misbehave. A rough specification of it would be

> *If a person is on pad 1 (the front pad) and there's no person on pad 2 (the rear pad), then open the door and stay open until there's no person on either pad 1 or pad 2.*

This can be modelled with two states: *closed* and *open*. So in our automaton, $Q = \{closed, open\}$. Now we need to capture all the combinations of people on pads: these will be the inputs to the system.

- (both) pad 1 and pad 2
- (front) pad 1 and not pad 2
- (rear) not pad 1 and pad 2
- (neither) not pad 1 and not pad 2

We will need 2 sensors, one for each pad, and some external mechanism to convert these two inputs into one of the possibilities. So our alphabet will be $\{b, f, r, n\}$, standing for $\{\mathit{both}, \mathit{front}, \mathit{rear}, \mathit{neither}\}$. Now the task is to define the transition function. This is most easily expressed as a diagram:



Finally, to complete the formal definition, we'd need to specify a start state. The set of final states would be empty, since one doesn't usually want a door controller to freeze the door in any particular position.

*Food for thought*. Should door controllers handle only finite inputs, or should they run forever? Is that even possible, or desired?

The formal definition of the door controller would be

$$M = (\{\mathit{closed}, \mathit{open}\}, \{f, r, n, b\}, \delta, \mathit{closed}, \emptyset)$$

where $\delta$ is defined as

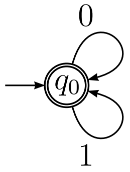
$$\begin{array}{rcl} \delta(\mathit{open}, x) & = & \texttt{if } x = n \texttt{ then } \mathit{closed} \texttt{ else } \mathit{open} \\ \delta(\mathit{closed}, x) & = & \texttt{if } x = f \texttt{ then } \mathit{open} \texttt{ else } \mathit{closed} \end{array}$$

That completes the door controller example. □

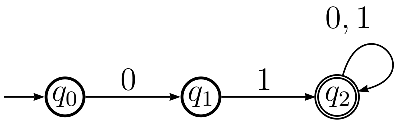
In the course, there are two main questions asked about automata:

- Given a DFA $M$, what is $\mathcal{L}(M)$?
- Given a language $L$, what is a DFA $M$ such that $\mathcal{L}(M) = L$?

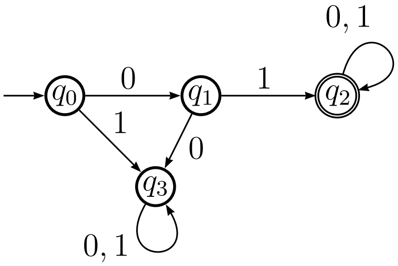
**Example 86.** Give a DFA for recognizing the set of all strings over $\{0,1\}$, *i.e.*, $\{0,1\}^*$. This is also known as the set of all *binary* strings. There is a very simple automaton for this:



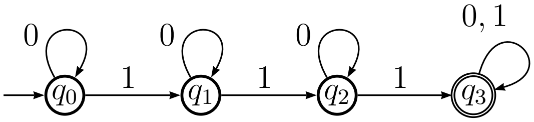
**Example 87.** Give a DFA for recognizing the set of all binary strings beginning with 01. Here's a first attempt (which doesn't quite work):



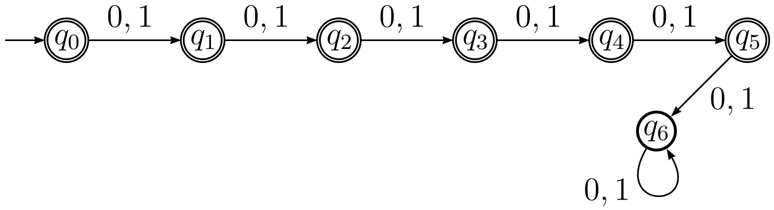
The problem is that this diagram does not describe a DFA: $\delta$ is not total. Here is a fixed version:



**Example 88.** Let $\Sigma = \{0,1\}$ and $L = \{w \mid w \text{ contains at least 3 1s}\}$. Show that $L$ is regular, *i.e.*, give a DFA that recognizes $L$.



**Example 89.** Let $\Sigma = \{0,1\}$ and $L = \{w \mid \mathsf{len}(w) \text{ is at most } 5\}$. Show that $L$ is regular.



### 5.1.2 The regular languages

We've now exercised our intuitions on the definition of a DFA. We should pause to formally define what it means for a machine $M$ to accept a string $w$, the language $\mathcal{L}(M)$ recognized by $M$, and the regular languages. We start by defining the notion of a computation path, which is the trace of an execution of $M$.

**Definition 26** (Computation path)**.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w = w_0 w_1 \dots w_{n-1}$ be a string over $\Sigma$, where each $w_i$ is an element of $\Sigma$. A *computation path*

$$q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{n-1}} q_n$$

is a sequence of states of $M$ labelled with symbols, fully describing the sequence of transitions made by $M$ in processing $w$. Moreover,

- $q_0$ is the start state of $M$
- each $q_i \in Q$
- $\delta(q_i, w_i) = q_{i+1}$ for $0 \leq i < n$

It is important to notice that, for a DFA $M$, there is *only one* computation path for any string. We will soon see other kinds of machines where this isn't true.

**Definition 27** (Language of a DFA)**.** The language defined by a DFA $M$, written $\mathcal{L}(M)$, is the set of strings accepted by $M$. Formally, let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w$ be a string over $\Sigma$.

$$\mathcal{L}(M) = \{w \mid \mathsf{execute}(M, w) \in F\}$$

A DFA $M$ is said to *recognize* language $A$ if $A = \mathcal{L}(M)$.

Now we give a name to the set of all languages that can be recognized by a DFA.

**Definition 28** (Regular languages)**.** A language is *regular* if there is a DFA that recognizes it:

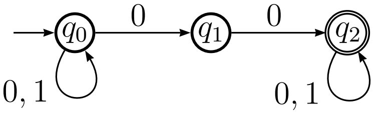
$$\textsf{Regular}(L) = \exists M.\ M \text{ is a DFA and } \mathcal{L}(M) = L$$

The regular languages give a uniform way to relate the languages recognized by DFAs and NFAs, long with the languages generated by regular expressions.

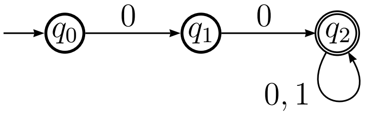
### 5.1.3 More examples

Now we turn to a few slightly harder examples.

**Example 90.** The set of all binary strings having a substring 00 is regular. To show this, we need to construct a DFA that recognizes all and only those strings having 00 as a substring. Here's a natural first try:

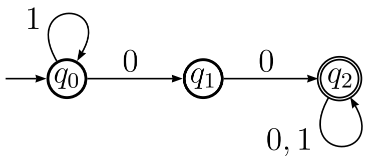


However, this is not a DFA (it is an *NFA*, which we will discuss in the next lecture). A second try can be constructed by trying to implement the following idea: we try to find 00 in the input string by 'shifting along' until a 00 is seen, whereupon we go to a success state. We start with a *preliminary* DFA, that expresses the part of the machine that detects successful input:

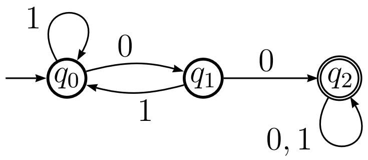


And now we consider, for each state, the moves needed to make the transition function total, *i.e.*, we need to consider all the missing cases.

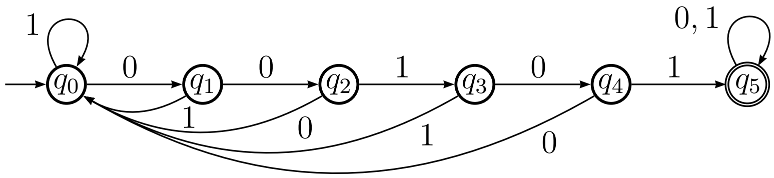
- If we are in $q_0$ and we get a 1, then we should try again, *i.e.*, stay in $q_0$. So $q_0$ is the machine state where it is *looking for a 0*. So the machine should look like



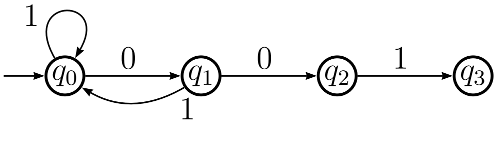
- If we are in $q_1$ and we get a 1, then we should start again, for we have seen a 01. This corresponds to shifting over by 2 in the input string. So the final machine looks like



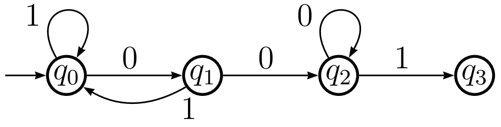
**Example 91.** Give a DFA that recognizes the set of all binary strings having a substring 00101. A straightforward—but incorrect—first attempt is the following:



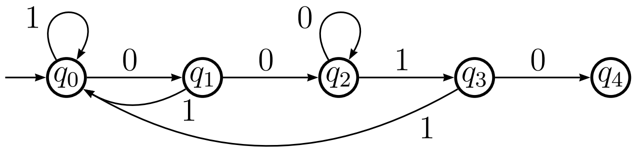
This doesn't work! Consider what happens at $q_2$:



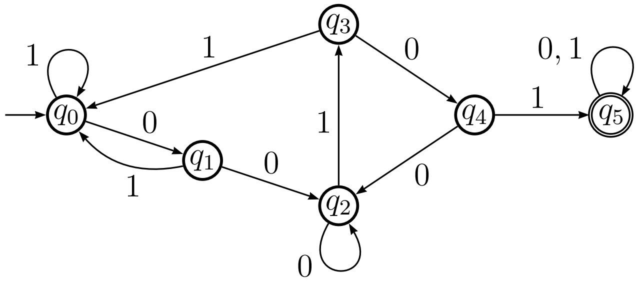
If the machine is in $q_2$, it has seen a 00. If we then get another 0, we could be seeing $0\underline{00101}$. In other words, if the next 3 symbols after 2 or more consecutive 0s are 101, we should accept. Therefore, once, we see 00, we should stay in $q_2$ as long as we see more 0s. Thus we can refine our diagram to



Next, what about $q_3$? When in $q_3$, we've seen something of the form $\ldots 001$ If we now see a 1, we have to restart, as in the original, naive diagram. If we see a 0, we proceed to $q_4$, as in the original.



Now $q_4$. We've seen $\ldots 0010$. If we now see a 1, then we've found our substring, and can accept. Otherwise, we've seen $\ldots 001\underline{00}$, *i.e.*, have seen a 00, therefore should go to $q_2$. This gives the final solution (somewhat rearranged):



## 5.2 Nondeterministic finite-state automata

A *nondeterministic finite-state automaton* (NFA) $N = (Q, \Sigma, \delta, q_0, F)$ is defined in the same way as a DFA except that the following liberalizations are allowed:

- multiple next states
- $\varepsilon$-transitions

### Multiple next states

This means that—in a state $q$ and with symbol $a$—there could be more than one next state to go to, *i.e.*, the value of $\delta(q, a)$ is a *subset* of $Q$. Thus $\delta(q, a) = \{q_1, \ldots, q_k\}$, which means that any one of $q_1, \ldots, q_k$ could be the next state.

There is a *special case*: $\delta(q, a) = \emptyset$. This means that there is *no* next state when the machine is in state $q$ and reading an $a$. How to understand this state of affairs? One way to think of it is that the machine hangs and the input will be rejected. This is equivalent to going into a failure state in a DFA.

### $\varepsilon$-Transitions

In an $\varepsilon$-transition, the tape head doesn't do anything—it doesn't read and it doesn't move. However, the state of the machine can be changed. Formally, the transition function $\delta$ is given the empty string. Thus

$$\delta(q, \varepsilon) = \{q_1, \ldots, q_k\}$$

means that the next state could be one of $q_1, \ldots, q_k$ without consuming the next input symbol. When an NFA executes, it makes transitions as a DFA does. However, after making a transition, it can make as many $\varepsilon$-transitions as are possible.

Formally, all that has changed in the definition of an automaton is $\delta$:

**DFA** $\delta : Q \times \Sigma \to Q$

**NFA** $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$

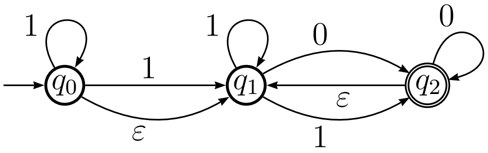
**Note.** Some authors write $\Sigma_\varepsilon$ instead of $\Sigma \cup \{\varepsilon\}$.

Don't let any of this formalism confuse you: it's just a way of saying that $\delta$ delivers a *set* of next states, each of which is a member of $Q$.

**Example 92.** Let $\delta$, the transition function, be given by the following table

|  | 0 | 1 | $\varepsilon$ |
|---|---|---|---|
| $q_0$ | $\emptyset$ | $\{q_0, q_1\}$ | $\{q_1\}$ |
| $q_1$ | $\{q_2\}$ | $\{q_1, q_2\}$ | $\emptyset$ |
| $q_2$ | $\{q2\}$ | $\emptyset$ | $\{q_1\}$ |

Also, let $F = \{q_2\}$. Note that we *must* take account of the possibility of $\varepsilon$ transitions in every state. Also note that each step can lead to one of a set of next states. The state transition diagram for this automaton is



**Note.** In a transition diagram for an NFA, we draw arrows for all transitions except those landing in the empty set (can one *land* in an empty set?).

**Note.** $\delta$ is still a total function, *i.e.*, we have to specify its behaviour in $\varepsilon$, for each state.

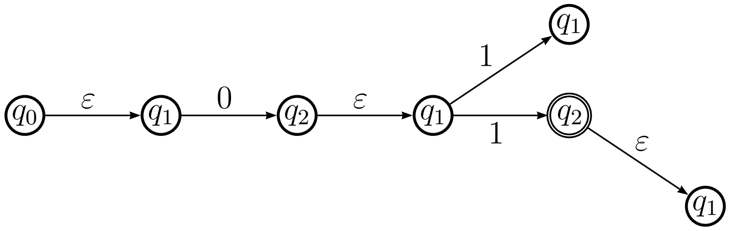
*Question* : Besides $\delta$, what changes when moving from DFA to NFA?
*Answer* : The notion that there is a single computation path for a string, and therefore, the definitions of acceptance and rejection of strings. Consequently, the definition of $\mathcal{L}(N)$, where $N$ is an NFA.

**Example 93.** Giving the input 01 to our example NFA allows 3 computation paths:

- $q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{0} q_2 \xrightarrow{\varepsilon} q_1 \xrightarrow{1} q_1$
- $q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{0} q_2 \xrightarrow{\varepsilon} q_1 \xrightarrow{1} q_2$
- $q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{0} q_2 \xrightarrow{\varepsilon} q_1 \xrightarrow{1} q_2 \xrightarrow{\varepsilon} q_1$

Notice that, in the last path, we can see that even after the input string has been consumed, the machine can still make $\varepsilon$-transitions. Also note that two paths, the first and third, do not end in a final state. The second path is the only one that ends in a final state

In general, the computation paths for input $x$ form a *computation tree*: the root is the start state and the paths branch out to (possibly) different states. For our example, with the input 01, we have the tree



Note that marking $q_2$ as a final state is just a marker to show that a path (the second) ends at that point; of course, the third path continues from that state.

An NFA *accepts* an input $x$ if *at least one path* in the computation tree for $x$ leads to a final state. In our example, 01 is accepted because $q_2$ is a final state.

**Definition 29** (Acceptance by an NFA)**.** Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. *$N$ accepts $w$* if we can write $w$ as $w_1 \cdot w_2 \cdot \ldots \cdot w_n$, where each $w_i$ is a member of $\Sigma \cup \{\varepsilon\}$ and a sequence of states $q_0, \ldots, q_k$ exists, with each $q_i \in Q$ such that the following conditions hold

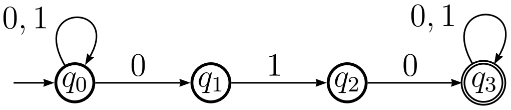
- $q_0$ is the start state of $N$
- $q_k$ is a final state of $N$ ($q_k \in F$)
- $q_{i+1} \in \delta(q_i, w_{i+1})$

As for DFAs, the language recognized by an NFA $N$ is the set of strings accepted by $N$.

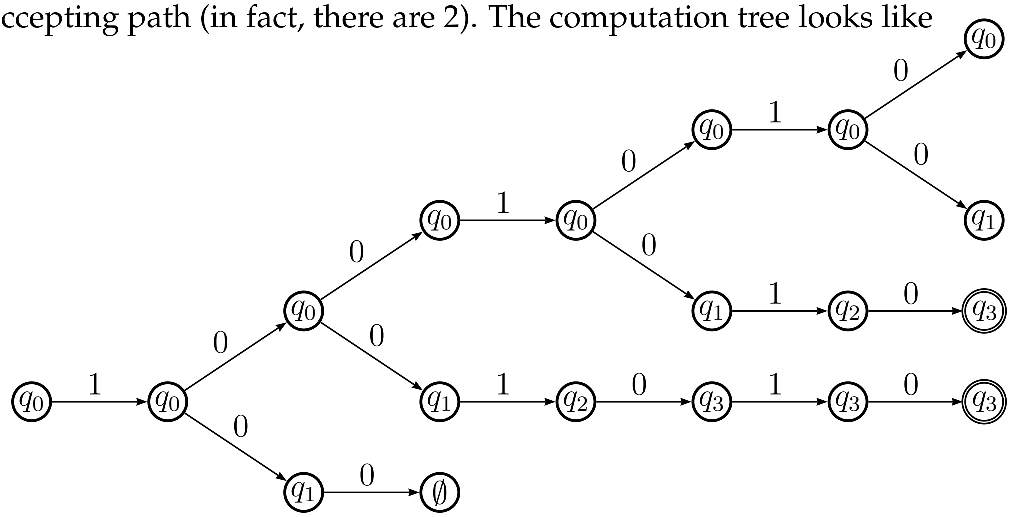
**Definition 30** (Language of an NFA)**.** The language of an NFA $N$ is written $\mathcal{L}(N)$ and defined

$$\mathcal{L}(N) = \{x \mid N \text{ accepts } x\}$$

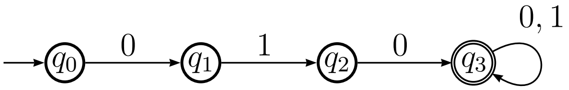
**Example 94.** A diagram for an NFA that accepts all binary string having a substring $010$ is the following:



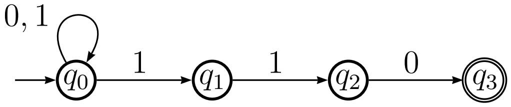
This machine accepts the string 1001010 because there exists at least one accepting path (in fact, there are 2). The computation tree looks like



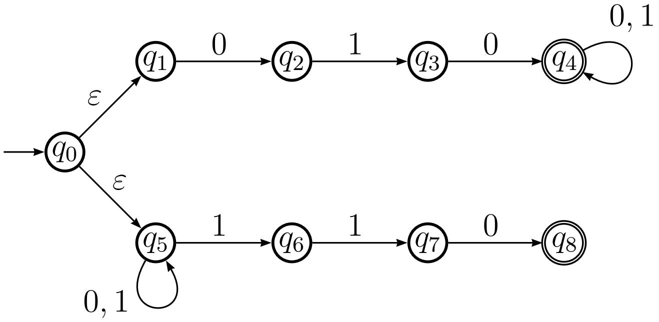
**Example 95.** Design an NFA that accepts the set of binary strings beginning with 010 or ending with 110. The solution to this uses a decomposition strategy: do the two cases separately then join the automata with $\varepsilon$-links. An automaton for the first case is the following



An automaton for the second case is the following



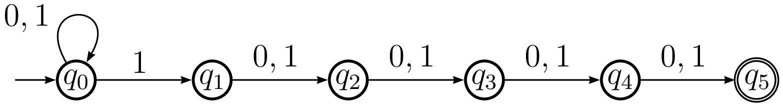
The joint automaton is



**Example 96.** Give an NFA for

$$L = \{x \in \{0,1\}^* \mid \text{ the fifth symbol from the right is } 1\}$$

The diagram for the requested automaton is



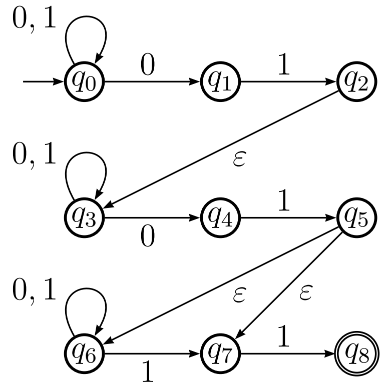
**Note.** There is much non-determinism in this example. Consider state $q_0$, where there are multiple transitions on a 1. Also consider state $q_5$, where there are no outgoing transitions.

This automaton accepts $L$ because

- For any string whose 5th symbol from the right is 1, there exists a sequence of legal transitions leading from $q_0$ to $q_5$.
- For any string whose 5th symbol from the right is 0 (or any string of length up to 4), there is no possible sequence of legal transitions leading from $q_0$ to $q_5$.

**Example 97.** Find an NFA that accepts the set of binary strings with at least 2 occurrences of 01, and which end in 11.

The solution uses $\varepsilon$-moves to connect 3 NFAs together:



**Note.** There is a special case to take care of the input ending in 011; whence the $\varepsilon$-transition from $q_5$ to $q_7$.

## 5.3 Constructions

OK, now we have been introduced to DFAs and NFAs and seen how they accept/reject strings. Now we are going to examine various *constructions* that operate on automata, yielding other automata. It's a way of building automata from components.

### 5.3.1 The product construction

We'll start with the *product* automaton. This creature takes 2 DFAs and delivers the product automaton, also a DFA. The product automaton is a single machine that, conceptually, runs its two component machines *in parallel* on the same input string. At each step of the computation, both machines access the (same) current input symbol, but they make transitions according to their respective $\delta$ functions.

This is easy to understand at a high level, but how do we make this precise? In particular, how can the resulting machine be a DFA? The key idea in solving this requirement is to make the states of the product automaton be *pairs of states* from the component automaton.

**Definition 31** (Product construction)**.** Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be DFAs. Notice that they share the same alphabet. The product of $M_1$ and $M_2$—sometimes written as $M_1 \times M_2$—is $(Q, \Sigma, \delta, q_0, F)$, where

- $Q = Q_1 \times Q_2$. Recall that this is $\{(p, q) \mid p \in Q_1 \land q \in Q2\}$ or informally as *all possible pairings of states in* $Q_1$ *with states in* $Q_2$. The size of $Q$ is the product of the sizes of $Q_1$ and $Q_2$.

- $\Sigma$ is unchanged. We require that $M_1$ and $M_2$ have the same input alphabet. *Question* : If they don't, what could we do?

- $\delta$ is defined by its behaviour on pairs of states: the transition is expressed by $\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$.

- $q_0 = (q_1, q_2)$, where $q_1$ is the start state for $M_1$ and $q_2$ is the start state for $M_2$.

- $F$ can be built in 2 ways. When the product automaton is run on an input string $w$, it eventually ends up in a state $(p, q)$, meaning that $p$ is the state $M_1$ would end up in on $w$, and similarly $q$ is the state $M_2$ would end up in on $w$. The choices are:

  **Union.** $(p, q)$ is an accept state if $p$ is an accept state for $M_1$, *or* if $q$ is an accept state for $M_2$.

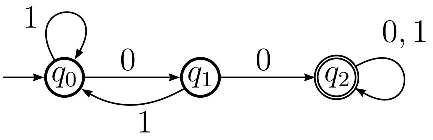
  **Intersection.** $(p, q)$ is an accept state if $p$ is an accept state for $M_1$, *and* if $q$ is an accept state for $M_2$.

  We will take up these later.

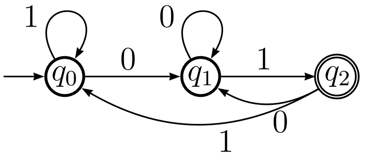
**Example 98.** Give a DFA that recognizes the set of all binary strings having a substring 00 or ending in 01. To answer this challenge, we notice that this can be regarded as the *union* of two languages:

$$\{x \mid x \text{ has a substring } 00\} \quad \cup \quad \{y \mid y \text{ ends in } 01\}$$

Let's build 2 automata separately and then use the product construction to join them. The first DFA, call it $M_1$, is



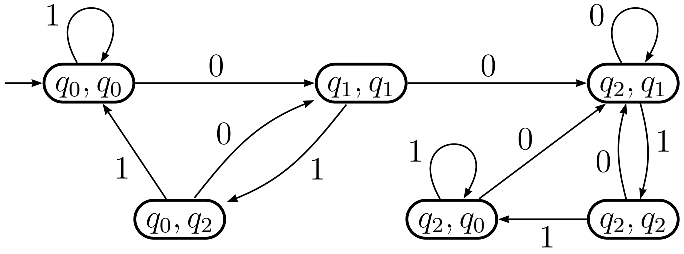
The second, call it $M_2$, is



The next thing to do is to construct the state space of the product machine, and use that to figure out $\delta$. The following table gives the details:

| $Q_1 \times Q_2$ | 0 | 1 |
|---|---|---|
| $(q_0\ q_0)$ | $(q_1, q_1)$ | $(q_0, q_0)$ |
| $(q_0\ q_1)$ | $(q_1, q_1)$ | $(q_0, q_2)$ |
| $(q_0\ q_2)$ | $(q_1, q_1)$ | $(q_0, q_0)$ |
| $(q_1\ q_0)$ | $(q_2, q_1)$ | $(q_0, q_0)$ |
| $(q_1\ q_1)$ | $(q_2, q_1)$ | $(q_0, q_2)$ |
| $(q_1\ q_2)$ | $(q_2, q_1)$ | $(q_0, q_0)$ |
| $(q_2\ q_0)$ | $(q_2, q_1)$ | $(q_2, q_0)$ |
| $(q_2\ q_1)$ | $(q_2, q_1)$ | $(q_2, q_2)$ |
| $(q_2\ q_2)$ | $(q_2, q_1)$ | $(q_2, q_0)$ |

This is of course easy to write out, once you get used to it (a bit mindless though). Now there are several of the combined states that aren't *reachable* from the start state and can be pruned. The following is a diagram of the result.



The final states of the automaton are $\{(q_0, \underline{q_2}), (\underline{q_2}, q_1), (\underline{q_2}, q_0), (\underline{q_2}, \underline{q_2})\}$ (underlined states are the final states in the component automata).

### 5.3.2 Closure under union

**Theorem 9.** *If $A$ and $B$ are regular sets, then $A \cup B$ is a regular set.*

*Proof.* Let $M_1$ be a DFA recognizing $A$ and $M_2$ be a DFA recognizing $B$. Then $M_1 \times M_2$ is a DFA recognizing $\mathcal{L}(M_1) \cup \mathcal{L}(M_2) = A \cup B$, provided that the final states of $M_1 \times M_2$ are those where one or the other components is a final state, *i.e.*, we require that the final states of $M_1 \times M_2$ are a subset of $(F_1 \times Q_2) \cup (Q_1 \times F_2)$. □

It must be admitted that this 'proof' is really just a construction: it tells how to build the automaton that recognizes $A \cup B$. The intuitive reason why the construction works is that we run $M_1$ and $M_2$ in parallel on the input string, accepting when either would accept.

### 5.3.3 Closure under intersection

**Theorem 10.** *If $A$ and $B$ are regular sets, then $A \cap B$ is a regular set.*

*Proof.* Use the product construction, as for union, but require that the final states of $M_1 \times M_2$ are $\{(p, q) \mid p \in F_1 \wedge q \in F_2\}$, where $F_1$ and $F_2$ are the final states of $M_1$ and $M_2$, respectively. In other words, accept an input to $M_1 \times M_2$ just when both $M_1$ and $M_2$ would accept it separately. □

*Remark.* If you see a specification of the form
*show* $\{x \mid \ldots \wedge \ldots\}$ *is regular* or
*show* $\{x \mid \ldots \vee \ldots\}$ *is regular*,
you should consider building a product automaton, in the intersection flavour (first spec.) or the union flavour (second spec).

### 5.3.4 Closure under complement

**Theorem 11.** *If $A$ is a regular set, then $\overline{A}$ is a regular set.*

*Proof.* Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing $A$. So $M$ accepts all strings in $A$ and rejects all others. Thus a DFA recognizing $\overline{A}$ is obtained by switching the final and non-final states of $M$, *i.e.*, the desired machine is $M' = (Q, \Sigma, \delta, q_0, Q - F)$. Note that $M'$ recognizes $\Sigma^* - \mathcal{L}(M)$. □

### 5.3.5 Closure under concatenation

If we can build a machine $M_1$ to recognize $A$ and a machine $M_2$ to recognize $B$, then we should be able to recognize language $A \cdot B$ by running $M_1$ on a string $w \in A \cdot B$ until it hits an accept state, and then running $M_2$ on the remainder of $w$. In other words, we somehow want to 'wire' the two machines together in series. To achieve this, we need to do several things:

- Connect the final states of $M_1$ to the start state of $M_2$. We will have to use $\varepsilon$-transitions to implement this, because reading a symbol off the input to make the jump from $M_1$ to $M_2$ will wreck things. (Why?)
- Make the start state for the combined machine be $q_{0_1}$.
- Make the final states for the combined machine be $F_2$.

The following makes this precise.

**Theorem 12.** *If $A$ and $B$ are regular sets, then $A \cdot B$ is a regular set.*

*Proof.* Let $M_1 = (Q_1, \Sigma, \delta_1, q_{0_1}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{0_2}, F_2)$ be DFAs that recognize $A$ and $B$, respectively. An automaton[2] for recognizing $A \cdot B$ is given by $M = (Q, \Sigma, \delta, q_0, F)$, where

- $Q = Q_1 \cup Q_2$
- $\Sigma$ is unchanged (assumed to be the same for both machines). This could be liberalized so that $\Sigma = \Sigma_1 \cup \Sigma_2$, but it would mean that $\delta$ would need extra modifications.
- $q_0 = q_{0_1}$
- $F = F_2$

[2] An NFA, in fact.

- $\delta(q, a)$ is defined by cases, as to whether it is operating 'in' $M_1$, transitioning between $M_1$ and $M_2$, or operating 'in' $M_2$:
  - $\delta(q, a) = \{\delta_1(q, a)\}$ when $q \in Q_1$ and $\delta_1(q, a) \notin F_1$.
  - $\delta(q, a) = \{q_{0_2}\} \cup \{\delta_1(q, a)\}$ when $q \in Q_1$ and $\delta_1(q, a) \in F_1$. Thus, when $M_1$ would enter one of its final states, it can stay in that state or make an $\varepsilon$-transition to the start state of $M_2$.
  - $\delta(q, a) = \delta_2(q, a)$ if $q \in Q_2$.

□

### 5.3.6 Closure under Kleene star

If we have a machine that recognizes $A$, then we should be able to build a machine that recognizes $A^*$ by making a loop of some sort. The details of this are a little bit tricky since the obvious way of doing this—simply making $\varepsilon$-transitions from the final states to the start state—doesn't work.

**Theorem 13.** *If $A$ is a regular set, then $A^*$ is a regular set.*

*Proof.* Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing $A$. An NFA $N$ recognizing $A^*$ is obtained by

- Adding a new start state $q_s$, with an $\varepsilon$-move to $q_0$. Since $\varepsilon \in A^*$, $q_s$ must be an accept state.

- Adding $\varepsilon$-moves from each final state to $q_0$.

The result is that $N$ accepts $\varepsilon$; it accepts $x$ if $M$ accepts $x$, and it accepts $w$ if $w = w_1 \cdot \ldots \cdot w_n$ such that $M$ accepts each $w_i$. So $N$ recognizes $A^*$.

Formally, $N = (Q \cup \{q_s\}, \Sigma, \delta', q_s, F \cup \{q_s\})$, where $\delta'$ is defined by cases:

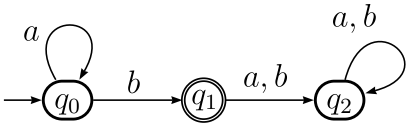
- Transitions from $q_s$. Thus $\delta'(q_s, \varepsilon) = \{q_0\}$ and $\delta'(q_s, a) = \emptyset$, for $a \neq \varepsilon$.

- Old transitions. $\delta'(q, a) = \delta(q, a)$, provided $\delta(q, a) \notin F$.

- $\varepsilon$-transitions from $F$ to $q_0$. $\delta'(q, a) = \delta(q, a) \cup \{q_0\}$, when $\delta(q, a) \in F$.

□

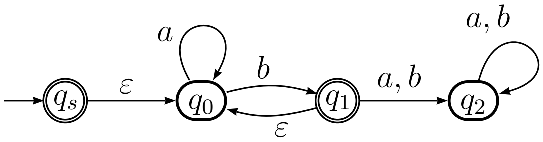
**Example 99.** Let $M$ be given by the following DFA:



A bit of thought reveals that $\mathcal{L}(M) = \{a^n b \mid n \geq 0\}$, and that

$$(\mathcal{L}(M))^* = \{\varepsilon\} \cup \{\text{all strings ending in b}\}.$$

If we apply the construction, we obtain the following NFA for $(\mathcal{L}(M))^*$.



### 5.3.7 The subset construction

Now we discuss the *subset* construction, which was invented by Dana Scott and Michael Rabin[3] in order to show that the expressive power of NFAs and DFAs is the same, *i.e.*, they both recognize the regular languages. The essential idea is that the subset construction can be used to map any NFA $N$ to a DFA $M$ such that $\mathcal{L}(N) = \mathcal{L}(M)$.

The underlying insight of the subset construction is to have the transition function of the corresponding DFA $M$ work over a set of states, rather than the single state used by the transition function of the NFA $N$:

| | $\delta$ |
|---|---|
| NFA | $\Sigma \cup \{\varepsilon\} \times Q \to 2^Q$ |
| DFA | $\Sigma \times 2^Q \to 2^Q$ |

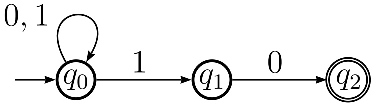
In other words, the NFA $N$ is always in a single state and can have multiple successor states for symbol $a$ via $\delta$. In contrast, the DFA $M$ is always in a

[3] They shared a Turing award for this; however, both researchers are famous for much other work as well.

set (possibly empty) of states and moves into a set of successor states via $\delta'$, which is defined in terms of $\delta$. This is formalized as follows:

$$\delta'(\{q_1, \ldots, q_k\}, a) = \delta(q_1, a) \cup \ldots \cup \delta(q_k, a).$$

**Example 100.** Let's consider the NFA $N$ given by the diagram



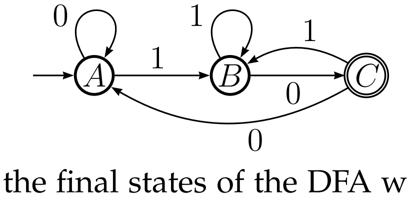
$N$ evidently accepts the language $\{x10 \mid x \in \{0,1\}^*\}$. The subset construction for $N$ proceeds by constructing a transition function over all subsets of the states of $N$. Thus we need to consider

$$\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}$$

as possible states of the DFA $M$ to be constructed. The following table describes the transition function for $M$.

| $Q$ | 0 | 1 | |
|---|---|---|---|
| $\emptyset$ | $\emptyset$ | $\emptyset$ | (unreachable) |
| $\{q_0\}$ | $\{q_0\}$ | $\{q_0, q_1\}$ | (**reachable**) |
| $\{q_1\}$ | $\{q_2\}$ | $\emptyset$ | (unreachable) |
| $\{q_2\}$ | $\emptyset$ | $\emptyset$ | (unreachable) |
| $\{q_0, q_1\}$ | $\{q_0, q_2\}$ | $\{q_0, q_1\}$ | (**reachable**) |
| $\{q_0, q_2\}$ | $\{q_0\}$ | $\{q_0, q_1\}$ | (**reachable**) |
| $\{q_1, q_2\}$ | $\{q_2\}$ | $\emptyset$ | (unreachable) |
| $\{q_0, q_1, q_2\}$ | $\{q_0, q_2\}$ | $\{q_0, q_1\}$ | (unreachable) |

And here's the diagram. Unreachable states have been deleted. State $A = \{q_0\}$ and $B = \{q_0, q_1\}$ and $C = \{q_0, q_2\}$.



Note that the final states of the DFA will be those that contain at least one final state of the NFA.

So by making the states of the DFA be sets of states of the NFA, we seem to get what we want: the DFA will accept just in case the NFA would accept. This apparently gives us the best of both characterizations: the expressive power of NFAs, coupled with the straightforward executability of DFAs. However, there is a flaw: some NFAs map to DFAs with exponentially more states. A class of examples with this property are those expressed as

> *Construct a DFA accepting the set of all binary strings in which the* n*th symbol from the right is 0.*

Also, we have not given a complete treatment: we still have to account for $\varepsilon$-transitions, via $\varepsilon$-closure.

### $\varepsilon$-Closure

Don't be confused by the terminology: $\varepsilon$-closure has noting to do with closure of regular languages under $\cap, \cup$, *etc.*

The idea of $\varepsilon$-closure is the following: when moving from a set of states $\mathcal{S}_i$ to a a set of states $\mathcal{S}_{i+1}$, we have to take account of all $\varepsilon$-moves that could be made after the transition. Why do we have to do that? Because the DFA is over the alphabet $\Sigma$, instead of $\Sigma \cup \{\varepsilon\}$, so we have to squeeze out all the $\varepsilon$-moves. Thus we define, for a set of states $Q$,

$$\mathsf{E}(Q) = \{q \mid q \text{ can be reached from a state in } Q \text{ by } 0 \text{ or more } \varepsilon\text{−moves}\}$$

and then a step in the DFA, originally

$$\delta'(\{q_1, \ldots, q_k\}, a) = \delta(q_1, a) \cup \ldots \cup \delta(q_k, a)$$

where $\delta$ is the transition function of the original NFA, instead becomes

$$\delta'(\{q_1, \ldots, q_k\}, a) = \mathsf{E}(\delta(q_1, a) \cup \ldots \cup \delta(q_k, a))$$

**Note.** We make a transition and then chase any $\varepsilon$-steps. But what about the start state? We need to chase any $\varepsilon$-steps from $q_0$ *before* we start making any transitions. So $q_0' = \mathsf{E}\{q_0\}$. Putting all this together gives the formal definition of the subset construction.

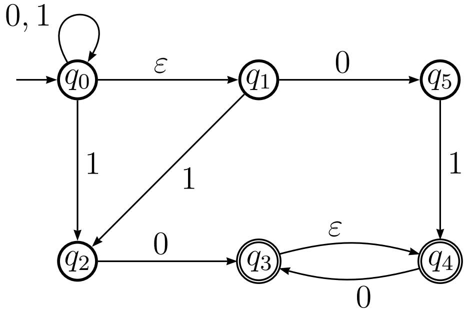
**Definition 32** (Subset construction)**.** Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. The DFA $M = (Q', \Sigma, \delta', q_0', F')$ given by the subset construction is specified by

- $Q' = 2^Q$
- $q'_0 = \mathsf{E}\{q_0\}$
- $$\begin{aligned} \delta'(\{q_1, \ldots, q_k\}, a) &= \mathsf{E}(\delta(q_1, a) \cup \ldots \cup \delta(q_k, a)) \\ &= \mathsf{E}(\delta(q_1, a)) \cup \ldots \cup \mathsf{E}(\delta(q_k, a)) \end{aligned}$$
- $F' = \{S \in 2^Q \mid \exists q \in S \wedge q \in F\}$

The essence of the argument for correctness of the subset construction amounts to noticing that the generated DFA mimicks the transition behaviour of the NFA and accepts and rejects strings exactly as the NFA does.

**Theorem 14** (Correctness of subset construction)**.** *If DFA $M$ is derived by applying the subset construction to NFA $N$, then $\mathcal{L}(M) = \mathcal{L}(N)$.*

**Example 101.** Convert the following NFA to an equivalent DFA.



The *very* slow way to do this would be to mechanically construct a table for the transition function, with the left column having all $2^6 = 64$ subsets of $\{q_0, q_1, q_2, q_3, q_4, q_5\}$:

| *states* | 0 | 1 |
|---|---|---|
| $\emptyset$ | | |
| $\{q_0\}$ | | |
| $\vdots$ | | |
| $\{q_5\}$ | | |
| $\{q_0, q_1\}$ | | |
| $\vdots$ | | |
| $\{q_0, q_1, q_2, q_3, q_4, q_5\}$ | | |

But that would lead to madness. Instead we should build the table in an *on-the-fly* manner, wherein we only write down the transitions for the *reachable* states, the ones we could actually get to by following transitions from the start state. First, we need to decide on the start state: it is not $\{q_0\}$! We have to take the $\varepsilon$-closure of $\{q_0\}$:

$$\mathsf{E}\{q_0\} = \{q_0, q_1\}$$

In the following, it also helps to name the reached state sets, for concision.

| $states$ | 0 | 1 |
|---|---|---|
| $A = \{q_0, q_1\}$ | $\{q_0, q_1, q_5\}$ | $\{q_0, q_1, q_2\}$ |
| $B = \{q_0, q_1, q_5\}$ | $B$ | $D$ |
| $C = \{q_0, q_1, q_2\}$ | $E$ | $C$ |
| $D = \{q_0, q_1, q_2, q_4\}$ | $E$ | $C$ |
| $E = \{q_0, q_1, q_3, q_4, q_5\}$ | $E$ | $D$ |

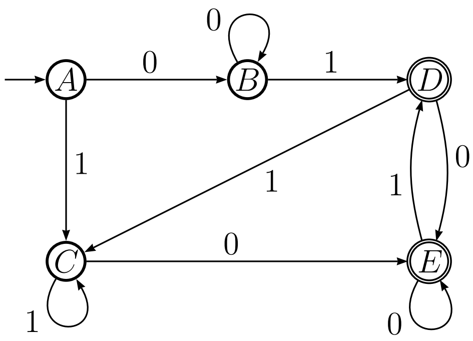
So for example, if we are in state $C$, the set of states we could be in after a $0$ on the input are:

$$\begin{array}{rcl} \mathsf{E}(\delta(q_0, 0)) \cup \mathsf{E}(\delta(q_1, 0)) \cup \mathsf{E}(\delta(q_2, 0)) & = & \{q_0, q_1\} \cup \{q_5\} \cup \{q_3, q_4\} \\ & = & \{q_0, q_1, q_3, q_4, q_5\} \\ & = & E \end{array}$$

Similarly, if we are in state $C$ and see a $1$, the set of states we could be in are:

$$\begin{array}{rcl} \mathsf{E}(\delta(q_0, 1)) \cup \mathsf{E}(\delta(q_1, 1)) \cup \mathsf{E}(\delta(q_2, 1)) & = & \{q_0, q_1\} \cup \{q_2\} \cup \emptyset \\ & = & \{q_0, q_1, q_2\} \\ & = & C \end{array}$$

A diagram of this DFA is



**Summary**

For every DFA, there's a corresponding (trivial) NFA. For every NFA, there's an equivalent DFA, via the subset construction. So the 2 models, apparently quite different, have the same power (in terms of the languages they accept). But notice the cost of 'compiling away' the non-determinism: the number of states in a DFA derived from the subset construction can be exponentially larger than in the orignal. Implementability has its price!

## 5.4 Regular Expressions

The *regular expressions* are another formal model of regular languages. Unlike automata, these are essentially given by bestowing a *syntax* on the regular languages and the operations they are closed under.

**Definition 33** (Syntax of regular expressions)**.** The set of regular expressions $\mathcal{R}$ formed from alphabet $\Sigma$ is the following:

- $a \in \mathcal{R}$, if $a \in \Sigma$
- $\varepsilon \in \mathcal{R}$
- $\emptyset \in \mathcal{R}$
- $r_1 + r_2 \in \mathcal{R}$, if $r_1 \in \mathcal{R} \wedge r_2 \in \mathcal{R}$
- $r_1 \cdot r_2 \in \mathcal{R}$, if $r_1 \in \mathcal{R} \wedge r_2 \in \mathcal{R}$
- $r^* \in \mathcal{R}$, if $r \in \mathcal{R}$
- Nothing else is in $\mathcal{R}$

*Remark.* This is an *inductive definition* of a set $\mathcal{R}$—the set is 'initialized' to have $\varepsilon$ and $\emptyset$ and all elements of $\Sigma$. Then we use the closure operations to build the rest of the (infinite, usually) set $\mathcal{R}$. The final clause disallows other random things being in the set.

**Note.** Regular expressions are *syntax trees* used to denote languages. The semantics, or meaning, of a regular expression is thus a set of strings, *i.e.*, a *language*.

**Definition 34** (Semantics of regular expressions)**.** The meaning of a regular expression $r$, written $\mathcal{L}(r)$ is defined as follows:

$$\begin{array}{lcl}
\mathcal{L}(a) & = & \{a\},\ \text{for } a \in \Sigma \\
\mathcal{L}(\varepsilon) & = & \{\varepsilon\} \\
\mathcal{L}(\emptyset) & = & \emptyset \\
\mathcal{L}(r_1 + r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1 \cdot r_2) & = & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(r^*) & = & \mathcal{L}(r)^*
\end{array}$$

Note the overloading. The occurrence of $\cdot$ and $^*$ on the right hand side of the equations are operations on languages, while on the left hand side, they are nodes in a tree structure.

*Convention.* For better readability, difference precedences are given to the infix and postfix operators. Also, we will generally omit the concatenation operator, just treating adjacent regular expressions as being concatenated. Thus we let $^*$ bind more strongly than $\cdot$, and that binds more strongly than the infix $+$ operator. Parentheses can be used to say exactly what you want. Thus

$$r_1 + r_2{r_3}^* = r_1 + r_2 \cdot {r_3}^* = r_1 + (r_2 \cdot ({r_3}^*))$$

Since the operations of $+$ and $\cdot$ are associative, bracketing doesn't matter in expressions like

$$a + b + c + d \qquad \text{and} \qquad abcd.$$

Yet more notation:

- We can use $\Sigma$ to stand for any member of $\Sigma$. That is, an occurrence of $\Sigma$ in a regular expression is an abbreviation for the regular expression $a_1 + \ldots + a_n$, where $\Sigma = \{a_1, \ldots, a_n\}$.
- $r^+ = rr^*$.

**Example 102** (Floating point constants)**.** Suppose [4]

$$\Sigma = \{\underline{+}, -\} \cup D \cup \{.\}$$

[4] We will underline the 'plus sign' $\underline{+}$ to distinguish it from the $+$ used to build the regular expression.

where $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Then

$$(\underline{+} \ + \ - \ + \ \varepsilon) \cdot (D^+ \ + \ D^+.D^* \ + \ D^*.D^+)$$

is a concise description of a simple class of floating point constants for a programming language. Examples of such constants are: $+3,\ -3.2,\ -.235$.

**Example 103.** Give a regular expression for the binary representation of the numbers which are powers of 4:

$$\{4^0, 4^1, 4^2, \ldots\} = \{1, 4, 16, 64, 256, \ldots\}$$

Merely transcribing to binary gives us the important clue we need:

$$\{1, 100, 10000, 1000000, \ldots\}$$

The regular expression generating this language is $1(00)^*$.

**Example 104.** Give a regular expression for the set of binary strings which have *at least* one occurrence of 001. One answer is

$$(0+1)^*001(0+1)^* \qquad \text{or} \qquad \Sigma^*001\Sigma^*$$

**Example 105.** Give a regular expression for the set of binary strings which have *no* occurrence of 001. This example is much harder, since the problem is phrased negatively. In fact, this is an instance where it is easier to build an automaton for recognizing the given set:

- Build an NFA for recognizing any string where 001 occurs. This is easy.
- Convert to a DFA. We know how to do this (subset construction).
- Complement the resulting automaton.[5]

However, we are required to come up with a regular expression. How to start? First, note that a string $w$ in the set can have no occurrence of $00$, unless $w$ is a member of the set denoted by $000^*$. The set of binary strings having no occurrence of $00$ and ending in $1$ is

$$(01+1)^*$$

And now we can append any number of $0$s to this and get the specified set:

$$(01+1)^*0^*$$

[5] Note that directly complementing the NFA won't work in general.

### 5.4.1 Equalities for regular expressions

The following equalities are useful when manipulating regular expressions. They should mostly be familiar and can be proved simply by reducing to the meaning in languages and using the techniques and theorems we have already seen.

$$
\begin{aligned}
&r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3\\
&r_1 + r_2 = r_2 + r_1\\
&r + r = r\\
&r + \emptyset = r\\
&\varepsilon r = r = r\varepsilon\\
&\emptyset r = \emptyset = r\emptyset\\
&\emptyset^* = \varepsilon\\
&r_1(r_2r_3) = (r_1r_2)r_3\\
&r_1(r_2 + r_3) = r_1r_2 + r_1r_3\\
&(r_1 + r_2)r_3 = r_1r_3 + r_2r_3\\
&\varepsilon + rr^* = r^*\\
&(\varepsilon + r)^* = r^*\\
&rr^* = r^*r\\
&r^*r^* = r^*\\
&r^{**} = r^*\\
&(r_1r_2)^*r_1 = r_1(r_2r_1)^*\\
&({r_1}^*r_2)^*{r_1}^* = (r_1 + r_2)^*
\end{aligned}
$$

**Example 106.** In the description of regular expressions, $\varepsilon$ is superfluous. The reason is that $\varepsilon = \emptyset^*$, since

$$
\begin{aligned}
\mathcal{L}(\varepsilon) = \{\varepsilon\} &= \mathcal{L}(\emptyset^*)\\
&= \mathcal{L}(\emptyset)^*\\
&= \emptyset^*\\
&= \emptyset^0 \cup \emptyset^1 \cup \emptyset^2 \cup \ldots\\
&= \{\varepsilon\} \cup \emptyset \cup \emptyset \cup \ldots\\
&= \{\varepsilon\}
\end{aligned}
$$

**Example 107.** Simplify the following regular expression: $(00)^*0 + (00)^*$. This is perhaps most easily seen by unrolling the subexpressions a few times:

$$(00)^*0 = \{0^1, 0^3, 0^5, \ldots\} = \{0^n \mid n \text{ is odd}\}$$

and

$$(00)^* = \{\varepsilon, 0^2, 0^4, 0^6, \ldots\} = \{0^n \mid n \text{ is even}\}$$

Thus $(00)^*0 + (00)^* = 0^*$.

**Example 108.** Simplify the following regular expression: $(0+1)(\varepsilon + 00)^+ + (0+1)$. By distributing $\cdot$ over $+$, we have

$$0(\varepsilon + 00)^+ + 1(\varepsilon + 00)^+ + (0+1)$$

We can use the lemma $(\varepsilon + r)^+ = r^*$ to get

$$0(00)^* + 1(00)^* + 0 + 1$$

but $0$ is already in $0(00)^*$, and $1$ is already in $1(00)^*$, so we are left with

$$0(00)^* + 1(00)^* \qquad \text{or} \qquad (0+1)(00)^*.$$

**Example 109.** Simplify the following regular expression: $(0 + \varepsilon)0^*1$.

$$\begin{array}{rcl} (0+\varepsilon)0^*1 & = & (0+\varepsilon)(0^*1) \\ & = & 0^+1 + 0^*1 \\ & = & 0^*1. \end{array}$$

**Example 110.** Show that $(0^2 + 0^3)^* = (0^2 0^*)^*$. Examining the left hand side, we have

$$\begin{array}{rcl} (0^2 + 0^3)^* & = & \{\varepsilon, 0^2, 0^3, 0^4, \ldots\} \\ & = & 0^* - \{0\}. \end{array}$$

On the right hand side, we have

$$\begin{array}{rcl} (0^2 0^*)^* & = & (000^*)^* \\ & = & \{\varepsilon\} \cup \{00, 000, 0^4, 0^5, \ldots\} \\ & = & \{\varepsilon\} \cup \{0^{k+2} \mid 0 \leq k\} \\ & = & 0^* - \{0\}. \end{array}$$

**Example 111.** Prove the identity $(0+1)^* = (1^*(0+\varepsilon)1^*)^*$ using the algebraic identities. We will work on the rhs, underlining subexpressions about to be changed.

$$\begin{aligned}
(0+1)^* &= (\underbrace{1^*(0+\varepsilon)1^*})^* && \\
&= (1^*01^* + \underbrace{1^*1^*})^* && a^*a^* = a^* \\
&= \underbrace{(1^*01^* + 1^*)^*} && a+b = b+a \\
&= (\underbrace{1^*}_{a} + \underbrace{1^*01^*}_{b})^* && (a+b)^* = (a^*b)^*a^* \\
&= (\underbrace{1^{**}1^*}01^*)^*\underbrace{1^{**}} && a^{**} = a^* = a^*a^* \\
&= (\underbrace{1^*}_{a}\underbrace{01^*}_{b})^*\underbrace{1^*}_{a} && (ab)^*a = a(ba)^* \\
&= 1^*(0\underbrace{1^*1^*})^* && a^*a^* = a^* \\
&= \underbrace{1^*}_{a}(\underbrace{0}_{b}\underbrace{1^*}_{a})^* && a(ba)^* = (ab)^*a \\
&= (\underbrace{1^*}_{a}\underbrace{0}_{b})^*\underbrace{1^*}_{a} && (a^*b)^*a^* = (a+b)^* \\
&= (0+1)^* &&
\end{aligned}$$

### 5.4.2 From regular expressions to NFAs

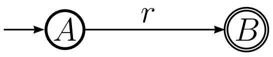
We have now seen 3 basic models of computation: DFA, NFA, and regular expressions. These are all equivalent, in that they all recognize the regular languages. We have seen the equivalence of DFAs and NFAs, which is proved by showing how a DFA can be mapped into an NFA (trivial), and *vice versa* (the subset construction). We are now going to fill in the rest of the picture by showing how to translate

- regular expressions into equivalent NFAs; and

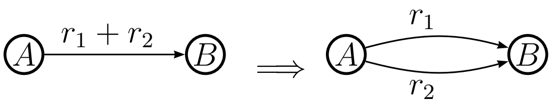
- DFAs into equivalent regular expressions

The translation of a regular expression to an NFA proceeds by exhaustive application of the following rules:

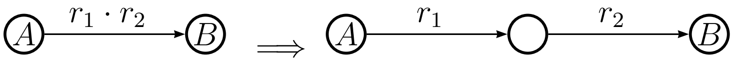
**(init)**



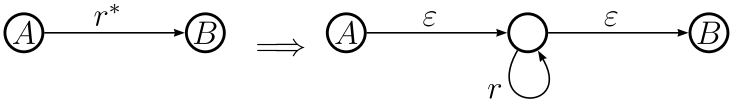
**(plus)**



**(concat)**



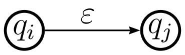
**(star)**



The *init* rule is used to start the process off: it sets the regular expression as a label between the start and accept states. The idea behind the rule applications is to iteratively replace each regular expression by a fragment of automaton that implements it.

Application of these rules, the *star* rule especially, can result in many useless $\varepsilon$-transitions. There is a complicated rule for eliminating these, which can be applied only after all the other rules can no longer be applied.

**Definition 35** (Redundant $\varepsilon$-edge elimination)**.** An edge



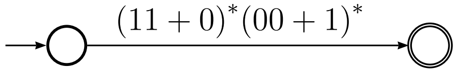
can be shrunk to a single node by the following rule:

- If the edge labelled with $\varepsilon$ is the only edge leaving $q_i$ then $q_i$ can be replaced by $q_j$. If $q_i$ is the start node, then $q_j$ becomes the new start state.

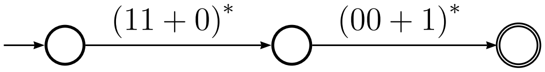
- If the edge labelled $\varepsilon$ is the only edge entering $q_j$ then $q_j$ can be replaced by $q_i$. If $q_j$ is a final state, then $q_i$ becomes a final state.

**Example 112.** Find an equivalent NFA for $(11+0)^*(00+1)^*$.

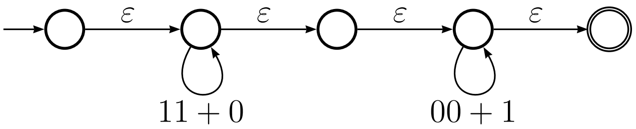
**init**



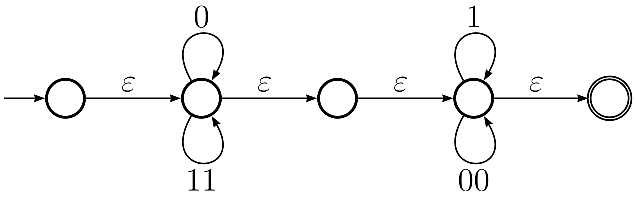
**concat**



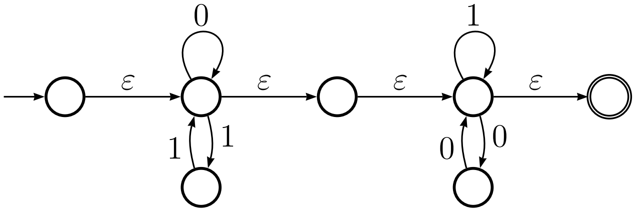
**star (twice)**



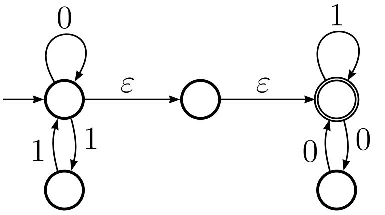
**plus (twice)**



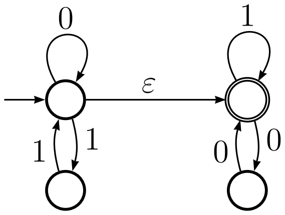
**concat (twice)**



That ends the elaboration of the regular expression into the corresponding NFA. All that remains is to eliminate redundant $\varepsilon$-transitions. The $\varepsilon$ transition from the start state can be dispensed with, since it is a unique out-edge from a non-final node; similarly, the $\varepsilon$ transition into the final state can be eliminated because it is a unique in-edge to a non-initial node. This yields



We are not yet done. One of the two middle $\varepsilon$-transitions can be eliminated—in fact the middle node has a unique in-edge *and* a unique out-edge—so the middle state can be dropped.



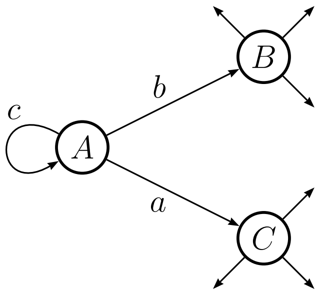
However, the remaining $\varepsilon$-edge can not be deleted; doing so would change the language to $(0 + 1)^*$. Thus application of the $\varepsilon$-edge elimination rule must occur one edge at a time.

### 5.4.3 From DFA to regular expression

It is possible to convert a DFA to an equivalent regular expression. There are several approaches; the one we will take is based on representing the automaton as a system of equations and then using Arden's lemma to solve the equations.

#### Representing an automaton as a system of equations

The basis of this representation is to think of a state in the machine as a regular expression representing the strings that would be accepted by running the machine on them from that state. Thus from state $A$ in the following fragment of a machine



any string that will eventually be accepted will be of one of the forms

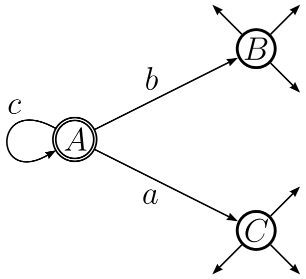
- $b \cdot t_1$, where $t_1$ is a string accepted from state $B$.
- $a \cdot t_2$, where $t_2$ is a string accepted from state $C$.
- $c \cdot t_3$, where $t_3$ is a string accepted from state $A$.

This can be captured in the equation

$$A = cA + bB + aC$$

the right hand size of which looks very much like a regular expression, except for the occurrences of the variables $A$, $B$, and $C$. Indeed, the equation solving process eliminates these variables so that the final expression is a *bona fide* regular expression. The goal, of course, is to solve for the variable representing the start state.

Accept states are somewhat special since the machine, if run from them, would accept the empty string. This has to be reflected in the equation. Thus



is represented by the equation

$$A = cA + bB + aC + \varepsilon$$

### Using Arden's Lemma to solve a system of equations

An important theorem about languages, proved earlier in these notes, is the following:

**Theorem 15** (Arden's Lemma). *Assume that $A$ and $B$ are two languages with $\varepsilon \notin A$. Also assume that $X$ is a language having the property $X = (A \cdot X) \cup B$. Then $X = A^* \cdot B$.*

What this theorem allows is the finding of *closed form* solutions to equations where the variable ($X$ in the theorem) appears on both sides. We can apply this theorem to the equations read off from DFAs quite easily: the side condition that $\varepsilon \notin A$ always holds, since DFAs have no $\varepsilon$-transitions. Thus, from our example, the equation characterizing the strings accepted from state $A$

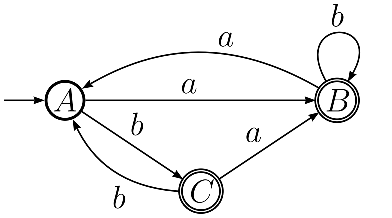
$$A = cA + bB + aC + \varepsilon$$

is equivalent, by application of Arden's Lemma to

$$A = c^*(bB + aC + \varepsilon)$$

Once the closed form $Q = rhs$ for a state $Q$ is found, $rhs$ can be substituted for $Q$ throughout the remainder of the equations. This is repeated until finally the start state has a regular expression representing its language.

**Example 113.** Give an equivalent regular expression for the following DFA:



We now make an equational presentation of the DFA:

$$\begin{aligned} A &= aB + bC \\ B &= bB + aA + \varepsilon \\ C &= aB + bA + \varepsilon \end{aligned}$$

We eventually need to solve for $A$, but can start with $B$ or $C$. Let's start with $B$. By application of Arden's lemma, we get

$$B = b^*(aA + \varepsilon)$$

and then we can substitute this in all the remaining equations:

$$\begin{array}{l} A = a(b^*(aA + \varepsilon)) + bC \\ C = a(b^*(aA + \varepsilon)) + bA + \varepsilon \end{array}$$

Now the equation for $C$ is not phrased in terms of $C$, so Arden's lemma doesn't apply and the $rhs$ may be substituted directly for $C$:

$$A = a(b^*(aA + \varepsilon)) + b(a(b^*(aA + \varepsilon)) + bA + \varepsilon)$$

And now we do some regular expression algebra to prepare for the final application of the lemma:

$$\begin{array}{rl} A = & a(b^*(aA + \varepsilon)) + b(a(b^*(aA + \varepsilon)) + bA + \varepsilon) \\ = & ab^*aA + ab^* + b(ab^*aA + ab^* + bA + \varepsilon) \\ = & ab^*aA + ab^* + bab^*aA + bab^* + bbA + b \\ = & (ab^*a + bab^*a + bb)A + ab^* + bab^* + b \end{array}$$

And then the lemma applies, and we obtain

$$A = (ab^*a + bab^*a + bb)^*(ab^* + bab^* + b)$$

Notice how this quite elegantly summarizes all the ways to loop back to $A$ when starting from $A$, followed by all the non-looping paths from $A$ to an accept state.
**End of example**

The DFA-to-regexp construction, together with the regexp-to-NFA construction, plus the subset construction, yield the following theorem.

**Theorem 16** (Kleene)**.** *Every regular language can be represented by a regular expression and, conversely, every regular expression describes a regular language.*

$$\forall L.\ \mathsf{Regular}(L) \text{ iff } \exists r.\ r \text{ is a regular expression} \wedge \mathcal{L}(r) = L$$

To summarize, we have seen methods for translating between DFAs, NFAs, and regular expressions:

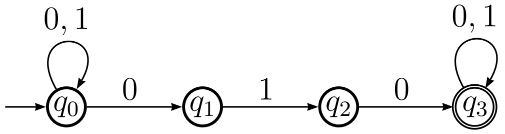
- Every DFA is an NFA.
- Every NFA can be converted to an equivalent DFA, by the subset construction.
- Every regular expression can be translated to an equivalent NFA, by the method in Section 5.4.2.
- Every DFA can be translated to a regular expression by the method in Section 7.1.3.

Notice that, in order to say that these translations work, *i.e.*, are correct, ′ we need to use the concept of formal language.

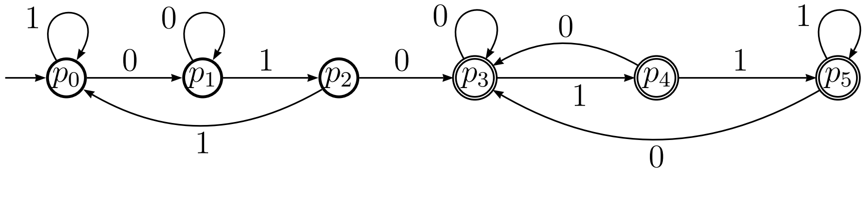
## 5.5 Minimization

Now we turn to examining how to reduce the size of a DFA such that it still recognizes the same language. This is useful because some transformations and tools will generate DFAs with a large amount of redundancy.

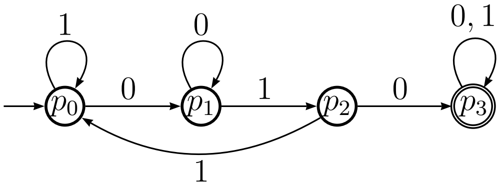
**Example 114.** Suppose we are given the following NFA:



The subset construction yields the following (equivalent) DFA:



which has 6 reachable states, out of a possible $2^4 = 16$. But notice that $p_3$, $p_4$, and $p_5$ are all accept states, and it's impossible to 'escape' from them. So you could collapse them to one big success state. Thus the DFA is equivalent to the following DFA with 4 states:



There are methods for systematically reducing DFAs to equivalent ones which are minimal in the number of states. Here's a rough outline of a minimization procedure:

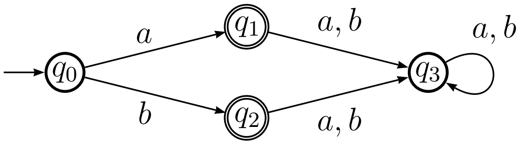
1. Eliminate inaccessible, or unreachable, states. These are states for which there is no string in $\Sigma^*$ that will take the machine to that state.

   How is this done? We have already been doing it, somewhat informally, when performing subset constructions. The idea is to start in $q_0$ and mark all states accessible in one step from it. Now repeat this from all the newly marked states until no new marked state is produced. Any unmarked states at the end of this are inaccessible and can be deleted.

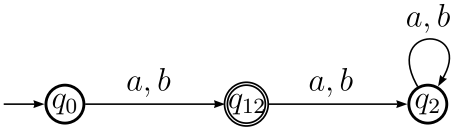
2. Collapse *equivalent* states. We will gradually see what this means in the following examples.

*Remark.* We will only be discussing minimization of DFAs. If asked to minimize an NFA, first convert it to a DFA.

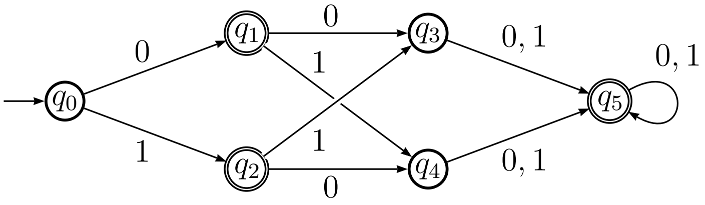
**Example 115.** The 4 state automaton



is clearly equivalent to the following $3$ state machine:



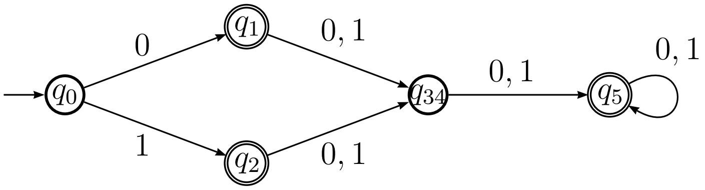
**Example 116.** The DFA



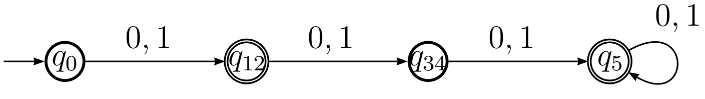
recognizes the language

$$\{0, 1\} \cup \{x \in \{0, 1\}^* \mid \mathsf{len}(x) \geq 3\}$$

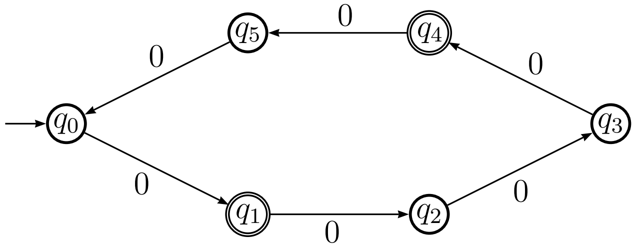
Now we observe that $q_3$ and $q_4$ are equivalent, since both go to $q_5$ on anything. Thus they can be collapsed to give the following equivalent DFA:



By the same reasoning, $q_1$ and $q_2$ both go to $q_{34}$ on anything, so we can collapse them to state $q_{12}$ to get the equivalent DFA



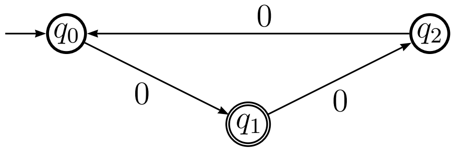
**Example 117.** The DFA



recognizes the language

$$\{0^n \mid \exists k.\, n = 3k + 1\}$$

This DFA minimizes to



How is this done, you may ask.

The main idea is a process that takes a DFA and combines states of it in a step-by-step fashion, where each steps yields an equivalent automaton. There are a couple of criteria that must be observed:

- We never combine a final state and a non-final state. Otherwise the language recognized by the automaton would change.

- If we merge states $p$ and $q$, then we have to combine $\delta(p, a)$ and $\delta(q, a)$, for each $a \in \Sigma$. Contrarily, if $\delta(p, a)$ and $\delta(q, a)$ are not equivalent states, then $p$ and $q$ can not be equivalent.

Thus if there is a string $x = x_1 \cdot \ldots \cdot x_n$ such that running the automaton $M$ from state $p$ on $x$ leaves $M$ in an accept state and running $M$ from state $q$ on $x$ leaves $M$ in a non-accept state, then $p$ and $q$ cannot be equivalent. However, if, for all strings $x$ in $\Sigma^*$, running $M$ on $x$ from $p$ yields the same acceptance verdict (accept/reject) as $M$ on $x$ from $q$, then $p$ and $q$ are equivalent. Formally we define equivalence $\approx$ as

**Definition 36** (DFA state equivalence)**.**

$$p \approx q \text{ iff } \forall x \in \Sigma^*.\ \Delta(p, x) \in F \text{ iff } \Delta(q, x) \in F$$

where $F$ is the set of final states of the automaton.

*Question*: What is $\Delta$?

*Answer* $\Delta$ is the extension of $\delta$ from symbols (single step) to strings (multiple steps). Its formal definition is as follows:

$$\begin{array}{rcl} \Delta(q, \varepsilon) & = & q \\ \Delta(q, a \cdot x) & = & \Delta(\delta(q, a), x) \end{array}$$

Thus $\Delta(q, x)$ gives the state after the machine has made a sequence of transitions while processing $x$. In other words, it's the state at the end of the computation path for $x$, where we treat $q$ as the start state.

*Remark.* $\approx$ is an equivalence relation, *i.e.*, it is reflexive, symmetric, and transitive:

- $p \approx p$
- $p \approx q \Rightarrow q \approx p$
- $p \approx q \wedge q \approx r \Rightarrow p \approx r$

An equivalence relation partitions the underlying set (for us, the set of states $Q$ of an automaton) into disjoint *equivalence classes*. This is denoted by $Q/\approx$. Each element of $Q$ is in one and only one partition of $Q/\approx$.

**Example 118.** Suppose we have a set of states $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$ and we define $q_i \approx q_j$ iff $i \text{ mod } 2 = j \text{ mod } 2$, *i.e.*, $q_i$ and $q_j$ are equivalent if $i$ and $j$ are both even or both odd. Then $Q/\approx = \{\{q_0, q_2, q_4\}, \{q_1, q_3, q_5\}\}$.

The equivalence class of $q \in Q$ is written $[q]$, and defined

$$[q] = \{p \mid p \approx q\} .$$

We have the equality

$$\underbrace{p \approx q}_{\text{equivalence of states}} \qquad \text{iff} \qquad \underbrace{([p] = [q])}_{\text{equality of sets of states}}$$

The *quotient* construction builds equivalence classes of states and then treats each equivalence class as a single state in the new automaton.

**Definition 37** (Quotient automaton)**.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The quotient automaton is $M/\approx \; = (Q', \Sigma, \delta', q_0', F')$ where

- $Q' = \{[p] \mid p \in Q\}$, *i.e.*, $Q/\approx$
- $\Sigma$ is unchanged
- $\delta'([p], a) = [\delta(p, a)]$, *i.e.*, transitioning from an equivalence class (where $p$ is an element) on a symbol $a$ is implemented by making a transition $\delta(p, a)$ in the original automaton and then returning the equivalence class of the state reached.
- $q_0' = [q_0]$, *i.e.*, the start state in the new machine is the equivalence class of the start state in the original.
- $F' = \{[p] \mid p \in F\}$, *i.e.*, the set of equivalence classes of the final states of the original machine.

**Theorem 17.** *If $M$ is a DFA that recognizes L, then $M/\approx$ is a DFA that recognizes L. There is no DFA that both recognizes L and has fewer states than $M/\approx$ .*

OK, OK, enough formalism! we still haven't addressed the crucial question, namely how do we calculate the equivalence classes?

There are several ways; we will use a *table-filling* approach. The general idea is to assume initially that all states are equivalent. But then we use our criteria to determine when states are not equivalent. Once all the non-equivalent states are marked as such, the remaining states must be equivalent.

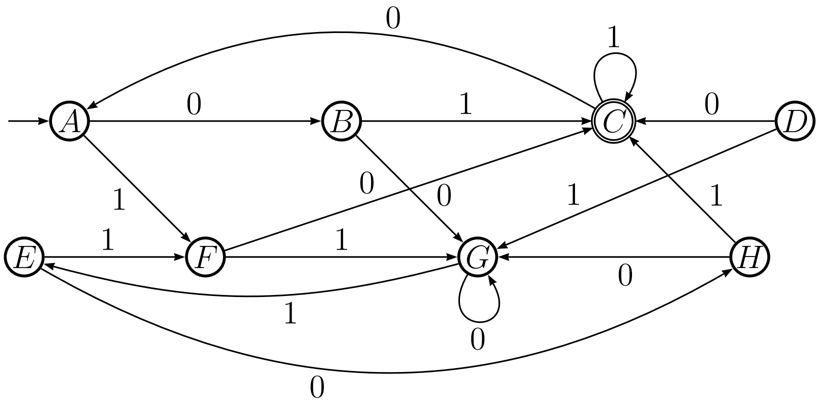
Consider all pairs of states $p$, $q$ in $Q$. A pair $p, q$ is *marked* once we know $p$ and $q$ are *not equivalent*. This leads to the following algorithm:

1. Write down a table for the pairs of states
2. Mark $(p, q)$ in the table if $p \in F$ and $q \notin F$, or if $p \notin F$ and $q \in F$.
3. Repeat until no change can be made to the table:
   - if there exists an unmarked pair $(p, q)$ in the table such that one of the states in the pair $(\delta(p, a), \delta(q, a))$ is marked, for some $a \in \Sigma$, then mark $(p, q)$.

4. Done. Read off the equivalence classes: if $(p, q)$ is not marked, then $p \approx q$.

*Remark.* We may have to revisit the same $(p, q)$ pair several times, since combining two states can suddenly allow hitherto equivalent states to be markable.

**Example 119.** Minimize the following DFA



We start by setting up our table. We will be able to restrict our attention to the lower left triangle, since equivalence is symmetric. Also, each box on the diagonal will be marked with $\approx$, since every state is equivalent to itself. We also notice that state $D$ is not reachable, so we will ignore it.

| | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ |
|---|---|---|---|---|---|---|---|---|
| $A$ | $\approx$ | – | – | – | – | – | – | – |
| $B$ | | $\approx$ | – | – | – | – | – | – |
| $C$ | | | $\approx$ | – | – | – | – | – |
| $D$ | – | – | – | – | – | – | – | – |
| $E$ | | | | – | $\approx$ | – | – | – |
| $F$ | | | | – | | $\approx$ | – | – |
| $G$ | | | | – | | | $\approx$ | – |
| $H$ | | | | – | | | | $\approx$ |

Now we split the states into final and non-final. Thus, a box indexed by $p, q$ will be labelled with an $X$ if $p$ is a final state and $q$ is not, or *vice versa*.

Thus we obtain

| | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ |
|---|---|---|---|---|---|---|---|---|
| $A$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $B$ | | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $C$ | $X_0$ | $X_0$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $D$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $E$ | | | $X_0$ | $-$ | $\approx$ | $-$ | $-$ | $-$ |
| $F$ | | | $X_0$ | $-$ | | $\approx$ | $-$ | $-$ |
| $G$ | | | $X_0$ | $-$ | | | $\approx$ | $-$ |
| $H$ | | | $X_0$ | $-$ | | | | $\approx$ |

State $C$ is inequivalent to all other states. Thus the row and column labelled by $C$ get filled in with $X_0$. (We will subscript each $X$ with the step at which it is inserted into the table.) However, note that $C, C$ is not filled in, since $C \approx C$. Now we have the following pairs of states to consider:

$$\{AB, AE, AF, AG, AH, BE, BF, BG, BH, EF, EG, EH, FG, FH, GH\}$$

Now we introduce some notation which compactly captures how the machine transitions from a pair of states to another pair of states. The notation

$$p_1p_2 \xleftarrow{0} q_1q_2 \xrightarrow{1} r_1r_2$$

means $q_1 \xrightarrow{0} p_1$ and $q_2 \xrightarrow{0} p_2$ and $q_1 \xrightarrow{1} r_1$ and $q_2 \xrightarrow{1} r_2$. If one of $p_1$, $p_2$, $r_1$, or $r_2$ are already marked in the table, then there is a way to distinguish $q_1$ and $q_2$: they transition to inequivalent states. Therefore $q_1 \not\approx q_2$ and the box labelled by $q_1q_2$ will become marked. For example, if we take the state pair $AB$, we have

$$BG \xleftarrow{0} AB \xrightarrow{1} FC$$

and since $FC$ is marked, $AB$ becomes marked as well.

| | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ |
|---|---|---|---|---|---|---|---|---|
| $A$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $B$ | $X_1$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $C$ | $X_0$ | $X_0$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $D$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $E$ | | | $X_0$ | $-$ | $\approx$ | $-$ | $-$ | $-$ |
| $F$ | | | $X_0$ | $-$ | | $\approx$ | $-$ | $-$ |
| $G$ | | | $X_0$ | $-$ | | | $\approx$ | $-$ |
| $H$ | | | $X_0$ | $-$ | | | | $\approx$ |

In a similar fashion, we examine the remaining unassigned pairs:

- $BH \xleftarrow{0} AE \xrightarrow{1} FF$. Unable to mark.
- $BC \xleftarrow{0} AF \xrightarrow{1} FG$. Mark, since $BC$ is marked.
- $BG \xleftarrow{0} AG \xrightarrow{1} FE$. Unable to mark.
- $BG \xleftarrow{0} AH \xrightarrow{1} FC$. Mark, since $FC$ is marked.
- $GH \xleftarrow{0} BE \xrightarrow{1} CF$. Mark, since $CF$ is marked.
- $GC \xleftarrow{0} BF \xrightarrow{1} CG$. Mark, since $CG$ is marked.
- $GG \xleftarrow{0} BG \xrightarrow{1} CE$. Mark, since $CE$ is marked.
- $GG \xleftarrow{0} BH \xrightarrow{1} CC$. Unable to mark.
- $HC \xleftarrow{0} EF \xrightarrow{1} FG$. Mark, since $CH$ is marked.
- $HG \xleftarrow{0} EG \xrightarrow{1} FE$. Unable to mark.
- $HG \xleftarrow{0} EH \xrightarrow{1} FC$. Mark, since $CF$ is marked.
- $CG \xleftarrow{0} FG \xrightarrow{1} GE$. Mark, since $CG$ is marked.
- $CG \xleftarrow{0} FH \xrightarrow{1} GC$. Mark, since $CG$ is marked.
- $GG \xleftarrow{0} GH \xrightarrow{1} EC$. Mark, since $EC$ is marked.

The resulting table is

| | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ |
|---|---|---|---|---|---|---|---|---|
| $A$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $B$ | $X_1$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $C$ | $X_0$ | $X_0$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $D$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $E$ | | $X_1$ | $X_0$ | $-$ | $\approx$ | $-$ | $-$ | $-$ |
| $F$ | $X_1$ | $X_1$ | $X_0$ | $-$ | $X_1$ | $\approx$ | $-$ | $-$ |
| $G$ | | $X_1$ | $X_0$ | $-$ | | $X_1$ | $\approx$ | $-$ |
| $H$ | $X_1$ | | $X_0$ | $-$ | $X_1$ | $X_1$ | $X_1$ | $\approx$ |

Next round. The following pairs need to be considered:

$$\{AE, AG, BH, EG\}$$

The previously calculated transitions can be re-used; all that will have changed is whether the 'transitioned-to' states have been subsequently marked with an $X_1$:

**AE:** unable to mark

**AG:** mark because $BG$ is now marked.

**BH:** unable to mark

**EG:** mark because $HG$ is now marked

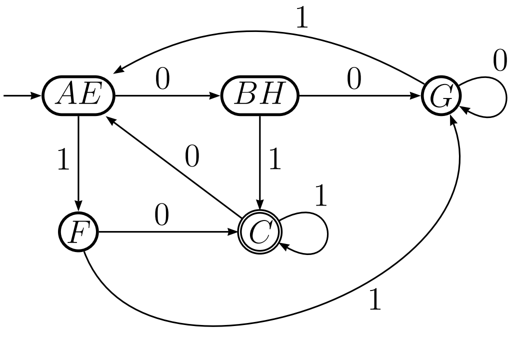
The resulting table is

| | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ |
|---|---|---|---|---|---|---|---|---|
| $A$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $B$ | $X_1$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $C$ | $X_0$ | $X_0$ | $\approx$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $D$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ |
| $E$ | | $X_1$ | $X_0$ | $-$ | $\approx$ | $-$ | $-$ | $-$ |
| $F$ | $X_1$ | $X_1$ | $X_0$ | $-$ | $X_1$ | $\approx$ | $-$ | $-$ |
| $G$ | $X_2$ | $X_1$ | $X_0$ | $-$ | $X_2$ | $X_1$ | $\approx$ | $-$ |
| $H$ | $X_1$ | | $X_0$ | $-$ | $X_1$ | $X_1$ | $X_1$ | $\approx$ |

Next round. The following pairs remain: $\{AE, BH\}$. However, neither makes a transition to a marked pair, so the round adds no new markings to the table. We are therefore done. The quotiented state set is

$$\{\{A, E\}, \{B, H\}, \{F\}, \{C\}, \{G\}\}$$

In other words, we have been able to merge states $A$ and $E$, and $B$ and $H$. The final automaton is given by the following diagram.



## 5.6 Decision Problems for Regular Languages

Now we will discuss some questions that can be asked about automata and regular expressions. These will tend to be from a general point of view, *i.e.*., involve arbitrary automata. A question that takes any automaton (or collection of automata) as input and asks for a terminating algorithm yielding a boolean (true or false) answer is called a *decision problem*, and a program that correctly solves such a problem is called a *decision algorithm*. Note well that a decision problem is typically a question about the (often infinite) set of strings that a machine must deal with; answers that involve running the machine on every string in the set are not useful, since they will take forever. That is not allowed: in every case, a decision algorithm must return a correct answer in finite time.

Here is a list of decision problems for automata and regular expressions:

1. Given a string $x$ and a DFA $M$, $x \in \mathcal{L}(M)$?

2. Given a string $x$ and an NFA $N$, $x \in \mathcal{L}(N)$?

3. Given a string $x$ and a regular expression $r$, $x \in \mathcal{L}(r)$?

4. Given DFA $M$, $\mathcal{L}(M) = \emptyset$?

5. Given DFA $M$, $\mathcal{L}(M) = \Sigma^*$?

6. Given DFAs $M_1$ and $M_2$, $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) = \emptyset$?

7. Given DFAs $M_1$ and $M_2$, $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$?

8. Given DFAs $M_1$ and $M_2$, $\mathcal{L}(M_1) = \mathcal{L}(M_2)$?

9. Given DFA $M$, is $\mathcal{L}(M)$ finite?

10. Given DFA $M$, is $M$ the DFA having the fewest states that recognizes $\mathcal{L}(M)$?

It turns out that all these problems *do* have algorithms that correctly answer the question. Some of the algorithms differ in how *efficient* they are; however, we will not delve very deeply into that issue, since this class is mainly oriented towards *qualitative* aspects of computation, *i.e.*, can the problems be solved at all? (For some decision problems, as we shall see later in the course, the answer is, surprisingly, no.)

### 5.6.1 Is a string accepted/generated?

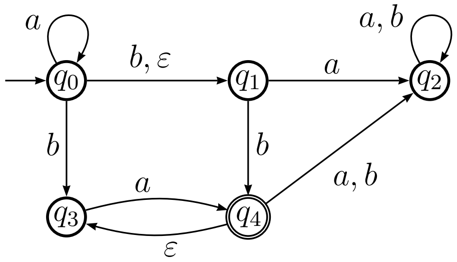
The problems

> Given a string $x$, DFA $M$, $x \in \mathcal{L}(M)$?
> Given a string $x$ and an NFA $N$, $x \in \mathcal{L}(N)$?
> Given a string $x$ and a regular expression $r$, $x \in \mathcal{L}(r)$?

are easily solved: for the first, merely run the string $x$ through the DFA and check whether the machine is in an accept state at the end of the run. For the second, first translate the NFA to an equivalent DFA by the subset construction and then run the DFA on the string. For the third, one must translate the regular expression to an NFA and then translate the NFA to a DFA before running the DFA on $x$.

However, we would like to avoid the step mapping from NFAs to DFAs, since the subset construction can create a DFA with exponentially more states than the NFA. Happily, it turns out that an algorithm that maintains the *set* of possible current states in an *on-the-fly* manner works relatively efficiently. The algorithm will be illustrated by example.

**Example 120.** Does the following NFA accept the string $aaaba$?



The initial set of states that the machine could be in is $\{q_0, q_1\}$. We then have the following table, showing how the set of possible current states changes with each new transition:

| input symbol | possible current states |
|---|---|
| | $\{q_0, q_1\}$ |
| $a$ $\downarrow$ | |
| | $\{q_0, q_1, q_2\}$ |
| $a$ $\downarrow$ | |
| | $\{q_0, q_1, q_2\}$ |
| $a$ $\downarrow$ | |
| | $\{q_0, q_1, q_2\}$ |
| $b$ $\downarrow$ | |
| | $\{q_1, q_2, q_3, q_4\}$ |
| $a$ $\downarrow$ | |
| | $\{q_2, q_3, q_4\}$ |

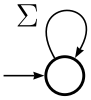
After the string has been processed, we examine the set of possible states $\{q_2, q_3, q_4\}$ and find $q_4$, so the answer returned is true.

In an implementation, the set of possible current states would be kept in a data structure, and each transition would cause states to be added or deleted from the set. Once the string was fully processed, all that needs to be done is to take the intersection between the accept states of the machine and the set of possible current states. If it was non-empty, then answer true; otherwise, answer false.

### 5.6.2 $\mathcal{L}(M) = \emptyset$?

There are a couple of possible approaches to checking language emptiness. The first idea is to minimize $M$ to an equivalent minimum state machine

$M'$ and check whether $M'$ is equal to the following DFA, which is a minimum (having only 1) state DFA that recognizes $\emptyset$, *i.e.*, accepts no strings:



This is a good idea; however, recall that the first step in minimizing a DFA is to first remove all unreachable states. A *reachable* state is one that some string will put the machine into. In other words, the reachable states are just those you can get to from the start state by making a finite number of transitions.

**Definition 38** (Reachable states)**.** Reachability is inductively defined by the following rules:

- $q_0$ is reachable.
- $q_j$ is reachable if there is a $q_i$ such that $q_i$ is reachable and $\delta(q_i, a) = q_j$, for some $a \in \Sigma$.
- no other states are reachable.

The following recursive algorithm computes the reachable states of a machine. It maintains a set of reachable states $R$, which is initialized to $\{q_0\}$:

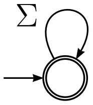
```
reachable $R$ =
    let $new = \{q' \mid \exists q\ a.\ q \in R \land q' \notin R \land a \in \Sigma \land \delta(q, a) = q'\}$
    in
      if $new = \emptyset$ then $R$ else reachable$(new \cup R)$
```

That leads us to the second idea: $\mathcal{L}(M) = \emptyset$ iff $F \cap R = \emptyset$ where $F$ is the set of accept states of $M$ and $R$ is the set of reachable states of $M$. Thus, in order to decide if $\mathcal{L}(M) = \emptyset$, we compute the reachable states of $M$ and check to see if any of them is an accept state. If one is, $\mathcal{L}(M) \neq \emptyset$; otherwise, the machine accepts no string.

### 5.6.3 $\mathcal{L}(M) = \Sigma^*$?

To decide whether a machine $M$ accepts all strings over its alphabet, we can use one of the following two algorithms:

1. Check if a minimized version of $M$ is equal to the following DFA:



2. Use closure under complementation: let $M'$ be the DFA obtained by swapping the accept and non-accept states of $M$; then apply the decision algorithm for language emptiness to $M'$. If the algorithm returns true then $M'$ accepts no string; thus $M$ must accept all strings, and we return true. Otherwise, $M'$ accepts some string $w$ and so $M$ must reject $w$, so we return false.

### 5.6.4 $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) = \emptyset$?

Given DFAs $M_1$ and $M_2$, we wish to see if there is some string that both machines accept. The following algorithm performs this task:

1. Build the product machine $M_1 \times M_2$, making the accept states be just those in which both machines accept: $\{(q_i, q_j) \mid q_i \in F_1 \wedge q_j \in F_2\}$. Thus $\mathcal{L}(M_1 \times M_2) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. This machine only accepts strings accepted by both $M_1$ and $M_2$.

2. Run the emptiness checker on $M_1 \times M_2$, and return its answer.

### 5.6.5 $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$?

Given DFAs $M_1$ and $M_2$, we wish to see if $M_2$ accepts all strings that $M_1$ accepts, and possibly more. Once we recall that $A - B = A \cap \overline{B}$ and that $A \subseteq B$ iff $A - B = \emptyset$, we can re-use our existing decision algorithms:

1. Build $M_2'$ by complementing $M_2$ (switch accept and non-accept states)

2. Run the decision algorithm for emptiness of language intersection on $M_1$ and $M_2'$, returning its answer.

### 5.6.6 $\mathcal{L}(M_1) = \mathcal{L}(M_2)$?

- One algorithm directly uses the fact $S_1 = S_2$ iff $S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$.
- Another decision algorithm would be to minimize $M_1$ and $M_2$ and test to see if the minimized machines are equal. Notice that we haven't yet said how this should be done. It is not quite trivial: we have to compare $M_1 = (Q, \Sigma, \delta, q_0, F)$ with $M_2 = (Q', \Sigma', \delta', q_0', F')$. The main problem here is that the states may have been named differently, *e.g.*, $q_0$ in $M_1$ may be $A$ in $M_2$. Therefore, we can't just test if the sets $Q$ and $Q'$ are identical. Instead, we have to check if there is a way of renaming the states of one machine into the other so that the machines become identical. We won't go into the details, which are straightforward but tedious.

  Therefore, we would be checking that the minimized machines are equal 'up to renaming of states' (another equivalence relation).

### 5.6.7 Is $\mathcal{L}(M)$ finite?

Does DFA $M$ accept only a finite number of strings? This decision problem seems more difficult than the others. We obviously can't blindly generate all strings, say in length-increasing order, and feed them to $M$: how could we stop if $M$ indeed did accept an infinite number of strings? We might see arbitrarily long stretches of strings being rejected, but couldn't be sure that eventually a longer string might come along that got accepted. Decision algorithms are not allowed to run for an infinitely long time before giving an answer.

However there is a direct approach to this decision problem. Intuitively, the algorithm would directly check to see if $M$ had a loop on a path from $q_0$ to any (reachable) accept state.

### 5.6.8 Does $M$ have as few states as possible?

Given DFA $M$, is there a machine $M'$ such that $\mathcal{L}(M) = \mathcal{L}(M')$ and there is no machine recognizing $\mathcal{L}(M)$ in fewer states than $M'$? This is solved by running the state minimization algorithm on $M$.

# Chapter 6

# The Chomsky Hierarchy

So far, we have not yet tied together the 3 different components of the course. What are the relationships between Regular, Context-Free, Decidable, Recognizable, and not-even-Recognizable languages?

It turns out that there is a (proper) inclusion hierarchy, known as the Chomsky hierarchy:

$$\mathit{Regular} \subset \mathit{CFL} \subset \mathit{Decidable} \subset \mathit{Recognizable} \subset \ldots$$

In other words, a language recognized by a DFA can always be generated by a CFG, but there are CFLs that no DFA can recognize. Every language generated by a CFG can be decided by a Turing machine, or a Register machine, but there are languages decided by TMs that cannot be generated by any CFG. Moreover, the Halting Problem is a problem that is not decidable, but is recognizable; and the complement of the Halting Problem is not even recognizable.

In order to show that a particular language $L$ is context-free but not regular, one would write down a CFG for $L$ and also show that $L$ could not possibly be regular. However, proving negative statements such as this can be difficult: in order to show that a language is regular, we need merely display an automaton or regular expression; conversely, to show that a language is not regular would (naively) seem to require examining all automata to check that each one fails to recognize at least one string in the language. But there is a better way!

## 6.1 The Pumping Lemma for Regular Languages

The pumping lemma provides one way out of this problem. It exposes a property, *pumpability*, that all regular sets have.

**Theorem 18** (Pumping lemma for regular languages)**.** *Suppose that $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA recognizing $L$. Let $p$ be the number of states in $Q$, and $s \in L$ be a string $w_0 \cdot \ldots \cdot w_{n-1}$ of length $n \geq p$. Then there exists $x$, $y$, and $z$ such that $s = xyz$ and*

**(a)** $xy^n z \in L$*, for all* $n \in \mathbb{N}$

**(b)** $y \neq \epsilon$ *(i.e.,* $\mathsf{len}(y) > 0$*)*

**(c)** $\mathsf{len}(xy) \leq p$

*Proof.* Suppose $M$ is a DFA with $p$ states which recognizes $L$. Also suppose there's an $s = w_0 \cdot \ldots \cdot w_{n-1} \in L$ where $n \geq p$. Then the computation path

$$q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{n-1}} q_n$$

for $s$ traverses at least $n + 1$ states. Now $n + 1 > p$, so, by the Pigeon Hole Principle[1], there's a state, call it $q$, which occurs at least twice in the computation path. Let $q_j$ and $q_k$ be the first and second occurrences of $q$ in the computation path. So we have

$$q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{j-1}} q_j \xrightarrow{w_j} \cdots \xrightarrow{w_{k-1}} q_k \xrightarrow{w_k} \cdots \xrightarrow{w_{n-1}} q_n$$

Now we partition the path into 3 as follows

$$\overbrace{q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{j-1}} q_j}^{x} \mid \overbrace{\xrightarrow{w_j} \cdots \xrightarrow{w_{k-1}}}^{y} \mid \overbrace{q_k \xrightarrow{w_k} \cdots \xrightarrow{w_{n-1}} q_n}^{z}$$

We have thus used our assumptions to construct a partition of $s$ into $x, y, z$. Note that this works for any string in $L$ with length not less than $p$. Now we simply have to show that the remaining conditions hold:

**(a)** The sub-path from $q_j$ to $q_k$ moves from $q$ to $q$, and thus constitutes a loop. We may go around the loop $0$, $1$, or more times to generate ever-larger strings, each of which is accepted by $M$ and is thus in $L$.

[1] The Pigeon Hole Principle is informally stated as: *given $n + 1$ pigeons and $n$ boxes, any assignment of pigeons to boxes must result in at least one box having at least 2 pigeons.*

**(b)** This is clear, since $q_j$ and $q_k$ are separated by at least one label (note that $j < k$).

**(c)** If $\mathsf{len}(xy) > p$, we could re-apply the pigeon-hole principle to obtain a state that repeats earlier than $q$, but that was ruled out by how we chose $q$.

□

The criteria (a) allows one to *pump* sufficiently long strings arbitrarily often, and thus gives us insight as to the nature of regular languages. However, it is the application of the pumping lemma to proofs of non-regularity of languages that is of interest.

### 6.1.1 Applying the pumping lemma

The use of the pumping lemma to prove non-regularity can be schematized as follows. Suppose you are to prove a statement of the following form: *Language L is not regular*. The standard proof, in outline, is as follows:

1. Towards a contradiction, assume that $L$ is regular. That means that there exists a DFA $M$ that recognizes $L$. This is boilerplate.

2. Let $p$ be the number of states in $M$. $p > 0$. This is boilerplate.

3. Supply a string $s$ of length $n \geq p$. **Creativity Required!** Typically, $s$ is phrased in terms of $p$.

4. Show that pumping $s$ leads to a contradiction *no matter how $s$ is partitioned into $x$, $y$, $z$*. In other words, find some $n$ such that $xy^nz \notin L$. This would contradict (a). One typically uses constraints (b) and (c) in the proof as well. Creativity is of course also required in this phase of the proof.

5. Shout 'Contradiction' and claim victory.

**Example 121.** The following language $L$ is not regular:

$$\{0^n1^n \mid n \geq 0\} \ .$$

*Proof.* Suppose the contrary, *i.e.*, that $L$ is regular. Then there's a DFA $M$ that recognizes $L$. Let $p$ be the number of states in $M$.

**Crucial Creative Step:** Let $s = 0^p 1^p$.

Now, $s \in L$ and $\mathsf{len}(s) \geq p$. Thus, the hypotheses of the pumping lemma hold, and we are given a partition of $s$ into $x$, $y$, and $z$ such that $s = xyz$ and

**(a)** $xy^n z \in L$, for all $n \in \mathbb{N}$

**(b)** $y \neq \epsilon$ (*i.e.*, $\mathsf{len}(y) > 0$)

**(c)** $\mathsf{len}(xy) \leq p$

all hold. Consider the string $xz$. By (a) $xz = xy^0 z \in L$. By (c) $xy$ is composed only of zeros, and hence $x$ is all zeros. By $b$, $x$ has fewer zeros than $xy$. So $xz$ has fewer than $p$ zeros, but has $p$ ones. Thus there is no way to express $xz$ as $0^k 1^k$, for any $k$. So $xz \notin L$. Contradiction. □

Here's a picture of the situation:

$$s = \underbrace{000\ldots}_{x}\underbrace{\ldots000\ldots}_{y}\underbrace{000111\ldots11}_{z}$$

Notice that $x$, $y$, and $z$ are abstract; we really don't know anything about them other than what we can infer by application of constraints $a - c$. We have $x = 0^u$ and $y = 0^v$ ($v \neq 0$) and $z = 0^w 1^p$. We know that $u + v + w = p$, but we also know that $u + w < p$, so we know $xz = 0^{u+w} 1^p \neq L$.

There's always huge confusion with the pumping lemma. Here's a slightly alternative view—*the pumping lemma protocol*—on how to use it to prove a language is not regular. Suppose there's an office $\mathcal{O}$ to support pumping lemma proofs.

1. To start the protocol, you inform $\mathcal{O}$ that $L$ is regular.

2. After some time, $\mathcal{O}$ responds with a $p$, which you know is greater than zero.

3. You then think about $L$ and invent a witness $s$. You send $s$ off to $\mathcal{O}$, along with some evidence (proofs) that $s \in L$ and $\mathsf{len}(s) \geq p$. Often this is very easy to see.

4. $\mathcal{O}$ checks your proofs. Then it divides $s$ into 3 pieces $x$, $y$, and $z$, *but it doesn't send them to you*. Instead $\mathcal{O}$ gives you permission to use (a), (b), and (c).

5. You don't know what $x$, $y$, and $z$ are, but you can use (a), (b) and (c), plus your knowledge of $s$ to deduce facts. After some ingenious steps, you find a contradiction, and send the proof of it off to $\mathcal{O}$.

6. $\mathcal{O}$ checks the proof and, if it is OK, sends you a final message confirming that $L$ is not regular after all.

**Example 122.** The following language $L$ is not regular:

$$\{w \mid w \text{ has an equal number of 0s and 1s}\} \ .$$

*Proof.* Towards a contradiction, suppose $L$ is regular. Then there's a DFA $M$ that recognizes $L$. Let $p$ be the number of states in $M$. Let $s = 0^p 1^p$. Now, $s \in L$ and $\mathsf{len}(s) \geq p$, so we know that $s = xyz$, for some $x$, $y$, and $z$. We also know (a), (b), and (c) from the statement of the pumping lemma. By (c) $xy$ is composed only of 0s. By (b) $xz = 0^k 1^p$ and $k < p$; thus $xz \notin L$. However, by (a), $xz = xy^0 z \in L$. Contradiction. □

So why did we choose $0^p 1^p$ for $s$? Why not $(01)^p$, for example? The answer comes from recognizing that, when $s$ is split into $x$, $y$, and $z$, *we have no control over how the split is made*. Thus $y$ can be *any* non-empty string of length $\leq p$. So if $s = 0101 \ldots 0101$, then $y$ could be $01$. In that case, repeated pumping will only ever lead to strings still in $L$ and we will not be able to obtain our desired contradiction.

**Upshot.** $s$ has to be chosen such that pumping it (adding in copies of $y$) will lead to a string not in $L$. Note that we can *pump down*, by adding in $0$ copies of $y$, as we have done in the last two proofs.

**Example 123.** The following language $L$ is not regular:

$$\{0^i 1^j \mid i > j\} \ .$$

*Proof.* Towards a contradiction, suppose $L$ is regular. Then there's a DFA $M$ that recognizes $L$. Let $p > 0$ be the number of states in $M$. Let $s = 0^{p+1} 1^p$. Now, $s \in L$ and $\mathsf{len}(s) \geq p$, so we know that $s = xyz$, for some $x$, $y$, and $z$. We also know (a), (b), and (c) from the statement of the pumping lemma. By (c) $xy$ is composed only of 0s. By (b) $xz$ has $k \leq p$ 0s and has $p$ 1s, so $xz \notin L$. However, by (a), $xz = xy^0 z \in L$. Contradiction. □

**Example 124.** The following language $L$ is not regular:

$$\{ww \mid w \in \{0,1\}^*\} \ .$$

*Proof.* Towards a contradiction, suppose $L$ is regular. Then there's a DFA $M$ that recognizes $L$. Let $p > 0$ be the number of states in $M$. Let $s = 0^p10^p1$. Now, $s \in L$ and $\mathsf{len}(s) \geq p$, so we know that $s = xyz$, for some $x$, $y$, and $z$. We also know (a), (b), and (c) from the statement of the pumping lemma. By (c) $xy$ is composed only of 0s. By (b) $xz = 0^k10^p1$ where $k < p$ so $xz \notin L$. However, by (a), $xz = xy^0z \in L$. Contradiction. □

Here's an example where pumping up is used.

**Example 125.** The following language $L$ is not regular:

$$\{1^{n^2} \mid n \geq 0\}.$$

*Proof.* This language is the set of all strings of 1s with length a square number. Towards a contradiction, suppose $L$ is regular. Then there's a DFA $M$ that recognizes $L$. Let $p > 0$ be the number of states in $M$. Let $s = 1^{p^2}$. This is the only natural choice; now let's see if it works! Now, $s \in L$ and $\mathsf{len}(s) \geq p$, so we have $s = xyz$, for some $x$, $y$, and $z$. We also know (a), (b), and (c) from the statement of the pumping lemma. Now we know that $1^{p^2} = xyz$. Let $i = \mathsf{len}(x)$, $j = \mathsf{len}(y)$ and $k = \mathsf{len}(z)$. Then $i+j+k = p^2$. Also, $\mathsf{len}(xyyz) = i + 2j + k = p^2 + j$. However, (b) and (c) imply that $0 < j \leq p$. Now the *next* element of $L$ larger than $1^{p^2}$ must be $1^{(p+1)^2} = 1^{p^2+2p+1}$, but $p^2 < p^2 + j < p^2 + 2p + 1$, so $xyyz \notin L$. Contradiction. □

And another.

**Example 126.** Show that $L = \{0^i1^j0^k \mid k > i + j\}$ is not regular.

*Proof.* Towards a contradiction, suppose $L$ is regular. Then there's a DFA $M$ that recognizes $L$. Let $p > 0$ be the number of states in $M$. Let $s = 0^p1^p0^{2p+1}$. Now, $s \in L$ and $\mathsf{len}(s) \geq p$, so we know that $s = xyz$, for some $x$, $y$, and $z$. We also know (a), (b), and (c) from the statement of the pumping lemma. By (c) we know

$$s = \underbrace{0^a}_{x}\underbrace{0^b}_{y}\underbrace{0^c1^p0^{2p+1}}_{z}$$

If we pump up $p + 1$ times, we obtain the string $0^a0^{b(p+1)}0^c1^p0^{2p+1}$, which is an element of $L$, by (a). However, $a + b(p + 1) + c + p \geq 2p + 1$, so this string cannot be in $L$. Contradiction. □

### 6.1.2 Is $\mathcal{L}(M)$ finite?

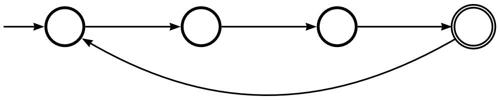
Recall the problem of deciding whether a regular language is finite. The ideas in the pumping lemma provide another way to provide an algorithm solving this problem. The idea is, given DFA $M$, to try $M$ on a finite set of strings and then render a verdict. Recall that the pumping lemma says, loosely, that every 'sufficiently long string in $L$ can be pumped': if we could find a sufficiently long string $w$ that $M$ accepts, then $\mathcal{L}(M)$ would be infinite.

All we have to do is figure out what 'sufficiently long' should mean. Two facts are important:

- For the pumping lemma to apply to a string $w$, it must be the case that $\mathsf{len}(w) \geq p$, where $p$ is the number of states of $M$. This means that, in order to be sure that $M$ has a path from the start state to an accept state and the path has a loop, we need to have $M$ accept a string of length at least $p$.

- Now we need to figure out when to stop. We want to set an upper bound $h$ on the length of the strings to be generated, so that we can reason as follows: if $M$ accepts no string $w$ where $p \leq \mathsf{len}(w) < h$ then $M$ accepts no string that can be pumped; if no strings can be pumped, then $\mathcal{L}(M)$ is finite, comprised solely of strings accepted by traversing loopless paths.

  Now, the longest single loop in a machine is going to be from a state back to itself, passing through all the other states. Here's an example:

  In the worst case for a machine with $p$ states, it will take a string of length $p-1$ to get to an accept state, plus another $p$ symbols in order to see if that state gets revisited. Thus our upper bound $h = 2p$.

The decision algorithm generates the (finite) set of strings having length at least $p$ and at most $2p-1$ and tests to see if $M$ accepts any of them. If it does, then $\mathcal{L}(M)$ is infinite; otherwise, it is finite.

## 6.2 The Pumping Lemma for Context-Free Languages

As for the regular languages, the context-free languages admit a *pumping lemma* which illustrates an interesting way in which every context-free language has a precise notion of repetition in its elements. For regular languages, the important idea in the proof was an application of the Pigeon Hole Principle in order to show that once an automaton $M$ made $n + 1$ transitions (where $n$ was the number of states of $M$) it would have to visit some state twice. If it could visit twice, it could visit any number of times. Thus we could *pump* any sufficiently long string in order to get longer and longer strings, all in the language.

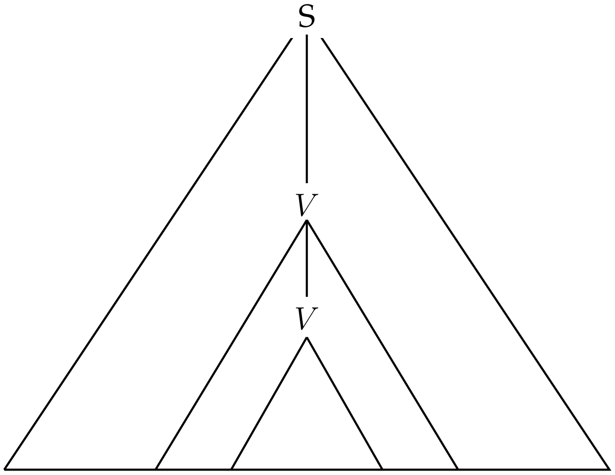
The same sort of argument, suitably adapted, can be applied to context-free languages. If a sufficiently long string is generated by a grammar $G$, then some rule in $G$ has to be applied at least twice, by appeal to the PHP. Therefore the rule can be repeatedly applied in order to pump the string.

**Theorem 19** (Pumping lemma for context-free languages)**.** *Let $L$ be a context-free language. Then there exists $p > 0$ such that, if $s \in L$ and $\mathsf{len}(s) \geq p$, then there exists $u, v, x, y, z$ such that $s = uvxyz$ and*

- $\mathsf{len}(vy) > 0$
- $\mathsf{len}(vxy) \leq p$
- $\forall i \geq 0.\ uv^ixy^iz \in L$

*Proof.* (The following is the beginning of a sketch, to be properly filled in later with more detail.)

Suppose that $L$ is context-free. Then there's a grammar $G$ that generates $L$. From the size of the right-hand sides of rules in $G$, we can compute the size of the smallest parse tree $T$ that will require some rule $V \longrightarrow \mathit{rhs}$ to be used at least twice in the derivation. This size is used to compute the minimum length $p$ of string $s \in L$ that will create a tree of that size. The so-called pumping length is $p$. Consider the longest path through $T$: by the Pigeon Hole Principle, it will have at least two (perhaps more) internal nodes labelled with $V$. Considering the "bottom-most" pair of such $V$s leads to a decomposition of $T$ as follows:



This implies a decomposition of $s$ into $u, v, x, y, z$. Now, the consequences can be established as follows:

- $\mathsf{len}(vy) > 0$. This holds since $T$ was chosen to be the smallest parse tree satisfying the other constraints: if both $v$ and $y$ were $\varepsilon$, then the resulting tree would be smaller than T.

- $\mathsf{len}(vxy) \leq p$. This holds since we chose the lowest two occurrences of $V$.

- $\forall i \geq 0.\ uv^ixy^iz \in L$. This holds, since the tree rooted at the bottom-most occurrence of $V$ can be replaced by the tree rooted at the next-higher-up occurrence of $V$. And so on, repeatedly.

□

Recall that the main application of the pumping lemma for regular languages was to show that various languages were *not* regular, by contradiction. The same is true for the context-free languages. However, the details of the proofs are more complex, as we shall see. We will go through one proof in full detail and then see how—sometimes—much of the complexity can be avoided.

**Example 127.** The language $L = \{a^nb^nc^n \mid n \geq 0\}$ is not context-free.

*Proof.* (As for regular languages, there is a certain amount of boilerplate at the beginning of the proofs.) Assume $L$ is a context-free language. Then there exists a pumping length $p > 0$. Consider $\boxed{s = a^p b^p c^p}$. (This is the first *creative* bit.) Evidently, $s \in L$, and $\mathsf{len}(s) \geq p$. Therefore, there exists $u, v, x, y, z$ such that $s = uvxyz$ and the following hold

1. $\mathsf{len}(vy) > 0$
2. $\mathsf{len}(vxy) \leq p$
3. $\forall i \geq 0.\ uv^i xy^i z \in L$

Now we consider where $vxy$ can occur in $s$. Pumping lemma proofs for context-free languages are all about case analysis. Here we have a number of cases (some of which have sub-cases): $vxy$ can occur

- completely within the leading $a^p$ symbols.
- completely within the middle $b^p$ symbols.
- completely within the trailing $c^p$ symbols.
- partly in the $a^p$ and partly in the $b^p$.
- partly in the $b^p$ and partly in the $c^p$.

What *cannot* happen is for $vxy$ to start with some $a$ symbols, span all $p$ $b$ symbols, and finish with some $c$ symbols: clause (2) above prohibits this.

Now, if $vxy$ occurs completely within the leading $a^p$ symbols, then pumping up once yields the string $s' = uv^2xy^2z = a^q b^p c^p$, where $p < q$, by (1). Thus $s' \notin L$, contradicting (3).

Similarly, if $vxy$ occurs completely within the middle $b^p$ symbols, pumping up once yields the string $s' = a^p b^q c^p$, where $p < q$. Contradiction. Now, of course, it can easily be seen that a very similar proof handles the case where $vxy$ occurs completely within the trailing $c^p$ symbols. We are now left with the *hybrid* cases, where $vxy$ spans two kinds of symbol. These need further examination.

Suppose $vxy$ occurs partly in the $a^p$ and partly in the $b^p$. Thus, at some point, $vxy$ changes from $a$ symbols to $b$ symbols. The change-over can happen in $v$, $x$, or in $y$:

- <u>in $v$</u>. Then we've deduced that the split of $s$ looks like

$$s = \underbrace{a^i}_{u} \underbrace{a^j b^k}_{v} \underbrace{b^\ell}_{x} \underbrace{b^m}_{y} \underbrace{b^n c^p}_{z}$$

If we now pump up, we obtain

$$s' = a^i \underbrace{a^j b^k}_{v} \underbrace{a^j b^k}_{v} b^\ell \underbrace{b^m}_{y} \underbrace{b^m}_{y} b^n c^p$$

which can't be an element of $L$, since we have a MIXED-UP JUMBLE $a^j b^k a^j b^k$ where we pumped $v$. On the other hand, if we pump down, there is no jumble; we obtain

$$s' = a^i b^\ell b^n c^p$$

which, however, is also not a member of $L$, since $i < p$ or $\ell + n < p$, by (1).

- <u>in $x$</u>. Thus the split of $s$ looks like

  $$s = \underbrace{a^i}_{u} \underbrace{a^j}_{v} \underbrace{a^k b^\ell}_{x} \underbrace{b^m}_{y} \underbrace{b^n c^p}_{z}$$

  In this situation, neither $v$ nor $y$ feature a change in symbol, so pumping up will not result in a JUMBLE. But, by pumping up we obtain

  $$s' = a^i a^{2j} a^k b^\ell b^{2m} b^n c^p$$

  Now, we know $i + j + k = p$ and $\ell + m + n = p$, therefore

  $$i + 2j + k > p \ \vee \ \ell + 2m + n > p$$

  hence $s' \notin L$. Contradiction. (Pumping down also leads to a contradiction.)

- <u>in $y$</u>. This case is very similar to the case where the change-over happens in $v$. We have

  $$s = \underbrace{a^i}_{u} \underbrace{a^j}_{v} \underbrace{a^k}_{x} \underbrace{a^\ell b^m}_{y} \underbrace{b^n c^p}_{z}$$

  and pumping up leads to a JUMBLE, while pumping down leads to $s' = a^{i+k} b^n c^p$, where $i + k < p$ or $n < p$, thus contradiction.

Are we done yet? No! We still have to consider what happens when $vxy$ occurs partly in the $b^p$ and partly in the $c^p$. Yuck! Let's review the skeleton of the proof: $vxy$ can occur

- completely within the leading $a^p$ symbols or completely within the middle $b^p$ symbols, or completely within the trailing $c^p$ symbols. These are all complete, and were easy.

- partly in the $a^p$ and partly in the $b^p$. This has just been completed. A subsidiary case analysis on where the change-over from $a$ to $b$ happens was needed: in $v$, in $x$, or in $y$ .

- partly in the $b^p$ and partly in the $c^p$. Not done, but requires case analysis on where the change-over from $b$ to $c$ happens: in $v$, in $x$, or in $y$. With minor changes, the arguments we gave for the previous case will establish this case, so we won't go through them.

□

Now that we have seen a fully detailed case analysis of the problem, it is worth considering whether there is a shorter proof. All that case analysis was pretty tedious! A different approach, which is sometimes a bit simpler for some (not all) pumping lemma proofs, is to use *zones*. Let's try the example again.

**Example 128** (Repeated)**.** The language $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.

*Proof.* Let the same boilerplate and witness be given. Thus we have the same facts at our disposal, but will make a different case analysis in the proof. Notice that $vxy$ can occur either in

- zone $A$

  $$s = \underbrace{a^p\ b^p}_{A}\ c^p$$

  In this case, if we pump up (to get $s'$), we will add a non-zero number of $a$ and/or $b$ symbols to zone $A$. Thus $\mathsf{count}(s', a) + \mathsf{count}(s', b) > 2 * \mathsf{count}(s', c)$, which implies that $s' \notin L$. Contradiction.

- or zone $B$:

  $$s = a^p\ \underbrace{b^p\ c^p}_{B}$$

  If we pump up (to get $s'$), we will add a non-zero number of $b$ and/or $c$ symbols to zone $B$. Thus $2 * \mathsf{count}(s', a) < \mathsf{count}(s', b) + \mathsf{count}(s', c)$, which implies that $s' \notin L$. Contradiction.

□

*Remark.* The argument for zone $A$ uses the following obvious lemma, which we will spell out for completeness.

$$\mathsf{count}(w,a) + \mathsf{count}(w,b) > 2 * \mathsf{count}(w,c) \Rightarrow w \notin \{a^n b^n c^n \mid n \geq 0\}$$

*Proof.* Assume $\mathsf{count}(w,a) + \mathsf{count}(w,b) > 2 * \mathsf{count}(w,c)$. Towards a contradiction, assume $w = a^n b^n c^n$, for some $n \geq 0$. Thus $\mathsf{count}(w,a) = n = \mathsf{count}(w,b) = \mathsf{count}(w,c)$, so $\mathsf{count}(w,a) + \mathsf{count}(w,b) = 2 * \mathsf{count}(w,c)$. Contradiction. □

The corresponding lemma for zone $B$ is similar.

The new proof using zones is quite a bit shorter. The reason for this is that it condensed all the similar case analyses of the first proof into just two cases. Notice that the JUMBLE cases still arise, but don't need to be explicitly addressed, since we rely on the lemmas about counting, which hold whether or not the strings are jumbled.

**Example 129.** $L = \{ww \mid w \in \{0,1\}^*\}$ is not a context-free language.

*Proof.* Assume $L$ is a context-free language. Then there exists a pumping length $p > 0$. Consider $\boxed{s = 0^p 1^p 0^p 1^p}$; thus $s \in L$, and $\mathsf{len}(s) \geq p$. Therefore, there exists $u, v, x, y, z$ such that $s = uvxyz$ and the following hold (1) $\mathsf{len}(vy) > 0$, (2) $\mathsf{len}(vxy) \leq p$, and (3) $\forall i \geq 0.\ uv^i xy^i z \in L$. Notice that $vxy$ can occur in zones $A$, $B$, or $C$:

- zone $A$ :

  $$s = \underbrace{0^p\ 1^p}_{A}\ \underbrace{0^p\ 1^p}_{C}$$

  In this case, if we pump down (to get $s'$), we will remove a non-zero number of $0$ and/or $1$ symbols from zone $A$, while zone $C$ does not change. Thus $s' = 0^i 1^j 0^p 1^p$, where $i < p$ or $j < p$. Thus zone $A$ in $s'$ becomes shorter than zone $C$. We still have to argue that $s'$ cannot be divided into two identical parts. Suppose it could be. In that case, the middle of $s'$ will be at $\frac{i+j+2p}{2} > i + j$, giving

  $$s' = \underbrace{0^i 1^j 0^k}\ \underbrace{0^\ell 1^p}$$

  where $i + j + k = \ell + p$ and $0 < k$, but then $s' \notin L$. Contradiction.

  The argument for zone $C$ is similar.

- zone $B$:

  $$s = 0^p \underbrace{1^p\ 0^p}_{B}\ 1^p$$

  Pumping down yields $s' = 0^p 1^i 0^j 1^p$ where $i + j < 2p$. Now $s' \in L$ provided $p = j$ and $i = p$. Contradiction.

- zone $C$: this is quite similar to the situation in zone $A$.

□

**Example 130.** $L = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$ is not context-free.

*Proof.* Assume $L$ is a context-free language. Then there exists a pumping length $p > 0$. Consider $\boxed{s = a^p b^p c^p}$. Thus $s \in L$, and $\mathsf{len}(s) \geq p$. Therefore, there exists $u, v, x, y, z$ such that $s = uvxyz$ and the following hold (1) $\mathsf{len}(vy) > 0$, (2) $\mathsf{len}(vxy) \leq p$, and (3) $\forall i \geq 0.\ uv^i xy^i z \in L$. Notice that $vxy$ can occur in zones $A$ or $B$:

- zone $A$:

  $$s = \underbrace{a^p\ b^p}_{A}\ c^p$$

  Here we have to pump up: pumping down could preserve the inequality. Thus $s' = a^i b^j c^p$, where $i > p \lor j > p$. In either case, $s' \notin L$. Contradiction.

- zone $B$:

  $$s = a^p\ \underbrace{b^p\ c^p}_{B}$$

  Here we pump down (pumping up could preserve the inequality). Thus $s' = a^p b^i c^j$, where $i < p \lor j < p$. In either case, $s' \notin L$. Contradiction.

□

Now here's an application of the pumping lemma featuring a language with only a single symbol in its alphabet. In such a situation, doing a case analysis on where $vxy$ occurs doesn't help; instead, we have to reason about the relative lengths of strings.

**Example 131.** $L = \{a^{n^2} \mid n \geq 0\}$ is not context-free.

*Proof.* Assume $L$ is a context-free language. Then there exists a pumping length $p > 0$. Consider $\boxed{s = a^{p^2}}$. Thus $s \in L$, and $\mathsf{len}(s) \geq p$. Therefore, there exists $u, v, x, y, z$ such that $s = uvxyz$ and the following hold (1) $\mathsf{len}(vy) > 0$, (2) $\mathsf{len}(vxy) \leq p$, and (3) $\forall i \geq 0.\ uv^ixy^iz \in L$.

By (1) and (2), we know $0 < \mathsf{len}(vy) \leq p$. Thus if we pump up once, we obtain a string $s' = a^n$, where $p^2 < n \leq p^2 + p$. Now consider $L$. The *next*[2] element of $L$ after $s$ must be of length $(p+1)^2$, *i.e.*, of length $p^2 + 2p + 1$. Since

$$p^2 < n < p^2 + p + 1$$

we conclude $s' \notin L$. Contradiction. □

[2] $L$ is a set, of course, and so has no notion of 'next'; however, for every element $x$ of $L$, there's an element $y \in L$ such that $\mathsf{len}(y) > \mathsf{len}(x)$ and $y$ is the shortest element of $L$ longer than $x$. Thus $y$ would be the *next* element of $L$ after $x$.

# Chapter 7

# Further Topics

## 7.1 Regular Languages

The subject of regular languages and related concepts, such as finite state machines, although established a long time ago, is still vibrant and influential. We have really only touched on the tip of the iceberg! In the following sections, we explore a few other related notions and applications.

### 7.1.1 Extended Regular Expressions

We have emphasized that regular languages are generated by regular expressions and accepted by machines. However, there was an asymmetry in our presentation: there are, seemingly, *fewer* operations on regular expressions than on machines. For example, to prove that regular languages are closed under complement, one usually thinks of a language as being represented by a DFA $M$, and the complement of the language is the language of a machine obtained by swapping accept and non-accept states of $M$. Is there something analogous from the perspective of regular expressions, *i.e.*, given a regular expression that generates a language, is there a regular expression that generates the complement of the language? We know that the answer is affirmative, but the typical route *uses machines*: the regular expression is translated to an NFA, the subset construction takes the NFA to an equivalent DFA, the DFA has its accept/non-accept states swapped, and then Kleene's construction is used to map the complemented DFA to an equivalent regular expression. How messy!

What happens if we avoid machines and stipulate a direct mapping from regular expressions to regular expressions? Is it possible? It turns out that the answer is "yes". In the first few pages of *Derivatives of Regular Expressions*,[1] Brzozowski defines an augmented set of regular expressions, and then introduces the idea of the *derivative*[2] of a regular expression with respect to a symbol of the alphabet. He goes on to give a recursive function to compute the derivative and shows how to use it in regular expression matching.

An extended regular expression adds $\cap$ and complementation operations to the set of regular expression operations. This allows any boolean operation on languages to be expressed.

**Definition 39** (Syntax of extended regular expressions)**.** The set of expressions $\mathcal{R}$ formed from alphabet $\Sigma$ is the following:

- $a \in \mathcal{R}, \text{if } a \in \Sigma$
- $\varepsilon \in \mathcal{R}$
- $\emptyset \in \mathcal{R}$
- $\overline{r} \in \mathcal{R}, \text{if } r \in \mathcal{R} \quad \text{(new)}$
- $r_1 \cap r_2 \in \mathcal{R}, \text{if } r_1 \in \mathcal{R} \wedge r_2 \in \mathcal{R} \quad \text{(new)}$
- $r_1 + r_2 \in \mathcal{R}, \text{if } r_1 \in \mathcal{R} \wedge r_2 \in \mathcal{R}$
- $r_1 \cdot r_2 \in \mathcal{R}, \text{if } r_1 \in \mathcal{R} \wedge r_2 \in \mathcal{R}$
- $r^* \in \mathcal{R}, \text{if } r \in \mathcal{R}$
- Nothing else is in $\mathcal{R}$

[1] Journal of the ACM, October 1964, pages 481 to 494.

[2] This is *not* the familiar notion from calculus, although it was so named because the algebraic equations are similar.

**Definition 40** (Semantics of extended regular expressions)**.** The meaning of an extended regular expression $r$, written $\mathcal{L}(r)$ is defined as follows:

$$\begin{array}{lcll}
\mathcal{L}(a) & = & \{a\}, \text{ for } a \in \Sigma & \\
\mathcal{L}(\varepsilon) & = & \{\varepsilon\} & \\
\mathcal{L}(\emptyset) & = & \emptyset & \\
\mathcal{L}(\overline{r}) & = & \overline{\mathcal{L}(r)} & \text{(new)} \\
\mathcal{L}(r_1 \cap r_2) & = & \mathcal{L}(r_1) \cap \mathcal{L}(r_2) & \text{(new)} \\
\mathcal{L}(r_1 + r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) & \\
\mathcal{L}(r_1 \cdot r_2) & = & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) & \\
\mathcal{L}(r^*) & = & \mathcal{L}(r)^* &
\end{array}$$

**Example 132.** If we wanted to specify the language of all binary strings with no occurrences of 00 or 11, the usual regular expressions could express this as follows:

$$\varepsilon + 0 + 1 + (01)^* + (10)^*$$

but it requires a few moments thought to make sure that this is a correct regular expression for the language. However, the following extended regular expression

$$\overline{\Sigma^*(00 + 11)\Sigma^*}$$

for the language is immediately understandable.

**Example 133.** The following extended regular expression generates the language of all binary strings with at least two consecutive zeros and not ending in 01.

$$(\Sigma^*00\Sigma^*) \cap \overline{\Sigma^*01}$$

One might think that this can be expressed just as simply with ordinary regular expressions: something like

$$\Sigma^*00\Sigma^*(10 + 11 + 00 + 0)$$

seems verbose but promising. However, it doesn't work, since it doesn't generate the string 00. Adding an $\varepsilon$ at the end

$$\Sigma^*00\Sigma^*(10 + 11 + 00 + 0 + \varepsilon)$$

also doesn't work: it allows 001, for example.

**Example 134.** The following extended regular expression generates the language of all strings with at least three consecutive ones and not ending in 01 or consisting of all ones.

$$(\Sigma^* 111 \Sigma^*) \cap \overline{(\Sigma^* 01 + 11^*)}$$

### Derivatives of extended regular expressions

In order to compute the derivative, it is necessary to be able to compute whether a regular expression $r$ generates the empty string, *i.e.*, whether $\varepsilon \in \mathcal{L}(r)$. If $r$ has this property, then $r$ is said to be *nullable*. This is easy to compute recursively:

**Definition 41** (Nullable)**.**

$$\begin{array}{rcl}
\mathsf{nullable}(\mathsf{a}) & = & \mathsf{false} \quad \text{if } \mathsf{a} \in \Sigma \\
\mathsf{nullable}(\varepsilon) & = & \mathsf{true} \\
\mathsf{nullable}(\emptyset) & = & \mathsf{false} \\
\mathsf{nullable}(\overline{r}) & = & \neg(\mathsf{nullable}(r)) \\
\mathsf{nullable}(r_1 \cap r_2) & = & \mathsf{nullable}(r_1) \wedge \mathsf{nullable}(r_2) \\
\mathsf{nullable}(r_1 + r_2) & = & \mathsf{nullable}(r_1) \vee \mathsf{nullable}(r_2) \\
\mathsf{nullable}(r_1 \cdot r_2) & = & \mathsf{nullable}(r_1) \wedge \mathsf{nullable}(r_2) \\
\mathsf{nullable}(r^*) & = & \mathsf{true}
\end{array}$$

**Definition 42** (Derivative)**.** The *derivative* of $r$ with respect to a string $u$ is defined to be the set of strings which when concatenated with $u$ yield an element of $\mathcal{L}(r)$:

$$\mathsf{Derivative}(u, r) = \{w \mid u \cdot w \in \mathcal{L}(r)\}$$

Thus the derivative is a language, *i.e.*, a set of strings. An immediate consequence is the following:

**Theorem 20.**

$$w \in \mathcal{L}(r) \text{ iff } \varepsilon \in \mathsf{Derivative}(w, r)$$

However, what we are after is an operation mapping regular expressions to regular expressions. An algorithm for computing the derivative for a single symbol is specified as follows.

**Definition 43** (Derivative of a symbol)**.** The derivative $\mathsf{D}(a, r)$ of a regular expression $r$ with respect to a symbol $a \in \Sigma$ is defined by

$$\begin{array}{rcl}
\mathsf{D}(a, \varepsilon) & = & \emptyset \\
\mathsf{D}(a, \emptyset) & = & \emptyset \\
\mathsf{D}(a, a) & = & \varepsilon \\
\mathsf{D}(a, b) & = & \emptyset \quad \text{if } a \neq b \\
\mathsf{D}(a, \overline{r}) & = & \overline{\mathsf{D}(a, r)} \\
\mathsf{D}(a, r_1 + r_2) & = & \mathsf{D}(a, r_1) + \mathsf{D}(a, r_2) \\
\mathsf{D}(a, r_1 \cap r_2) & = & \mathsf{D}(a, r_1) \cap \mathsf{D}(a, r_2) \\
\mathsf{D}(a, r_1 \cdot r_2) & = & (\mathsf{D}(a, r_1) \cdot r_2) + \mathsf{D}(a, r_2) \quad \text{if } \mathsf{nullable}(r_1) \\
\mathsf{D}(a, r_1 \cdot r_2) & = & \mathsf{D}(a, r_1) \cdot r_2 \quad \text{if } \neg\mathsf{nullable}(r_1) \\
\mathsf{D}(a, r^*) & = & \mathsf{D}(a, r) \cdot r^*
\end{array}$$

Consider $r' = \mathsf{D}(a, r)$. Intuitively, $\mathcal{L}(r')$ is the set of strings in $\mathcal{L}(r)$ from which a leading $a$ has been dropped. Formally,

**Theorem 21.**

$$w \in \mathcal{L}(\mathsf{D}(a, r)) \text{ iff } a \cdot w \in \mathcal{L}(r)$$

The function $\mathsf{D}$ can be used to compute $\mathsf{Derivative}(u, r)$ by iteration on the symbols in $u$.

**Definition 44** (Derivative of a string)**.**

$$\begin{array}{rcl}
\mathsf{Der}(\varepsilon, r) & = & r \\
\mathsf{Der}(a \cdot w, r) & = & \mathsf{Der}(w, \mathsf{D}(a, r))
\end{array}$$

The following theorem is then easy to show:

**Theorem 22.**

$$\mathcal{L}(\mathsf{Der}(w, r)) = \mathsf{Derivative}(w, r)$$

Recall that the standard way to check if $w \in \mathcal{L}(r)$ requires the translation of $r$ to a state machine, followed by running the state machine on $w$. In contrast, the use of derivatives allows one to merely evaluate $\mathsf{nullable}(\mathsf{Der}(w, r))$, *i.e.*, to stay in the realm of regular expressions. However, this can be inefficient, since taking the derivative can substantially increase the size of the regular expression.

### Generating automata from extended regular expressions

Instead, Brzozowski's primary purpose in introducing derivatives was to use them as a way of directly producing minimal DFAs from extended regular expressions.The process works as follows. Suppose $\Sigma = \{a_1, \ldots, a_n\}$ and $r$ is a regular expression. We think of $r$ as representing the start state of the desired DFA. Since the transition function $\delta$ of the DFA is total, the successor states may be obtained by taking the derivatives $\mathsf{D}(a_1, r), \ldots \mathsf{D}(a_n, r)$. This is repeated until no new states can be produced. Final states are just those that are nullable. The resulting state machine accepts the language generated by $r$. This is an amazingly elegant procedure, especially in comparison to the translation to automata. However, it depends on being able to decide when two regular expressions have the same language (so that seemingly different states can be equated, which is necessary for the process to terminate).

**Example 135.** We will translate $(0 + 1)^*1$ to a DFA. To start, we assign state $q_0$ to $(0 + 1)^*1$. Then we take the derivatives to get the successor states, and build up the transition function $\delta$ along the way.

$$\begin{array}{rcl}
\mathsf{D}(0, (0+1)^*1) & = & \mathsf{D}(0, (0+1)^*)1 + \mathsf{D}(0, 1) \\
& = & \mathsf{D}(0, (0+1)^*)1 \\
& = & \mathsf{D}(0, (0+1))(0+1)^*1 \\
& = & (\mathsf{D}(0, 0) + \mathsf{D}(0, 1))(0+1)^*1 \\
& = & (\varepsilon + \emptyset)(0+1)^*1 \\
& = & (0+1)^*1
\end{array}$$

So $\delta(q_0, 0) = q_0$. What about $\delta(q_0, 1)$?

$$\begin{array}{rcl}
\mathsf{D}(1, (0+1)^*1) & = & \mathsf{D}(1, (0+1)^*)1 + \mathsf{D}(1, 1) \\
& = & \mathsf{D}(1, (0+1)^*)1 + \varepsilon \\
& = & \mathsf{D}(1, (0+1))(0+1)^*1 + \varepsilon \\
& = & (\mathsf{D}(1, 0) + \mathsf{D}(1, 1))(0+1)^*1 + \varepsilon \\
& = & (\emptyset + \varepsilon)(0+1)^*1 + \varepsilon \\
& = & (0+1)^*1 + \varepsilon
\end{array}$$

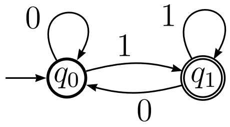
Since this regular expression is not equal to that associated with any other state, we allocate a new state $q_1 = (0 + 1)^*1 + \varepsilon$. Note that $q_1$ is a final state because its associated regular expression is nullable. We now compute the successors to $q_1$:

$$\begin{array}{rcl}\mathsf{D}(0,(0+1)^*1+\varepsilon) & = & \mathsf{D}(0,(0+1)^*1)+\mathsf{D}(0,\varepsilon) \\ & = & (0+1)^*1+\emptyset\end{array}$$

So $\delta(q_1,0)=q_0$. Also

$$\begin{array}{rcl}\mathsf{D}(1,(0+1)^*1+\varepsilon) & = & \mathsf{D}(1,(0+1)^*1)+\mathsf{D}(1,\varepsilon) \\ & = & ((0+1)^*1+\varepsilon)+\emptyset\end{array}$$

So $\delta(q_1,1)=q_1$. There are no more states to consider, so the final, minimal, equivalent DFA is



In the previous discussion we have assumed a 'full' equality test, *i.e.*, one with the property $r_1 = r_2$ iff $\mathcal{L}(r_1) = \mathcal{L}(r_2)$. If the algorithm uses this test, the resulting DFA is guaranteed to be minimal. However, such a test is computationally expensive. It is an interesting fact that we can approximate the equality test and still obtain an equivalent DFA, which may, however, not be minimal.

Let $r_1 \approx r_2$ iff $r_1$ and $r_2$ are syntactically equal modulo the use of the equalities

$$\begin{array}{l} r+r=r \\ r_1+r_2=r_2+r_1 \\ (r_1+r_2)+r_3=r_1+(r_2+r_3)\end{array}$$

The state-generation procedure outlined above will still terminate with $\approx$ being used to implement the test for regular expression equality rather than full equality. Also note that implementations take advantage of standard simplifications for regular expressions in order to keep the regular expressions in a reduced form.

### 7.1.2 How to Learn a DFA

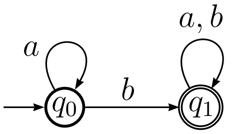
### 7.1.3 From DFAs to regular expressions (Again)

[ *The following subsection takes a traditional approach to the translation of DFAs to regexps. In the body of the notes, I have instead used the friendlier (to the instructor and the student) approach based on representing the automaton by systems of equations and then iteratively using Arden's lemma to solve for the starting state.* ]

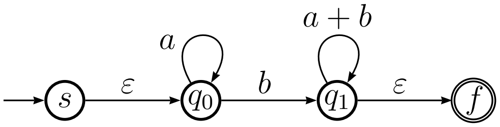
The basic idea in translating an automaton $M$ into an equivalent regular expression is to translate $M$ into a regular expression through a series of steps. Each step will preserve $\mathcal{L}(M)$. At each step we will drop a state from the automaton, and in order to still recognize $\mathcal{L}(M)$, we will have to 'patch up' the labels on the edges between the remaining states. The technical device for accomplishing this is the so-called *GNFA*, which is an NFA with arbitrary regular expressions labelling transitions. (You can think of the intermediate automata in the just-seen regular expression-to-NFA translation as being GNFA.)

We will look at a very simple example of the translation to aid our intuition when thinking about the general case.

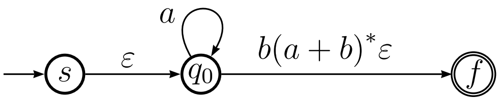
**Example 136.** Let the example automaton be given by the following diagram



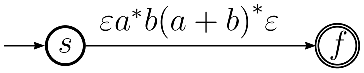
The first step is to add a new start state and a single new final state, connected to the initial automaton by $\varepsilon$-transitions. Also, multiple edges from a source to a target are agglomerated into one, by joining the labels via a $+$ operation.



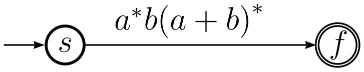
Now we iteratively delete nodes. It doesn't matter in which order we delete them—the language will remain the same—although pragmatically, the right choice of node to delete can make the work much simpler.[3] Let's delete $q_1$. Now we have to patch the hole left. In order to still accept the same set of strings, we have to account for the $b$ label, the $a+b$ label on the self-loop of $q_1$, and the $\varepsilon$ label leading from $q_1$ to $f$. Thus the following automaton:



Similarly, deleting $q_0$ yields the final automaton:



which by standard identities is



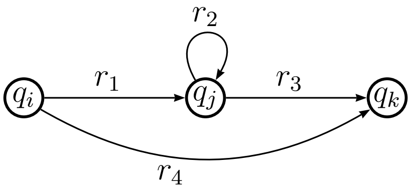
### Constructing a GNFA

To make an initial GNFA (call it $GNFA_0$) from an NFA $N = (Q, \Sigma, \delta, q_0, F)$ requires the following steps:

1. Make a new start state with an $\varepsilon$-transition to $q_0$.

2. Make a new final state with $\varepsilon$-transitions from all the states in $F$. The states in $F$ are no longer considered to be final states in $GNFA_0$.

3. Add edges to $N$ to ensure that every state $q_j$ in $Q$ has the shape

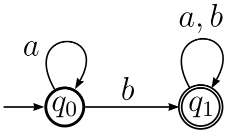


[3] The advice of the experts is to delete the node which 'disconnects' the automaton as much as possible.

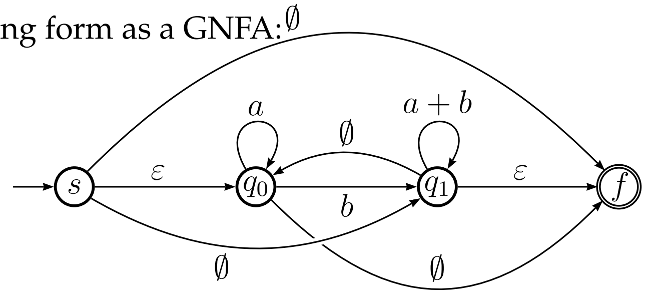
To achieve this may require adding in lots of weird new edges. In particular, a GNFA must have the following special form:

- The new start state must have arrows going to every other state (but no arrows coming in to it).
- The new final state must have arrows coming into it from every other state (but no arrows going out of it).
- For all other states (namely all those in $Q$) there must be a *single* arrow to every other state, plus a self loop. In order to agglomerate multiple edges from the same source to a target, we make a 'sum' of all the labels.

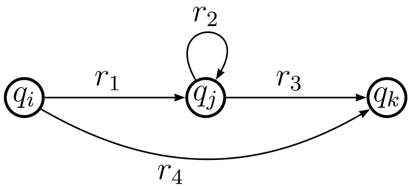
Note that if a transition didn't exist between two states in $N$, one would have to be created. For this purpose, such an edge would be labelled with $\emptyset$, which fulfills the syntactic requirement without actually enabling any new behaviour by the machines (since transitions labelled with $\emptyset$ can never be followed). Thus, our simple example



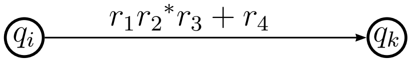
has the following form as a GNFA:



(In order to avoid too many superfluous $\emptyset$-transitions, we will often omit them from our GNFAs, with the understanding that they are still there, lurking just out of sight.) Now we can describe the step that is taken each time a state is eliminated when passing from $GNFA_i$ to $GNFA_{i+1}$. To eliminate state $q_j$ in

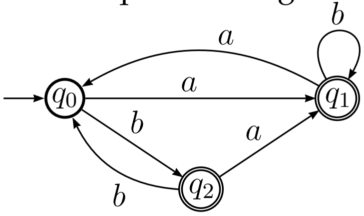


we replace it by

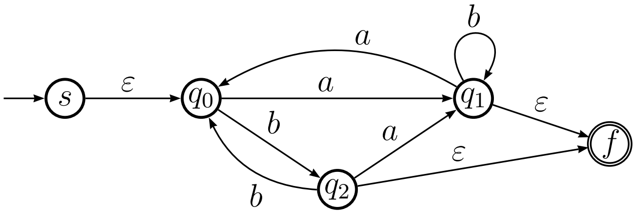


A clue why this 'works' is obtained by considering an arbitrary string $w$ accepted by $GNFA_i$ and showing it is still accepted by $GNFA_{i+1}$. Consider the sequence of states traversed in an accepting run of the automaton $GNFA_i$. Either $q_j$ appears in it or it doesn't. If it appears, then the portion of $w$ processed while passing through $q_j$ is evidently matched by the regular expression $r_1{r_2}^*r_3$. On the other hand, if $q_j$ does not appear in the state sequence, that means that the 'bypass' from $q_i$ to $q_k$ has been taken (since all states have transitions among themselves). In that case $r_4$ will match.

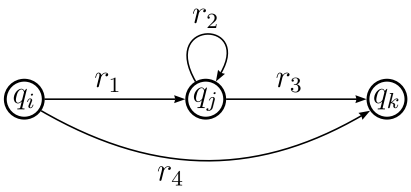
**Example 137.** Give an equivalent regular expression for the following DFA:



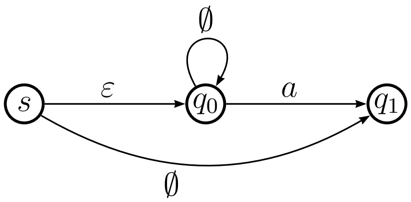
The initial GNFA is ($\emptyset$-transitions have not been drawn):



The 'set-up' of the initial GNFA means that, for any state $q_j$, except $s$ and $f$, the following pattern holds:

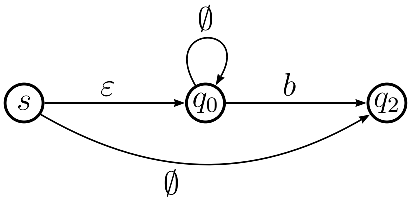


In other words, for any such $q_j$ there is guaranteed to be at least one $q_i$ and $q_k$ such that one step, labelled $r_1$, moves from $q_i$ to $q_j$, and one step, labelled $r_3$, moves from $q_j$ to $q_k$. In our example, the required pattern holds for $q_0$, in the following form:

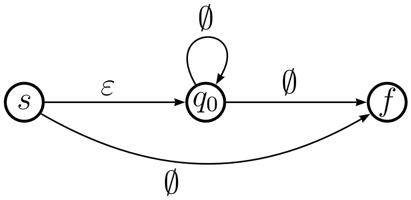


However, we have to consider all such patterns, *i.e.*, all pairs of states that $q_0$ lies between. There are surprisingly many (eight more, in all):

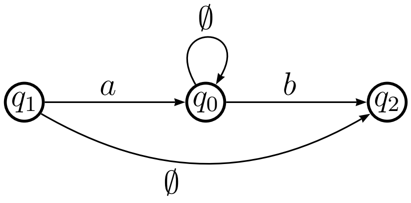
1.



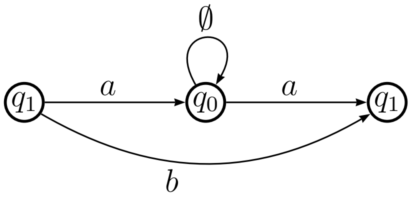
2.



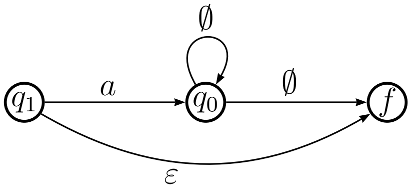
3.


$\emptyset$
$q_1$
$a$
$q_0$
$b$
$q_2$
$\emptyset$

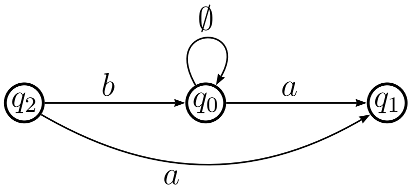
4.


$\emptyset$
$q_1$
$a$
$q_0$
$a$
$q_1$
$b$

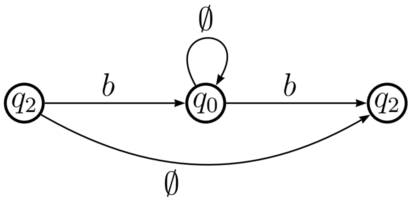
5.

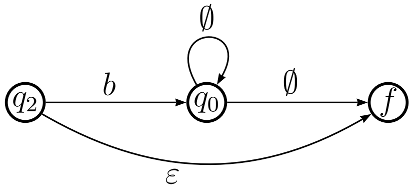

$\emptyset$
$q_1$
$a$
$q_0$
$\emptyset$
$f$
$\varepsilon$

6.


$\emptyset$
$q_2$
$b$
$q_0$
$a$
$q_1$
$a$

7.

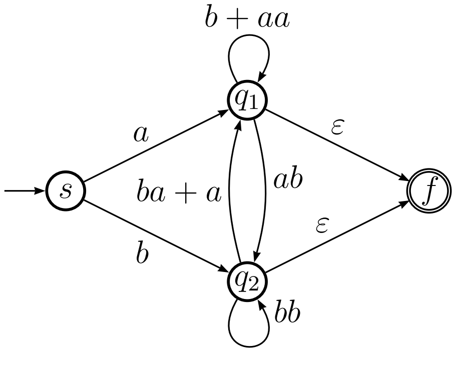

$\emptyset$
$q_2$
$b$
$q_0$
$b$
$q_2$
$\emptyset$

8. 

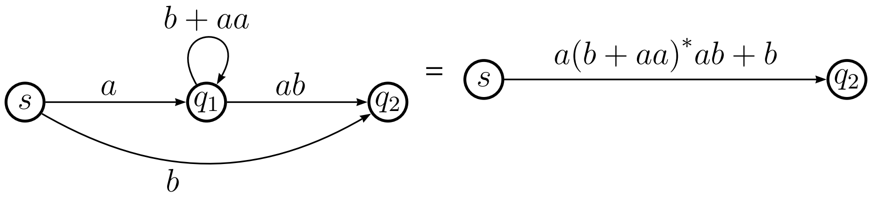
Now we apply our rule to get the following new transitions, which replace any old ones:

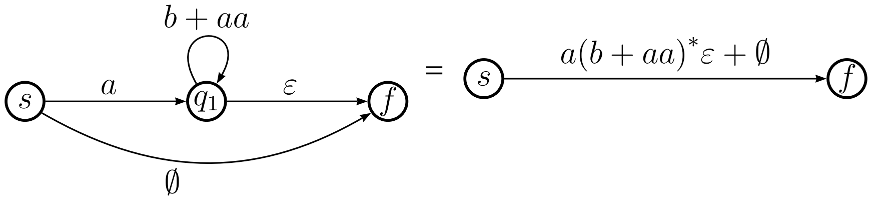
1. $s \xrightarrow{\varepsilon\emptyset^*a+\emptyset} q_1 = s \xrightarrow{a} q_1$
2. $s \xrightarrow{\varepsilon\emptyset^*b+\emptyset} q_2 = s \xrightarrow{b} q_2$
3. $s \xrightarrow{\varepsilon\emptyset^*\emptyset+\emptyset} f = s \xrightarrow{\emptyset} f$
4. $q_1 \xrightarrow{a\emptyset^*b+\emptyset} q_2 = q_1 \xrightarrow{ab} q_2$
5. $q_1 \xrightarrow{a\emptyset^*a+b} q_1 = q_1 \xrightarrow{aa+b} q_1$
6. $q_1 \xrightarrow{a\emptyset^*\emptyset+\varepsilon} f = q_1 \xrightarrow{\varepsilon} f$
7. $q_2 \xrightarrow{b\emptyset^*a+a} q_1 = q_2 \xrightarrow{ba+a} q_1$
8. $q_2 \xrightarrow{b\emptyset^*b+\emptyset} q_2 = q_2 \xrightarrow{bb} q_2$
9. $q_2 \xrightarrow{b\emptyset^*\emptyset+\varepsilon} f = q_2 \xrightarrow{\varepsilon} f$

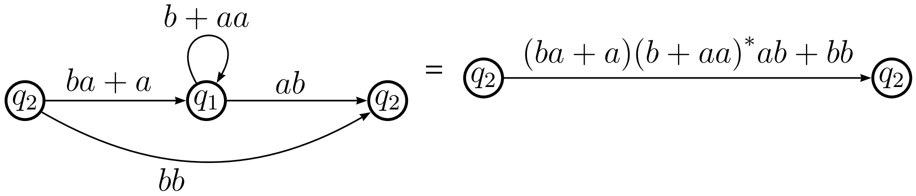
Thus $GNFA_1$ is

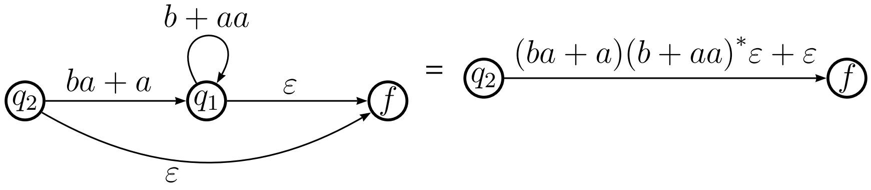


Now let's toss out $q_1$. We therefore have to consider the following cases:

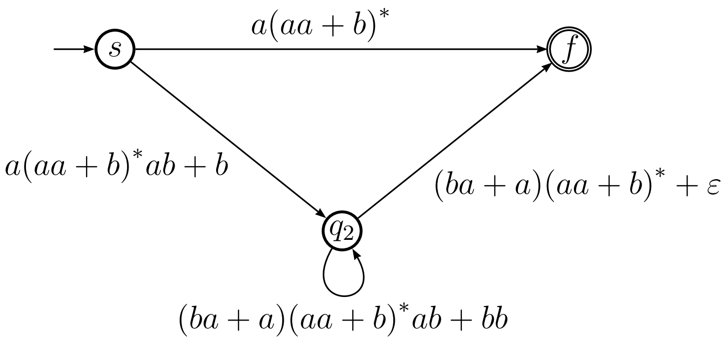
- 

- 

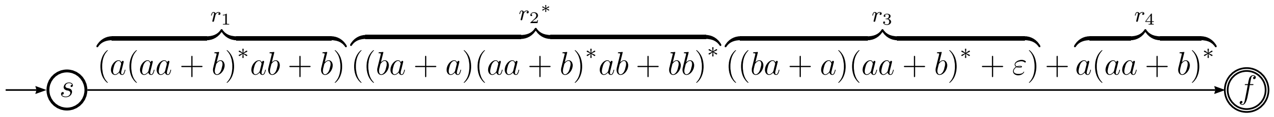
- 

- 

Thus $GNFA_2$ is :



And finally *GNFA*$_3$ is immediate:


$s$
$\overbrace{(a(aa+b)^*ab+b)}^{r_1}\overbrace{((ba+a)(aa+b)^*ab+bb)^*}^{{r_2}^*}\overbrace{((ba+a)(aa+b)^*+\varepsilon)}^{r_3}+\overbrace{a(aa+b)^*}^{r_4}$
$f$

### 7.1.4 Summary

We have now seen a detailed example of translating a DFA to a regular expression, the denotation of which is just the language accepted by the DFA. The translations used to convert back and forth can be used to prove the following important theorem.