Languages and Computation (COMP2012/G52LAC) Lecture notes Spring 2019

Thorsten Altenkirch, Venanzio Capretta, and Henrik Nilsson

March 1, 2019

Contents

List of exercises

1 Introduction

This module is about two fundamental notions in computer science, languages and computation, and how they are related. Specific topics include:

- Automata Theory
- Formal Languages
- Models of Computation
- Complexity Theory

The module starts with an investigation of classes of formal languages and related abstract machines, considers practical uses of this theory such as parsing, and finishes with a discussion on what computation is, what can and cannot be computed at all, and what can be computed efficiently, including famous results such as the Halting Problem and open problems such as P versus NP.

COMP2012/G52LAC builds on COMP1001/G51MSC Mathematics for Computer Scientists. It feeds into modules such as COMP3012/G53CMP Compilers, COMP3001/G53COM Computability, and COMP4001/G54FOP Mathematical Foundations of Programming. To give you a more concrete idea about what you will encounter in this module, as well as the broader significance of the module and a bit of historical context, we will illustrate with some examples.

1.1 Example: Valid Java programs

Consider the following Java fragment:

```
class Foo {
    int n;
    void printNSqrd() {
        System.out.println(n * n);
    }
}
```

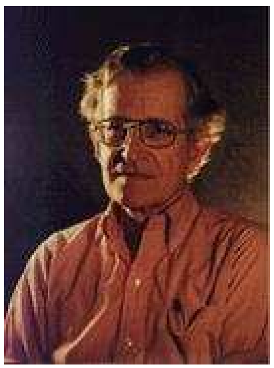
As written using a text editor or as stored in a file, it is just a string of characters. But not any string is a valid Java program. For example, Java uses specific keywords, have rules for what identifiers must look like, and requires proper nesting, such as a definition of a method inside a definition of a class.

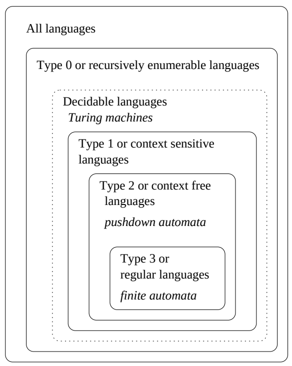
This raises a number of questions:

- How to describe the set of strings that are valid Java programs?
- Given a string, how to determine if it is a valid Java program or not?
- How to recover the structure of a Java program from a "flat" string?

To answer such questions, we will study *regular expressions* and *grammars* to give precise descriptions of languages, and various kinds of *automata* to decide if a string belongs to a language or not. We will also consider how to systematically derive programs that efficiently answer this type of questions, drawing directly from the theory. Such programs are key parts of compilers, web browsers and web servers, and in fact of any program that uses structured text in one way or another.

A little bit of history. Context-free grammars were invented by American linguist, philosopher, and cognitive scientist *Noam Chomsky* (1928–) in an attempt to describe natural languages formally. He also introduced the *Chomsky Hierarchy* which classifies grammars and languages and their descriptive power:





1.2 Example: The halting problem

Consider the following program. Does it terminate for all values of $\texttt{n} \geq 1$?

```
while (n > 1) {
    if even(n) {
        n = n / 2;
    } else {
        n = n * 3 + 1;
    }
}
```

This is not as easy to answer as it might first seem. Say we start with $\texttt{n} = 7$, for example:

7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

Note how the numbers both increase and decrease in a way that is very hard to describe, which is exactly why it is so hard to analyse this program. The sequence involved is known as the *hailstone sequence*, and *Collatz conjecture* says that the number 1 will always be reached. And, in fact, for all numbers that have been tried (all numbers up to 2^{60}!), the sequence does indeed terminate. But so far, no one has been able to *prove* that it always will! The famous mathematician Paul Erdős even said: "Mathematics may not be ready for such problems." (See Collatz conjecture, Wikipedia.)

The following important decidability result should then perhaps not come as a total surprise:

> It is impossible to write a program that decides if another, *arbitrary*, program terminates (halts) or not.

This is known as the *Halting Problem* and it is thus one example of an *undecidable* problem: the answer cannot be determined mechanically in general[1].

The undecidability of the Halting Problem was first proved by British mathematician, logician, and computer scientist *Alan Turing* (1912–1954):

Turing proved this result using *Turing Machines*, an abstract model of computation that he introduced in 1936 to give a precise definition of what problems are "effectively calculable" (can be solved mechanically). Turing was further instrumental in the success of British code breaking efforts during World War II and is also famous for the Turing test to decide if a machine exhibits intelligent behaviour equivalent to, or indistinguishable from, that of a human. Andrew Hodges has written a very good biography of Turing: *Alan Turing: the Enigma* (`http://www.turing.org.uk/turing/`).

[1] This does not mean that it is impossible to compute the answer for every instance of such a problem. On the contrary, in specific cases, the answer can often be computed very easily, and programs that attempt to solve undecidable problems can be very useful. But if we do write such a program, we must necessarily be prepared to give up one way or another with a "don't know" answer.

1.3 Example: The λ-calculus

The λ-calculus is a theory of pure functions. It is very simple, having only two constructs: definition and application of functions. The folloing is an example of a λ-calculus term:

$$(\lambda x.x)(\lambda y.y)$$

Like Turing machines, the λ-calculus is a universal model of computation. It was introduced by American mathematician and logician *Alonzo Church* (1903–1995), also in 1936, a little earlier than Turing's machine:

Alan Turing subsequently became a PhD student of Alonzo Church. They proved that Turing machines and the λ-calculus are equivalent in terms of computational expressiveness. In fact, all proposed universal models of computation to date have proved to be equivalent in that sense. This is captured by the *Church-Turing thesis*:

> What is effectively calculable is exactly what can be computed by a Turing machine.

Functional programming languages, like Haskell, and many proof assistants implement (variations of) the λ-calculus. This is thus an example of a theory with very direct and practical applications.

1.4 P versus NP

Here is a seemingly innocuous question:

> Can every problem whose solution can be *checked* quickly by a computer also be *solved* quickly by a computer?"

"Quickly" here means in time proportional to a *polynomial* in the size of the problem. Whether or not this is ths case is known as the *P versus NP problem*

and it is likely the most famous open problem in computer science, dating back to the 1950s. Here, "P" refers to the class of problems that can be solved in polynomial time, while "NP" refers to problems that can be solved in nondeterministic polynomial time, and the question is thus whether these two classes of problems actually are the same, or P = NP.

There is an abundance of important problems where solutions can be checked quickly, but where the best *known* algorithm for finding a solution is *exponential* in the size of the problem.

As an example, here is one, *the Subset Sum Problem*: Does some nonempty subset of given set of integers sum to zero? For example, given $\{3, -2, 8, -5, 4, 9\}$, the nonempty subset $\{-5, -2, 3, 4\}$ sums to 0.

It is easy to check a proposed solution: just add all the numbers. If the initial set contains n integers, any proposed solution (being a subset) contains at most n integers, so we can sum all the elements with at most n additions meaning the total time taken is proportional to n (assuming addition is a constant time operation).

However, for *finding* a solution, no better way is known than essentially checking each possible subset one after another. As there are 2^n subsets of a set with n elements, this means finding a solution takes exponential time.

Whether or not there is a better way to solve the Subset Sum Problem might not seem particularly important, but if it were the case that P = NP, then this would have monumental practical implications. For example, public key cryptography, on which pretty much all secure Internet communication, such as HTTPS, hinges, would no longer provide adequate security, and the entire Internet security infrastructure would have to be redesigned and reimplemented. The question here is if it is possible to quickly find the prime factors of (very) large numbers. As long as that is *not* the case, public key cryptography is considered secure.

2 Formal Languages

In this course we will use the terms *language* and *word* in a different way than in everyday language:

- A *language* is a set of words.
- A *word* is a sequence, or string, of symbols.

We will write ϵ for the *empty* word; i.e., a sequence of length 0.

This leaves us with the question: what is a symbol? The answer is: anything, but it has to come from an alphabet Σ that is a finite set. A common (and important) instance is $\Sigma = \{0, 1\}$. Note that ϵ will never be a symbol to avoid confusion.

Mathematically we say: Given an alphabet Σ we define the set Σ^* as set of words (or sequences) over Σ: the empty word $\epsilon \in \Sigma^*$ and given a symbol $x \in \Sigma$ and a word $w \in \Sigma^*$ we can form a new word $xw \in \Sigma^*$. These are all the ways elements on Σ^* can be constructed (this is called an *inductive definition*). This unary $*$-operator is known as the *Kleene star* (or *Kleene operator* or *Kleene closure*).

With $\Sigma = \{0, 1\}$, typical elements of Σ^* are 0010, 00000000,ϵ. Note, that we only write ϵ if it appears on its own, instead of 00ϵ we just write 00.

Note further that Σ^* by definition is *always* nonempty as the empty word ϵ belongs to Σ^* for *any* alphabet Σ, including $\Sigma = \emptyset$. Moreover, for any nonempty alphabet Σ, Σ^* is an infinite set.

It is important to realise that although there are infinitely many words over a nonempty alphabet Σ, each word has a finite length. At first this may seem strange: how can it be that all elements of a set with infinitely many elements can be finite? A good way to think of an infinite set is as a *process* that can generate a new element whenever we need one, as many times as we like[2]. But each such element can obviously be of finite size as we at any point in time will only have asked for finitely many elements. Conversely, if we make a set containing a single (notionally) "infinite" element, such as a number ∞ larger than any number except itself, or an infinitely long string, that does not make the set itself infinite: it would still contain exactly one element.

We can now define the notion of a language L over an alphabet Σ precisely: $L \subseteq \Sigma^*$ or equivalently $L \in \mathcal{P}(\Sigma^*)$[3].

Here are some informal examples of languages:

- The set $\{0010, 00000000, \epsilon\}$ is a language over $\Sigma = \{0, 1\}$. This is an example of a finite language.
- The set of words with odd length over $\Sigma = \{1\}$.
- The set of words that contain the same number of 0s and 1s is a language over $\Sigma = \{0, 1\}$.

[2] Indeed, this is *exactly* how infinite data structures, such as infinite lists, are realised in lazy languages like Haskell.

[3] Given a set A, $\mathcal{P}(A)$ is the *powerset* of A; that is, the set of all possible subsets of A. For example, if $A = \{a, b\}$, then $\mathcal{P}(A) = \{\,\emptyset, \{a\}, \{b\}, \{a, b\}\,\}$. The number of elements $|A|$ of a set A, its *cardinality*, and the number of elements in its power set are related by $|\mathcal{P}(A)| = 2^{|A|}$. Hence powerset.

- The set of words that contain the same number of 0s and 1s modulo 2 (i.e., both are even or odd) is a language over $\Sigma = \{0, 1\}$.
- The set of palindromes using the English alphabet, e.g. words that read the same forwards and backwards like `abba`. This is a language over $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$.
- The set of correct Java programs. This is a language over the set of UNICODE "characters" (which correspond to numbers between 0 and $17 \cdot 2^{16} - 1$, less some invalid subranges, 1112062 valid encodings in all).
- The set of programs that, if executed on a Windows machine, prints the text "Hello World!" in a window. This is a language over $\Sigma = \{0, 1\}$.

Note the distinction between ϵ, $\emptyset$, and $\{\epsilon\}$!

- ϵ denotes the empty word, a *sequence* of symbols of length 0.
- $\emptyset$ denotes the empty *set*, a set with no elements.
- $\{\epsilon\}$ is a set with exactly *one* element: the empty word.

In particular, note that ϵ is a different type (a sequence) from $\emptyset$ and $\{\epsilon\}$ (that are both sets).

An important operation on Σ^* is concatenation. This is denoted by juxtapositioning (or, if you prefer, by an "invisible operator"): given $u, v \in \Sigma^*$ we can construct a new word $uv \in \Sigma^*$ simply by concatenating the two words. We can define this operation by primitive recursion:

$$\begin{aligned} \epsilon v &= v \\ (xu)v &= x(uv) \end{aligned}$$

Concatenation is associative and has unit ϵ:

$$\begin{aligned} u(vw) &= (uv)w \\ \epsilon u = u &= u\epsilon \end{aligned}$$

where u, v, w are words. We use exponent notation to denote concatenation of a word with itself. For example, $u^2 = uu$, $u^3 = uuu$, and so on. By definition, $u^1 = u$ and $u^0 = \epsilon$, the unit of concatenation. Thus we can simplify repeated concatenation using familiar-looking laws. For example: $u^1u^0u^2 = u^3$.

Concatenation of words is extended to concatenation of languages by:

$$MN = \{uv \mid u \in M \wedge v \in N\}$$

For example:

$$\begin{aligned} M &= \{\epsilon, a, aa\} \\ N &= \{b, c\} \\ MN &= \{uv \mid u \in \{\epsilon, a, aa\} \wedge v \in \{b, c\}\} \\ &= \{\epsilon b, \epsilon c, ab, ac, aab, aac\} \\ &= \{b, c, ab, ac, aab, aac\} \end{aligned}$$

Some important properties of language concatenation are:

- Concatenation of languages is associative:

$$L(MN) = (LM)N$$

- Concatenation of languages has zero $\emptyset$:

$$L\emptyset = \emptyset = \emptyset L$$

- Concatenation of languages has unit $\{\epsilon\}$:

$$L\{\epsilon\} = L = \{\epsilon\}L$$

- Concatenation distributes through set union:

$$\begin{aligned} L(M \cup N) &= LM \cup LN \\ (L \cup M)N &= LN \cup MN \end{aligned}$$

Note that concatenation does *not* distribute through intersection! Counterexample. Let $L = \{\epsilon, a\}$, $M = \{\epsilon\}$, $N = \{a\}$. Then:

$$\begin{array}{l} L(M \cap N) = L\emptyset = \emptyset \\ LM \cap LN = \{\epsilon, a\} \cap \{a, aa\} = \{a\} \end{array}$$

Exponent notation is used to denote iterated language concatenation: $L^1 = L$, $L^2 = LL$, $L^3 = LLL$, and so on. By definition, $L^0 = \{\epsilon\}$ (for *any* language, including $\emptyset$), which is the unit for language concatenation (just as $u^0 = \epsilon$ is the unit for concatenation of words).

The Kleene star can also be applied to languages. This intuitively means language concatenation iterated 0 or more times:

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

Note that $\epsilon \in L^*$ for any language L, including $L = \emptyset$, the empty language. As an example, if $L = \{\mathsf{a}, \mathsf{ab}\}$, then $L^* = \{\epsilon, \mathsf{a}, \mathsf{ab}, \mathsf{aab}, \mathsf{aba}, \mathsf{aaab}, \mathsf{aaba}, \ldots\}$.

Alternatively (and more abstractly), L^* can be described as the least language (with respect to $\subseteq$) that contains L and the empty word, ϵ, and is closed under concatenation:

$$u \in L^* \wedge v \in L^* \implies uv \in L^*$$

Note the subtle difference between using the Kleene star on an alphabet Σ, a set of *symbols*, as in Σ^*, and on using the Kleene star on a language $L \subseteq \Sigma^*$, a set of *words*. While the result in both cases is a set of words, the types of the arguments to the two variants of the Kleene star operation differ.

2.1 Exercises

Exercise 2.1

Let the alphabet $\Sigma = \{3, 5, 7, 9\}$, and let the language $L = \{w \mid w \in \Sigma^*, 1 \leq |w| \leq 2\}$. (If w is a word, $|w|$ denotes the length of that word. If X is a finite set, like an alphabet or finite language, $|X|$ denotes the number of elements in that set, its *cardinality*.) Answer the following questions:

1. Describe L in plain English.
2. Enumerate all the words in L.
3. In general, for an arbitrary alphabet Σ_1 and $0 \leq m \leq n$, how many words are there in the language $L_1 = \{w \mid w \in \Sigma_1^*, m \leq |w| \leq n\}$? That is, write down an expression for $|L_1|$.
4. How many words would there be in L_1 if $\Sigma_1 = \Sigma$, $m = 3$, and $n = 7$?

Exercise 2.2

Let the alphabet $\Sigma = \{a, b, c\}$ and let $L_1 = \{\epsilon, b, ac\}$ and $L_2 = \{a, b, ca\}$ be two languages over Σ. Enumerate the words in the following languages, showing your calculations in some detail:

1. $L_3 = L_1 \cup L_2$
2. $L_4 = L_1\{\epsilon\}(L_2 \cap L_1)$
3. $L_5 = L_3\emptyset L_4$

Exercise 2.3

Let the alphabet $\Sigma = \{a, b, c\}$ and let $L_1 = \{\epsilon, b, bb\}$ and $L_2 = \{a, ab, abc\}$ be two languages over Σ. Enumerate the words in the following languages, showing your calculations in some detail:

1. $L_3 = L_1 \cap L_2$
2. $L_4 = (L_2\{\epsilon\}L_1) \cap \Sigma^*$
3. $L_5 = L_3\emptyset \cap L_4$

Exercise 2.4

Let the alphabet $\Sigma = \{a, b, c\}$. Enumerate the words in

$$L = \{w \mid w \in \{\epsilon, a, b, bc\}^*, |w| \leq 3\}$$

3 Finite Automata

Finite automata correspond to a computer with a fixed finite amount of memory[4]. We will introduce *deterministic finite automata* (DFA) first and then move to *nondeterministic finite automata* (NFA). An automaton will accept certain words (sequences of symbols of a given alphabet Σ) and reject others. The set of accepted words is called the language of the automaton. We will show that the class of languages that are accepted by DFAs and NFAs is the same.

3.1 Deterministic finite automata

3.1.1 What is a DFA?

A *deterministic finite automaton* (DFA) $A = (Q, \Sigma, \delta, q_0, F)$ is given by:

1. A finite set of *states* Q
2. A finite set of input symbols, the *alphabet*, Σ
3. A *transition function* $\delta \in Q \times \Sigma \to Q$
4. An *initial state* $q_0 \in Q$
5. A set of *final states* $F \subseteq Q$

The initial states are sometimes called *start states*, and the final states are sometimes called *accepting states*.

As an example consider the following automaton

$$D = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta_D, q_0, \{q_2\})$$

where

$$\delta_D = \{((q_0, 0), q_1), ((q_0, 1), q_0), ((q_1, 0), q_1), ((q_1, 1), q_2), ((q_2, 0), q_2), ((q_2, 1), q_2)\}$$

if we view a function as a set of argument-result pairs. Alternatively, we can define it case by case:

$$\begin{aligned}
\delta_D(q_0, 0) &= q_1 \\
\delta_D(q_0, 1) &= q_0 \\
\delta_D(q_1, 0) &= q_1 \\
\delta_D(q_1, 1) &= q_2 \\
\delta_D(q_2, 0) &= q_2 \\
\delta_D(q_2, 1) &= q_2
\end{aligned}$$

A DFA may be more conveniently represented by a *transition table*. The transition table for the DFA D is:

	δ_D	0	1
$\to$	q_0	q_1	q_0
	q_1	q_1	q_2
$*$	q_2	q_2	q_2

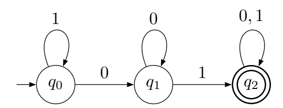
[4] However, that does not mean that finite automata are a good model of general purpose computers. A computer with n bits of memory has 2^n possible states. That is an absolutely enormous number even for very modest memory sizes, say 1024 bits or more, meaning that describing a computer using finite automata quickly becomes infeasible. We will encounter a better model of computers later, the *Turing Machines*.

A transition table represents the transition function δ of a DFA; i.e., the value of $\delta(q, x)$ is given by the row labelled q in the column labelled x. In addition, the initial state is identified by putting an arrow $\to$ to the left of it, and all final states are similarly identified by a star $*$. The inclusion of this additional information makes a transition table a self-contained representation of a DFA.

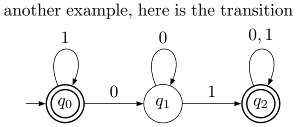
Note that the initial state also can be final (accepting). For example, for a variation D' of the DFA D where q_0 also is final:

	$\delta_{D'}$	0	1
$\to *$	q_0	q_1	q_0
	q_1	q_1	q_2
$*$	q_2	q_2	q_2

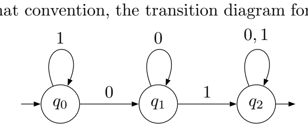
Another way to represent a DFA is through a *transition diagram*. The transition diagram for the DFA D is:



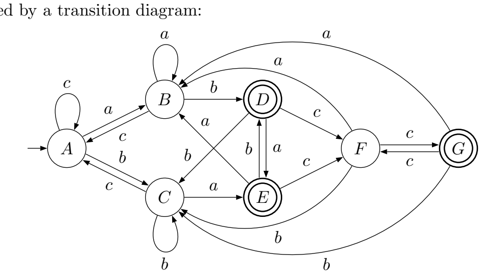
The initial state is identified by an incoming arrow. Final states are drawn with a double outline. If $\delta(q, x) = q'$ then there is an arrow from state q to q' that is labelled x. For another example, here is the transition diagram for the DFA D':



An alternative to the double outline for a final state is to use an outgoing arrow. Using that convention, the transition diagram for the DFA D is:



Here is an example of a larger DFA over the alphabet $\Sigma = \{a, b, c\}$ represented by a transition diagram:



The states are named by capital letters this time for a bit of variation: $Q = \{A, B, C, D, E, F, G\}$. While it is common to use symbols q_i, $i \in \mathbb{N}$ to name states, we can pick any names we like. Another common choice is to use natural numbers; i.e., $Q \subset \mathbb{N} \wedge Q$ is finite.

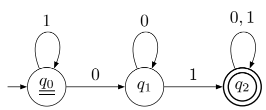
The representation of the above DFA as a transition table is:

	δ	a	b	c
$\rightarrow$	A	B	C	A
	B	B	D	A
	C	E	C	A
$*$	D	E	C	F
$*$	E	B	D	F
	F	B	C	G
$*$	G	B	C	F

3.1.2 The language of a DFA

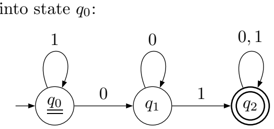
We will now discuss how a DFA accepts or rejects words over its alphabet of input symbols. The set of words accepted by a DFA A is called the *language* $L(A)$ of the DFA. Thus, for a DFA A with alphabet Σ, $L(A) \subseteq \Sigma^*$.

To determine whether a word $w \in L(A)$, the machine starts in its initial state. Taking the DFA D above as an example, it would start in state q_0. We indicate the state of a DFA by underlining the state name:

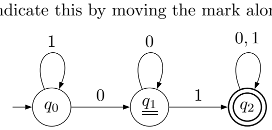


Then, whenever an input symbol is read from w, the machine transitions to a new state by following the edge labelled with this symbol. Once all symbols in the input word w have been read, the word is *accepted* if the state is final, meaning $w \in L(A)$, otherwise the word is *rejected*, meaning $w \notin L(A)$.

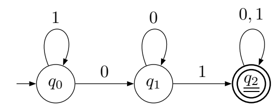
To continue with the example, suppose $w = 101$. The machine D would thus first read 1 and transition to a new state by following the edge labelled 1. As that edge in this case forms a loop back to state q_0, the machine D would transition back into state q_0:



The machine would then read 0 and transition to state q_1 by following the edge labelled 0. We indicate this by moving the mark along that edge to q_1:



Finally, the machine would read the last 1 in the input word, moving to q_2:



As q_2 is a final state, the DFA D accepts the word $w = 101$, meaning $101 \in L(D)$. In the same way, we can determine that $0 \notin L(D)$, $110 \notin L(D)$, but $011 \in L(D)$. Verify this. Indeed, a little bit of thought reveals that

$$L(D) = \{w \mid w \text{ contains the substring } 01\}$$

To make the notion of the language of a DFA precise, we now give a formal definition of $L(A)$. First we define the *extended transition function* $\hat{\delta} \in Q \times \Sigma^* \rightarrow Q$. Intuitively, $\hat{\delta}(q, w) = q'$ if the machine starting from state q ends up in state q' when reading the word w. Formally, $\hat{\delta}$ is defined by primitive recursion:

$$\hat{\delta}(q, \epsilon) = q \qquad (1)$$

$$\hat{\delta}(q, xw) = \hat{\delta}(\delta(q, x), w) \qquad (2)$$

where $x \in \Sigma$ and $w \in \Sigma^*$. Thus, xw stands for a nonempty word the first symbol of which is x and the rest of which is w. For example, if $xw = 010$, then $x = 0$ and $w = 10$. Note that w may be empty; e.g., if $xw = 0$, then $x = 0$ and $w = \epsilon$.

As an example, we calculate $\hat{\delta}_D(q_0, 101) = q_1$:

$$\begin{array}{rcll}
\hat{\delta}_D(q_0, 101) & = & \hat{\delta}_D(\delta_D(q_0, 1), 01) & \text{by (2)} \\
& = & \hat{\delta}_D(q_0, 01) & \text{because } \delta_D(q_0, 1) = q_0 \\
& = & \hat{\delta}_D(\delta_D(q_0, 0), 1) & \text{by (2)} \\
& = & \hat{\delta}_D(q_1, 1) & \text{because } \delta_D(q_0, 0) = q_1 \\
& = & \hat{\delta}_D(\delta_D(q_1, 1), \epsilon) & \text{by (2)} \\
& = & \hat{\delta}_D(q_2, \epsilon) & \text{because } \delta_D(q_1, 1) = q_2 \\
& = & q_2 & \text{by (1)}
\end{array}$$

Using the extended transition function $\hat{\delta}$, we define the language $L(A)$ of a DFA A formally:

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$$

Returning to our example, we thus have that $101 \in L(D)$ because $\hat{\delta}_D(q_0, 101) = q_2$ and $q_2 \in F_D$.

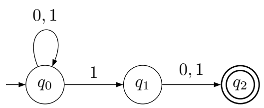
3.2 Nondeterministic finite automata

3.2.1 What is an NFA?

Nondeterministic finite automata (NFA) have transition functions that map a given state and an input symbol to zero or more successor states. We can think of this as the machine having a "choice" whenever there are two or more possible transitions from a state on an input symbol. In this presentation, we will further allow an NFA to have more than one initial state[5]. Again, we can think of this as the machine having a “choice” of where to start. An NFA accepts a word w if there is at least one *possible* way to get from one of the initial states to one of the final states along edges labelled with the symbols of w in order.

It is important to note that although an NFA has a nondetermistic transition function, it can always be determined whether or not a word belongs to its language. Indeed, we shall see that every NFA can be translated into an DFA that accepts the same language.

Here is an example of an NFA C that accepts all words over $\Sigma = \{0, 1\}$ such that the symbol before the last is 1:



A *nondeterministic finite automaton* (NFA) $A = (Q, \Sigma, \delta, S, F)$ is given by:

1. A finite set of *states* Q,
2. A finite set of input symbols, the *alphabet*, Σ,
3. A *transition function* $\delta \in Q \times \Sigma \to \mathcal{P}(Q)$,
4. A set of *initial states* $S \subseteq Q$,
5. A set of *final* (or *accepting*) states $F \subseteq Q$.

Thus, in contrast to a DFA, an NFA may have many initial states, not just one, and its transition function maps a state and an input symbol to a set of possible successor states, not just a single state. As an example we have that

$$C = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta_C, \{q_0\}, \{q2\})$$

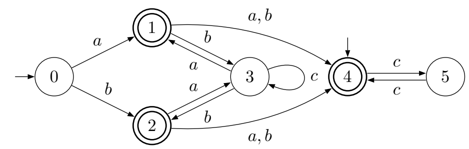
where δ_C is given by

δ_C	0	1
$\to\ q_0$	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
$*\ q_2$	$\emptyset$	$\emptyset$

Note that the entries in the table are *sets* of states, and that these sets may be empty ($\emptyset$), here exemplified by the entries for state q_2. Again, the (in this case only) initial state has been marked with $\to$ and the (in this case only) final state marked with $*$ to make this a self-contained representation of the NFA.

Here is another example of an NFA, this time over the alphabet $\Sigma = \{a, b, c\}$ and with states $Q = \{0, 1, 2, 3, 4, 5\} \subset \mathbb{N}$:

[5] Note that we diverge slightly from the definition in the book [HMU01], which uses a single initial state instead of a set of initial states. Permitting more than one initial state allows us to avoid introducing ϵ-NFAs (see [HMU01], section 2.5).



The transition table for this NFA is:

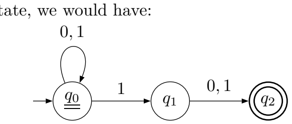
δ	a	b	c
$\to\ 0$	$\{1\}$	$\{2\}$	$\emptyset$
$*\ 1$	$\{4\}$	$\{3, 4\}$	$\emptyset$
$*\ 2$	$\{3, 4\}$	$\{4\}$	$\emptyset$
3	$\{1\}$	$\{2\}$	$\{3\}$
$\to *\ 4$	$\emptyset$	$\emptyset$	$\{5\}$
5	$\emptyset$	$\emptyset$	$\{4\}$

Note that this NFA has multiple initial states, multiple final states, one initial state that also is final, and that there in some cases are no possible successor states and in other cases more than one.

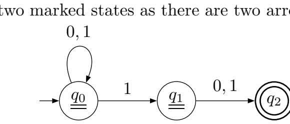
3.2.2 The language accepted by an NFA

To see whether a word $w \in \Sigma^*$ is accepted by an NFA A, we have to consider *all* possible states the machine could be in after having read a sequence of input symbols. Initially, an NFA can be in any of its initial states. Each time an input symbol is read, all successor states on the read symbol for each current possible state become the new possible states. After having read a complete word w, if at least one of the possible states is final (accepting), then that word is accepted, meaning $w \in L(A)$, otherwise it is rejected, meaning $w \notin L(A)$.

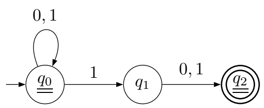
We will illustrate by showing how the NFA C rejects the word 100. We will again mark the current states of the NFA by underlining the state names, but this time there may be more than one marked state at once. Initially, as q_0 is the only initial state, we would have:



Each time when we read a symbol we look at all the marked states. We remove the old markers and put markers at all the states that are *reachable* via an arrow marked with the current input symbol. This may include one or more states that were marked previously. It may also be the case that no states are reachable, in which case all marks are removed and the word rejected (as it no longer is possible to reach any final states). In our example, after reading 1, there would be two marked states as there are two arrows from q_0 labelled 1:

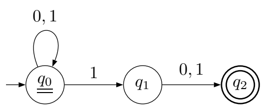


After reading 0, the next symbol in the word 100, there would still be two marked states as the machine on input 0 can reach q_0 from q_0 and q_2 from q_1:



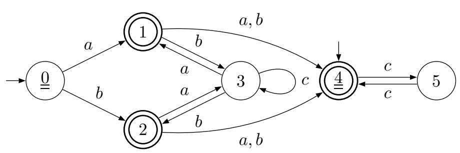
Note that one of the marked states is a final (accepting) state, meaning the word read so far (10) is accepted by the NFA.

However, there is one symbol left in our example word 100, and after having read the final 0, the final state would no longer be marked because it cannot be reached from any of the marked states:

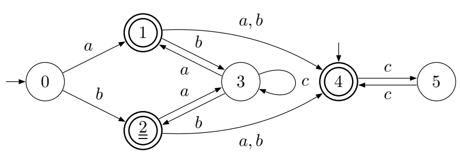


The NFA C thus rejects the word 100.

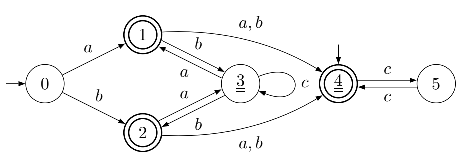
For another example, consider the NFA at the end of section 3.2.1. Convince yourself that you understand how this NFA accepts the words ϵ, *abcc*, *abcca*, and rejects *abccaac*. We illustrate by tracing its operation on the word *bacac*. We start by marking all initial states. Then it is just a matter of systematically exploring all possibilities:



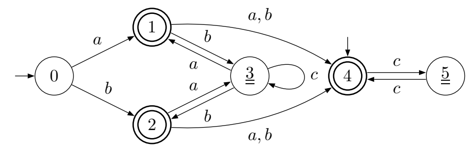
After reading b:



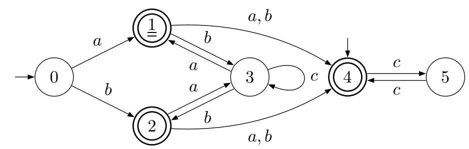
After reading a:



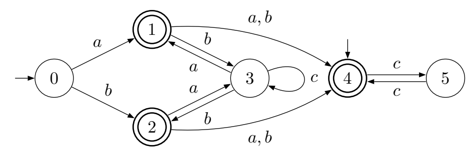
After reading c:



After reading a:



After reading c:



The machine thus rejects *bacac* as no final state is marked. In fact, as there are *no* marked states left at all, this shows that this NFA will reject *all* words that start *bacac*. Can you find other such prefixes?

To define the *extended transition function* $\hat{\delta}$ for NFAs we use a generalisation of the union operation $\cup$ on sets over a (finite) set of sets:

$$\bigcup\{A_1, A_2, \dots A_n\} = A_1 \cup A_2 \cup \dots \cup A_n$$

In the special cases of the empty set of sets and a one element set of sets:

$$\bigcup\emptyset = \emptyset \qquad \bigcup\{A\} = A$$

As an example

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

Alternatively, we can define $\bigcup$ by comprehension, which also extends the operation to infinite sets of sets (although we don't need this here):

$$\bigcup B = \{x \mid \exists A \in B . x \in A\}$$

We define $\hat{\delta} \in \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ such that $\hat{\delta}(P, w)$ is the set of states that are reachable from one of the states in P on the word w:

$$\hat{\delta}(P, \epsilon) = P \tag{3}$$

$$\hat{\delta}(P, xw) = \hat{\delta}(\bigcup\{\delta(q, x) \mid q \in P\}, w) \tag{4}$$

where $x \in \Sigma$ and $w \in \Sigma^*$. Intuitively, if P are the possible states, then $\hat{\delta}(P, w)$ are the possible states after having read a word w.

To illustrate, we calculate $\hat{\delta}_C(q_0, 100)$:

$$\begin{array}{rcll}
\hat{\delta}_C(\{q_0\}, 100) & = & \hat{\delta}_C(\bigcup\{\delta_C(q, 1) \mid q \in \{q_0\}\}, 00) & \text{by (4)} \\
& = & \hat{\delta}_C(\delta_C(q_0, 1), 00) & \\
& = & \hat{\delta}_C(\{q_0, q_1\}, 00) & \\
& = & \hat{\delta}_C(\bigcup\{\delta_C(q, 0) \mid q \in \{q_0, q_1\}\}, 0) & \text{by (4)} \\
& = & \hat{\delta}_C(\delta_C(q_0, 0) \cup \delta_C(q_1, 0), 0) & \\
& = & \hat{\delta}_C(\{q_0\} \cup \{q_2\}, 0) & \\
& = & \hat{\delta}_C(\{q_0, q_2\}, 0) & \\
& = & \hat{\delta}_C(\bigcup\{\delta_C(q, 0) \mid q \in \{q_0, q_2\}\}, \epsilon) & \text{by (4)} \\
& = & \hat{\delta}_C(\delta_C(q_0, 0) \cup \delta_C(q_2, 0), 0) & \\
& = & \hat{\delta}_C(\{q_0\} \cup \emptyset, \epsilon) & \\
& = & \{q_0\} & \text{by (3)}
\end{array}$$

Of course, we already knew this from the worked example above illustrating how the NFA C rejects 100. Make sure you see how the marked states after each step coincides with the set of possible states in the calculation.

The language of an NFA can now be defined using $\hat{\delta}$:

$$L(A) = \{w \mid \hat{\delta}(S, w) \cap F \neq \emptyset\}$$

Thus, $100 \notin L(C)$ because

$$\hat{\delta}_C(S_C, 100) \cap F_C = \hat{\delta}_C(\{q_0\}, 100) \cap \{q_2\} = \{q_0\} \cap \{q_2\} = \emptyset$$

3.2.3 The subset construction

DFAs can be viewed as a special case of NFAs; i.e., those for which the there is precisely one start state ($S = \{q_0\}$) and for which the transition function always returns singleton (one-element) sets ($\delta(q, x) = \{q'\}$ for all $q \in Q$ and $x \in \Sigma$).

The opposite is also true, however: NFAs are really just DFAs "in disguise". We show this by for a given NFA systematically constructing an *equivalent* DFA; i.e., a DFA that accepts the same language as the given NFA. NFAs are thus no more powerful than DFAs; i.e., NFAs cannot describe more languages than DFAs. However, in some cases, NFAs need a lot fewer states than the corresponding DFA, and they may be easier to construct in the first place.

The subset construction: Given an NFA $A = (Q, \Sigma, \delta, S, F)$ we construct the equivalet DFA:

$$D(A) = (\mathcal{P}(Q), \Sigma, \delta_{D(A)}, S, F_{D(A)})$$

where

$$\delta_{D(A)}(P, x) = \bigcup\{\delta(q, x) \mid q \in P\} \tag{5}$$

$$F_{D(A)} = \{P \in \mathcal{P}(Q) \mid P \cap F \neq \emptyset\} \tag{6}$$

The basic idea of this construction is to define a DFA whose states are *sets of NFA states*. A set of possible NFA states thus becomes a single DFA state. The DFA transition function is given by considering all reachable NFA states for each of the current possible NFA states for each input symbol. The resulting set of possible NFA states is again just a single DFA state. A DFA state is final if that set that contains at least one final NFA state.

As an example, let us construct a DFA $D(C)$ equivalent to C above:

$$D(C) = (\mathcal{P}(\{q_0, q_1, q_2\}, \{0, 1\}, \delta_{D(C)}, \{q_0\}, F_{D(C)})$$

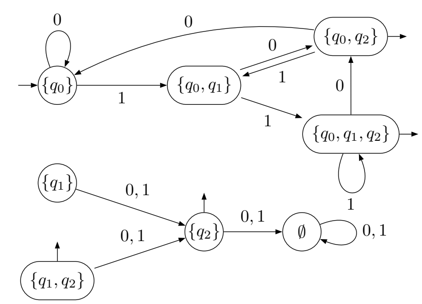
where $\delta_{D(C)}$ is given by:

	$\delta_{D(C)}$	0	1
	$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow$	$\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
	$\{q_1\}$	$\{q_2\}$	$\{q_2\}$
$*$	$\{q_2\}$	$\emptyset$	$\emptyset$
	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
$*$	$\{q_0, q_2\}$	$\{q_0\}$	$\{q_0, q_1\}$
$*$	$\{q_1, q_2\}$	$\{q_2\}$	$\{q_2\}$
$*$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$

and $F_{D(C)}$ (all the states marked with $*$ above) by:

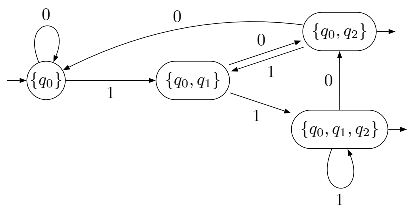
$$F_{D(C)} = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$$

The transition diagram is:



Accepting states have been marked by outgoing arrows.

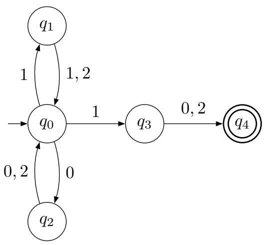
Note that some of the states ($\emptyset, \{q_1\}, \{q_2\}, \{q_1, q_2\}$) cannot be reached from the initial state. This means that they can be omitted without changing the language. We thus obtain the following automaton:



It is possible to avoid having to perform calculations for states that cannot be reached by carrying out the subset construction in a "demand driven" way. The idea is to start from the initial DFA state, which is just the set of initial NFA states S, and then only consider the DFA states (subsets of NFA states) that appear during the course of the calculations. We illustrate this approach by an example. Consider the following NFA N:

$$N = (Q_N = \{q_0, q_1.q_2, q_3, q_4\},\ \Sigma_N = \{0,\, 1,\, 2\},\ \delta_N,\ S_N = \{q_0\},\ F_N = \{q_4\})$$

where δ_N is given by the transition diagram:



Note that N has 5 states which means that the DFA $D(N)$ has $|\mathcal{P}(Q_N)| = 2^5 = 32$ states. However, as we will see, only a handful of those 32 states can actually be reached from the initial state S_N of $D(N)$. We would thus waste quite a bit of effort if we were to tabulate all of them.

We start from $S_N = \{q_0\}$, the set of start states of N, and we we compute $\bigcup\{\delta(q, x) \mid q \in S_N\}$ for each $x \in \Sigma_N$ (equation (5)). In this case we get:

	$\delta_{D(N)}$	0	1	2
$\rightarrow$	$\{q_0\}$	$\{q_2\}$	$\{q_1, q_3\}$	$\emptyset$

Whenever we encounterq a state $P \subseteq Q$ of $D(N)$ that has not been considered before, we add P to the table, marking any final states as such. In this case, three new DFA states emerge ($\{q_2\}$, $\{q_1, q_3\}$, and $\emptyset$), none of which is final:

	$\delta_{D(N)}$	0	1	2
$\rightarrow$	$\{q_0\}$	$\{q_2\}$	$\{q_1, q_3\}$	$\emptyset$
	$\{q_2\}$			
	$\{q_1, q_3\}$			
	$\emptyset$			

We then proceed to tabulate $\delta_{D(N)}$ for each of the new states for each $x \in \Sigma$, adding any further new states to the table:

	$\delta_{D(N)}$	0	1	2
$\rightarrow$	$\{q_0\}$	$\{q_2\}$	$\{q_1, q_3\}$	$\emptyset$
	$\{q_2\}$	$\{q_0\}$	$\emptyset$	$\{q_0\}$
	$\{q_1, q_3\}$	$\emptyset \cup \{q_4\} = \{q_4\}$	$\{q_0\} \cup \emptyset = \{q_0\}$	$\{q_0\} \cup \{q_4\} = \{q_0, q_4\}$
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

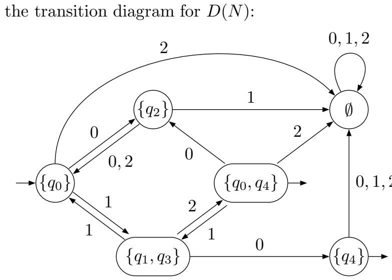
Here, two new states emerge ($\{q_4\}$ and $\{q_0, q_4\}$), both final (because $\{q_4\} \cap F_N \neq \emptyset$ and $\{q_0, q_4\} \cap F_N \neq \emptyset$):

	$\delta_{D(N)}$	0	1	2
$\rightarrow$	$\{q_0\}$	$\{q_2\}$	$\{q_1, q_3\}$	$\emptyset$
	$\{q_2\}$	$\{q_0\}$	$\emptyset$	$\{q_0\}$
	$\{q_1, q_3\}$	$\emptyset \cup \{q_4\} = \{q_4\}$	$\{q_0\} \cup \emptyset = \{q_0\}$	$\{q_0\} \cup \{q_4\} = \{q_0, q_4\}$
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$*$	$\{q_4\}$			
$*$	$\{q_0, q_4\}$			

This process is repeated until no new states emerges. Tabulating for the last two new states reveals that no further states emerge in this case and we are thus done, having only had to tabulate for 6 reachable out of the 32 DFA states:

	$\delta_{D(N)}$	0	1	2
$\rightarrow$	$\{q_0\}$	$\{q_2\}$	$\{q_1, q_3\}$	$\emptyset$
	$\{q_2\}$	$\{q_0\}$	$\emptyset$	$\{q_0\}$
	$\{q_1, q_3\}$	$\emptyset \cup \{q_4\} = \{q_4\}$	$\{q_0\} \cup \emptyset = \{q_0\}$	$\{q_0\} \cup \{q_4\} = \{q_0, q_4\}$
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$*$	$\{q_4\}$	$\emptyset$	$\emptyset$	$\emptyset$
$*$	$\{q_0, q_4\}$	$\{q_2\} \cup \emptyset = \{q_2\}$	$\{q_1, q_3\} \cup \emptyset = \{q_1, q_3\}$	$\emptyset \cup \emptyset = \emptyset$

After double checking that we have not forgotten to mark any final states, we can draw the transition diagram for $D(N)$:



Accepting states have been marked by outgoing arrows.

3.2.4 Correctness of the subset construction

We still have to convince ourselves that the subset construction actually works; i.e., that for a given NFA A it really is the case that $L(A) = L(D(A))$. We start by proving the following lemma, which says that the extended transition functions coincide:

Lemma 3.1

$$\hat{\delta}_{D(A)}(P, w) = \hat{\delta}_A(P, w)$$

The result of both functions is a set of states of the NFA A: for the left-hand side because the extended transition function on NFAs returns a set of states, and for the right-hand side because the states of $D(A)$ are sets of states of A.

Proof. We show this by *induction* over the length of the word w, $|w|$.

$|w| = 0$ Then $w = \epsilon$ and we have

$$\begin{aligned} \hat{\delta}_{D(A)}(P, \epsilon) &= P && \text{by (1)} \\ &= \hat{\delta}_A(P, \epsilon) && \text{by (3)} \end{aligned}$$

$|w| = n + 1$ Then $w = xv$ with $|v| = n$.

$$\begin{aligned} \hat{\delta}_{D(A)}(P, xv) &= \hat{\delta}_{D(A)}(\delta_{D(A)}(P, x), v) && \text{by (2)} \\ &= \hat{\delta}_A(\delta_{D(A)}(P, x), v) && \text{ind.hyp.} \\ &= \hat{\delta}_A(\bigcup\{\delta_A(q, x) \mid q \in P\}, v) && \text{by (5)} \\ &= \hat{\delta}_A(P, xv) && \text{by (4)} \end{aligned}$$

□

We can now use the lemma to show

Theorem 3.2

$$L(A) = L(D(A))$$

Proof.

$$\begin{aligned} & w \in L(A) \\ \iff & \text{Definition of } L(A) \text{ for NFAs} \\ & \hat{\delta}_A(S, w) \cap F \neq \emptyset \\ \iff & \text{Lemma 3.1} \\ & \hat{\delta}_{D(A)}(S, w) \cap F \neq \emptyset \\ \iff & \text{Definition of } F_{D(A)} \\ & \hat{\delta}_{D(A)}(S, w) \in F_{D(A)} \\ \iff & \text{Definition of } L(A) \text{ for DFAs} \\ & w \in L_{D(A)} \end{aligned}$$

□

Corollary 3.3 *NFAs and DFAs recognise the same class of languages.*

Proof. We have noticed that DFAs are just a special case of NFAs. On the other hand the subset construction introduced above shows that for every NFA we can find a DFA that recognises the same language. □

3.3 Exercises

Exercise 3.1

Let the alphabet $\Sigma_A = \{a, b\}$ and consider the following DFA A:

$$\begin{aligned} A &= (Q_A = \{0,1,2,3\},\ \Sigma_A,\ \delta_A,\ q_0 = 0,\ F_A = \{1,2\}) \\ \delta_A &= \{((0,a),1),\ ((0,b),2),\ ((1,a),0),\ ((1,b),3),\ ((2,a),3),\ ((2,b),0), \\ &\qquad ((3,a),2),\ ((3,b),1)\} \end{aligned}$$

(Here tuple notation is used to define the mapping of the transition function δ_A; thus $\delta_A(0, a) = 1$, $\delta_A(0, b) = 2$, etc.) For the DFA A:

1. Draw its transition diagram.
2. Determine which of the following words belong to $L(A)$:
   1. ϵ
   2. b
   3. *abaab*
   4. *bababbba*
3. Explicitly calculate $\hat{\delta}_A(0, abba)$.
4. Describe the language that the automaton recognises in English.

Exercise 3.2

Construct a DFA B over $\Sigma_B = \{a, b, c, d\}$ accepting all words where the number of a's is a multiple of 3. E.g. $abdaca \in L(B)$ (3 a's), but $ddaabaa \notin L(B)$ (4 a's, 4 is not a multiple of 3). Explain your construction. In particular, explain why you chose to have the number of states you did, and explain the purpose (or "meaning") of each state.

Exercise 3.3

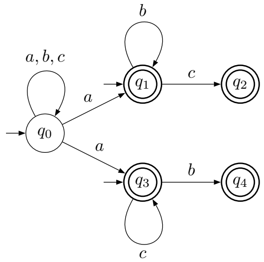
For the alphabet $\Sigma_C = \{a, b, c\}$, construct a DFA C that recognises all words where the number of a's is odd and the number of b's is divisible by 3. (There may thus be any number of c's.) For example, $a \in L(C)$ (odd number of a's, the number of b's is 0 which is divisible by 3), $cbcbabc \in L(C)$ (odd number of a's, the number of b's is 3 which is divisible by 3), but $\epsilon \notin L(C)$ (even number of a's). Give a brief explanation of your construction, that clearly conveys the key ideas, and give the transition diagram for your DFA as the final answer.

Exercise 3.4

For the alphabet $\Sigma_D = \{0, 1, 2, 3\}$, construct a DFA D that precisely recognizes the words for which the arithmetic sum of the constituent symbols is divisible by 5. For example, $\epsilon \in L(D)$ (there are no symbols in the empty string, the sum is thus 0 which is divisible by 5), $0 \in L(D)$ (the sum is again 0), and $23131 \in L(D)$ ($2 + 3 + 1 + 3 + 1 = 10$ which is divisible by 5), but $133 \notin L(D)$ ($1 + 3 + 3 = 7$ which is not divisible by 5). Explain your construction.

Exercise 3.5

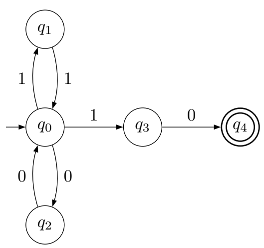
Consider the following NFA A over $\Sigma_A = \{a, b, c\}$:



1. Which of the following words are accepted by A and which are not?
   (a) ϵ
   (b) *aaa*
   (c) *bbc*
   (d) *cbc*
   (e) *abcacb*
2. Construct a DFA $D(A)$ equivalent to A using the "subset construction". Clearly show each step of your calculations in a transition table.
   Hint: Some of the 32 states (i.e., the $2^{|Q_A|} = 2^5 = 32$ possible subsets of Q_A) that would arise by applying the subset construction blindly to A may be unreachable. You can therefore adopt a strategy where you only consider states reachable from the initial state, S_A.
3. Draw the transition diagram for $D(A)$, ignoring unreachable states.

Exercise 3.6

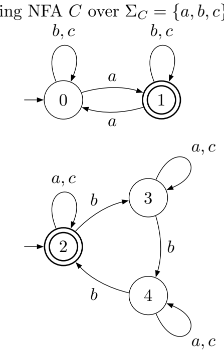
Consider the following NFA B over $\Sigma_B = \{0, 1\}$:



1. Construct a DFA $D(B)$ equivalent to B using the "subset construction" and draw the transition diagram for $D(B)$, ignoring unreachable states. Clearly show each step of your calculations, e.g. in a transition table.
2. Carry out a *sanity check* on your resulting DFA $D(B)$ as follows.
   (a) Give two words over Σ_B that are accepted by the NFA B and two that are not. At least two of those should be four symbols long or longer.
   (b) Check that the DFA $D(B)$ accepts the first two words and rejects the other two, exactly like the NFA B. Justify your answer by listing the sequence of states the DFA $D(B)$ goes through for each word, and stating whether or not the last state of that sequence is accepting.

Exercise 3.7

Consider the following NFA C over $\Sigma_C = \{a, b, c\}$:



1. Which of the following words are accepted by C and which are not?
   (a) ϵ
   (b) *aa*
   (c) *bb*
   (d) *abcabc*
   (e) *abcabca*
2. Describe the language accepted by C in English.
3. Construct a DFA $D(C)$ equivalent to C using the "subset construction". Clearly show each step of your calculations in a transition table. Indicate which DFA state that is initial and which DFA states that are accepting. Only consider states reachable from the initial state of the resulting DFA.
4. Draw the transition diagram for $D(C)$.

4 Regular Expressions

To recapitulate, given an alphabet Σ, a language is a set of words $L \subseteq \Sigma^*$. So far, we have described languages either using set theory (explicit enumeration or set comprehensions) or through finite automata. The key benefits of using automata is that they can describe infinite languages (unlike enumeration) and that they directly give a mechanical way to determine language membership (unlike comprehensions). However, from an automaton, it is not usually immediately obvious what the language of that automaton is, and conversely, given a high-level description of a language, it is often not obvious if it is possible to describe the language using a finite automaton.

This section introduces *regular expressions*: a concise and much more direct way to describe languages. Moreover, a regular expression can mechanically be translated into a finite automaton that accept precisely the language described. This opens up for many practical applications as languages can both be described and recognised with ease. In fact, the opposite is also true: given a finite automaton, it is possible to translate that into an equivalent regular expression. Finite automata and regular expressions are thus *interconvertible*, meaning that they describe the exact *same* class of languages: the *regular languages* or, according to the Chomsky hierarchy, type 3 languages (section 1.1).

One application of regular expressions is to define patterns in programs such as `grep`. Given a regular expression and a sequence of text lines as input, `grep` returns those lines that match the regular expression, where matching means that the line contains a substring that is in the language denoted by the regular expression. The syntax used by `grep` for regular expressions is slightly different from the one used here, and `grep` further supports some convenient abbreviations. However, the underlying theory is exactly the same.

Other applications for regular expressions include defining the *lexical* syntax of programming languages; i.e., what basic symbols, or *tokens*, such as identifiers, keywords, numeric literals look like, as well other lexical aspects such as white space and comments. The context-free syntax (see section 7) of a programming language is then defined in terms of the tokens; i.e., the tokens effectively constitute the alphabet of the language.

In fact, regular expression matching has so many applications that many programming languages provide support for this capability, either built-in or via libraries. Examples include Perl, PHP, Python, and Java. In the past, some of those implementations were a bit naive as the regular expressions were not compiled into finite automata. As a result, matching could be *very* slow, as explained in the paper *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)* [Cox07]. This paper is a very good read, and once you have read these lecture notes up to and including the present section, you will be able to appreciate it fully.

4.1 What are regular expressions?

Given an alphabet Σ (e.g., $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$), the *syntax* (i.e., *form*) of regular expressions over Σ is defined inductively as follows:

1. $\emptyset$ is a regular expression.
2. ϵ is a regular expression.

3. For each $x \in \Sigma$, $\mathbf{x}$ is a regular expression[6].
4. If E and F are regular expressions then $E + F$ is a regular expression.
5. If E and F are regular expressions then EF (juxtapositioning; just one after the other) is a regular expression.
6. If E is a regular expression then E^* is a regular expression.
7. If E is a regular expression then $\mathbf{(}E\mathbf{)}$ is a regular expression[7].

These are all regular expressions.

To illustrate, here are some examples of regular expressions:

- ϵ
- $\mathbf{hallo}$
- $\mathbf{hallo} + \mathbf{hello}$
- $\mathbf{h}(\mathbf{a} + \mathbf{e})\mathbf{llo}$
- $\mathbf{a}^*\mathbf{b}^*$
- $(\epsilon + \mathbf{b})(\mathbf{ab})^*(\epsilon + \mathbf{a})$

As in arithmetic, there are conventions for reading regular expressions:

- $*$ binds stronger than juxtapositioning and $+$. For example, $\mathbf{ab}^*$ is read as $\mathbf{a}(\mathbf{b}^*)$. Parentheses must be used to enforce the reading $(\mathbf{ab})^*$.
- Juxtapositioning binds stronger than $+$. For example, $\mathbf{ab} + \mathbf{cd}$ is read as $(\mathbf{ab}) + (\mathbf{cd})$. Parentheses must be used to enforce the reading $\mathbf{a}(\mathbf{b} + \mathbf{c})\mathbf{d}$.

4.2 The meaning of regular expressions

In the previous section, we defined the *syntax* of regular expressions, their *form*. We now proceed to define the *semantics* of regular expressions; i.e., what they *mean*, what language a regular expression denotes.

To answer this question, first recall the definition of concatenation of concatenation of languages from section 2:

$$L_1 L_2 = \{uv \mid u \in L_1 \land v \in L_2\}$$

We further recall the *the Kleene star* operation from the same section (2):

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

To each regular expression E over Σ we assign a language $L(E) \subseteq \Sigma^*$ as its meaning or semantics. We do this by *induction over the definition of the syntax*:

[6] Note that the regular expression here is typeset in boldface, like $\mathbf{a}$, to distinguish is from the corresponding *symbol*, like `a`, typeset in a type-writer font in this and the next section (and on occasion later on as well). Underlining is sometimes used as an alternative to boldface.

[7] The parentheses have been typeset in boldface to emphasise that they are part of the syntax of the regular expression.

1. $L(\emptyset) = \emptyset$
2. $L(\epsilon) = \{\epsilon\}$
3. $L(\mathbf{x}) = \{x\}$ where $x \in \Sigma$.
4. $L(E + F) = L(E) \cup L(F)$
5. $L(EF) = L(E)L(F)$
6. $L(E^*) = L(E)^*$
7. $L(\mathbf{(}E\mathbf{)}) = L(E)$

Subtle points: In (1), the symbol $\emptyset$ is used both as a regular expression and as the empty set (empty language). Similarly, ϵ in (2) is used in two ways: as a regular expression and as the empty word. In (3), the regular expression is typeset in boldface to distinguish it from the corresponding symbol. In (6), the $*$-operator is used both to construct a regular expression (part of the syntax) and as an operation on languages. In (7), the inner parentheses on the left-hand side, typeset in boldface, are part of the syntax of regular expressions.

Let us now calculate the meaning of each of the regular expression examples from the previous section; i.e., the language denoted in each case:

- ϵ:

  By (2):

$$L(\epsilon) = \{\epsilon\}$$

- **hallo**:

  Consider $L(\mathbf{ha})$. By (3):

$$\begin{aligned} L(\mathbf{h}) &= \{\mathtt{h}\} \\ L(\mathbf{a}) &= \{\mathtt{a}\} \end{aligned}$$

  Hence, by (5) and language concatenation (section 2):

$$\begin{aligned} L(\mathbf{ha}) &= L(\mathbf{h})L(\mathbf{a}) \\ &= \{uv \mid u \in L(\mathbf{h}) \wedge vL(\mathbf{a}) \\ &= \{uv \mid u \in \{\mathtt{h}\} \wedge v \in \{\mathtt{a}\}\} \\ &= \{\mathtt{ha}\} \end{aligned}$$

  Continuing the same reasoning we obtain:

$$L(\mathbf{hallo}) = \{\mathtt{hallo}\}$$

- **hallo + hello**:

  From above we know $L(\mathbf{hallo}) = \{\mathtt{hallo}\}$ and $L(\mathbf{hello}) = \{\mathtt{hello}\}$. By (4) we then get:

$$\begin{aligned} L(\mathbf{hallo} + \mathbf{hello}) &= \{\mathtt{hallo}\} \cup \{\mathtt{hello}\}\} \\ &= \{\mathtt{hallo}, \mathtt{hello}\} \end{aligned}$$

- $\mathbf{h(a+e)llo}$:

  By (3) and (4) we know $L(\mathbf{a} + \mathbf{e}) = \{\mathtt{a}, \mathtt{e}\}$. Thus, using (5) and language concatenation, we obtain:

$$\begin{aligned} L(\mathbf{h(a+e)llo}) &= L(\mathbf{h})L(\mathbf{a}+\mathbf{e})L(\mathbf{llo}) \\ &= \{uvw \mid u \in L(\mathbf{h}) \wedge v \in L(\mathbf{a}+\mathbf{e}) \wedge w \in L(\mathbf{llo})\} \\ &= \{uvw \mid u \in \{\mathtt{h}\} \wedge v \in \{\mathtt{a}, \mathtt{e}\} \wedge w \in \{\mathtt{llo}\}\} \\ &= \{\mathtt{hallo}, \mathtt{hello}\} \end{aligned}$$

- $\mathbf{a}^*\mathbf{b}^*$:

  By (6):

$$\begin{aligned} L(\mathbf{a}^*) &= L(\mathbf{a})^* \\ &= \{\mathtt{a}\}^* \\ &= \bigcup_{n=0}^{\infty} \{\mathtt{a}\}^n \\ &= \bigcup_{n=0}^{\infty} \{w_1 w_1 \ldots w_n \mid 1 \le i \le n,\ w_i \in \{\mathtt{a}\}\} \\ &= \bigcup_{n=0}^{\infty} \{\mathtt{a}^n\} \\ &= \{\mathtt{a}^n \mid n \in \mathbb{N}\} \end{aligned}$$

  Using (5) and language concatenation, this allows us to conclude:

$$\begin{aligned} L(\mathbf{a}^*\mathbf{b}^*) &= L(\mathbf{a}^*)L(\mathbf{b}^*) \\ &= \{uv \mid u \in L(\mathbf{a}^*) \wedge v \in L(\mathbf{b}^*)\} \\ &= \{uv \mid u \in \{\mathtt{a}^m \mid m \in \mathbb{N}\} \wedge v \in \{\mathtt{b}^n \mid n \in \mathbb{N}\}\} \\ &= \{\mathtt{a}^m\mathtt{b}^n \mid m, n \in \mathbb{N}\} \end{aligned}$$

  That is, $L(\mathbf{a}^*\mathbf{b}^*)$ is the set of all words that start with a (possibly empty) sequence of `a`'s, followed by a (possibly empty) sequence of `b`'s.

- $(\epsilon + \mathbf{b})(\mathbf{ab})^*(\epsilon + \mathbf{a})$:

  Let us analyse the parts:

$$\begin{aligned} L(\epsilon + \mathbf{b}) &= \{\epsilon, \mathtt{b}\} \\ L((\mathbf{ab})^*) &= \{(\mathtt{ab})^n \mid n \in \mathbb{N}\} \\ L(\epsilon + \mathbf{a}) &= \{\epsilon, \mathtt{a}\} \end{aligned}$$

  Thus, we have:

$$L((\epsilon + \mathbf{b})(\mathbf{ab})^*(\epsilon + \mathbf{a})) = \{u(\mathtt{ab})^n v \mid u \in \{\epsilon, \mathtt{b}\} \wedge n \in \mathbb{N} \wedge v \in \{\epsilon, \mathtt{a}\}\}$$

  In English: $L((\epsilon + \mathbf{b})(\mathbf{ab})^*(\epsilon + \mathbf{a}))$ is the set of (possibly empty) sequences of alternating `a`'s and `b`'s.

4.3 Algebraic laws

The semantics of regular expressions not only allows us to find out the meaning of specific regular expressions, but also allows us to prove useful laws about regular expression in general. Let us illustrate by proving the following distributive law for regular expressions:

$$E(F+G) = EF + EG$$

Note that E, F, G are *variables* standing for some specific but arbitrary regular expressions, and that $=$ here is *semantic* (as opposed to syntactic) equality. That is, what we need to prove is that a regular expression of the form $E(F+G)$ and one of the form $EF+EG$ always have the same meaning, i.e., denote the same language.

We thus start from $L(E(F+G))$ and show step by step that this is equal to $L(EF+EG)$ without making any assumptions about the constituent regular expressions E, F, and G, other than that their semantics is given by $L(E)$ etc.

$$\begin{array}{ll}
 & L(E(F+G)) \\
= & \quad\quad \{\text{ Semantics of r.e. (concat.) }\} \\
 & L(E)L(E+F) \\
= & \quad\quad \{\text{ Semantics of r.e. } (+) \ \} \\
 & L(E)(L(F) \cup L(G)) \\
= & \quad\quad \{\text{ Def. concat. of languages }\} \\
 & \{w_1w_2 \mid w_1 \in L(E) \wedge w_2 \in (L(F) \cup L(G))\} \\
= & \quad\quad \{\text{ Def. set union }\} \\
 & \{w_1w_2 \mid w_1 \in L(E) \wedge w_2 \in \{w \mid w \in L(F) \vee w \in L(G)\}\} \\
= & \quad\quad \{\text{ Duality between sets and predicates }\} \\
 & \{w_1w_2 \mid w_1 \in L(E) \wedge (w_2 \in L(F) \vee w_2 \in L(G))\} \\
= & \quad\quad \{\text{ Conjunction } (\wedge) \text{ distributes over disjunction } (\vee) \ \} \\
 & \{w_1w_2 \mid (w_1 \in L(E) \wedge w_2 \in L(F)) \vee (w_1 \in L(E) \wedge w_2 \in L(G))\} \\
= & \quad\quad \{\text{ Def. set union }\} \\
 & \{w_1w_2 \mid (w_1 \in L(E) \wedge w_2 \in L(F))\} \cup \{w_1w_2 \mid (w_1 \in L(E) \wedge w_2 \in L(G))\} \\
= & \quad\quad \{\text{ Def. concat. languages (twice) }\} \\
 & L(E)L(F) \cup L(E)L(G) \\
= & \quad\quad \{\text{ Semantics of r.e. (conacat., twice) }\} \\
 & L(EF) \cup L(EG) \\
= & \quad\quad \{\text{ Semantics of r.e. } (+) \ \} \\
 & L(EF+EG)
\end{array}$$

Other laws for regular expressions can be proved similarly, although induction is sometimes needed. As an exercise, prove (some of) the following:

$$\begin{array}{rcl}
\epsilon E & = & E \\
E\epsilon & = & E \\
\emptyset E & = & \emptyset \\
E\emptyset & = & \emptyset \\
E + (F+G) & = & (E+F)+G \\
E(FG) & = & (EF)G \\
(E^*)^* & = & E^* \\
\epsilon + EE^* & = & E^*
\end{array}$$

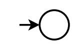
4.4 Translating regular expressions into NFAs

Theorem 4.1 *A regular expression E can be translated into an equivalent NFA $N(E)$ such that $L(N(E)) = L(E)$.*

We refer to this translation as the "Graphical Construction". It is a variation of the standard way of translating regular expressions into NFAs known as Thompson's construction[8].

Proof. The proof is by induction on the syntax of regular expressions:

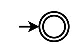
1. $N(\emptyset)$:



which will reject everything (there are no final states). Thus:

$$\begin{array}{rcl}
L(N(\emptyset)) & = & \emptyset \\
 & = & L(\emptyset)
\end{array}$$

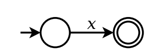
2. $N(\epsilon)$:



This automaton accepts the empty word but rejects everything else. Thus

$$\begin{array}{rcl}
L(N(\epsilon)) & = & \{\epsilon\} \\
 & = & L(\epsilon)
\end{array}$$

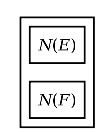
3. $N(\mathbf{x})$:



This automaton only accepts the word x. Thus:

$$\begin{array}{rcl}
L(N(\mathbf{x})) & = & \{\mathrm{x}\} \\
 & = & L(\mathbf{x})
\end{array}$$

4. $N(E+F)$: We merge the diagrams for $N(E)$ and $N(F)$ into one:



[8] https://en.wikipedia.org/wiki/Thompson%27s_construction

Intuitively, the NFAs for the subexpressions E and F are placed side by side. Thus if either of the NFA accepts a word, the combined NFA accepts this word. However, we have to ensure that the states of the constituent NFAs do not get confused with each other. We therefore have to use the *disjoint union*, defined as follows:

$$A \uplus B = \{(0,a) \mid a \in A\} \cup \{(1,b) \mid b \in B\}$$

That is, each element of the disjoint union is "tagged" with an index that shows from which of the two sets it originated. Thus the elements of the constituent sets will remain distinct. The transition function of the combined NFA also has to be defined to work on tagged states.

Thus, given the NFAs for the subexpressions E and F:

$$\begin{aligned} N(E) &= (Q_E, \Sigma, \delta_E, S_E, F_E) \\ N(F) &= (Q_F, \Sigma, \delta_F, S_F, F_F) \end{aligned}$$

we construct the combined NFA for the regular expression $E + F$:

$$N(E+F) = (Q_{E+F}, \Sigma, \delta_{E+F}, S_{E+F}, F_{E+F})$$

where

$$\begin{aligned} Q_{E+F} &= Q_E \uplus Q_F \\ \delta_{E+F}((0,q),x) &= \{(0,q') \mid q' \in \delta_E(q,x)\} \\ \delta_{E+F}((1,q),x) &= \{(1,q') \mid q' \in \delta_F(q,x)\} \\ S_{E+F} &= S_E \uplus S_F \\ F_{E+F} &= F_E \uplus F_F \end{aligned}$$

It remains to prove $L(N(E+F)) = L(E+F)$. We first observe that

$$L(N(E+F)) = L(N(E)) \cup L(N(F))$$

because the initial states S_{E+F} of the combined NFA is the (disjoint) union of the initial states of the constituent NFAs, and because the combined NFA accepts a word whenever one of the constituent NFAs does.

The proof then proceeds by induction; that is, we assume that the translation is correct for the subexpressions:

$$\begin{aligned} L(N(E)) &= L(E) \\ L(N(F)) &= L(F) \end{aligned}$$

Thus:

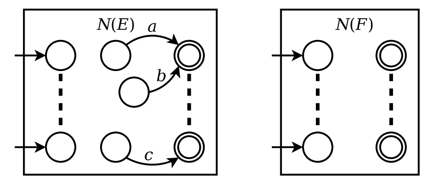
$$\begin{aligned} L(N(E+F)) &= L(N(E)) \cup L(N(F)) && \text{Above} \\ &= L(E) \cup L(F) && \text{Induction hypothesis} \\ &= L(E+F) && \text{By (4)} \end{aligned}$$

This is what is meant by *induction over the syntax of regular expressions*.

5. $N(EF)$: Recall that $L(EF) = L(E)L(F)$. Thus, a word $w \in L(EF)$ iff w can be divided into two words u and v, $w = uv$, such that $u \in L(E)$ and $v \in L(F)$. Consequently, if we have an NFA $N(E)$ recognising $L(E)$ and another NFA $N(F)$ recognising $L(F)$, we can construct an NFA recognising words $w = uv \in L(EF)$ if we join the NFAs $N(E)$ and $N(F)$ in sequence in such a way that the machine moves from $N(E)$ to an initial state of $N(F)$ on the last symbol of a word $u \in L(E)$.

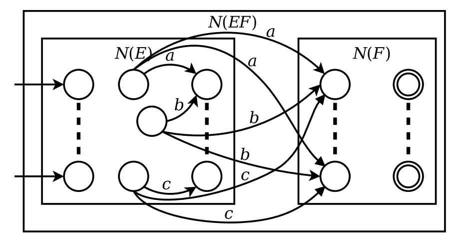
   Of course, it could be that $\epsilon \in L(E)$, meaning that one or more of the initial states of $N(E)$ are accepting. In this case, for a word $w = uv$ with $u = \epsilon$, the machine needs to start in an initial state of $N(F)$ directly as there is no last symbol of $u = \epsilon$ to move on.

   Therefore, we consider two cases: $\epsilon \notin L(E)$, meaning no initial state of $N(E)$ is accepting, and $\epsilon \in L(E)$, meaning at least one initial state of $N(E)$ is accepting. We start with the former:



   The dashed lines here suggest "one or more". So there could be one or more initial states and one or more final states in both $N(E)$ and $N(F)$. This will be made precise shortly; the figure just conveys the general idea.

   We identify every state of $N(E)$ that *immediately precedes* an accepting state; i.e., every state from which an accepting state can be reached on a single input symbol. We then join $N(E)$ and $N(F)$ into a combined NFA $N(EF)$ by adding an edge from each of the identified states to *all* of the initial states of $N(F)$ for each symbol on which an accepting state of $N(E)$ can be reached. The initial states of $N(EF)$ are the initial states of $N(E)$ and the final states of $N(EF)$ are the final states of $N(F)$:



   This ensures that the NFA $N(EF)$, once it has read a word u accepted by $N(E)$, is ready to try to accept the remainder v of a word $w = uv$

by effectively passing v to $N(F)$, allowing the latter to try to accept the remaining part v of w from any of its initial states.

We now formalise this construction. The set of states of the combined NFA $N(EF)$ is again given by the *disjoint* union of the states of $N(E)$ and $N(F)$ to avoid confusion, and the transition function δ_{EF} as well as the initial states S_{EF} and final states F_{EF} are defined accordingly.

Thus, given the NFAs for the subexpressions E and F:

$$\begin{array}{rcl} N(E) & = & (Q_E, \Sigma, \delta_E, S_E, F_E) \\ N(F) & = & (Q_F, \Sigma, \delta_F, S_F, F_F) \end{array}$$

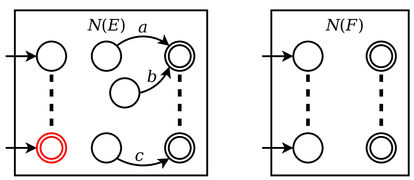
we construct the combined NFA for the regular expression EF:

$$N(EF) \quad = \quad (Q_{EF}, \Sigma, \delta_{EF}, S_{EF}, F_{EF})$$

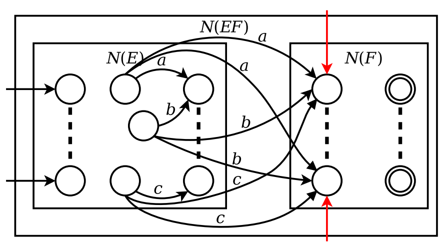
where

$$\begin{array}{rcl} Q_{EF} & = & Q_E \uplus Q_F \\ \delta_{EF}((0,q),x) & = & \{(0,q') \mid q' \in \delta_E(q,x)\} \\ & & \cup \{(1,q') \mid \delta_E(q,x) \cap F_E \neq \emptyset \ \wedge \ q' \in S_F\} \\ \delta_{EF}((1,q),x) & = & \{(1,q') \mid q' \in \delta_F(q,x)\} \\ S_{EF} & = & \{(0,q) \mid q \in S_E\} \\ F_{EF} & = & \{(1,q) \mid q \in F_F\} \end{array}$$

Now let us consider the second case: at least one of the initial states of $N(E)$ is accepting:



As was discussed above, this simply means that we have to arrange that the initial states of $N(F)$ also be initial states of the combined NFA $N(EF)$:



We thus refine the formal definition of the initial states of $N(EF)$ to account for this, yielding a definition that covers both cases:

$$\begin{array}{rcl} S_{EF} & = & \{(0,q) \mid q \in S_E\} \\ & & \cup \{(1,q) \mid S_E \cap F_E \neq \emptyset \ \wedge \ q \in S_F\} \end{array}$$

It remains to prove $L(N(EF)) = L(EF)$. From the construction above, it is clear that

$$L(N(EF)) = \{uv \mid u \in L(N(E)) \ \wedge \ v \in L(N(F))\}$$

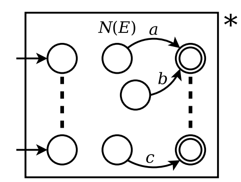
The proof again proceeds by induction; that is, we assume that the translation is correct for the subexpressions:

$$\begin{array}{rcl} L(N(E)) & = & L(E) \\ L(N(F)) & = & L(F) \end{array}$$

Thus:

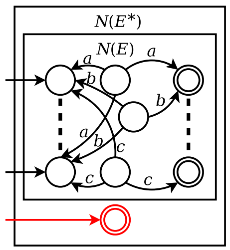
$$\begin{array}{rcll} L(N(EF)) & = & \{uv \mid u \in L(N(E)) \ \wedge \ v \in L(N(F))\} & \text{Above} \\ & = & \{uv \mid u \in L(E) \ \wedge \ v \in L(F)\} & \text{Ind. hyp.} \\ & = & L(E)L(F) & \text{Lang. concat.} \\ & = & L(EF) & \text{By (5)} \end{array}$$

6. $N(E^*)$: Recall that $L(E^*) = L(E)^*$. Thus, a word $w \in L(E^*)$ iff w can be divided into a sequence of $n \in \mathbb{N}$ words u_i, $w = u_1u_2 \ldots u_n$, such that $\forall i \in [1,n] \ . \ u_i \in L(E)$. Consequently, if we have an NFA $N(E)$ recognising $L(E)$, we can construct an NFA recognising words $w \in L(E^*)$ by connecting 0 or more NFAs $N(E)$ in sequence in a similar way to what we did for the case $N(EF)$ above:



Here we use the * to informally suggest sequential composition of 0 or more NFAs.

However, we need to construct a *single* NFA, and there is no upper bound on the number of NFAs $N(E)$ that we need to connect in sequence. We resolve this by taking a single NFA $N(E)$ and construct and NFA for $N(E^*)$ by making it loop back to all of its own initial states from each state that immediately precedes an accepting state. As we also need to allow for iteration 0 times, we further have to add one extra state that is both initial and final thus accepting ϵ:



We now formalise this construction. This time, the states of $N(E^*)$ are almost the same as those of $N(E)$. But we need one extra state to ensure that $N(E^*)$ can accept the empty word, ϵ, and we have to make sure that this one extra state cannot be confused with any other state in $N(E)$. We label the new state ϵ, suggestive of its role to ensure acceptance of the empty word, and we then form the states of $N(E^*)$ using disjoint union to ensure states cannot be accidentally confused. Like before, we have to take this into account when defining the transition function δ_{E^*} as well as the initial states S_{E^*} and final states F_{E^*} of the NFA $N(E^*)$.

Thus, given the NFA resulting from translating the subexpression E:

$$N(E) \quad = \quad (Q_E, \Sigma, \delta_E, S_E, F_E)$$

we construct the NFA for the regular expression E^*:

$$N(E^*) \quad = \quad (Q_{E^*}, \Sigma, \delta_{E^*}, S_{E^*}, F_{E^*})$$

where

$$\begin{array}{rcl}
Q_{E^*} & = & Q_E \uplus \{\epsilon\} \\
\delta_{E^*}((0,q),x) & = & \{(0,q') \mid q' \in \delta_E(q,x)\} \\
 & & \cup \{(0,q') \mid \delta_E(q,x) \cap F_E \neq \emptyset \wedge q' \in S_E\} \\
\delta_{E^*}((1,\epsilon),x) & = & \emptyset \\
S_{E^*} & = & S_E \uplus \{\epsilon\} \\
F_{E^*} & = & F_E \uplus \{\epsilon\}
\end{array}$$

It remains to prove $L(N(E^*)) = L(E^*)$. Given the construction above, we claim that

$$L(N(E^*)) = \{u_1 u_2 \ldots u_n \mid n \in \mathbb{N} \wedge \forall i \in [1,n] \,.\, u_i \in L(N(E))\}$$

The intuition is that we can run through the automaton one or more times and that the new state ϵ allows the NFA to accept the empty word.

The proof then again proceeds by induction; that is, we assume that the translation is correct for the subexpression:

$$L(N(E)) \quad = \quad L(E)$$

Thus:

$$\begin{array}{rcll}
L(N(E^*)) & = & \{\, u_1 u_2 \ldots u_n & \text{Above} \\
 & & \mid\, n \in \mathbb{N} \wedge \forall i \in [1,n] \,.\, u_i \in L(N(E)) \,\} & \\
 & = & \{\, u_1 u_2 \ldots u_n & \text{Ind. hyp.} \\
 & & \mid\, n \in \mathbb{N} \wedge \forall i \in [1,n] \,.\, u_i \in L(E) \,\} & \\
 & = & \bigcup_{n=0}^{\infty} L(E)^n & \text{Lang. concat.} \\
 & = & L(E)^* & \text{Def. Kleene star} \\
 & = & L(E^*) & \text{By (6)}
\end{array}$$

7. $N((E)) = N(E)$: Parentheses are just used for grouping and does not change anyting.

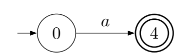
   We need to prove $L(N((E))) = L((E))$. The proof is again by induction, so we assume $L(N(E)) = L(E)$ and then we proceed as follows:

$$\begin{array}{rcll}
L(N((E))) & = & L(N(E)) & \text{By construction} \\
 & = & L(E) & \text{Induction hypothesis} \\
 & = & L((E)) & \text{By (7)}
\end{array}$$

□

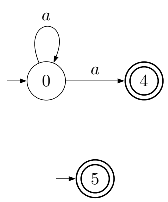
It is worth pausing briefly to reflect on what we just have accomplished. In effect, we have implemented a *compiler* that translates regular expressions into NFAs, and we have proved it correct; that is, the translation preserves the meaning (here, the described language), which after all is what we generally expect of an accurate translation. Of course, it is a very simple compiler. Yet, in essence, it reflects how real tools that handle regular expressions work; for example, scanner (or lexer) generators such as Flex, Ragel, or Alex[9]. Moreover, while proving the correctness of compilers for typical programming languages is vastly more complicated than what we have seen here, there are methodological similarities, such as proof by induction over the structure of the language.

Let us illustrate how to apply the Graphical Construction. As a first example, we construct $N(\mathbf{a}^*\mathbf{b}^*)$. We start with the innermost subexpressions and then join the NFAs together step by step. The states are named according to how they will be named in the final NFA to make it easier to follow the derivation. It is fine to leave states unnamed until the end, and that is what normally is done. We begin with the NFA for $\mathbf{a}$:

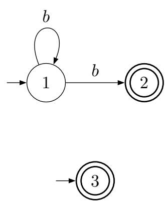


The NFA for $\mathbf{a}^*$ is obtained by adding a loop on a from state 0 to itself as this state precedes a final state and is the only initial state, and by adding the extra state for accepting ϵ:

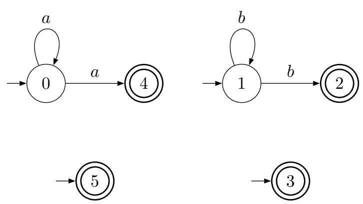
[9] `https://en.wikipedia.org/wiki/Lexical_analysis`



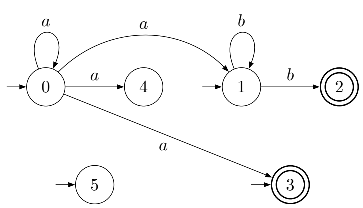
The NFA for $\mathbf{b}^*$ is constructed in the same way:



Now we have to join these two NFAs in sequence:

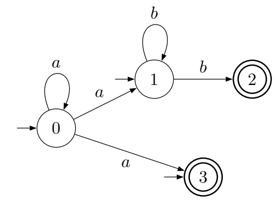


We have to pay extra attention because the automaton for the subexpression $\mathbf{a}^*$ contains a state that is both initial and final, namely state 5, resulting in "extra" initial states when composing that automaton with the automaton for the subexpression $\mathbf{b}^*$:

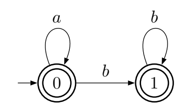


The states 4 and 5 have manifestly become "dead ends": there is no way to reach a final state from either. For NFAs, such dead ends can simply be removed

(along with associated edges) without changing the accepted language. If we do that, we obtain:

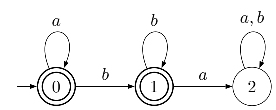


You may have noted that, though correct, this NFA is unnecessary complicated. For example, the following NFA also accepts $N(\mathbf{a}^*\mathbf{b}^*)$, but has fewer states:



This is typical: the translation of regular expressions into NFAs does generally not yield the simplest possible automata.

If we are interested in obtaining the smallest possible machine, one approach is to first convert the resulting NFA into a DFA using the subset construction of section 3.2.3 and then *minimize* this DFA as explained in section 5. If we do this for the four-state NFA above, we obtain the following DFA:

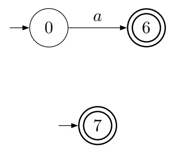


As it happens, just applying the subset construction to the four-state NFA yields this DFA directly[10]: it is already minimal. Try it! It is quick and a good exercise.

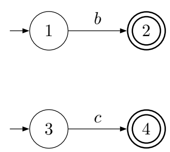
Let us do one more, somewhat larger, example: constructing an NFA for $((\mathbf{a}+\epsilon)(\mathbf{b}+\mathbf{c}))^*$. We again start with the innermost subexpressions and then join the NFAs together step by step. We have to (again) pay extra attention because the automaton for the subexpression $(\mathbf{a}+\epsilon)$ contains a state that is both initial and final, resulting in "extra" initial states when composing that automaton with the automaton for the subexpression $(\mathbf{b}+\mathbf{c})$. Also, it makes sense to eliminate dead ends as soon as they occur, here *before* closing the loop due to the top-level Kleene star. The states are named according to how they will be named in the final NFA to make it easier to follow the derivation, but could be left unnamed until the end if you prefer.

First, an NFA for $\mathbf{a}+\epsilon$:

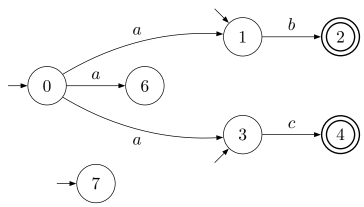
[10] Or one isomorphic to it: the states will probably be named differently.



NFA for $\mathbf{b} + \mathbf{c}$

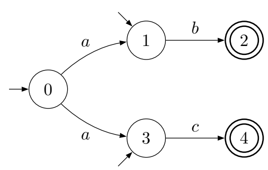


Join the above two NFAs to obtain an NFA for $(\mathbf{a} + \epsilon)(\mathbf{b} + \mathbf{c})$:

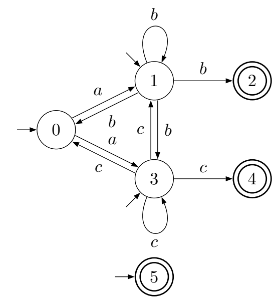


Note that both state 1 and 3 *remain* initial states states because the left automaton has an initial state that is also accepting, meaning we need to be able to get to the start states of the right automaton without consuming any input.

States 6 and 7 have now manifestly become dead ends because there is no way to reach an accepting state from either. Let us remove them and all associated edges:



The last step is to carry out the construction corresponding to the $*$-operator. States 1 and 3 both immediately precede a final state, and we should thus add corresponding transition edges from those back to all initial states. There are three initial states, 0, 1, and 3. Thus we need an edge labelled b from 1 to each of 0, 1, and 3 (i.e., a loop back to itself on b) and and an edge labelled c from 3 to each of 0, 1, and 3 (i.e., a loop back to itself on c). Additionally, we must not forget to add an extra initial state which is also final (here state 5) to ensure the NFA accepts ϵ.



Note that the isolated state 5 thus also is part of the final automaton.

4.5 Summing up

From the previous section we know that a language given by regular expression is also recognized by a NFA. What about the other way: Can a language recognized by a finite automaton (DFA or NFA) also be described by a regular expression? The answer is yes:

Theorem 4.2 *Given a DFA A there is a regular expression $R(A)$ that recognizes the same language $L(A) = L(R(A))$.*

We omit the proof (which can be found in the [HMU01] on pp.91-93). However, we conclude:

Corollary 4.3 *Given a language $L \subseteq \Sigma^*$ the following is equivalent:*

1. *L is given by a regular expression.*
2. *L is the language accepted by an NFA.*
3. *L is the language accepted by a DFA.*

Proof. We have that 1. $\Longrightarrow$ 2 by theorem 4.1. We know that 2. $\Longrightarrow$ 3. by3.2 and 3. $\Longrightarrow$ 1. by 4.2. $\square$

As indicated in the introduction, the languages that are characterised by any of the three equivalent conditions are called *regular languages* or *type 3 languages*.

4.6 Exercises

Exercise 4.1

Give regular expressions defining the following languages over the alphabet $\Sigma = \{a, b, c\}$:

1. All words that contain exactly one a.
2. All words that contain at least two bs.
3. All words that contain at most two cs.
4. All words such that all b's appear before all c's.
5. All words that contain exactly one b and one c (but any number of a's).
6. All words such that the number of a's plus the number of b's is odd.
7. All words that contain the sequence $abba$ at least once.

Exercise 4.2

Using the formal definition of the meaning of regular expressions, compute the set denoted by the regular expression

$$(\mathbf{aa} + \epsilon\mathbf{b}^*\emptyset)(\mathbf{b} + \mathbf{c})$$

simplifying as far as possible. Provide a step-by-step account.

Exercise 4.3

Construct an NFA for the regular expression $(\mathbf{a}(\mathbf{b}+\mathbf{c}))^*$ using the "graphical construction" from the lecture notes. Provide a step-by-step account.

For NFAs it is possible to omit "dead ends", i.e., states from which no final state possibly can be reached, without changing the language of the automaton. Do this as soon as dead ends emerges to reduce your work.

Exercise 4.4

Systematically construct an NFA for the regular expression

$$(\mathbf{a}(\emptyset^* + \mathbf{b}))^*(\mathbf{c} + \epsilon + \emptyset)$$

by following the graphical construction from the lecture notes. Make sure it is clear how you undertake the construction by showing the major steps. Eliminate "dead ends" (states from which no final state can be reached) when they appear. The states in the final NFA should be named, but as long as it is clear what you are doing, you can leave the states of intermediate NFAs unnamed.

5 Minimization of Finite Automata

[FOR REFERENCE: MINIMIZATION NOT TAUGHT SPRING 2019]

As we saw when translating regular expressions into NFAs, the resulting automaton is not necessarily the smallest possible one. Similarly, when employing the subset construction to translate an NFA into a DFA, the result is not always the smallest possible DFA. It is often desirable to make automatons as small as possible. For example, if we wish to implement an automaton, the implementation will be more efficient the smaller the automaton is.

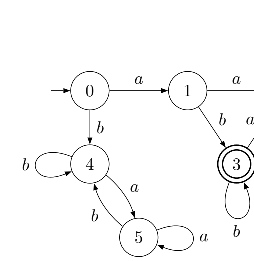
Given an automaton, the question, then, is how to construct an *equivalent* but smaller automaton. Recall that two automatons are equivalent of they accept the same language. In the following we will study a method for minimizing DFAs: the table-filling algorithm.

Another interesting question is if there, in general, is one unique automaton that is the smallest equivalent one, or if there can be many distinct equivalent automatons, none of which can be made any smaller. It turns out the answer is that the minimal equivalent DFA is unique up to naming of the states. This, in turn, means that we have obtained a mechanical decision procedure for determining whether two regular languages are equal: simply convert their respective representation (be it a DFA, an NFA, or a regular expression) to DFAs and minimize them. Because the minimal DFAs are unique, the languages are equal if and only if the minimal DFAs are equal.

5.1 The table-filling algorithm

For a DFA $(Q, \Sigma, \delta, q_0, F)$, $p, q \in Q$ are *equivalent* states if and only if, for all $w \in \Sigma^*$, $\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$. If two states are *not* equivalent, then they are *distinguishable*.

Consider the following DFA, where $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3, 4, 5\}$, $F = \{2, 3\}$:



The states 1 and 2 are distinguishable on ϵ because $\hat{\delta}(1, \epsilon) = 1 \notin F$ while $\hat{\delta}(2, \epsilon) = 2 \in F$. Similarly, 0 and 1 are distinguishable on e.g. b because $\hat{\delta}(0, b) = 4 \notin F$ while $\hat{\delta}(1, b) = 3 \in F$. On the other hand, in this case, we can easily see that 4 and 5 are not distinguishable on any word because it is not possible to reach any accepting (final) state from either 4 or 5.

The *Table-Filling Algorithm* recursively constructs the set of distinguishable pairs of states for a DFA. When all distinguishable state pairs have been identified, any remaining pairs of states must be equivalent. Such states can be merged, thereby minimizing the automaton. Assume a DFA $(Q, \Sigma, \delta, q_0, F)$:

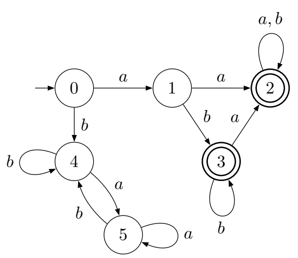
BASIS For $p, q \in Q$, if $(p \in F \wedge q \notin F) \vee (p \notin F \wedge qinF)$, then (p, q) is a distinguishable pair of states. (The states p and q are distinguishable on ϵ.)

INDUCTION For $p, q, r, s \in Q$, $a \in \Sigma$, if $(r, s) = (\delta(p, a), \delta(q, a))$ is a distinguishable pair of states, then (p, q) is also distinguishable. (If the states r and s are distinguishable on a word w, then p and q are distinguishable on aw.)

Theorem 5.1 *If two states are* not *distinguishable by the table-filling algorithm, then they are* equivalent.

5.2 Example of DFA minimization using the table-filling algorithm

This section illustrates how the table-filling algorithm can be used to minimize a DFA when working by hand through a fully worked example. We will minimize the following DFA, where $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3, 4, 5\}$, $F = \{2, 3\}$:



First construct a table over all pairs of distinct states. That is, we do not consider pairs $(p \in Q, p \in Q)$ because a state obviously cannot be distinguishable from itself. An easy way of doing constructing the table is to order the states (e.g. numerically or alphabetically), and then list all states except the last one in *ascending* order along the top, and all states except the first one in *descending* order down the left-hand side of the table. The resulting table for our 6-state DFA looks like this:

	0	1	2	3	4
5					
4					
3					
2					
1					

Then mark the state pairs that are distinguishable according to the basis of the table-filling algorithm; i.e., the pairs where one state is accepting, and one is not. The accepting states of our DFA are 2 and 3. Thus the state pairs (0, 2), (0, 3), (1, 2), (1, 3), (2, 4), (2, 5), (3, 4), and (3, 5) have to be marked. List all remaining state pairs to the right: these are the potentially equivalent states that we now have to investigate further:

	0	1	2	3	4
5			x	x	
4			x	x	
3	x	x			
2	x	x			
1					

$\underline{(0,1)}$ $\underline{(0,4)}$ $\underline{(0,5)}$ $\underline{(1,4)}$

$\underline{(1,5)}$ $\underline{(2,3)}$ $\underline{(4,5)}$

Recall that the induction step of the table-filling algorithm says that for $p, q, r, s \in Q$ and $a \in \Sigma$, if $(r, s) = (\delta(p, a), \delta(q, a)$ is distinguishable (on some word w), then (p, q) is (on the word aw). If, during systematic investigation of all state combinations, we, from a state pair (p, q) on some input symbol a, reach a state pair (r, s) for which it is not yet known whether it is distinguishable or not, we record (p, q) under the heading for (r, s). If it later becomes clear that (r, s) is distinguishable, that means that (p, q) also is distinguishable, and recording this implication allows us to carry out the deferred marking at that point.

Investigate all potentially equivalent state pairs on all input symbols (unless we find that a pair is distinguishable, which means we can stop):

$(0,1)$:	$(\delta(0,a), \delta(1,a)) = (1,2)$	Distinguishable! Mark in table.
$(0,4)$:	$(\delta(0,a), \delta(4,a)) = (1,5)$	Unknown as yet. Add $(0,1)$ under $(1,5)$.
	$(\delta(0,b), \delta(4,b)) = (4,4)$	Same state, no info.

Our table now looks as follows (we strike a line across the pairs we have considered):

	0	1	2	3	4
5			x	x	
4			x	x	
3	x	x			
2	x	x			
1	x				

$\cancel{\underline{(0,1)}}$ $\cancel{\underline{(0,4)}}$ $\underline{(0,5)}$ $\underline{(1,4)}$

$\underline{(1,5)}$ $\underline{(2,3)}$ $\underline{(4,5)}$
$(0,4)$

We continue:

$(0,5)$:	$(\delta(0,a), \delta(5,a)) = (1,5)$	Unknown as yet. Add $(0,5)$ under $(1,5)$.
	$(\delta(0,b), \delta(5,b)) = (4,4)$	Same state, no info.
$(1,4)$:	$(\delta(1,a), \delta(4,a)) = (2,5)$	Distinguishable! Mark in table.

Table:

	0	1	2	3	4
5			x	x	
4		x	x	x	
3	x	x			
2	x	x			
1	x				

$\cancel{\underline{(0,1)}}$ $\cancel{\underline{(0,4)}}$ $\cancel{\underline{(0,5)}}$ $\cancel{\underline{(1,4)}}$

$\underline{(1,5)}$ $\underline{(2,3)}$ $\underline{(4,5)}$
$(0,4)$
$(0,5)$

Now we have come to the state pair $(1,5)$. If we can determine that $(1,5)$ is a distinguishable pair, then we also know that the pairs $(0,4)$ and $(0,5)$ are distinguishable:

$$(1,5): \quad (\delta(1,a), \delta(5,a)) = (2,5) \quad \text{Distinguishable!}$$

Thus we should mark $(1,5)$ along with and $(0,4)$ and $(0,5)$:

	0	1	2	3	4
5	x	x	x	x	
4	x	x	x	x	
3	x	x			
2	x	x			
1	x				

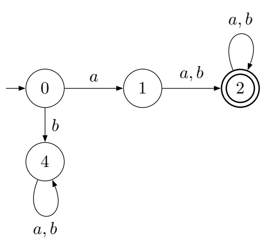
$\cancel{(0,1)}$ $\cancel{(0,4)}$ $\cancel{(0,5)}$ $\cancel{(1,4)}$

$\cancel{(1,5)}$ $(2,3)$ $(4,5)$
$\cancel{(0,4)}$
$\cancel{(0,5)}$

It remains to check the pairs $(2,3)$ and $(4,5)$:

$$\begin{array}{lll}
(2,3): & (\delta(2,a), \delta(3,a)) = (2,2) & \text{Same state, no info.} \\
 & (\delta(2,b), \delta(3,b)) = (2,3) & \text{No point in adding } (2,3) \text{ below } (2,3). \\
(4,5): & (\delta(4,a), \delta(5,a)) = (5,5) & \text{Same state, no info.} \\
 & (\delta(4,b), \delta(5,b)) = (4,4) & \text{Same state, no info.}
\end{array}$$

We have now systematically checked all potentially equivalent state pairs. Two pairs remain unmarked; i.e., we have not been able to show that they are distinguishable: $(2,3)$ and $(4,5)$. We can therefore conclude that these states are pairwise equivalent: $2 \equiv 3$ and $4 \equiv 5$. We thus proceed to merge these states by, informally, placing them "on top" of each other and "dragging along" the edges. The result is the following minimal DFA (where the merged states have been given the names 2 and 4):



6 Disproving Regularity

Regular languages are languages that can be recognized by a computer with finite memory. Such a computer corresponds to a DFA. However, there are many languages that cannot be recognized using only finite memory. A simple example is the language

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

That is,the language of words that start with a number of 0s followed by the *same* number of 1s. Note that this is different from $L(\mathbf{0}^*\mathbf{1}^*)$, which is the language of words of a sequences of 0s followed by a sequence of 1s but not necessarily of the same length. (We know this to be regular because it is given by a regular expression.)

Why can L not be recognized by a computer with finite memory? Suppose we have 32 MiB of memory; that is, we have $32 * 1024 * 1024 * 8 = 268435456$ bits. Such a computer corresponds to an enormous DFA with $2^{268435456}$ states (imagine drawing the transition diagram!). However, this computer can only count to $2^{268435456} - 1$; if it reads a word starting with more than $2^{268435456} - 1$ 0s, it will necessarily lose count! The same reasoning applies whatever finite amount of memory we equip our computer with. Thus, an unbounded amount of memory is needed recognize L. (Of course, $2^{268435456} - 1$ is a *very* large number indeed, so for practical purposes the machine will almost certainly be able to count much further than we ever will need.)

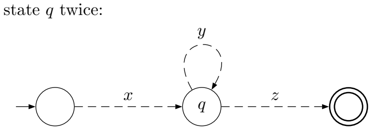
We will now show a general theorem called *the pumping lemma* (for regular languages) that allows us to prove that a certain language is not regular.

6.1 The pumping lemma

Theorem 6.1 *Given a regular language L, then there is a number $n \in \mathbb{N}$ such that all words $w \in L$ that are longer than n ($|w| \geq n$) can be split into three words $w = xyz$ s.t.*

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. $\forall k \in \mathbb{N} \,.\, xy^k z \in L$

Proof. For a regular language L there exists a DFA A s.t. $L = L(A)$. Let us assume that A has n states. If A accepts a word w with $|w| \geq n$, it must have visited some state q twice:



We choose q such that it is the first cycle; hence $|xy| \leq n$. We also know that y is nonempty (otherwise there is no cycle). Now, consider what happens if a word of the form $xy^k z$ is given to the automaton. The automaton will clearly accept this word; it just has to go round the cycle k times, whatever k is, including $k = 0$. Thus $\forall k \in \mathbb{N} \,.\, xy^k z \in L$.

□

6.2 Applying the pumping lemma

Theorem 6.2 *The language $L = \{0^n1^n \mid n \in \mathbb{N}\}$ is not regular.*

Proof. Assume L would be regular. We will show that this leads to contradiction using the pumping lemma.

By the pumping lemma, there is an n such that we can split each word that is longer than n such that the properties given by the pumping lemma hold. Consider $0^n1^n \in L$. This is certainly longer than n. We have that $xyz = 0^n1^n$ and we know that $|xy| \leq n$, hence y can only contain 0s. Further, because $y \neq \epsilon$, it must contain at least one 0. Now, according to the pumping lemma, $xy^0z \in L$. However, this cannot be the case because it contains at least one fewer 0s than 1s. Our assumption that L is regular must thus have been wrong. □

It is easy to see that the language

$$\{1^n \mid n \text{ is even}\}$$

is regular (just construct the appropriate DFA or use a regular expression). However what about

$$\{1^n \mid n \text{ is a square}\}$$

where by saying n is a square we mean that is there is an $k \in \mathbb{N}$ s.t. $n = k^2$. We may try as we like: there is no way to find out whether we have a got a square number of 1s by only using finite memory. And indeed:

Theorem 6.3 *The language $L = \{1^n \mid n$ is a square$\}$ is not regular.*

Proof. We apply the same strategy as above. Assume L is regular. Then there is a number n such we can split all longer words according to the pumping lemma. Let us take $w = 1^{n^2}$; this is certainly long enough. By the pumping lemma, we know that we can split $w = xyz$ s.t. the conditions of the pumping lemma hold. In particular we know that

$$1 \leq |y| \leq |xy| \leq n$$

Using the 3rd condition we know that

$$xyyz \in L$$

that is $|xyyz|$ is a square. However we know that

$$\begin{aligned}
n^2 &= |w| \\
&= |xyz| \\
&< |xyyz| && \text{because } 1 \leq |y| \\
&= |xyz| + |y| \\
&\leq n^2 + n && \text{because } |y| \leq n \\
&< n^2 + 2n + 1 \\
&= (n+1)^2
\end{aligned}$$

To summarize, we have

$$n^2 < |xyyz| < (n+1)^2$$

That is $|xyyz|$ lies between two subsequent squares. But then it cannot be a square itself, and hence we have a contradiction to $xyyz \in L$. We conclude L is not regular. □

Given a word $w \in \Sigma^*$ we write w^R for the word read backwards. E.g. $\mathtt{abc}^R = \mathtt{bca}$. Formally this can be defined as

$$\begin{aligned}
\epsilon^R &= \epsilon \\
(xw)^R &= w^R x
\end{aligned}$$

We use this to define the language of even length palindromes

$$L_{\text{pali}} = \{ww^R \mid w \in \Sigma^*\}$$

E.g. for $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ we have $\mathtt{abba} \in L_{\text{pali}}$. Using the intuition that finite automata can only use finite memory, it should be clear that this language is not regular either: to check whether the 2nd half is the same as the 1st half read backwards, we have to remember the first half, however long it is. Indeed, we can show:

Theorem 6.4 *Given $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ we have that L_{pali} is not regular.*

Proof. We use the pumping lemma: We assume that L_{pali} is regular. Now given a pumping number n we construct $w = \mathtt{a}^n\mathtt{bba}^n \in L_{\text{pali}}$, this word is certainly longer than n. From the pumping lemma we know that there is a splitting of the word $w = xyz$ s.t. $|xy| \leq n$ and hence y may only contain $\mathtt{a}$'s and because $y \neq \epsilon$ at least one. We conclude that $xz \in L_{\text{pali}}$ where $xz = \mathtt{a}^m\mathtt{bba}^n$ where $m < n$. However, this word is not a palindrome, because the sequence of $\mathtt{a}$'s at the beginning is shorter that the sequence of $\mathtt{a}$'s at the end. Hence our assumption L_{pali} is regular must be wrong. □

The proof works for any alphabet with at least 2 different symbols. However, if Σ contains only one symbol, as in $\Sigma = \{1\}$, then L_{pali} is the language of an even number of 1s and this is regular: $L_{\text{pali}} = (11)^*$.

6.3 Exercises

Exercise 6.1

Apply the pumping lemma for regular languages to show that the following languages are not regular:

1. $L_1 = \{a^nb^mc^{n+m} \mid m, n \in \mathbb{N}\}$ over the alphabet $\Sigma_1 = \{a, b, c\}$.
   (E.g., $aabbbccccc \in L_1$, but $aabbcc \notin L_1$.)
2. $L_2 = \{w \in \Sigma^* \mid \#_a(w) = 2 \times \#_b(w) \;\wedge\; \#_b(w) = 2 \times \#_c(w)\}$ over the alphabet $\Sigma = \{a, b, c\}$, where $\#_a(w)$ denotes the number of a's in a word w, $\#_b(w)$ the number of b's, etc.
   (E.g., $abcaaba \in L_2$, but $aaabbc \notin L_2$, and $aaaaabbc \notin L_2$.)

7 Context-Free Grammars

This section introduces *context-free grammars* (CFGs) as a formalism to define languages that is more general than regular expressions; that is, there are more languages definable by CFGs than by regular expressions and finite automata. The class of languages definable by CFGs is known as the *context-free languages* or *type 2 languages* (section 1.1). We will define the notion of automata corresponding to CFGs, the *push down automata* (PDA), later, in section 9.

Context-free grammars have an abundance of applications. A prominent example is the definition of (aspects of) the syntax programming languages, like C, Java, or Haskell. Another application is the document type definition (DTD) for the SGML-family markup languages, like XML and HTML. Applications of CFGs are discussed further in section 7.6.

7.1 What are context-free grammars?

We start be defining what context-free grammars are, their *syntax*, deferring what CFGs mean, the languages they describe or their *semantics*, to section 7.2. A context-free grammar $G = (N, T, P, S)$ is given by

- A finite set N of *nonterminal symbols* or *nonterminals*.
- A finite set T of *terminal symbols* or *terminals*.
- $N \cap T = \emptyset$; i.e., the sets N and T are disjoint.
- A finite set $P \subseteq N \times (N \cup T)^*$ of productions. A production (A, α), where $A \in N$ and $\alpha \in (N \cup T)^*$ is a sequence of nonterminal and terminal symbols. It is written as $A \to \alpha$ in the following.
- $S \in N$: the distinguished start symbol.

Nonterminals are also referred to as *variables* and consequently the set of nonterminals is sometimes denoted by V. Yet another term for the the same thing is *syntactic categories*. The terminals are the alphabet of the language defined by a CFG, and for that reason the set of terminals is sometimes denoted by Σ. Indeed, we will occasionally use that convention as well in the following.

Note that the right-hand side of a production may be empty. This is known as an ϵ-production and written

$$A \to \epsilon$$

Also note that it is perfectly permissible for the *same* nonterminal to occur both to the left and to the right of the arrow in a production. In fact, this is essential: if that were not allowed, CFGs would only amount to a (possibly) compact description of finite languages and be of little interest. A production for a nonterminal A where the same nonterminal is the first symbol of the right-hand side, in the leftmost position, is called *immediately left-recursive*; e.g.,

$$A \to A\alpha$$

where $\alpha \in (N \cup T)^*$. A production for a nonterminal A where the same nonterminal is the last symbol of the right-hand side, in the rightmost position, is called *immediately right-recursive*; e.g.,

$$A \to \alpha A$$

Recursion can also be *indirect*: the left-hand side non-terminal of a production can be reached again from the right-hand side via one or more other productions. This is very common: see the following example for an illustration.

As an example we define a grammar for the language of arithmetic expressions over a using only $+$ and $*$. As we will see in section 7.2 where the language described by a CFG is defined, the elements of this language are words like $a + (a * a)$ or $(a + a) * (a + a)$. On the other hand, words like $a + +a$ or $)(a$, which manifestly do not correspond to well-formed arithmetic expressions, do not belong to the language:

$$G_{\text{arith}} = (\{E, T, F\},\ \{(,), a, +, *\},\ P,\ E)$$

where P is given by:

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

Here, the choice E, T, F for the nonterminal symbols is meant to suggest *Expression*, *Term*, and *Factor*, respectively. Note that some of the productions for E and T are immediately left-recursive, and that one of the productions for F recursively refer back to the start symbol S. The latter is an example of indirect recursion.

Somewhat unfortunately, T is used here both as one of the nonterminals and to denote the set of terminals, as per the conventions outlined above. Do not let this confuse you: the nonterminal symbol T and the set of terminal symbols are quite distinct! Occasional name clashes are a fact of life.

To save space, we may combine all the rules with the same left-hand side, separating the alternatives with a vertical bar. Using this convention, our set of productions can be written

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

In practice, the set of productions is often given by just listing the productions, without explicit braces indicating a set:

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

Either way, these are just a more convenient ways to write down exactly the same set of productions.

7.2 The meaning of context-free grammars

How can we check if a word $w \in T^*$ is in the language of a grammar? We start with the start symbol S. Note that this is a word in $(N \cup T)^*$. If there is a production $S \to \alpha$, we can obtain a new word in $(N \cup T)^*$ by replacing the S by the right-hand side α of the production. Should there be further nonterminals in the resulting word, the process is repeated by looking for a production where the left-hand side is one of those nonterminals and replacing that nonterminal by the right-hand side of the production. This process is called a *derivation*. It is often is the case that there is a choice between derivation steps as there can be more than one nonterminal in a word and more than one production for a nonterminal. Any word $w \in T^*$ derived in this way belongs to the language defined by the grammar.

Let us consider our expression grammar G_{arith} from section 7.1. In this case, the start symbol is E. The following is one possible derivation:

$$
\begin{aligned}
E &\underset{G_{\text{arith}}}{\Rightarrow} E + T \\
&\underset{G_{\text{arith}}}{\Rightarrow} T + T \\
&\underset{G_{\text{arith}}}{\Rightarrow} F + T \\
&\underset{G_{\text{arith}}}{\Rightarrow} a + T \\
&\underset{G_{\text{arith}}}{\Rightarrow} a + F \\
&\underset{G_{\text{arith}}}{\Rightarrow} a + (E) \\
&\underset{G_{\text{arith}}}{\Rightarrow} a + (T) \\
&\underset{G_{\text{arith}}}{\Rightarrow} a + (T * F) \\
&\underset{G_{\text{arith}}}{\Rightarrow} a + (F * F) \\
&\underset{G_{\text{arith}}}{\Rightarrow} a + (a * F) \\
&\underset{G_{\text{arith}}}{\Rightarrow} a + (a * a)
\end{aligned}
$$

Generally, given a grammar G, the symbol $\underset{G}{\Rightarrow}$ stands for the relation *derives in one step in grammar* G or *directly derives in grammar* G. It has nothing to do with implication. When the grammar used is clear from the context, it is conventional to drop the subscript G and simply use $\Rightarrow$, read *directly derives*. In the example above, we always replaced the leftmost nonterminal symbol. This is called a *leftmost* derivation. However, as was remarked before, this is not necessary: in general, we are free to pick any nonterminal for replacement. An alternative would be to always pick the rightmost one, which results in a *rightmost* derivation. Or we could pick whichever nonterminal seems more convenient to expand. The symbols $\underset{\text{lm}}{\Rightarrow}$ and $\underset{\text{rm}}{\Rightarrow}$ are sometimes used to indicate leftmost and rightmost derivation steps, respectively.

Given any grammar $G = (N, T, P, S)$ we define the relation *directly derives in grammar* G as follows:

$$
\begin{aligned}
&\underset{G}{\Rightarrow} \subseteq (N \cup T)^* \times (N \cup T)^* \\
&\alpha A \gamma \underset{G}{\Rightarrow} \alpha\beta\gamma \iff A \to \beta \in P
\end{aligned}
$$

The relation *derives in grammar* G, derivation in 0 or more steps, is defined as:

$$
\begin{aligned}
&\overset{*}{\underset{G}{\Rightarrow}} \subseteq (N \cup T)^* \times (N \cup T)^* \\
&\alpha_0 \overset{*}{\underset{G}{\Rightarrow}} \alpha_n \iff \alpha_0 \underset{G}{\Rightarrow} \alpha_1 \underset{G}{\Rightarrow} \dots \alpha_{n-1} \underset{G}{\Rightarrow} \alpha_n
\end{aligned}
$$

where $n \in \mathbb{N}$. Thus $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$, as n may be 0. We will also occasionally use $\overset{+}{\underset{G}{\Rightarrow}}$ meaning derivation in one or more steps[12].

A word $\alpha \in (N \cup T)^*$ such that $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ is called a *sentential form*. The *language* of a grammar, $L(G) \subseteq T^*$, consists of all terminal sentential forms:

$$L(G) = \{w \in T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

A language that can be defined by a context-free grammar is called a context-free language (CFL).

7.3 The relation between regular and context-free languages

A grammar in which each production has at most one non-terminal symbol in its right-hand side is *linear*. For example,

$$G_1 = (\{S\},\ \{0,1\},\ \{S \to \epsilon \mid 0S1\},\ S)$$

is a linear grammar. There are two special cases of linear grammars:

- A linear grammar is *left-linear* if each right-hand side nonterminal is the leftmost (first) symbol in its right-hand side.
- A linear grammar is *right-linear* if each right-hand side nonterminal is the rightmost (last) symbol in its right-hand side.

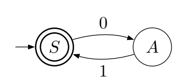
For example,

$$G_2 = (\{S, A\},\ \{0,1\},\ \{S \to \epsilon \mid A1,\ A \to S0\},\ S)$$

is left-linear, and

$$G_3 = (\{S, A\},\ \{0,1\},\ \{S \to \epsilon \mid 0A,\ A \to 1S\},\ S)$$

is right-linear. Collectively, left-linear and right-linear grammars are called *regular* grammars because the languages they describe are *regular*. We will not prove this fact, but it is easy to see how right-linear grammars correspond directly to NFAs. For example, G_3 corresponds to the following NFA:



[12] $\overset{*}{\underset{G}{\Rightarrow}}$ is the *reflexive transitive closure* of $\underset{G}{\Rightarrow}$, and $\overset{+}{\underset{G}{\Rightarrow}}$ is the *transitive closure* of $\underset{G}{\Rightarrow}$.

Thus we see that context-free grammars can be used to describe at least some regular languages. On the other hand, some of the languages that we have shown not to be regular are actually context-free. For example, the (linear) grammar G_1 above describes the language $\{0^n1^n \mid n \in \mathbb{N}\}$ that we proved (theorem 6.2) not to be regular. It is worth noting that if we allow left-linear and right-linear productions to be mixed, the resulting language is not necessarily regular. For example, the grammar

$$G_1' = (\{S, A\},\ \{0, 1\},\ \{S \to \epsilon \mid 0A,\ A \to S1\},\ S)$$

is equivalent to G_1; i.e., describes the same, non-regular, language. Further, we proved (theorem 6.4) that the language of even-length palindromes

$$L_{\text{pali}} = \{ww^R \ \mid\ w \in \{a, b\}^*\}$$

is not regular. The following context-free grammar is one way of defining this language, demonstrating that L_{pali} is a context-free language:

$$G_{\text{pali}} = (\{S\},\ \{\text{a}, \text{b}\},\ \{S \to \epsilon \mid \text{a}S\text{a} \mid \text{b}S\text{b}\},\ S)$$

So, what is the relation between the regular and context-free languages? Are there languages that are regular but not context-free? The answer is no:

Theorem 7.1 *All regular languages are context-free.*

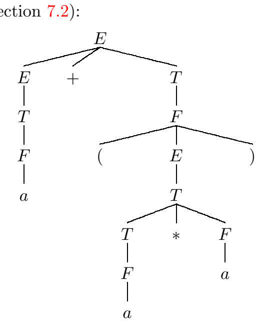
Again, we do not give a proof, but the idea is that regular expressions can be translated into (regular) context-free grammars. For example, $\mathbf{a}^*\mathbf{b}^*$ can be translated into:

$$(\{A, B\},\ \{a, b\},\ \{A \to aA \mid B,\ B \to bB \mid \epsilon\},\ A)$$

As a consequence of theorem 7.1 and the fact that we have seen that there are at least some languages that are context-free but not regular, we have established that the regular languages form a *proper subset* of the context-free ones.

7.4 Derivation trees

A derivation in a context-free grammar induce a corresponding *derivation tree* that reflects the *structure* of the derivation: how each nonterminal was rewritten. As an example, consider the tree representation of the derivation of $a + (a * a)$ in grammar G_{arith} (section 7.2):



The central point is that the parent-child relationship reflects a derivation step according to a production in the grammar. For example, the production $F \to (E)$ was used once in the derivation of $a + (a * a)$, and consequently there is a node labelled F in the derivation tree with child nodes labelled (, E,), ordered from left to right.

In more detail, a tree is a *derivation tree* for a CFG $G = (N, T, P, S)$ iff:

1. Every node has a label from $N \cup T \cup \{\epsilon\}$.
2. The label of the root node is S.
3. Labels of internal nodes belong to N.
4. If a node n has label A and nodes $n_1, n_2, \ldots, n_k$ are children of n, from left to right, with labels $X_1, X_2, \ldots X_k$, respectively, then $A \to X_1X_2\ldots X_k$ is a production in P.
5. If a node n has label ϵ, then n is a leaf and the only child of its parent.

Through the notion of the *yield* of a derivation tree, the relationship between a derivation tree and corresponding derivations can be made precise:

- The string of *leaf labels* read from left to right, eliding any ϵ bar one if it is the only remaining symbol, constitute the *yield* of the tree.
- For a CFG $G = (N, T, P, S)$, a string $\alpha \in (N \cup T)^*$ is the yield of some derivation tree iff $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$.

Note that the leaf nodes may be labelled with either terminal or nonterminal symbols. The yield is thus a *sentential form* (see section 7.2) in general, and not necessarily a word in the language of the context-free grammar.

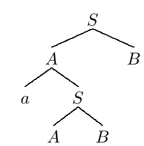
To illustrate the above point, along with elision of superfluous ϵs from the yield, consider the grammar

$$\begin{array}{rcl} S & \to & AB \\ A & \to & aS \mid \epsilon \\ B & \to & Sb \mid \epsilon \end{array}$$

One derivation in this grammar is

$$S \Rightarrow AB \Rightarrow aSB \Rightarrow aABB$$

The corresponding derivation tree is

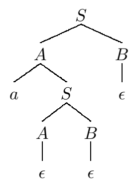


Here, the yield is the sentential form $aABB$.

We can continue the derivation, using the ϵ-productions for the nonterminals A and B:

$$aABB \Rightarrow aBB \Rightarrow aB \Rightarrow a$$

The derivation tree for the entire derivation is

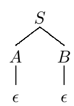


with the yield being just a.

For an example where the yield is empty, consider the derivation

$$S \Rightarrow AB \Rightarrow B \Rightarrow \epsilon$$

The derivation tree is



with yield ϵ.

7.5 Ambiguity

A CFG $G = (N, T, P, S)$ is *ambiguous* iff there is at least one word $w \in L(G)$ such that there are

- two different *derivation trees*, or equivalently
- two different *leftmost derivations*, or equivalently
- two different *rightmost derivations*

for w. This is usually a bad thing because it entails that there is more than one way to interpret a word; i.e., it leads to semantic ambiguity.

As an example consider the following variation of a grammar for simple arithmetic expressions (SAE):

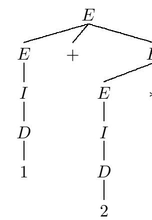
$$SAE = (N = \{E, I, D\}, T = \{+, *, (,), 0, 1, \ldots, 9\}, P, E)$$

where P is given by:

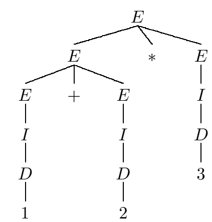
$$\begin{array}{rcl} E & \rightarrow & E + E \\ & | & E * E \\ & | & (E) \\ & | & I \\ I & \rightarrow & DI \mid D \\ D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

The grammar SAE allows expressions involving numbers to be derived, unlike G_{arith}. Also note that this grammar is simpler in that there only is one nonterminal (E) involved at the level of expressions proper, as opposed to three (E, T, F) for G_{arith}.

Consider the word $1 + 2 * 3$. The following derivation tree shows that this word belongs to $L(SAE)$:



Note that the yield is the word above, $1 + 2 * 3$. But there is *another* tree with the same yield:



Thus, there is *one* word for which there are *two* derivation trees. This shows that the grammar SAE is ambiguous.

As per the definition of ambiguity, another way to demonstrate ambiguity is to find two leftmost or two rightmost derivations for a word. It is easy to see that there is a one-to-one correspondence between a derivation tree and a leftmost and a rightmost derivation. To illustrate, here are the two leftmost derivations for $1 + 2 * 3$, corresponding to the first and to the second tree respectively:

$$\begin{aligned} E &\underset{\text{lm}}{\Rightarrow} E + E \underset{\text{lm}}{\Rightarrow} I + E \underset{\text{lm}}{\Rightarrow} D + E \underset{\text{lm}}{\Rightarrow} 1 + E \underset{\text{lm}}{\Rightarrow} 1 + E * E \\ &\underset{\text{lm}}{\Rightarrow} 1 + I * E \underset{\text{lm}}{\Rightarrow} 1 + D * E \underset{\text{lm}}{\Rightarrow} 1 + 2 * E \\ &\underset{\text{lm}}{\Rightarrow} 1 + 2 * I \underset{\text{lm}}{\Rightarrow} 1 + 2 * D \underset{\text{lm}}{\Rightarrow} 1 + 2 * 3 \end{aligned}$$

and

$$\begin{aligned} E &\underset{\text{lm}}{\Rightarrow} E * E \underset{\text{lm}}{\Rightarrow} E + E * E \underset{\text{lm}}{\Rightarrow} I + E * E \underset{\text{lm}}{\Rightarrow} D + E * E \\ &\underset{\text{lm}}{\Rightarrow} 1 + E * E \underset{\text{lm}}{\Rightarrow} 1 + I * E \underset{\text{lm}}{\Rightarrow} 1 + D * E \underset{\text{lm}}{\Rightarrow} 1 + 2 * E \\ &\underset{\text{lm}}{\Rightarrow} 1 + 2 * I \underset{\text{lm}}{\Rightarrow} 1 + 2 * D \underset{\text{lm}}{\Rightarrow} 1 + 2 * 3 \end{aligned}$$

Note, *one* word, *two* different *leftmost* derivations. Thus the grammar is ambiguous. Exercise: Find the *rightmost* derivation corresponding to each tree.

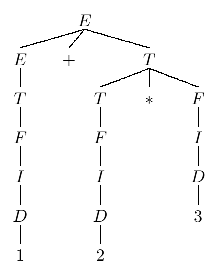
There are two reasons for why ambiguity is problematic. The first we already alluded to: semantic ambiguity. Suppose we wish to assign a meaning to a word beyond the word itself. In our case, the meaning might be the result of evaluating the expression, for instance. Then the first tree suggests that the expression should be read as $1+(2*3)$, which evaluates to 7, while the second tree suggests the expression should be read as $(1+2)*3$, which evaluates to 9[13]. Note how the bracketing reflects the tree structure in each case. We have *one* word with *two* different *interpretations* and thus semantic ambiguity.

The other reason is that many methods for *parsing*, i.e. determining if a given word belongs to the language defined by a CFG, do not work for ambiguous grammars. In particular, this applies to efficient methods for parsing that usually are what we would like to use for that very reason. We return to parsing in section 10.

Fortunately, it is often possible to change an ambiguous grammar into an unambiguous one that is equivalent; i.e., the language is unchanged. Such *grammar transformations* are discussed 8. But for now, let us note that G_{arith} was carefully structured so as to be unambiguous. We can restructure SAE similarly:

$$\begin{array}{rcl} E & \rightarrow & E+T \mid T \\ T & \rightarrow & T*F \mid F \\ F & \rightarrow & (E) \mid I \\ I & \rightarrow & DI \mid D \\ D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

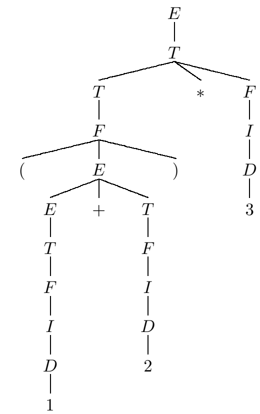
Now there is only one possible derivation tree for $1+2*3$:



Convince yourself that this is the case. Note that this tree corresponds to the reading $1+(2*3)$ of the expression; i.e. a reading where multiplication has higher precedence than addition. This is, of course, is the standard convention.

[13] Strictly speaking, this is just a natural and often convenient convention, because it implies that the meaning of the whole can be understood in terms of the meaning of the parts in the obvious way implied by the structure of the tree. However, sometimes, in real compilers, it might be necessary or convenient to parse things in a way that does not reflect the semantics as directly. But then the parse tree is usually transformed later into a simplified, internal version (an *Abstract Syntax Tree*), the structure of which reflects the intended semantics as described here.

Now, if the interpretation $(1+2)*3$ is desired, *explicit* parentheses have to be used. The (only!) derivation tree for this word is:



Note that the addition now is a subexpression (subtree) of the overall expression, which is what was desired.

7.6 Applications of context-free grammars

An important example for context-free languages is the syntax of programming languages. For example, the specification of the Java programming language[14] [GJS+15] uses a context-free grammar to specify the syntax of Java. Another example is the specification of the syntax of the language used in the G53CMP Compilers module [Nil16].

In practice, the exact details of how a grammar is presented often differs a little bit from the conventions introduced here. The Java language specification being a case in point. Further common variations include Backus-Naur form (BNF)[15] and Extended Backus-Naur form (EBNF)[16]. But these differences are entirely superficial. For example, the symbol ::= is often used instead of $\rightarrow$. For another example, EBNF introduces a shorthand notation for “zero or more of”, commonly denoted by enclosing a grammar symbol (or symbols) in curly braces or using a Kleene-star like notation; e.g.:

$$A \rightarrow \ldots\{B\}\ldots$$

or

$$A \rightarrow \ldots B^{*}\ldots$$

[14] `http://docs.oracle.com/javase/specs/`
[15] `https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form`
[16] `https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form`

It is easy to see that this iterative construct really just is a shorthand for basic productions as it readily can be expressed by introducing an auxiliary nonterminal and a couple of associated productions, either using left recursion or right recursion. The example above can be translated into the equivalent immediately left-recursive (section 7.1) productions

$$\begin{array}{rcl} A & \rightarrow & \ldots A_1 \ldots \\ A_1 & \rightarrow & \epsilon \mid A_1 B \end{array}$$

or into the equivalent immediately right-recursive productions

$$\begin{array}{rcl} A & \rightarrow & \ldots A_1 \ldots \\ A_1 & \rightarrow & \epsilon \mid B A_1 \end{array}$$

where A_1 is a the new, auxiliary, nonterminal.

Note that not all syntactic aspects of common programming languages are captured by the context-free grammar. For example, requirements regarding declaring variables before they are used and type correctness of expressions cannot be captured through context-free languages. However, at least in the area of programming languages, it is common practice to use the notion of "syntactically correct" in the more limited sense of "conforming to the context-free grammar of the language". Aspects such as type-correctness are considered separately.

Another application is the document type definition (DTD)[17] for the SGML-family markup languages, like XML and HTML. A DTD defines the document structure including legal elements and attributes. It can be declared inline, as a header of the document to which the definition applies, or as an external reference.

Extensions of context-free grammars are used in computer linguistics to describe natural languages. In fact, as mentioned in section 1.1, context-free grammars were originally invented by Noam Chomsky for describing natural languages. As an example, consider the following set of terminals, each an English word:

$$T = \{\text{the}, \text{dog}, \text{cat}, \text{that}, \text{bites}, \text{barks}, \text{catches}\}$$

We can then define a grammar $G = (\{S, N, NP, VI, VT, VP\}, T, P, S)$ where P is the following set of productions:

$$\begin{array}{rcl} S & \rightarrow & NP\ VP \\ N & \rightarrow & \text{cat} \mid \text{dog} \\ NP & \rightarrow & \text{the}\ N \mid NP\ \text{that}\ VP \\ VI & \rightarrow & \text{barks} \mid \text{bites} \\ VT & \rightarrow & \text{bites} \mid \text{catches} \\ VP & \rightarrow & VI \mid VT\ NP \end{array}$$

This grammar allows us to derive interesting sentences like:

the dog that catches the cat that bites barks

[17] `https://en.wikipedia.org/wiki/Document_type_definition`

7.7 Exercises

Exercise 7.1

Consider the following Context-Free Grammar (CFG) G:

$$\begin{array}{rcl} S & \rightarrow & X \mid Y \\ X & \rightarrow & aXb \mid \epsilon \\ Y & \rightarrow & cYd \mid \epsilon \end{array}$$

S, X, Y are nonterminal symbols, S is the start symbol, and a, b, c, d are terminal symbols.

1. Derive the following words using the grammar G. Answer by giving the entire derivation sequence from the start symbol S:
   (a) ϵ
   (b) $aabb$
   (c) $cccddd$
2. Does the string $aaaddd$ belong to the language $L(G)$ generated by the grammar G? Provide a brief justification.
3. Give a set expression (using set comprehensions and operations on sets like union) denoting the language $L(G)$.

Exercise 7.2

Construct a context free grammar generating the language

$$L = \{(ab)^m(bc)^n(cb)^n(ba)^m \mid m, n \geq 1\}\{d^n \mid n \geq 0\} \cup \{d^n \mid n \geq 2\}$$

over the alphabet $\{a, b, c, d\}$ (parentheses are only used for grouping). Note that set concatenation has higher precedence than set union. Explain your construction.

Exercise 7.3

Consider the Context-Free Grammar (CFG) $G = (N, T, P, S)$ where $N = \{S, X, Y\}$ are the nonterminal symbols, $T = \{a, b, c\}$ are the terminal symbols, S is the start symbol, and the set of productions P is:

$$\begin{array}{rcl} S & \rightarrow & X \mid Y \\ X & \rightarrow & aXb \mid ab \\ Y & \rightarrow & bYc \mid \epsilon \end{array}$$

Recall that the relation "derives directly in G" is a relation on strings of terminals and non-terminals; i.e.

$$\underset{G}{\Rightarrow} \subseteq (N \cup T)^* \times (N \cup T)^*$$

such that for all $\alpha, \gamma \in (N \cup T)^*$

$$\alpha A \gamma \underset{G}{\Rightarrow} \alpha\beta\gamma \iff A \rightarrow \beta \in P$$

List all pairs (ϕ, θ) of the relation $\underset{G}{\Rightarrow}$ for the cases where either $\phi \in \{X, XY, aXbYc, cc\}$ or $\theta = a$.

Exercise 7.4

Consider the following Context-Free Grammar (CFG) *Exp*:

$$
\begin{array}{rcl}
T & \to & T+T \mid F \\
F & \to & F*F \mid P \\
P & \to & N(A) \mid (T) \mid I \\
N & \to & f \mid g \mid h \\
A & \to & T \mid \epsilon \\
I & \to & DI \mid D \\
D & \to & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

T, F, P, N, A, I, D are nonterminals; $+$, $*$, f, g, h, (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are terminals; T is the start symbol.

1. Derive the following words in the grammar *Exp* where possible. If it is possible to derive the word, give the entire *left-most* derivation; i.e. always expand the left-most non-terminal of the sentential form[18]. If it is not possible to derive the word, give a brief explanation as to why not.
   (a) (789)
   (b) $7+g(3*5)*(f())$
   (c) $1+2*3)$
   (d) $1+7(9)$
2. Draw a derivation tree for the word $7+(8*h(1))+9$ in the grammar *Exp*.
3. Draw another derivation tree for the word $7+(8*h(1))+9$ from 2. What does the fact that there are two different derivation trees for one word tell about the grammar *Exp*?
4. Modify the relevant productions of the grammar *Exp* so that a function symbol (one of f, g, h) can be applied to zero, one, or more arguments, separated by a single comma when there are more than one argument, instead of just zero or one argument. For example, it should be possible to derive words like
   $$f(2, g(), h(3+4))$$
   Explain your construction.

[18] *Sentential form:* word derivable from the start symbol.

8 Transformations of context-free grammars

In this section, we discuss how context free grammars can be restructured systematically without changing the language that they describe. There are many reasons for doing this, such as simplifying a grammar, putting it into some specific form, or eliminating ambiguities.

8.1 Equivalence of context-free grammars

Two grammars G_1 and G_2 are *equivalent* iff

$$L(G_1) = L(G_2)$$

Whenever a grammar is being transformed, it is assumed that the resulting grammar is equivalent to the original one.

For example, the following two grammars are equivalent:

$$
G_1: \quad \begin{array}{rcl} S & \to & \epsilon \mid A \\ A & \to & a \mid aA \end{array}
\qquad\qquad
G_2: \quad \begin{array}{rcl} S & \to & A \\ A & \to & \epsilon \mid Aa \end{array}
$$

$L(G_1) = \{a\}^* = L(G_2)$. The equivalence of CFGs is in general *undecidable* (section 11).

8.2 Elimination of uselsss productions

In a context free grammar $G = (N, T, P, S)$, a nonterminal $X \in N$ is

- *reachable* iff $S \overset{*}{\Rightarrow} \alpha X \beta$ for some $\alpha, \beta \in \{N \cup T\}^*$
- *productive* iff $X \overset{*}{\Rightarrow} w$ for some $w \in T^*$

Phrased differently, a nonterminal X is reachable if it occurs in some sentential form. Productions for nonterminals that are unreachable or unproductive, or where an unproductive nonterminal occurs on the right-hand side, are collectively knows as *useless productions*. Any useless production can be removed from a grammar without changing the language.

For example, consider the following grammar, where $N = \{S, A, B\}$, $T = \{a, b\}$, and S is the start symbol:

$$
\begin{array}{rcl}
S & \to & aAB \mid b \\
A & \to & aA \mid a \\
B & \to & bB
\end{array}
$$

The nonterminal B is unproductive as there is no way to derive a word of only terminal symbols from it. This makes the productions $S \to aAB$ and $B \to bB$ useless. Removing those productions leaves us with:

$$
\begin{array}{rcl}
S & \to & b \\
A & \to & aA \mid a
\end{array}
$$

But now A has clearly become an unreachable nonterminal, making the productions $A \to aA$ and $A \to a$ useless too. If we remove those as well we obtain:

$$S \rightarrow b$$

Of course, any terminal and nonterminals that no longer occur in any productions can also be eliminated. Thus the set of nonterminals is now just $\{S\}$ and the set of terminals just $\{b\}$.

8.3 Substitution

As a direct consequence of how derivation in a grammar is defined, it follows that an occurrence of a non-terminal in a right-hand side may be replaced by the right-hand sides of the productions for that non-terminal if done in all possible ways. This is a form of *substitution*.

For example, consider the following grammar fragment. We assume it includes *all* productions for B, and we wish to eliminate the occurrence of B in the right-hand side of the production for A:

$$\begin{array}{lcl} A & \rightarrow & XBY \\ B & \rightarrow & C \mid D \\ B & \rightarrow & \epsilon \end{array}$$

Note that there are three productions for B. By substitution, this grammar fragment can be transformed into:

$$\begin{array}{lcl} A & \rightarrow & XCY \mid XDY \mid XY \\ B & \rightarrow & C \mid D \\ B & \rightarrow & \epsilon \end{array}$$

Thus we get one new production for A for each alternative for B. Note that we cannot necessarily remove the productions for B: there may be other occurrences of B in the grammar (as the above was only a fragment). However, if substitution renders some symbols unreachable, then the productions for those symbols become useless and can consequently be removed (section 8.2).

Substitution can of course also be performed on recursive productions. That is often not very useful, though, as the productions just get bigger and more numerous without eliminating any nonterminals. For example, assume the following productions are *all* productions for A in some grammar. Note that $A \to Ab$ is recursive (immediately left-recursive, as it happens):

$$A \rightarrow \epsilon \mid Ab$$

We can substitute the right-hand sides of all productions for A for the one occurrence of A in the right-hand side of $A \to Ab$. That would leave us with:

$$A \rightarrow \epsilon \mid b \mid Abb$$

Thus we did not gain anything here, unless there was some specific reason for wanting a single production that shows that A can yield a single b.

8.4 Left factoring

Sometimes it is useful to factor out a common prefix of the right-hand sides of a group of productions. This is called *left factoring*. Left factoring can make a grammar easier to read and understand as it captures a recurring pattern in one place. It is sometimes also necessary for putting a grammar into a form suitable for use with certain parsing methods (section 10).

Consider the following two productions for A. Note the common prefix XY:

$$A \rightarrow XYX \mid XYZZY$$

After left factoring:

$$\begin{array}{lcl} A & \rightarrow & BX \mid BZZY \\ B & \rightarrow & XY \end{array}$$

8.5 Disambiguating context-free grammars

Given an ambiguous context-free grammar G, it is often possible to construct an equivalent but *unambiguous* grammar G'. Some context-free languages are *inherently ambiguous*, meaning that every CFG generating the language is necessarily ambiguous, but most languages of practical interest, such as programming languages, can be given unambiguous CFGs.

In this section, we focus on how expression languages (like arithmetic expressions or regular expressions) can be given unambiguous CFGs by structuring the grammar to account for *operator precedence* and *operator associativity*.

The following is a CFG for simple arithmetic expressions. For simplicity, we only consider the numbers 0, 1, and 2:

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E * E \mid E \uparrow E \mid (E) \mid N \\ N & \rightarrow & 0 \mid 1 \mid 2 \end{array}$$

E and N are nonterminals, E is the start symbol, $+$, $*$, $\uparrow$, (,), 0, 1, 2 are terminals. The grammar is ambiguous. For example, the word $0+1+2$ has two different leftmost derivations:

$$\begin{array}{l} E \underset{\text{lm}}{\Rightarrow} E + E \underset{\text{lm}}{\Rightarrow} N + E \underset{\text{lm}}{\Rightarrow} 0 + E \underset{\text{lm}}{\Rightarrow} 0 + E + E \underset{\text{lm}}{\Rightarrow} 0 + N + E \underset{\text{lm}}{\Rightarrow} 0 + 1 + E \\ \quad \underset{\text{lm}}{\Rightarrow} 0 + 1 + N \underset{\text{lm}}{\Rightarrow} 0 + 1 + 2 \\ E \underset{\text{lm}}{\Rightarrow} E + E \underset{\text{lm}}{\Rightarrow} E + E + E \underset{\text{lm}}{\Rightarrow} N + E + E \underset{\text{lm}}{\Rightarrow} 0 + E + E \underset{\text{lm}}{\Rightarrow} 0 + N + E \\ \quad \underset{\text{lm}}{\Rightarrow} 0 + 1 + E \underset{\text{lm}}{\Rightarrow} 0 + 1 + N \underset{\text{lm}}{\Rightarrow} 0 + 1 + 2 \end{array}$$

We now wish to construct an equivalent but unambiguous version of the the above grammar by making it reflect the following conventions regarding operator precedence and associativity:

Operators	Precedence	Associativity
$\uparrow$	highest	right
$*$	medium	left
$+$	lowest	left

For example, the word

$$1 + 2 * 2 \uparrow 2 \uparrow 2 + 0$$

should be read as

$$(1 + (2 * (2 \uparrow (2 \uparrow 2)))) + 0$$

That is, the structure of the (one and only) derivation tree should reflect this reading.

To impart operator precedence on the grammar, it has to be *stratified*: First the expressions have to be partitioned into different categories of expressions, one category for each operator precedence level. We thus need to introduce one nonterminal (or syntactic category) for each precedence level. Then it must be arranged so that expressions belonging to the category for operators of one precedence level only occur as *subexpressions* of expressions belonging to the category for operators of the next lower precedence level. Bracketing (enclosing an expression in some form of parentheses) should have higher precedence than any operator. Bracketed subexpressions should thus only be allowed as subexpressions of expressions in the category for the highest operator precedence. The expression enclosed in the parentheses, however, should be of the category for the lowest operator precedence.

In our example, there are three operator precedence levels, so we introduce three additional nonterminals (E_1, E_2, E_3) to stratify the grammar into four levels: one level for each operator precedence category, and one more for the innermost expression level, the subexpressions of expressions involving operators of the highest precedence. The resulting grammar is:

$$\begin{array}{rcl} E & \to & E_1 + E_1 \mid E_1 \\ E_1 & \to & E_2 * E_2 \mid E_2 \\ E_2 & \to & E_3 \uparrow E_3 \mid E_3 \\ E_3 & \to & (E) \mid N \\ N & \to & 0 \mid 1 \mid 2 \end{array}$$

This grammar is fine in that it is unambiguous. But for practical purposes, it is inconvenient as expressions involving more than one operator at the same level has to be explicitly bracketed. For example, the word $0 + 1 + 2$ cannot be derived in this grammar (try it!), but a user of this little expression language would have to write either $(0 + 1) + 2$ or $0 + (1 + 2)$.

This is where operator associativity comes into the picture: by adopting a convention regarding associativity, expressions involving more than operator of some specific precedence level can be implicitly bracketed. In our example, $+$ is left-associative (as per standard mathematical conventions), which means the word $0+1+2$ should be read as $(0+1)+2$; i.e. the derivation tree for $0+1+2$ should branch to the left, suggesting that $0+1$ is a subexpression of the overall expression.

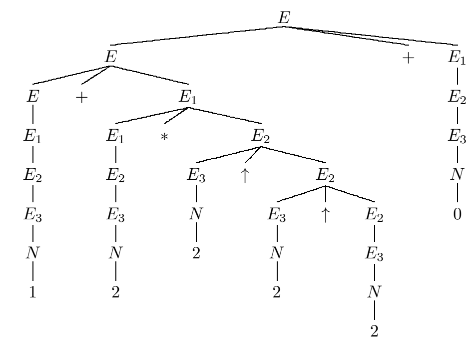
Imparting of associativity is achieved by making productions for *left*-associative operators *left*-recursive, and those for *right*-associative operators *right*-recursive, as this results in the desired left-branching or right-branching structure, respectively, of the derivation tree. Note that the requirement that subexpressions at one precedence level must all be expressions at next higher precedence level is relaxed a little, but only in a way that does not make the grammar unambiguous.

In our example, the operators $+$ and $*$ are left-associative, and the operator $\uparrow$ is right-associative. We thus make the corresponding productions left- and right-recursive, respectively, arriving at the final version of the grammar:

$$\begin{array}{rcl} E & \to & E + E_1 \mid E_1 \\ E_1 & \to & E_1 * E_2 \mid E_2 \\ E_2 & \to & E_3 \uparrow E_2 \mid E_3 \\ E_3 & \to & (E) \mid N \\ N & \to & 0 \mid 1 \mid 2 \end{array}$$

This grammar is (again) unambiguous, but now allows words like $0 + 1 + 2$. Verify that there only is one derivation tree, and that this has the desired left-branching structure!

As an example, let us consider the word $1 + 2 * 2 \uparrow 2 \uparrow 2 + 0$ from above. Recall that we want the reading $(1 + (2 * (2 \uparrow (2 \uparrow 2)))) + 0$. The derivation tree for the word in the final version of the grammar is:



Make sure you understand how the branching structure corresponds to the desired reading of the word; i.e., how precedence and associativity allowed for implicit bracketing. As an exercise, draw the derivation tree for the *explicitly* bracketed word, and compare the structure of the two trees. The trees will not be the same, of course, as the words are not the same (as words, symbol by symbol), but the branching structure (what is a subexpression of what) will agree.

8.6 Elimination of left recursion

A CFG is *left-recursive* if there is some non-terminal A such that $A \overset{+}{\Rightarrow} A\alpha$[19]. Certain parsing methods cannot handle left-recursive grammars. An example is recursive decent parsing as described in section 10. If we want to use such a parsing method for parsing a language $L = L(G)$ given by a left-recursive

[19] Recall that $\overset{+}{\Rightarrow}$ means derives in one or more steps; i.e., the transitive closure of $\Rightarrow$.

grammar G, then it first has to be transformed into an equivalent grammar G' that is *not* left-recursive.

We first consider *immediate* left recursion (section 7.1); i.e., productions of the form $A \to A\alpha$. We assume that α cannot derive ϵ.

The key idea of the transformation is simple. Let us first consider a simplified scenario, with one immediately left-recursive production for a nonterminal A and one non-recursive production:

$$\begin{array}{rcl} A & \to & A\alpha \\ A & \to & \beta \end{array}$$

Then observe that all strings derivable from A using these two productions have the form $\beta(\alpha)^*$ once the last A has been replaced by β; i.e., β followed by zero or more α. Now it is easy to see that the following alternative grammar generates strings of exactly the same form:

$$\begin{array}{rcl} A & \to & \beta A' \\ A' & \to & \alpha A' \mid \epsilon \end{array}$$

This is arguably a more direct way to generate strings of the form $\beta(\alpha)^*$. In essence, the productions say: "start with β, and then tag on zero or more α".

We now generalise and formalise this idea. In order to transform an immediately left-recursive grammar to an equivalent grammar that is not left recursive, proceed as follows. For each nonterminal A defined by some left-recursive production, group the productions for A

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

such that no β_i begins with an A. Then replace the A productions by

$$\begin{array}{rcl} A & \to & \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\ A' & \to & \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{array}$$

Assumption: no α_i derives ϵ.

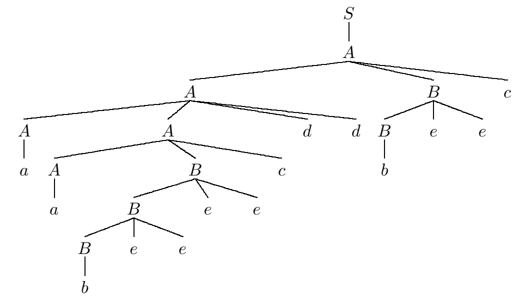
To illustrate, consider the immediately left-recursive grammar

$$\begin{array}{rcl} S & \to & A \mid B \\ A & \to & ABc \mid AAdd \mid a \mid aa \\ B & \to & Bee \mid b \end{array}$$

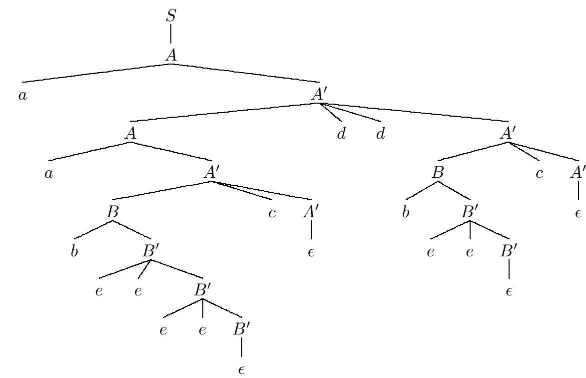
Applying the transformation rules above yields the following equivalent right-recursive grammar:

$$\begin{array}{rclcrcl} S & \to & A \mid B & \quad & B & \to & bB' \\ A & \to & aA' \mid aaA' & & B' & \to & eeB' \mid \epsilon \\ A' & \to & BcA' \mid AddA' \mid \epsilon & & & & \end{array}$$

Let us do a sanity check on the new grammar by picking a word in the language of the original grammar and make sure it can also be derived in the new grammar. The derivation of the word should ideally make use of all recursive productions in the grammar. Let us pick the word *aabeeeecddbeec*. The following is the derivation tree for this word in the original grammar, demonstrating that the word indeed is in the language generated by the grammar:



The derivation tree for the same word in the transformed grammar:



Note that, while the yield is the same in both cases, the structure of the trees are very different: the first tree is more left-branching, due to the left recursion, and the other more right-branching, due to the right-recursion.

To eliminate *general* left recursion, the grammar is first transformed into an *immediately* left-recursive grammar through systematic substitution (section 8.3). After this the elimination scheme set out above can be applied.

Consider the following grammar. Note that no production is immediately left-recursive:

$$\begin{array}{rcl} A & \to & BaB \\ B & \to & Cb \mid \epsilon \\ C & \to & Ab \mid Ac \end{array}$$

The grammar is, however, left recursive because, for example, $A \Rightarrow BaB \Rightarrow CbaB \Rightarrow AbbaB$. Thus we have $A \stackrel{+}{\Rightarrow} A\alpha$ (for $\alpha = bbaB$ in this case), demonstrating that the grammar is left recursive.

To eliminate the left recursion, let us first transform this grammar into an equivalent immediately left-recursive grammar. Let us start by eliminating C by substituting all alternatives for C into the right-hand side of the production $B \to Cb$. Note that this makes C an unreachable nonterminal, making the productions for C useless meaning they can be removed (section 8.2):

$$\begin{array}{rcl} A & \to & BaB \\ B & \to & Abb \mid Acb \mid \epsilon \end{array}$$

Then we can eliminate B wherever it occurs in the *leftmost* position of a right-hand side in the productions for A. The grammar is now immediately left-recursive:

$$\begin{array}{rcl} A & \to & AbbaB \mid AcbaB \mid aB \\ B & \to & Abb \mid Acb \mid \epsilon \end{array}$$

Alternatively, B can be eliminated completely from the productions for A, making B an unreachable terminal allowing the productions for B to be removed:

$$\begin{array}{rcl} A & \to & AbbaAbb \mid AcbaAbb \mid aAbb \\ & \mid & AbbaAcb \mid AcbaAcb \mid aAcb \\ & \mid & Abba \mid Acba \mid a \end{array}$$

Let us go with the smaller version (fewer productions):

$$\begin{array}{rcl} A & \to & AbbaB \mid AcbaB \mid aB \\ B & \to & Abb \mid Acb \mid \epsilon \end{array}$$

Only the productions for A are immediately left-recursive. Applying the transformation to eliminate left recursion gives us:

$$\begin{array}{rcl} A & \to & aBA' \\ A' & \to & bbaBA' \mid cbaBA' \mid \epsilon \\ B & \to & Abb \mid Acb \mid \epsilon \end{array}$$

Note that A appears to the left in B-productions; yet the grammar is no longer left-recursive. Why?

8.7 Exercises

Exercise 8.1

The grammar *Exp* from exercise 7.4 is ambiguous. Fix this problem; i.e., modify the grammar, *without* changing the language of the grammar, so that all words in the language have exactly one derivation tree. You do not need to prove that this holds for the resulting grammar, but you should explain what you did and why.

Exercise 8.2

1. Construct a simple, *unambiguous* grammar according to the following:
   - The integer literals are the only primitive expressions. An integer literal is either 0, or a non-empty word of decimal digits (0, 1, ..., 9) not starting with 0 and with a single optional minus sign ($-$) in front. E.g. 0, 1, 42, -234 are all valid integer literals, but 01, -0, $--1$ are not.
   - There are four binary operators:

Operators	Precedence	Associativity
$<$	1 (lowest)	non-associative
$\oplus$	2	left
$\otimes$	3	left
$\uparrow$	4 (highest)	right

   - Additionally, it should be possible to use parentheses for grouping in the standard way.
2. Draw the derivation tree for $42 < 0 \otimes -10 \otimes (1 \oplus 7) \uparrow 2$ and verify that its structure reflects the specification above.

Exercise 8.3

The following context-free grammar is immediately left-recursive:

$$\begin{array}{rcl} S & \to & Sa \mid XbS \mid a \\ X & \to & XXX \mid YYY \mid XYY \mid YYX \\ Y & \to & cY \mid dY \mid e \end{array}$$

S, X, and Y are nonterminals, S is the start symbol, and a, b, c, d, and e are terminals.

Transform it into an equivalent *right-recursive* grammar. Explicitly show the result of the *grouping step* in addition to the final result after applying the actual *transformation step* to the productions that need transformation.

9 Pushdown Automata

[FOR REFERENCE: PDA NOT TAUGHT SPRING 2019]

We will now consider a new notion of automata *Pushdown Automata* (PDA). PDAs are finite automata with a stack; i.e., a data structure that can be used to store an arbitrary number of symbols (hence PDAs have an infinite set of states) but which can be only accessed in a *last-in-first-out* (LIFO) fashion. The languages that can be recognized by PDA are precisely the context-free languages.

9.1 What is a pushdown automaton?

A Pushdown Automaton $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is given by the following data

- A finite set Q of states,
- A finite set Σ of input symbols (the alphabet),
- A finite set Γ of stack symbols,
- A transition function

  $$\delta \in Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}_{\mathrm{fin}}(Q \times \Gamma^*)$$

  Here $\mathcal{P}_{\mathrm{fin}}(A)$ are the finite subsets of a set; i.e., this can be defined as

  $$\mathcal{P}_{\mathrm{fin}}(A) = \{X \mid X \subseteq A \wedge X \text{is finite.}\}$$

  Thus, PDAs are in general nondeterministic because they may have a choice of transitions from any state. However, there are always only finitely many choices.

- An initial state $q_0 \in Q$,
- An initial stack symbol $Z_0 \in \Gamma$,
- A set of final states $F \subseteq Q$.

As an example we consider a PDA P that recognizes the language of even length palindromes over $\Sigma = \{0, 1\}$: $L = \{ww^R \mid w \in \{0, 1\}^*\}$. Intuitively, this PDA pushes the input symbols on the stack until it *guesses* that it is in the middle and then it compares the input with what is on the stack, popping of symbols from the stack as it goes. If it reaches the end of the input precisely at the time when the stack is empty, it accepts.

$$P_0 = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, \#\}, \delta, q_0, \#, \{q_2\})$$

where δ is given by the following equations:

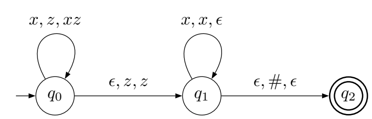
$$\begin{array}{lcl}
\delta(q_0, 0, \#) & = & \{(q_0, 0\#)\} \\
\delta(q_0, 1, \#) & = & \{(q_0, 1\#)\} \\
\delta(q_0, 0, 0) & = & \{(q_0, 00)\} \\
\delta(q_0, 1, 0) & = & \{(q_0, 10)\} \\
\delta(q_0, 0, 1) & = & \{(q_0, 01)\} \\
\delta(q_0, 1, 1) & = & \{(q_0, 11)\} \\
\delta(q_0, \epsilon, \#) & = & \{(q_1, \#)\} \\
\delta(q_0, \epsilon, 0) & = & \{(q_1, 0)\} \\
\delta(q_0, \epsilon, 1) & = & \{(q_1, 1)\} \\
\delta(q_1, 0, 0) & = & \{(q_1, \epsilon)\} \\
\delta(q_1, 1, 1) & = & \{(q_1, \epsilon)\} \\
\delta(q_1, \epsilon, \#) & = & \{(q_2, \epsilon)\} \\
\delta(q, w, z) & = & \emptyset \quad \text{everywhere else}
\end{array}$$

To save space we may abbreviate this by writing:

$$\begin{array}{lcl}
\delta(q_0, x, z) & = & \{(q_0, xz)\} \\
\delta(q_0, \epsilon, z) & = & \{(q_1, z)\} \\
\delta(q_1, x, x) & = & \{(q_1, \epsilon)\} \\
\delta(q_1, \epsilon, \#) & = & \{(q_2, \epsilon)\} \\
\delta(q, x, z) & = & \emptyset \quad \text{everywhere else}
\end{array}$$

where $q \in Q, x \in \Sigma, z \in \Gamma$. We obtain the previous table by expanding all the possibilities for q, x, z.

We draw the transition diagram of P by labelling each transition with a triple x, Z, γ with $x \in \Sigma, Z \in \Gamma, \gamma \in \Gamma^*$:



9.2 How does a PDA work?

At any time the state of the computation of a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is given by:

- the state $q \in Q$ the PDA is in,
- the input string $w \in \Sigma^*$ that still has to be processed,
- the contents of the stack $\gamma \in \Gamma^*$.

Such a triple $(q, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*$ is called an Instantaneous Description (ID).

We define a relation $\underset{P}{\vdash} \subseteq \mathrm{ID} \times \mathrm{ID}$ between IDs that describes how the PDA can change from one ID to the next one. Because PDAs in general are nondeterministic, this is a relation (not a function); i.e., there may be more than one possibility.

There are two possibilities for $\underset{P}{\vdash}$:

1. $(q, xw, z\gamma) \underset{P}{\vdash} (q', w, \alpha\gamma)$ if $(q', \alpha) \in \delta(q, x, z)$

2. $(q, w, z\gamma) \underset{P}{\vdash} (q', w, \alpha\gamma)$ if $(q', \alpha) \in \delta(q, \epsilon, z)$

In the first case the PDA reads an input symbol and consults the transition function δ to calculate a possible new state q' and a sequence of stack symbols α that replaces the currend symbol on the top z.

In the second case the PDA ignores the input and silently moves into a new state and modifies the stack as above. The input is unchanged.

Consider the word 0110. What are possible sequences of IDs for P_0 starting with $(q_0, 0110, \#)$?

$$\begin{array}{lll}
(q_0, 0110, \#) & \underset{P_0}{\vdash} (q_0, 110, 0\#) & \text{1. with } (q_0, 0\#) \in \delta(q_0, 0, \#) \\
 & \underset{P_0}{\vdash} (q_0, 10, 10\#) & \text{1. with } (q_0, 10) \in \delta(q_0, 1, 0) \\
 & \underset{P_0}{\vdash} (q_1, 10, 10\#) & \text{2. with } (q_1, 1) \in \delta(q_0, \epsilon, 1) \\
 & \underset{P_0}{\vdash} (q_1, 0, 0\#) & \text{1. with } (q_1, \epsilon) \in \delta(q_1, 1, 1) \\
 & \underset{P_0}{\vdash} (q_1, \epsilon, \#) & \text{1. with } (q_1, \epsilon) \in \delta(q_1, 0, 0) \\
 & \underset{P_0}{\vdash} (q_2, \epsilon, \epsilon) & \text{2. with } (q_2, \epsilon) \in \delta(q_1, \epsilon, \#)
\end{array}$$

We write $(q, w, \gamma) \overset{*}{\underset{P}{\vdash}} (q', w', \gamma)$ if the PDA can move from (q, w, γ) to (q', w', γ') in a (possibly empty) sequence of moves. Above we have shown that

$$(q_0, 0110, \#) \overset{*}{\underset{P_0}{\vdash}} (q_2, \epsilon, \epsilon).$$

However, this is not the only possible sequence of IDs for this input. E.g. the PDA may just guess the middle wrong:

$$\begin{array}{lll}
(q_0, 0110, \#) & \underset{P_0}{\vdash} (q_0, 110, 0\#) & \text{1. with } (q_0, 0\#) \in \delta(q_0, 0, \#) \\
 & \underset{P_0}{\vdash} (q_1, 110, 0\#) & \text{2. with } (q_1, 0) \in \delta(q_0, \epsilon, 0)
\end{array}$$

We have shown $(q_0, 0110, \#) \overset{*}{\underset{P_0}{\vdash}} (q_1, 110, 0\#)$. Here the PDA gets stuck as there is no state after $(q_1, 110, 0\#)$.

If we start with a word that is not in the language L (like 0011) then the automaton will always get stuck before reaching a final state.

9.3 The language of a PDA

There are two ways to define the language of a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ($L(P) \subseteq \Sigma^*$) because there are two notions of acceptance:

Acceptance by final state

$$L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \gamma) \wedge q \in F\}$$

That is the PDA accepts the word w if there is any sequence of IDs starting from (q_0, w, Z_0) and leading to (q, ϵ, γ), where $q \in F$ is one of the final

states. Here it doesn't play a role what the contents of the stack are at the end.

In our example the PDA P_0 would accept 0110 because $(q_0, 0110, \#) \overset{*}{\underset{P_0}{\vdash}} (q_2, \epsilon, \epsilon)$ and $q_2 \in F$. Hence we conclude $0110 \in L(P_0)$.

On the other hand, because there is no successful sequence of IDs starting with $(q_0, 0011, \#)$, we know that $0011 \notin L(P_0)$.

Acceptance by empty stack

$$L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \epsilon)\}$$

That is the PDA accepts the word w if there is any sequence of IDs starting from (q_0, w, Z_0) and leading to (q, ϵ, ϵ), in this case the final state plays no role.

If we specify a PDA for acceptance by empty stack we will leave out the set of final states F and just use $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

Our example automaton P_0 also works if we leave out F and use acceptance by empty stack.

We can always turn a PDA that uses one acceptance method into one that uses the other. Hence, both acceptance criteria specify the same class of languages.

9.4 Deterministic PDAs

We have introduced PDAs as nondeterministic machines that may have several alternatives how to continue. We now define Deterministic Pushdown Automata (DPDA) as those that never have a choice.

To be precise we say that a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is deterministic (is a DPDA) iff

$$|\delta(q, x, z)| + |\delta(q, \epsilon, z)| \leq 1$$

Remember, that $|X|$ stands for the number of elements in a finite set X.

That is: a DPDA may get stuck but it has never any choice.

In our example the automaton P_0 is not deterministic, e.g. we have $\delta(q_0, 0, \#) = \{(q_0, 0\#)\}$ and $\delta(q_0, \epsilon, \#) = \{(q_1, \#)\}$ and hence $|\delta(q_0, 0, \#)| + |\delta(q_0, \epsilon, \#)| = 2$.

Unlike the situation for finite automata, there is in general no way to translate a nondeterministic PDA into a deterministic one. Indeed, there is no DPDA that recognizes the language L! **Nondeterministic PDAs are more powerful than deterministic PDAs.**

However, we can define a similar language L' over $\Sigma = \{0, 1, \$\}$ that can be recognized by a deterministic PDA:

$$L' = \{w\$w^R \mid w \in \{0, 1\}^*\}$$

That is L' contains palindroms with a marker \$ in the middle, e.g. $01\$10 \in L'$.

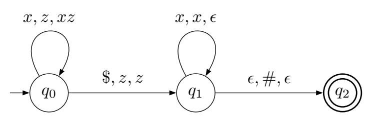
We define a DPDA P' for L':

$$P' = (\{q_0, q_1, q_2\}, \{0, 1, \$\}, \{0, 1, \#\}, \delta', q_0, \#, \{q_2\})$$

where δ' is given by:

$$\begin{array}{rcll}
\delta'(q_0,x,z) & = & \{(q_0,xz)\} & x\in\{0,1\},\ z\in\{0,1,\#\} \\
\delta'(q_0,\$,z) & = & \{(q_1,z)\} & z\in\{0,1,\#\} \\
\delta'(q_1,x,x) & = & \{(q_1,\epsilon)\} & x\in\{0,1\} \\
\delta'(q_1,\epsilon,\#) & = & \{(q_2,\epsilon)\} & \\
\delta'(q,x,z) & = & \emptyset & \text{everywhere else}
\end{array}$$

The transition graph is:



We can check that this automaton is deterministic. In particular the 3rd and 4th line cannot overlap because $\#$ is not an input symbol.

In contrast to PDAs in general, the two acceptance methods are not equivalent for DPDAs: acceptance by final state makes it possible to define a bigger class of langauges. We will consequently always use acceptance by final state for DPDAs in the following.

9.5 Context-free grammars and push-down automata

Theorem 9.1 *For a language $L\subseteq\Sigma^*$ the following two statements are equivalent:*

1. *L is given by a CFG G, $L=L(G)$.*
2. *L is the language of a PDA P, $L=L(P)$.*

To summarize: Context-Free Languages (CFLs) can be described by a Context-Free Grammar (CFG) and can be processed by a pushdown automaton.

We will he only show how to construct a PDA from a grammar - the other direction is shown in [HMU01] (6.3.2, pp. 241).

Given a CFG $G=(V,\Sigma,P,S)$, we define a PDA

$$P(G)=(\{q_0\},\ \Sigma,\ V\cup\Sigma,\ \delta,\ q_0,\ S)$$

where δ is defined as follows:

$$\begin{array}{rcll}
\delta(q_0,\epsilon,A) & = & \{(q_0,\alpha)\ \mid\ A\to\alpha\in P\} & \text{for all } A\in V \\
\delta(q_0,a,a) & = & \{(q_0,\epsilon)\} & \text{for all } a\in\Sigma.
\end{array}$$

We haven't given a set of final states because we use acceptance by empty stack. Yes, we use only one state!

Take as an example

$$G=(\{E,T,F\},\ \{\texttt{(},\texttt{)},\texttt{a},\texttt{+},\texttt{*}\},\ E,\ P)$$

where the set of productions P are given by

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

we define

$$P(G)=(\{q_0\},\ \{\texttt{(},\texttt{)},\texttt{a},\texttt{+},\texttt{*}\},\ \{E,T,F,\texttt{(},\texttt{)},\texttt{a},\texttt{+},\texttt{*}\},\ \delta,\ q_0,\ E)$$

where

$$\begin{array}{rcll}
\delta(q_0,\epsilon,E) & = & \{(q_0,T),\ (q_0,E\texttt{+}T)\} & \\
\delta(q_0,\epsilon,T) & = & \{(q_0,F),\ (q_0,T\texttt{*}F)\} & \\
\delta(q_0,\epsilon,F) & = & \{(q_0,\texttt{a}),\ (q_0,\texttt{(}E\texttt{)})\} & \\
\delta(q_0,\texttt{(},\texttt{(}) & = & \{(q_0,\epsilon)\} & \\
\delta(q_0,\texttt{)},\texttt{)}) & = & \{(q_0,\epsilon)\} & \\
\delta(q_0,\texttt{a},\texttt{a}) & = & \{(q_0,\epsilon)\} & \\
\delta(q_0,\texttt{+},\texttt{+}) & = & \{(q_0,\epsilon)\} & \\
\delta(q_0,\texttt{*},\texttt{*}) & = & \{(q_0,\epsilon)\} & \\
\delta(q,x,z) & = & \emptyset & \text{everywhere else}
\end{array}$$

How does the $P(G)$ accept $\texttt{a+(a*a)}$?

$$\begin{array}{rcl}
(q_0,\ \texttt{a+(a*a)},\ E) & \vdash & (q_0,\ \texttt{a+(a*a)},\ E\texttt{+}T) \\
 & \vdash & (q_0,\ \texttt{a+(a*a)},\ T\texttt{+}T) \\
 & \vdash & (q_0,\ \texttt{a+(a*a)},\ F\texttt{+}T) \\
 & \vdash & (q_0,\ \texttt{a+(a*a)},\ \texttt{a+}T) \\
 & \vdash & (q_0,\ \texttt{+(a*a)},\ \texttt{+}T) \\
 & \vdash & (q_0,\ \texttt{(a*a)},\ T) \\
 & \vdash & (q_0,\ \texttt{(a*a)},\ F) \\
 & \vdash & (q_0,\ \texttt{(a*a)},\ \texttt{(}E\texttt{)}) \\
 & \vdash & (q_0,\ \texttt{a*a)},\ E\texttt{)}) \\
 & \vdash & (q_0,\ \texttt{a*a)},\ T\texttt{)}) \\
 & \vdash & (q_0,\ \texttt{a*a)},\ T\texttt{*}F\texttt{)}) \\
 & \vdash & (q_0,\ \texttt{a*a)},\ F\texttt{*}F\texttt{)}) \\
 & \vdash & (q_0,\ \texttt{a*a)},\ \texttt{a*}F\texttt{)}) \\
 & \vdash & (q_0,\ \texttt{*a)},\ \texttt{*}F\texttt{)}) \\
 & \vdash & (q_0,\ \texttt{a)},\ F\texttt{)}) \\
 & \vdash & (q_0,\ \texttt{a)},\ \texttt{a)}) \\
 & \vdash & (q_0,\ \texttt{)},\ \texttt{)}) \\
 & \vdash & (q_0,\ \epsilon,\ \epsilon)
\end{array}$$

Hence $\texttt{a+(a*a)}\in L(P(G))$.

This above example illustrates the general idea:

$$\begin{array}{rcl}
w\in L(G) & \iff & S\overset{*}{\Rightarrow}w \\
 & \iff & (q_0,w,S)\overset{*}{\vdash}(q_0,\epsilon,\epsilon) \\
 & \iff & w\in L(P(G))
\end{array}$$

The automaton we have constructed is very nondeterministic: Whenever we have a choice between different rules the automaton may silently choose one of the alternatives.

10 Recursive-Descent Parsing

10.1 What is parsing?

According to Merriam-Webster OnLine[20] *parse* means:

> To resolve (as a sentence) into component parts of speech and describe them grammatically.

In a computer science context, we take this to mean answering whether or not

$$w \in L(G)$$

for a CFG G by analysing the structure of w according to G; i.e. to *recognise* the language generated by a grammar G.

A *parser* is a program that carries out parsing. For a CFG, this amounts to a realisation of a PDA. For most practical applications, a parser will also return a structured representation of a word $w \in L(G)$. This could be the derivation (or *parse) tree* for the word, or, more commonly, a simplified version of this, a so called *Abstract Syntax Tree*. Further, for a word not in the language, $w \notin L(G)$, a practical parser would normally provide an error message explaining why. An important application of parsers is in the front-end of compilers and interpreters where they turn a text-based representation of a program into a structured representation for further analysis and translation or execution.

In this section we study how to systematically construct a parser from a given CFG using the recursive-decent parsing method. To make the discussion concrete, we implement the parsers we develop in Haskell. Thus you can easily try out the examples in this section yourself using a Haskell system such as GHCi, and it is recommended that you do so! That said, we are essentially just using a small fragment of Haskell as a notation for writing simple functions, so it should not be difficult to follow the development even if you are not very familiar with Haskell. There are plenty of Haskell resources on-line, both for learning and for downloading and installing Haskell systems[21].

10.2 Parsing strategies

There are two basic strategies for parsing: *top-down* and *bottom up*.

- A top-down parser attempts to carry out a derivation matching the input starting from the start symbol; i.e., it constructs the parse tree for the input *from the root downwards* in preorder.
- A bottom-up parser tries to construct the parse tree *from the leaves upwards* by using the productions "backwards".

Top-down parsing is, in essence, what we have been doing so far whenever we have derived some specific word in a language from the start symbol of a grammar generating that language. For example, consider the grammar:

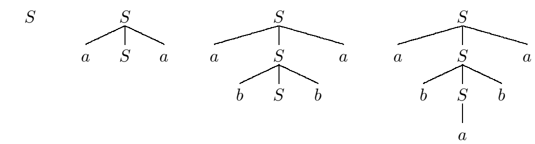
$$S \quad \rightarrow \quad aSa \mid bSb \mid a \mid b$$

[20] `http://www.webster.com`
[21] `http://www.haskell.org`, `http://learnyouahaskell.com`

If given a string *ababa*, a top-down parser for this grammar would try to derive this string from the start symbol S and thus proceed as:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow ababa$$

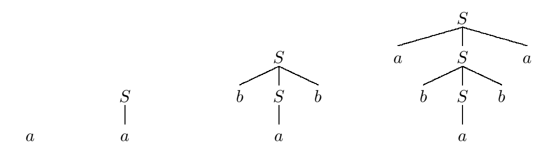
If we draw the derivation tree after each derivation step, we see how the tree gets constructed from the root downwards:



In contrast, a bottom-up parser would start from the leaves, and step by step group them together by applying productions in reverse:

$$ababa \Leftarrow abSba \Leftarrow aSa \Leftarrow S$$

This is a *rightmost derivation in reverse*. The derivation tree thus gets constructed from the bottom upwards:



The key difficulty of bottom-up parsing is to decide when to reduce. For example, in this case, how does the parser know not to reduce neither the first input symbol a nor the second input symbol b to S, but wait until it sees the middle a and only then do the first reduction step? Such questions are answered by *LR parsing theory*[22] We will not consider LR parsing further here, except noting that it is covered in more depth in the Compilers module [Nil16] that builds on much of the material in this module. Instead we turn our attention to recursive-decent parsing, which is a type of top-down parsing.

10.3 Basics of recursive-descent parsing

Recursive-descent parsing is a way to implement top-down parsing. We are just going to focus on the language recognition problem: $w \in L(G)$? This suggests the following type for the parser:

```
parser :: [Token] -> Bool
```

Token is "compiler speak" for (input) symbol; i.e., an element of the alphabet.
Consider a typical production in some CFG G:

$$S \quad \rightarrow \quad AB$$

[22] `https://en.wikipedia.org/wiki/LR_parser`

Let $L(X)$ be the language $\{w \in T^* \mid X \overset{*}{\underset{G}{\Rightarrow}} w\}$, $X \in N$. Note that

$$w \in L(S) \Leftarrow \exists w_1, w_2 \,.\; \begin{array}{l} w = w_1 w_2 \\ \wedge\, w_1 \in L(A) \\ \wedge\, w_2 \in L(B) \end{array}$$

That is, given a parser for $L(A)$ and a parser for $L(B)$, we can construct a parser for $L(S)$ by asking the first parser if a prefix w_1 of w belongs to $L(A)$, and then asking the other parser if the remaining suffix w_2 of w belongs to $L(B)$. If the answer two both questions is yes, then w belongs to $L(S)$.

However, we need to find the right way to divide the input word w! In general, there are $|w| + 1$ possibilities. We could, of course, blindly try them all. But as the prefix and suffix recursively also have to be split in all possible ways, and so on until we get down to individual akphabet symbols, it is clear that this approach would lead to a combinatorial explosion that would render such a parser useless for all but very short words.

Instead we need to let the input guide the search. To that end, we initially adopt the following idea:

- Each parser tries to derive a *prefix* of the input according to the productions for the nonterminal
- Each parser returns the remaining *suffix* if successful, allowing this to be passed to the next parser for analysis.

This gives us the following refined type for parsers:

```
parseX :: [Token] -> Maybe [Token]
```

Recall that `Maybe` is Haskell's option type:

```
data Maybe a = Nothing | Just a
```

Of course, we should be a little suspicious: There could be *more* than one prefix derivable from a non-terminal, and if so, how can we then know which *one* to pick? Picking the *wrong* prefix might make it impossible to derive the suffix from the non-terminal that follows. We will return to these points later.

Now we can construct a parser for $L(S)$

$$S \to AB$$

in terms of parsers for $L(A)$ and $L(B)$:

```
parseS :: [Token] -> Maybe [Token]
parseS ts =
    case parseA ts of
        Nothing  -> Nothing
        Just ts' ->
            case parseB ts' of
                Nothing   -> Nothing
                Just ts'' -> Just ts''
```

Note that the case analysis on the result of `parseB ts'` simply passes on the result unchanged. Thus we can simplify the code to:


```
parseS :: [Token] -> Maybe [Token]
parseS ts =
    case parseA ts of
        Nothing  -> Nothing
        Just ts' -> parseB ts'
```

This approach is called recursive-descent parsing because the parse functions (usually) end up being (mutually) recursive. What does this have to do with realising a PDA? Fundamental to the implementation of a recursive computation is a that keeps track of the *state* of the computation and allows for *subcomputations* (to any depth). In a language that supports recursive functions and procedures, the stack is usually not explicitly visible, but internally, it is a central datastructure. Thus, a recursive-descent parser is a kind of PDA.

Let us develop this example a little further into code that can be executed. First, for simplicity, let us pick the type `Char` for token:

```
type Token = Char
```

Recall that (basic) strings are just lists of characters in Haskell; that is, `String` = `[Char]` = `[Token]`. Thus, a string literal like `"abcd"` is just a shorthand notation for the list of characters `['a','b','c','d']`.

Now, suppose the productions for A and B are the following

$$\begin{array}{lcl} A & \to & a \\ B & \to & b \end{array}$$

Thus, in this case, it is clear that if the parsing function `parseA` for the nonterminal A sees input starting with an a, then that a is the desired prefix, and whatever remains of the input is the suffix that should be returned as part if the indication of having been able to successfully derive a prefix of the input from the nonterminal in question. Otherwise, if the input does not start with an a, it is equally clear that it is not possible to derive any prefix of the input from the nonterminal A, and the parsing function must thus indicate failure. In essence, this is how the input is used to guide the search for the right prefix.

We can implement `parseA` in Haskell, using pattern matching, as follows:

```
parseA :: [Token] -> Maybe [Token]
parseA ('a' : ts) = Just ts
parseA _          = Nothing
```

The case and code for the parsing function `parseB` for the nonterminal B is of course analogous:

```
parseB :: [Token] -> Maybe [Token]
parseB ('b' : ts) = Just ts
parseB _          = Nothing
```

Now we can evaluate parseA, `parseB`, and `parseS` on `"abcd"` with the following results:

```
parseA "abcd"  ⇒  Just "bcd"
parseB "abcd"  ⇒  Nothing
parseS "abcd"  ⇒  Just "cd"
```

This tells us that a prefix of *abcd* can be derived from A, leaving a remaining suffix *bcd*, that no prefix of *abcd* can be derived from B, but that a prefix of *abcd* also can be derived from S, leaving a suffix *cd*, as we would expect.

10.4 Handling choice

Of course, there are usually more than one one production for a nonterminal. Thus need a way to handle *choice*, as in

$$S \to AB \mid CD$$

We are first going to consider the case when the choice is obvious, as in

$$S \to aAB \mid cCD$$

That is, we assume it is manifest from the grammar that we can choose between productions with a one-symbol *lookahead*.

As an example, let us construct a parser for the grammar:

$$\begin{array}{rcl} S & \to & aA \mid bBA \\ A & \to & aA \mid \epsilon \\ B & \to & bB \mid \epsilon \end{array}$$

We are going to need one parsing function for each non-terminal:

- `parseS :: [Token] -> Maybe [Token]`
- `parseA :: [Token] -> Maybe [Token]`
- `parseB :: [Token] -> Maybe [Token]`

We again take `type Token = Char` for simplicity.

Code for `parseS`. Note how the pattern matching makes use of a one-symbol lookahead to chose between the two productions for S:

```
parseS :: [Token] -> Maybe [Token]
parseS ('a' : ts) =
    parseA ts
parseS ('b' : ts) =
    case parseB ts of
        Nothing  -> Nothing
        Just ts' -> parseA ts'
parseS _ = Nothing
```

The code for `parseA` and `parseB` similarly make use of the one-symbol lookahead to chose between productions, but this time, because $A \Rightarrow \epsilon$ and $B \Rightarrow \epsilon$, it is *not* a syntax error if the next token is not, respectively, a and b. Thus both functions can succeed *without* consuming any input:

```
parseA :: [Token] -> Maybe [Token]
parseA ('a' : ts) = parseA ts
parseA ts         = Just ts

parseB :: [Token] -> Maybe [Token]
parseB ('b' : ts) = parseB ts
parseB ts         = Just ts
```


Now consider a more challenging scenario:

$$\begin{array}{rcl} S & \to & aA \mid aBA \\ A & \to & aA \mid \epsilon \\ B & \to & bB \mid \epsilon \end{array}$$

In the parsing function `parseS` for nonterminal S, should `parseA` or `parseB` be called once a has been read?

We could try the alternatives in order; i.e., a limited form of *backtracking*:

```
parseS ('a' : ts) =
    case parseA ts of
        Just ts' -> Just ts'
        Nothing ->
            case parseB ts of
                Nothing  -> Nothing
                Just ts' -> parseA ts'
```

Of course, the choice to try `parseA` first is arbitrary, a point we will revisit shortly.

Similarly, there are two alternatives for the nonterminal A. In fact, we already encountered this situation above, and as we did there, let us try to consume an input prefix (here a) if possible, and only if that is not possible succeed without consuming any input:

```
parseA :: [Token] -> Maybe [Token]
parseA ('a' : ts) = parseA ts
parseA ts         = Just ts
```

The code for B is of course similar. This may seem like an obvious ordering: after all, if we opted to "try" without consuming any input first, then that would always succeed, and no other alternatives would ever be tried. Nevertheless, picking the order we did still amounts to an arbitrary choice, and in fact it is *not* always the right one.

The problem here is that limited backtracking is *not* an exhaustive search. For many grammars, there simply is no one order that *always* will work, meaning that a parser that nevertheless commits to one particular order is liable to get stuck in "blind alleys".

Consider the following grammar

$$\begin{array}{rcl} S & \to & AB \\ A & \to & aA \mid \epsilon \\ B & \to & ab \end{array}$$

and corresponding parsing functions:

```
parseA ('a' : ts) = parseA ts
parseA ts         = Just ts

parseB ('a' : 'b' : ts) = Just ts
parseB ts               = Nothing
```

```
parseS ts =
    case parseA ts of
        Nothing  -> Nothing
        Just ts' -> parseB ts'
```

Will it work? Let us try it on ab. Clearly derivable from the grammar:

$$S \Rightarrow AB \Rightarrow B \Rightarrow ab$$

However, if we run our parser on `"ab"`:

$$\texttt{parseS "ab"} \;\Rightarrow\; \texttt{Nothing}$$

Our parser thus says "no". Why? Because

$$\texttt{parseA "ab"} \;\Rightarrow\; \texttt{Just "b"}$$

That is, the code for `parseA` committed to the choice $A \to a$ too early and will never try $A \to \epsilon$. That was the wrong choice in this case and the parser got stuck in a "blind alley".

Would it have been better to try $A \to \epsilon$ first? Then the parser would work for the word ab, but it would still fail on other words that should be accepted, such as aab. To successfully parse that word, `parseA` must somehow consume the first a but not the second, and neither ordering of the productions for A will achieve that.

One principled approach addressing this dilemma is to try *all* alternatives; i.e., *full backtracking* (aka *list of successes*):

- Each parsing function returns a *list* of *all* possible suffixes. Type:

  ```
  parseX :: [Token] -> [[Token]]
  ```

- Translate $A \to \alpha \mid \beta$ into

  ```
  parseA ts = parseAlpha ts ++ parseBeta ts
  ```

- An empty list indicates no possible parsing.

However:

- Full backtracking is computationally expensive.
- In error reporting, it becomes difficult to pinpoint the exact location of a syntax error: where exactly lies the problem if it only *after* an exhaustive search becomes apparent that there is no possible way to parse a word?

In section 10.6, we are going to look at another principled approach that avoids backtracking: *predictive parsing*. The price we have to pay is that the grammar must satisfy certain conditions. But at least we will know *statically, at construction time* if the parser is going to work or not. And if not, we can try to modify the grammar (without changing the language) until the prerequisite conditions are met. First, however, we will consider the problem of left-recursion and context-free grammars.

10.5 Recursive-descent parsing and left-recursion

Consider the grammar

$$A \;\to\; Aa \mid \epsilon$$

and the corresponding recursive-descent parsing function:

```
parseA :: [Token] -> Maybe [Token]
parseA ts =
    case parseA ts of
        Just ('a' : ts') -> Just ts'
        _                -> Just ts
```

Any problem? Yes, because the function calls itself without consuming any input, it will loop forever.

The problem here is that the grammar is left-recursive. Recall that a this means that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some nonterminal A. As each derivation step corresponds to one parsing function calling another, it is clear that recursive-descent parsing functions derived from a left-recursive grammar will end up looping forever as soon as one of the parsing functions for a left-recursive non-terminal is invoked because no input is consumed before the same function is entered again, directly or indirectly.

Recursive-descent parsers thus *cannot*[23] deal with *left-recursive* grammars. The standard way of resolving this is to transform a left-recursive grammar into an equivalent grammar that is not left recursive as described in section 8.6, and then deriving the parser from the non-left-recursive version of the grammar.

10.6 Predictive parsing

In a recursive-decent parsing setting, we want a parsing function to be successful *exactly* when a prefix of the input *can* be derived from the corresponding nonterminal. This can be achieved by:

- Adopting a suitable parsing strategy, specifically regarding how to handle *choice* between two or more productions for one nonterminal.
- Impose *restrictions* on the grammar to ensure success of the chosen parsing strategy.

Predictive parsing is when all parsing decisions can be made based on a *lookahead* of limited length, typically one symbol. We have already seen cases where predictive parsing clearly is possible; for example, this is manifestly the case when the right-hand side of each possible production starts with a distinct terminal, as here:

$$S \;\to\; aB \;\mid\; cD$$

But we also saw that the choice is not always this obvious, and that if we then make arbitrary choices regarding which order in which to try productions, the resulting parser is likely to be flawed. In the following, we are going to

[23] At least not when implemented in the standard way described here. It is possible, by keeping track of more contextual information, to detect when no progress is being made and limit the recursion depth at that point. See https://en.wikipedia.org/wiki/Top-down_parsing.

look into *exactly* when the next input symbol suffices to make all choices. As a consequence, if we are faced with a grammar were a one-symbol lookahead is not enough, we will know this, and we can take corrective action, such as trying to transform the grammatr in a way that will reolve the problem.

Before we start, let us just give an example that illustrates that a one-symbol lookahead can be enough even if the RHSs start with nonterminals:

$$\begin{array}{rcl} S & \rightarrow & AB \mid CD \\ A & \rightarrow & a \mid b \\ C & \rightarrow & c \mid d \end{array}$$

Here, if the input starts with an a or b, we should clearly attempt to parse by the production $S \rightarrow AB$, and if it starts with a c or a d, we should attempt to parse by $A \rightarrow CD$. This suggests that the key is going to be an *analysis* of the grammar: for each nonterminal, we need to know what symbols that may start words derived from that nonterminal.

More generally, consider productions for a nonterminal X

$$X \quad \rightarrow \quad \alpha \ \mid \ \beta$$

and the corresponding parsing code:

```
parseX (t : ts) =
    | t ??      -> parse α
    | t ??      -> parse β
    | otherwise -> Nothing
```

The questions is, what should the conditions be on the lookahead symbol `t`, here indicated by ??, to decide whether or not to parse by the production corresponding to each case?

The idea of predictive parsing is this:

- Compute the *set* of terminal symbols that can *start* strings derived from each alternative, the *first set*.
- If there is a choice between two or more alternatives, insist that the first sets for those are *disjoint* (a grammar restriction).
- The right choice can now be made simply by determining to which alternative's first set the next input symbol belongs.

We can now refine the code to:

```
parseX (t : ts) =
    | t ∈  first(α) -> parse α
    | t ∈  first(β) -> parse β
    | otherwise  -> Nothing
```

But the situation could be a bit more involved as it sometimes is possible to derive the empty word from a nonterminal, and the empty word does of course not begin with any symbol at all. For a concrete example, consider again $X \rightarrow \alpha \mid \beta$, and suppose it can be the case that $\beta \stackrel{*}{\Rightarrow} \epsilon$.

Clearly, the next input symbol could in this case be a terminal that can *follow* a string derivable form X, meaning we need to refine the parsing code further:


```
parseX (t : ts) =
    | t ∈  first(α)              -> parse α
    | t ∈  first(β)  ∪  follow(X) -> parse β
    | otherwise                  -> Nothing
```

Of course, the branches must be mutually exclusive! Otherwise a one-symbol lookahead is not enough to decide which choice to make.

10.6.1 First and follow sets

We will now develop tease ideas in more detail. The presentation roughly follows "The Dragon Book" [ASU86]. For a CFG $G = (N,\ T,\ P,\ S)$:

$$\begin{array}{rcl} \text{first}(\alpha) & = & \{a \in T \mid \alpha \underset{G}{\stackrel{*}{\Rightarrow}} a\beta\} \\ \text{follow}(A) & = & \{a \in T \mid S \underset{G}{\stackrel{*}{\Rightarrow}} \alpha A a \beta\} \\ & & \cup \{\$ \mid S \underset{G}{\stackrel{*}{\Rightarrow}} \alpha A\} \end{array}$$

where $\alpha,\ \beta \in (N \cup T)^*$, $A \in N$, and where \$ is a special "end of input" marker.

To illustrate these definitions, consider the grammar:

$$\begin{array}{rclcrcl} S & \rightarrow & ABC & \qquad & B & \rightarrow & b \mid \epsilon \\ A & \rightarrow & aA \mid \epsilon & & C & \rightarrow & c \mid d \end{array}$$

First sets:

$$\begin{array}{rcl} \text{first}(C) & = & \{c,\ d\} \\ \text{first}(B) & = & \{b\} \\ \text{first}(A) & = & \{a\} \\ \text{first}(S) & = & \text{first}(ABC) \\ & = & [\text{because } A \stackrel{*}{\Rightarrow} \epsilon \text{ and } B \stackrel{*}{\Rightarrow} \epsilon] \\ & & \text{first}(A) \cup \text{first}(B) \cup \text{first}(C) \\ & = & \{a,\ b,\ c,\ d\} \end{array}$$

Follow sets:

$$\begin{array}{rcl} \text{follow}(C) & = & \{\$\} \\ \text{follow}(B) & = & \text{first}(C) = \{c,\ d\} \\ \text{follow}(A) & = & [\text{because } B \stackrel{*}{\Rightarrow} \epsilon] \\ & & \text{first}(B) \cup \text{first}(C) \\ & = & \{b,\ c,\ d\} \end{array}$$

10.6.2 LL(1) grammars

Now consider *all* productions for a nonterminal A in some grammar:

$$A \quad \rightarrow \quad \alpha_1 \mid \alpha_2 \mid \ \ldots \ \mid \alpha_n$$

In the parsing function for A, on input symbol t, we should parse according to α_i if

- $t \in \text{first}(\alpha_i)$.
- $t \in \text{follow}(A)$, if $\alpha_i \stackrel{*}{\Rightarrow} \epsilon$

Thus, if:

- $\text{first}(\alpha_i) \cap \text{first}(\alpha_j) = \emptyset$ for $1 \leq i < j \leq n$, and
- if $\alpha_i \stackrel{*}{\Rightarrow} \epsilon$ for some i, then, for all $1 \leq j \leq n$, $j \neq i$,
  - $\alpha_j \stackrel{*}{\not\Rightarrow} \epsilon$, and
  - $\text{follow}(A) \cap \text{first}(\alpha_j) = \emptyset$

then it is always clear what do do! A grammar satisfying these conditions is said to be an *LL(1)* grammar.

10.6.3 Nullable nonterminals

In order to compute the first and follow sets for a grammar $G = (N,\ T,\ P,\ S)$, we first need to know all nonterminals $A \in N$ such that $A \stackrel{*}{\Rightarrow} \epsilon$; i.e. the set $N_\epsilon \subseteq N$ of *nullable* nonterminals.

Let $\text{syms}(\alpha)$ denote the *set* of symbols in a string α:

$$\begin{array}{rcl} \text{syms} & \in & (N \cup T)^* \to \mathcal{P}(N \cup T) \\ \text{syms}(\epsilon) & = & \emptyset \\ \text{syms}(X\alpha) & = & \{X\} \cup \text{syms}(\alpha) \end{array}$$

The set N_ϵ is the *smallest* solution to the equation

$$N_\epsilon \quad = \quad \{A \mid A \to \alpha \in P \ \wedge\ \forall X \in \text{syms}(\alpha)\ .\ X \in N_\epsilon\}$$

Note that $A \in N_\epsilon$ if $A \to \epsilon \in P$ because $\text{syms}(\epsilon) = \emptyset$ and $\forall X \in \emptyset\ .\ \ldots$ is trivially true. Also note that we really need to look for the *smallest* solution. For example, consider a grammar $S \to SS \mid a$. $N_\epsilon = \{S\}$ is clearly a solution to the equation defining N_ϵ for this grammar, but S is also clearly not nullable. That is because $N_\epsilon = \{S\}$ is *not* the smallest solution. The smallest solution in this case is $N_\epsilon = \emptyset$; i.e. there are no nullable nonterminals.

We can now define a predicate nullable on *strings* of grammar symbols:

$$\begin{array}{rcl} \text{nullable} & \in & (N \cup T)^* \to \text{Bool} \\ \text{nullable}(\epsilon) & = & \text{true} \\ \text{nullable}(X\alpha) & = & X \in N_\epsilon \wedge \text{nullable}(\alpha) \end{array}$$

The equation for N_ϵ can be solved iteratively as follows:

1. Initialize N_ϵ to $\{A \mid A \to \epsilon \in P\}$.
2. If there is a production $A \to \alpha$ such that $\forall X \in \text{syms}(\alpha)\ .\ X \in N_\epsilon$, then add A to N_ϵ.
3. Repeat step 2 until no further nullable nonterminals can be found.

Consider the following grammar:

$$\begin{array}{rclcrcl} S & \to & ABC \mid AB & \qquad & B & \to & b \mid \epsilon \\ A & \to & aA \mid BB & & C & \to & c \mid d \end{array}$$

- Because $B \to \epsilon$ is a production, $B \in N_\epsilon$.
- Because $A \to BB$ is a production and $B \in N_\epsilon$, additionally $A \in N_\epsilon$.
- Because $S \to AB$ is a production, and $A, B \in N_\epsilon$, additionally $S \in N_\epsilon$.
- No more production with nullable RHSs. The set of nullable symbols $N_\epsilon = \{S, A, B\}$.

10.6.4 Computing first sets

For a CFG $G = (N,\ T,\ P,\ S)$, the sets $\text{first}(A)$ for $A \in N$ are the smallest sets satisfying:

$$\begin{array}{rcl} \text{first}(A) & \subseteq & T \\ \text{first}(A) & = & \displaystyle\bigcup_{A \to \alpha\ \in\ P} \text{first}(\alpha) \end{array}$$

For strings, first is defined as (note the *overloaded* notation):

$$\begin{array}{rcl} \text{first} & \in & (N \cup T)^* \to \mathcal{P}(T) \\ \text{first}(\epsilon) & = & \emptyset \\ \text{first}(a\alpha) & = & \{a\} \\ \text{first}(A\alpha) & = & \text{first}(A) \cup \begin{cases} \text{first}(\alpha), & \text{if } A \in N_\epsilon \\ \emptyset, & \text{if } A \notin N_\epsilon \end{cases} \end{array}$$

where $a \in T$, $A \in N$, and $\alpha \in (N \cup T)^*$.

The solutions can often be obtained directly by expanding out all definitions. If necessary, the equations can be solved by iteration in a similar way to how N_ϵ is computed. Note that the smallest solution to a set equation of the type $X = X \cup Y$ when there are no other constraints on X is simply $X = Y$.

Consider (again):

$$\begin{array}{rclcrcl} S & \to & ABC & \qquad & B & \to & b \mid \epsilon \\ A & \to & aA \mid \epsilon & & C & \to & c \mid d \end{array}$$

First compute the nullable nonterminals: $N_\epsilon = \{A, B\}$. Then the first sets:

$$
\begin{aligned}
\text{first}(A) &= \text{first}(aA) \cup \text{first}(\epsilon) \\
&= \{a\} \cup \emptyset = \{a\} \\
\text{first}(B) &= \text{first}(b) \cup \text{first}(\epsilon) \\
&= \{b\} \cup \emptyset = \{b\} \\
\text{first}(C) &= \text{first}(c) \cup \text{first}(d) \\
&= \{c\} \cup \{d\} = \{c, d\} \\
\text{first}(S) &= \text{first}(ABC) \\
&= [A \in N_\epsilon] \\
&\quad\ \text{first}(A) \cup \text{first}(BC) \\
&= [B \in N_\epsilon \wedge C \notin N_\epsilon] \\
&\quad\ \text{first}(A) \cup \text{first}(B) \cup \text{first}(C) \cup \emptyset \\
&= \{a\} \cup \{b\} \cup \{c, d\} = \{a, b, c, d\}
\end{aligned}
$$

10.6.5 Computing follow sets

For a CFG $G = (N,\ T,\ P,\ S)$, the sets follow(A) are the smallest sets satisfying:

- $\{\$\} \subseteq \text{follow}(S)$
- If $A \to \alpha B \beta\ \in\ P$, then $\text{first}(\beta) \subseteq \text{follow}(B)$
- If $A \to \alpha B \beta\ \in\ P$, and nullable$(\beta)$ then $\text{follow}(A) \subseteq \text{follow}(B)$

$A, B \in N$, and $\alpha, \beta \in (N \cup T)^*$.
(It is assumed that there are no *useless* symbols; i.e., all symbols can appear in the derivation of some sentence.)

Here is our example grammar again:

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
S & \to & ABC & B & \to & b \mid \epsilon \\
A & \to & aA \mid \epsilon & C & \to & c \mid d
\end{array}
$$

Constraints for follow(S):

$$\{\$\} \quad \subseteq \quad \text{follow}(S)$$

Constraints for follow(A) (note: $\neg$nullable(BC)):

$$
\begin{aligned}
\text{first}(BC) &\subseteq \text{follow}(A) \\
\text{first}(\epsilon) &\subseteq \text{follow}(A) \\
\text{follow}(A) &\subseteq \text{follow}(A)
\end{aligned}
$$

Constraints for follow(B) (note: $\neg$nullable(C)):

$$\text{first}(C) \quad \subseteq \quad \text{follow}(B)$$

Constraints for follow(C) (note: nullable(ϵ)):

$$
\begin{aligned}
\text{first}(\epsilon) &\subseteq \text{follow}(C) \\
\text{follow}(S) &\subseteq \text{follow}(C)
\end{aligned}
$$

In general:

$$X \subseteq Z\ \wedge\ Y \subseteq Z \quad \Longleftrightarrow \quad X \cup Y \subseteq Z$$

Also, constraints like $\emptyset \subseteq X$ and $X \subseteq X$ are trivially satisfied and can be omitted.

The constraints for our example can thus be written as:

$$
\begin{aligned}
\{\$\} &\subseteq \text{follow}(S) \\
\text{first}(BC) \cup \text{first}(\epsilon) &\subseteq \text{follow}(A) \\
\text{first}(C) &\subseteq \text{follow}(B) \\
\text{first}(\epsilon) \cup \text{follow}(S) &\subseteq \text{follow}(C)
\end{aligned}
$$

Using

$$
\begin{aligned}
\text{first}(\epsilon) &= \emptyset \\
\text{first}(C) &= \{c, d\} \\
\text{first}(BC) &= \text{first}(B) \cup \text{first}(C) \cup \emptyset \\
&= \{b\} \cup \{c, d\} = \{b, c, d\}
\end{aligned}
$$

the constraints can be simplified further:

$$
\begin{aligned}
\{\$\} &\subseteq \text{follow}(S) \\
\{b, c, d\} &\subseteq \text{follow}(A) \\
\{c, d\} &\subseteq \text{follow}(B) \\
\text{follow}(S) &\subseteq \text{follow}(C)
\end{aligned}
$$

Finally, looking for the smallest sets satisfying these constraints, we get:

$$
\begin{aligned}
\text{follow}(S) &= \{\$\} \\
\text{follow}(A) &= \{b, c, d\} \\
\text{follow}(B) &= \{c, d\} \\
\text{follow}(C) &= \text{follow}(S) = \{\$\}
\end{aligned}
$$

10.6.6 Implementing a predictive parser

Let us now implement a predictive parser for our sample grammar:

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
S & \to & ABC & B & \to & b \mid \epsilon \\
A & \to & aA \mid \epsilon & C & \to & c \mid d
\end{array}
$$

For this grammar, as per the calculations above:

- Nullable symbols: $N_\epsilon = \{S, A, B\}$
- First sets:

$$
\begin{aligned}
\text{first}(S) &= \{a, b, c, d\} \\
\text{first}(A) &= \{a\} \\
\text{first}(B) &= \{b\} \\
\text{first}(C) &= \{c, d\}
\end{aligned}
$$

- Follow sets:

$$\begin{aligned}\text{follow}(S) &= \{\$\}\\ \text{follow}(A) &= \{b, c, d\}\\ \text{follow}(B) &= \{c, d\}\\ \text{follow}(C) &= \{\$\}\end{aligned}$$

Recall the “template code” for a parsing function for a pair of productions like $X \to \alpha \mid \beta$. If no RHS is nullable:

```
parseX (t : ts) =
    | t ∈ first(α) -> parse α
    | t ∈ first(β) -> parse β
    | otherwise  -> Nothing
```

If one RHS is nullable, say nullable(β):

```
parseX (t : ts) =
    | t ∈ first(α)              -> parse α
    | t ∈ first(β) ∪ follow(X) -> parse β
    | otherwise                 -> Nothing
```

We can now implement predictive parsing functions for the nonterminals S, A, B, and C as follows in pseudo Haskell. The function for S is straightforward as there is only one production for S, thus no choice:

```
parseS ts =
    case parseA ts of
        Just ts' ->
            case parseB ts' of
                Just ts'' ->
                    parseC ts''
                Nothing ->
                    Nothing
        Noting ->
            Nothing
```

For A, there are two productions: $A \to aA \mid \epsilon$. Note that aA is trivially not nullable, while ϵ trivially is nullable. We thus compute first(aA) $= \{a\}$ and first(ϵ) $\cup$ follow(A) $= \emptyset \cup \{b, c, d\} = \{b, c, d\}$. The parsing function for A thus becomes:

```
parseA (t : ts) =
    | t ∈ {a}     -> parseA ts
    | t ∈ {b,c,d} -> Just (t : ts)
    | otherwise   -> Nothing
```

Note how the case for the ϵ-production only does checking on the next input symbol, but does not consume it.

For B, there are also two productions: $B \to b \mid \epsilon$. Note that b is trivially not nullable, while ϵ trivially is nullable. We thus compute first(b) $= \{b\}$ and first(ϵ) $\cup$ follow(B) $= \emptyset \cup \{c, d\} = \{c, d\}$. The resulting parsing function for B:


```
parseB (t : ts) =
    | t ∈ {b}   -> Just ts
    | t ∈ {c,d} -> Just (t : ts)
    | otherwise -> Nothing
```

Note how the case for b checks that the next input indeed is a b and if so succeeds, consuming that one input symbol, while the ϵ-production again only checks the next input symbol without consuming it.

Finally, the productions for C are $C \to c \mid d$, where both RHSs are trivially not nullable. We compute first(c) $= \{c\}$ and first(d) $= \{d\}$. The parsing function for C:

```
parseC (t : ts) =
    | t ∈ {c}  -> Just ts
    | t ∈ {d}  -> Just ts
    | otherwise -> Nothing
```

This can of course be simplified a little, which is the case whenever there are multiple productions with the RHSs being a single terminal:

```
parseC (t : ts) =
    | t ∈ {c,d} -> Just ts
    | otherwise -> Nothing
```

10.6.7 LL(1), left-recursion, and ambiguity

As we have seen, the LL(1) conditions impose a number of restrictions on a grammar. In particular, no left-recursive or ambiguous grammar can be LL(1)!

Let us prove that a left-recursive grammar cannot be LL(1). Recall that a grammar is left-recursive iff there exists $A \in N$ and $\alpha \in (N \cup T)^*$ such that $A \stackrel{+}{\Rightarrow} A\alpha$ (section 8.6). We can assume without loss of generality that there are no useless symbols and productions in the grammar as any useless productions can be removed from a grammar without changing the language (section 8.2). It thus follows that a derivation $A \stackrel{+}{\Rightarrow} w$, $w \in T^*$, must also exist.

Let us assume that all derivations are leftmost. Clearly, $A\alpha \neq w$, and thus there must have been a choice at some point differentiating these two derivations. That is, there must exist some $B \in N$ for which there are at least two *distinct* productions $B \to \beta_1 \mid \beta_2$ such that

$$A \stackrel{*}{\Rightarrow} B\gamma \Rightarrow \beta_1\gamma \stackrel{*}{\Rightarrow} A\alpha$$

and

$$A \stackrel{*}{\Rightarrow} B\gamma \Rightarrow \beta_2\gamma \stackrel{*}{\Rightarrow} w$$

Let us now observe that if there is a derivation $\alpha \stackrel{*}{\Rightarrow} \beta$, then first($\alpha$) $\supseteq$ first(β). Let us also observe that if $\neg$nullable(α), then first($\alpha\beta$) $=$ first(α), and if nullable(α), then first($\alpha\beta$) $=$ first(α)$\cup$first(β). (These should really be proved as auxiliary lemmas, but they are fairly obvious.)

Now let us consider β_1 and β_2. If both nullable(β_1) and nullable(β_2), then that is an immediate violation of the LL(1) conditions, so we need not consider that case further. Moreover, we can assume $w \neq \epsilon$: if the only terminal string derivable from A is ϵ, then, under the assumption of no useless productions, it must be the case that both nullable(β_1) and nullable(β_2) which again violates the LL(1) conditions. Thus, because $A \stackrel{+}{\Rightarrow} w$ and $w \neq \epsilon$, we have:

$$\text{first}(A) \supseteq \text{first}(w) \neq \emptyset$$

Suppose $\neg$nullable(β_1) and $\neg$nullable(β_2). Because $\neg$nullable(β_1), $\beta_1\gamma \stackrel{*}{\Rightarrow} A\alpha$, and first($A\alpha$) $\supseteq$ first(A) by definition, we have:

$$\text{first}(\beta_1) = \text{first}(\beta_1\gamma) \supseteq \text{first}(A\alpha) \supseteq \text{first}(A) \supseteq \text{first}(w)$$

Because $\neg$nullable(β_2) and $\beta_2\gamma \stackrel{*}{\Rightarrow} w$, we have:

$$\text{first}(\beta_2) = \text{first}(\beta_2\gamma) \supseteq \text{first}(w)$$

Thus

$$\text{first}(\beta_1) \cap \text{first}(\beta_2) \supseteq \text{first}(w) \neq \emptyset$$

This proves that the intersection between the first sets of the RHSs of the two productions $B \to \beta_1 \mid \beta_2$ is nonempty, and we have a violation of the LL(1) conditions.

Suppose nullable(β_1) and $\neg$nullable(β_2). The LL(1) conditions now require first(β_1) $\cup$ follow(B) and first(β_2) to be disjoint. Becasue $A \stackrel{*}{\Rightarrow} B\gamma$ (assuming no uselss symbols or productions), follow(B) $\supseteq$ first(γ). It follows that

$$\text{first}(\beta_1) \cup \text{follow}(B) \supseteq \text{first}(\beta_1) \cup \text{first}(\gamma) = \text{first}(\beta_1\gamma)$$

But then, because $\beta_1\gamma \stackrel{*}{\Rightarrow} A\alpha$ we have

$$\text{first}(\beta_1\gamma) \supseteq \text{first}(A\alpha) \supseteq \text{first}(A) \supseteq \text{first}(w)$$

Thus, first(β_1) $\cup$ follow(B) $\supseteq$ first(w). But as before, as $\neg$nullable(β_2), we have first(β_2) $\supseteq$ first(w). As first(w) $\neq \emptyset$, we can conclude that these two sets are not disjoint, and therefore the LL(1) conditions are violated.

Suppose instead $\neg$nullable(β_1) and nullable(β_2). The LL(1) conditions now require first(β_1) and first(β_2) $\cup$ follow(B) to be disjoint. Again, follow(B) $\supseteq$ first(γ). It follows that

$$\text{first}(\beta_2) \cup \text{follow}(B) \supseteq \text{first}(\beta_2) \cup \text{first}(\gamma) = \text{first}(\beta_2\gamma)$$

Then, because $\beta_2\gamma \stackrel{*}{\Rightarrow} w$ we have

$$\text{first}(\beta_1\gamma) \supseteq \text{first}(w)$$

Thus, first(β_2)$\cup$follow(B) $\supseteq$ first(w). But given $\neg$nullable(β_1), we have first(β_1) $\supseteq$ first(w). As first(w) $\neq \emptyset$, we can again conclude that these two sets are not disjoint, and therefore the LL(1) conditions are violated.

Thus, no left-recursive grammar can satisfy the LL(1) conditions, which means that a left-recursive grammar first must be transformed into an equivalent grammar that is not left-recursive if we wish to develop an LL(1) parser for the language described by the grammar.

Let us now prove that no ambiguous grammar can be LL(1). Recall that a grammar is ambiguous if a single word w can be derived in two (or more) essentially different ways, for example there exist two different leftmost derivations for w.

Assume that a given grammar is ambiguous and pick two different leftmost derivations for some word w in the langauge of the grammar. Consider the first place where these derivations differ:

$$S \stackrel{*}{\Rightarrow} a_1 \dots a_i A\alpha \Rightarrow a_1 \dots a_i \beta_1 \alpha \stackrel{*}{\Rightarrow} a_1 \dots a_i a_{i+1} \dots a_n = w$$

and

$$S \stackrel{*}{\Rightarrow} a_1 \dots a_i A\alpha \Rightarrow a_1 \dots a_i \beta_2 \alpha \stackrel{*}{\Rightarrow} a_1 \dots a_i a_{i+1} \dots a_n = w$$

Thus there are two productions $A \to \beta_1 \mid \beta_2$ in the grammar. Assuming $a_{i+1} \dots a_n \neq \epsilon$, we have the following possibilities:

- $a_{i+1} \in \text{first}(beta_1)$ and $a_{i+1} \in \text{first}(beta_2)$, meaning the LL(1) conditions are violated;
- One of β_1 or β_2 derives ϵ, implying $a_{i+1} \in \text{follow}(A)$ and $a_{i+1} \in \text{first}(beta_1)$ or $a_{i+1} \in \text{first}(beta_2)$, meaning the LL(1) conditions are violated either way;
- Both β_1 or β_2 derive ϵ, a direct violation of the LL(1) conditions.

Assuming $a_{i+1} \dots a_n = \epsilon$, then it must be the case that both $\beta 1$ and $\beta 2$ are nullable, which violates the LL(1) conditions.

Thus, no ambiguous grammar can satisfy the LL(1) conditions, which means that an ambiguous grammar first must be transformed into an equivalent unambiguous grammar if we wish to develop an LL(1) parser for the described language.

10.6.8 Satisfying the LL(1) conditions

Not all grammars satisfy the LL(1) conditions. If we have such a grammar, and we wish to develop an LL(1) parser, the grammar first has to be transformed. In particular, left-recursion and ambiguity must be eliminated because the LL(1) conditions are necessarily violated otherwise (see section 10.6.7). There is no point in computing first and follow sets (for the purpose of constructing a LL(1) parser) before this is done. Of course, transforming a grammar in this way is not always possible: the grammar may be inherently ambiguous, for example (section 8.5). But often transformation into an equivalent grammar satisfying the LL(1) conditions is possible.

Section 8 covered a number of transformations, including ones for eliminating left recursion and disambiguating grammars. Sometimes other transformations are needed. A common problem is the following. Note that the productions are not left recursive, nor would these rules in isolation make a grammar ambiguous as one of the rules derive a word with the terminal c, and the other one without the terminal c:

$$S \quad \to \quad aXbY \mid aXbYcZ$$

This grammar is clearly not suitable for predictive parsing as the first sets for the RHSs of both productions are the same, $\{a\}$. But it is also clear that the problem in this case is relatively simple: there is a common prefix of the RHSs of the two productions. Thus, we can try to *postpone* the choice by factoring out the common prefix. That could be enough to satisfy the LL(1) conditions.

Thus, what we need here is *left factoring* (section 8.4). After left factoring:

$$\begin{array}{rcl} S & \rightarrow & aXbYS' \\ S' & \rightarrow & \epsilon \mid cZ \end{array}$$

As it turns out, this is now suitable for LL(1) parsing!

10.7 Beyond hand-written parsers: use parser generators

The restriction to LL(1) has a number of disadvantages: In many case a natural grammar like has to be changed to satisfy the LL(1) conditions. This may even be impossible: some context-free *languages* cannot be generated by any LL(1) grammar.

Luckily, there is a more powerful approach called LR(1). LR(1) is a bottom-up method and was briefly discussed in section 10.2. In particular, in contrast to LL(1), LR(1) can handle both left-recursive and right-recursive grammars without modification.

The disadvantage with LR(1) and the related approach LALR(1) (which is slightly less powerful but much more efficient) is that it is very hard to construct LR-parsers by hand. Hence there are automated tools that get the grammar as an input and produce a parser as the output. One of the first of those *parser generators* was YACC for C. Nowadays one can find parser generators for many languages such as JAVA CUP for Java [Hud99] and Happy for Haskell [Mar01].

However, there are also very serious tools based on LL(k) parsing technology, such as ANTLR (ANother Tool for Language Recognition) [Par05], that overcome some of the problems of basic LL parsing and that provides as much automation as any other parser generator tool. It is true that a grammar still may have to be changed a little bit to parseable, but that is true for LR parsing too. In practice, once one become familiar with a tool, it is not that hard to write a grammar in such a way so as to avoid the most common pitfalls from the outset. Additionally, most tools provide mechanics such as declarative specification of disambiguation rules that allows grammars to be written in a natural way without worrying too much about the details of the underlying parsing technology. Today, when it comes to choosing a tool, the underlying parsing technology is probably less important than other factors such as tool quality, supported development languages, feature set, etc.

10.8 Exercises

Exercise 10.1

Consider the following Context-Free Grammar (CFG):

$$\begin{array}{rcl} S & \rightarrow & ABB \mid BBC \mid CA \\ A & \rightarrow & aA \mid \epsilon \\ B & \rightarrow & Bb \mid \epsilon \\ C & \rightarrow & cC \mid d \end{array}$$

S, A, B, and C are nonterminals, a, b, c, and d are terminals, and S is the start symbol.

1. What is the set N_ϵ of *nullable* nonterminals? Provide a brief justification.
2. Systematically compute the *first sets* for all nonterminals, i.e. first(S), first(A), first(B), and first(C), by setting up and solving the equations according to the definitions of first sets for nonterminals and strings of grammar symbols. Show your calculations.
3. Set up the subset constraint system that defines the *follow sets* for all nonterminals, i.e. follow(S), follow(A), follow(B), and follow(C). Simplify where possible using the law

$$X \subseteq Z \;\wedge\; Y \subseteq Z \quad\Longleftrightarrow\quad X \cup Y \subseteq Z$$

   and the fact that constraints like $\emptyset \subseteq X$ and $X \subseteq X$ are trivially satisfied and can be omitted.
4. Solve the subset constraint system for the follow sets from the previous question by finding the smallest sets satisfying the constraints.

Exercise 10.2

Consider the following Context-Free Grammar (CFG):

$$\begin{array}{rcl} S & \rightarrow & AS \mid AB \\ A & \rightarrow & aA \mid \epsilon \\ B & \rightarrow & BCDb \mid \epsilon \\ C & \rightarrow & cD \mid \epsilon \\ D & \rightarrow & dC \mid e \end{array}$$

S, A, B, C, and D are nonterminals, a, b, c, d, and e are terminals, and S is the start symbol.

1. What is the set N_ϵ of *nullable* nonterminals? Provide a brief justification.
2. Systematically compute the *first sets* for all nonterminals, i.e., first(S), first(A), first(B), first(C), and first(D), by setting up and solving the equations according to the definitions of first sets for nonterminals and strings of grammar symbols. Show your calculations.

3. Set up the subset constraint system that defines the *follow sets* for all nonterminals; i.e., follow(S), follow(A), follow(B), follow(C), and follow(D). Simplify where possible using the law

$$X \subseteq Z \;\wedge\; Y \subseteq Z \quad\Longleftrightarrow\quad X \cup Y \subseteq Z$$

and by removing trivially satisfied constraints such as $\emptyset \subseteq X$ and $X \subseteq X$.

4. Solve the subset constraint system for the follow sets from the previous question by finding the *smallest* sets satisfying the constraints.

11 Turing Machines

A Turing machine (TM) is a generalization of a PDA that uses a tape instead of a stack. Turing machines are an abstract version of a computer: they have been used to define formally what is *computable*. There are a number of alternative approaches to formalize the concept of computability (for example, the λ-calculus (section 12), μ-recursive functions, the von Neumann architecture, and so on) but they all turn out to be equivalent. That this is the case for any reasonable notion of computation is called the *Church-Turing Thesis*.

On the other side there is a generalization of context-free grammars called phrase structure grammars or just grammars. Here we allow several symbols on the left hand side of a production, e.g. we may define the context in which a rule is applicable. Languages definable by grammars correspond precisely to the ones that may be accepted by a Turing machine and those are called *Type-0-languages* or the *recursively enumerable languages* (or *semidecidable languages*)

Turing machines behave differently from the previous machine classes we have seen: they may run forever, without stopping. To say that a language is accepted by a Turing machine means that the TM will stop in an accepting state for each word that is in the language. However, if the word is not in the language the Turing machine may stop in a non-accepting state or loop forever. In this case we can never be sure whether the given word is in the language; i.e., the Turing machine doesn't decide the word problem.

We say that a language is *recursive* (or *decidable*), if there is a TM that accepts it and always stop on any word. There are *type-0-languages* that are not recursive; the most famous one is the *halting problem*. This is the language of encodings of Turing machines that will always stop.

There is no type of grammars that captures all recursive languages (and for theoretical reasons there cannot be one). However there is a subset of recursive languages, called the *context-sensitive languages*, that can be characterized by *context-sensitive grammars*. These are grammars where the left-hand side of a production is always shorter than the right-hand side. Context-sensitive languages on the other hand correspond to linear bounded TMs, that use only a tape whose length can be given by a linear function over the length of the input.

11.1 What is a Turing machine?

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is:

- A finite set Q of states;
- A finite set Σ of symbols (the alphabet);
- A finite set Γ of tape symbols s.t. $\Sigma \subseteq \Gamma$. This is the case because we use the tape also for the input;
- A transition function

  $$\delta \in Q \times \Gamma \to \{\text{stop}\} \cup Q \times \Gamma \times \{\text{L}, \text{R}\}$$

  The transition function defines how the machine behaves if is in state q and the symbol on the tape is x. If $\delta(q, x) = \text{stop}$ then the machine stops otherwise if $\delta(q, x) = (q', y, d)$ the machines gets into state q', writes y on the tape (replacing x) and moves left if $d = \text{L}$ or right, if $d = \text{R}$;

- An initial state $q_0 \in Q$;
- The blank symbol $B \in \Gamma$ but $B \notin \Sigma$. Initially, only a finite section of the tape containing the input is non-blank;
- A set of final states $F \subseteq Q$.

In [HMU01] the transition function is defined without an explicit option to stop with type $\delta \in Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$. However, they allow δ to be undefined which corresponds to our function returning stop.

The above defines deterministic Turing machines; for nondeterministic TMs the type of the transition function is changed to:

$$\delta \in Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$$

In this case, the transition function returning the empty set plays the role of stop. As for finite automata (and unlike for PDAs) there is no difference in the strength of deterministic or nondeterministic TMs.

As for PDAs, we define instantaneous descriptions for Turing machines: $\mathrm{ID} = \Gamma^* \times Q \times \Gamma^*$. An element $(\gamma_L, q, \gamma_R) \in \mathrm{ID}$ describes a situation where the TM is in state Q, the non-blank portion of the tape on the left of the head is γ_L and the non-blank portion of the tape on the right, including the square under the head, is γ_R.

We define the next-state relation $\underset{M}{\vdash}$ similarly to PDAs:

$$\begin{array}{rrcll}
1. & (\gamma_{\mathrm{L}},\, q,\, x\gamma_{\mathrm{R}}) & \underset{M}{\vdash} & (\gamma_{\mathrm{L}}y,\, q',\, \gamma_{\mathrm{R}}) & \text{if } \delta(q,x) = (q', y, \mathrm{R}) \\
2. & (\gamma_{\mathrm{L}}z,\, q,\, x\gamma_{\mathrm{R}}) & \underset{M}{\vdash} & (\gamma_{\mathrm{L}},\, q',\, zy\gamma_{\mathrm{R}}) & \text{if } \delta(q,x) = (q', y, \mathrm{L}) \\
3. & (\epsilon,\, q,\, x\gamma_{\mathrm{R}}) & \underset{M}{\vdash} & (\epsilon,\, q',\, By\gamma_{\mathrm{R}}) & \text{if } \delta(q,x) = (q', y, \mathrm{L}) \\
4. & (\gamma_{\mathrm{L}},\, q,\, \epsilon) & \underset{M}{\vdash} & (\gamma_{\mathrm{L}}y,\, q',\, \epsilon) & \text{if } \delta(q,B) = (q', y, \mathrm{R}) \\
5. & (\gamma_{\mathrm{L}}z,\, q,\, \epsilon) & \underset{M}{\vdash} & (\gamma_{\mathrm{L}},\, q',\, zy) & \text{if } \delta(q,B) = (q', y, \mathrm{L}) \\
6. & (\epsilon,\, q,\, \epsilon) & \underset{M}{\vdash} & (\epsilon,\, q',\, By) & \text{if } \delta(q,B) = (q', y, \mathrm{L})
\end{array}$$

The cases 3 to 6 are only needed to deal with the situation of having reached the end of the non-blank part of the tape.

We say that a TM M *accepts* a word if it goes into an accepting state; i.e., the language of a TM is defined as

$$L(M) = \{w \in \Sigma^* \mid (\epsilon, q_0, w) \overset{*}{\underset{M}{\vdash}} (\gamma_L, q', \gamma_R) \wedge q' \in F\}$$

That is, the TM stops automatically if it goes into an accepting state. However, it may also stop in a non-accepting state if δ returns stop. In this case the word is rejected. A TM M *decides* a language if it accepts it and it never loops (in the negative case).

To illustrate, we define a TM M that accepts the language

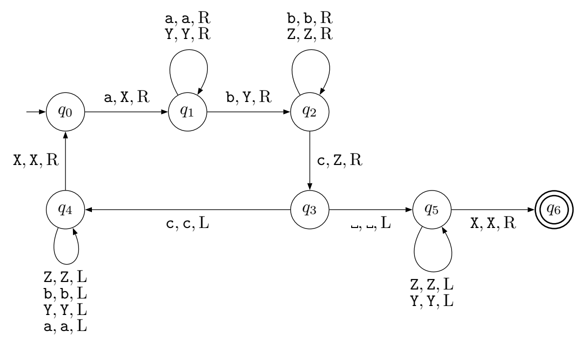
$$L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}\}$$

This is a language that cannot be recognized by a PDA or be defined by a CFG. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

$$\begin{array}{rcl}
Q & = & \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\} \\
\Sigma & = & \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \\
\Gamma & = & \Sigma \cup \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \textvisiblespace\} \\
q_0 & = & q_0 \\
B & = & \textvisiblespace \\
F & = & \{q_6\}
\end{array}
\qquad
\begin{array}{rcll}
\delta(q_0, \textvisiblespace) & = & (q_6, \textvisiblespace, \mathrm{R}) & \\
\delta(q_0, \mathtt{a}) & = & (q_1, \mathtt{X}, \mathrm{R}) & \\
\delta(q_1, \mathtt{a}) & = & (q_1, \mathtt{a}, \mathrm{R}) & \\
\delta(q_1, \mathtt{Y}) & = & (q_1, \mathtt{Y}, \mathrm{R}) & \\
\delta(q_1, \mathtt{b}) & = & (q_2, \mathtt{Y}, \mathrm{R}) & \\
\delta(q_2, \mathtt{b}) & = & (q_2, \mathtt{b}, \mathrm{R}) & \\
\delta(q_2, \mathtt{Z}) & = & (q_2, \mathtt{Z}, \mathrm{R}) & \\
\delta(q_2, \mathtt{c}) & = & (q_3, \mathtt{Z}, \mathrm{R}) & \\
\delta(q_3, \textvisiblespace) & = & (q_5, \textvisiblespace, \mathrm{L}) & \\
\delta(q_3, \mathtt{c}) & = & (q_4, \mathtt{c}, \mathrm{L}) & \\
\delta(q_4, \mathtt{Z}) & = & (q_4, \mathtt{Z}, \mathrm{L}) & \\
\delta(q_4, \mathtt{b}) & = & (q_4, \mathtt{b}, \mathrm{L}) & \\
\delta(q_4, \mathtt{Y}) & = & (q_4, \mathtt{Y}, \mathrm{L}) & \\
\delta(q_4, \mathtt{a}) & = & (q_4, \mathtt{a}, \mathrm{L}) & \\
\delta(q_4, \mathtt{X}) & = & (q_0, \mathtt{X}, \mathrm{R}) & \\
\delta(q_5, \mathtt{Z}) & = & (q_5, \mathtt{Z}, \mathrm{L}) & \\
\delta(q_5, \mathtt{Y}) & = & (q_5, \mathtt{Y}, \mathrm{L}) & \\
\delta(q_5, \mathtt{X}) & = & (q_6, \mathtt{X}, \mathrm{R}) & \\
\delta(q, x) & = & \text{stop} & \text{everywhere else}
\end{array}$$

The machine replaces an `a` by `X` (q_0) and then looks for the first `b` replaces it by `Y` (q_1) and looks for the first `c` and replaces it by a `Z` (q_2). If there are more `c`'s left it moves left to the next `a` (q_4) and repeats the cycle. Otherwise it checks whether there are no `a`'s and `b`'s left (q_5) and if so goes into an accepting state (q_6).

Graphically the machine can be represented by the following transition diagram, where the edges are labelled by (read-symbol, write-symbol, move-direction):



E.g. consider the sequence of IDs on `aabbcc`:

$$\begin{array}{rcl}
(\epsilon, q_0, \texttt{aabbcc}) & \vdash & (\texttt{X}, q_1, \texttt{abbcc}) \\
& \vdash & (\texttt{Xa}, q_1, \texttt{bbcc}) \\
& \vdash & (\texttt{XaY}, q_2, \texttt{bcc}) \\
& \vdash & (\texttt{XaYb}, q_2, \texttt{cc}) \\
& \vdash & (\texttt{XaYbZ}, q_3, \texttt{c}) \\
& \vdash & (\texttt{XaYb}, q_4, \texttt{Zc}) \\
& \vdash & (\texttt{XaY}, q_4, \texttt{bZc}) \\
& \vdash & (\texttt{Xa}, q_4, \texttt{YbZc}) \\
& \vdash & (\texttt{X}, q_4, \texttt{aYbZc}) \\
& \vdash & (\epsilon, q_4, \texttt{XaYbZc}) \\
& \vdash & (\texttt{X}, q_0, \texttt{aYbZc}) \\
& \vdash & (\texttt{XX}, q_1, \texttt{YbZc}) \\
& \vdash & (\texttt{XXY}, q_1, \texttt{bZc}) \\
& \vdash & (\texttt{XXYY}, q_2, \texttt{Zc}) \\
& \vdash & (\texttt{XXYYZ}, q_2, \texttt{c}) \\
& \vdash & (\texttt{XXYYZZ}, q_3, \epsilon) \\
& \vdash & (\texttt{XXYYZ}, q_5, \texttt{Z\textvisiblespace}) \\
& \vdash & (\texttt{XXYY}, q_5, \texttt{ZZ\textvisiblespace}) \\
& \vdash & (\texttt{XXY}, q_5, \texttt{YZZ\textvisiblespace}) \\
& \vdash & (\texttt{XX}, q_5, \texttt{YYZZ\textvisiblespace}) \\
& \vdash & (\texttt{X}, q_5, \texttt{XYYZZ\textvisiblespace}) \\
& \vdash & (\texttt{XX}, q_6, \texttt{YYZZ\textvisiblespace})
\end{array}$$

We see that M accepts `aabbcc`. Because M never loops, it does actually decide L.

11.2 Grammars and context-sensitivity

Let us define grammars $G = (N, T, P, S)$ in the same way as context-free grammars were defined before, with the *only* difference that there may now be *several* symbols on the left-hand side of a production; i.e., $P \subseteq (N \cup T)^+ \times (N \cup T)^*$. Here $(N \cup T)^+$ means that at least one symbol has to present. The relation derives $\underset{G}{\Rightarrow}$ (and $\underset{G}{\overset{*}{\Rightarrow}}$) is defined as before:

$$\begin{array}{rcl}
\underset{G}{\Rightarrow} & \subseteq & (N \cup T)^* \times (N \cup T)^* \\
\alpha\beta\gamma \underset{G}{\Rightarrow} \alpha\beta'\gamma & \iff & \beta \to \beta' \in P
\end{array}$$

Also as before, the language of G is defined as:

$$L(G) = \{w \in T^* \mid S \underset{G}{\overset{*}{\Rightarrow}} w\}$$

We say that a grammar is context-sensitive (or type 1) if the right-hand side of a production is at least as long as the left-hand side. That is, for each $\alpha \to \beta \in P$, we have $|\beta| \geq |\alpha|$.

Here is an example of a context-sensitive grammar $G = (N, T, P, S)$ with $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N} \wedge n \geq 1\}$ where

- $N = \{S, B, C\}$
- $T = \{a, b, c\}$
- P is the set of productions:

$$\begin{array}{lcl}
S & \to & aSBC \\
S & \to & aBC \\
aB & \to & ab \\
CB & \to & BC \\
bB & \to & bb \\
bC & \to & bc \\
cC & \to & cc
\end{array}$$

- S is the start symbol

We present without proof:

Theorem 11.1 *For a language $L \subseteq T^*$ the following is equivalent:*

1. *L is accepted by a Turing machine M; i.e., $L = L(M)$*
2. *L is given by a grammar G; i.e., $L = L(G)$*

Theorem 11.2 *For a language $L \subseteq T^*$ the following is equivalent:*

1. *L is accepted by a Turing machine M; i.e., $L = L(M)$ such that the length of the tape is bounded by a linear function in the length of the input; i.e., $|\gamma_L| + |\gamma_R| \leq f(x)$ where $f(x) = ax + b$ with $a, b \in \mathbb{N}$.*
2. *L is given by a context-sensitive grammar G; i.e., $L = L(G)$*

11.3 The halting problem

Turing showed that there are languages that are accepted by a TM (i.e., type 0 languages) but that are undecidable. The technical details of this construction are involved but the basic idea is simple and is closely related to Russell's paradox, which we have seen in MCS.

Let's fix a simple alphabet $\Sigma = \{0, 1\}$. As computer scientist we are well aware that everything can be coded up in bits and hence we accept that there is an encoding of TMs in binary. That is, we can agree on a way to express every TM M as a string of bits $\ulcorner M \urcorner \in \{0,1\}^*$. We assume that the string contains its own length at the beginning, so that we know when the encoding ends. This allows us to put both the TM and an input for it, one after the other, on the same tape. We can determine when the encoding of the machine ends and the subsequent input on starts.

Now we define the following language

$$L_{\text{halt}} = \{\ulcorner M \urcorner w \mid M \text{ halts on input } w.\}$$

It is easy (although the details are quite daunting) to define a TM that accepts this language: we just simulate M and accept if M stops. However, Turing showed that there is no TM that decides this language.

Let us prove this by assuming that the language is decidable and deriving a contradiction from it. Suppose there is a TM H that decides L. That is, when run on a word v, H always terminates and the final state is accepting if and only if v has the form $v = \lceil M \rceil w$ for some machine M and input w and when M is run on w it terminates. In all other cases, if v is in such form but M does not terminate on w or if v is not in that form at all, H will terminate in a rejecting state.

Now using H we construct a new TM F that is a bit obnoxious. When we run F on input x, it computes H on the duplicated input xx. If H says *yes* (it accepts xx), then F goes into an infinite loop; otherwise, if H says *no* (it rejects xx), F stops.

What happens if we run F on its own code $\lceil F \rceil$? Will it terminate or loop forever? If we consider each of the two possibilities, we get the contradictory conclusion that the opposite should be true: if we assume that it terminates, we can prove that it must loop; if we assume that it loops, we can prove that it must terminate!

Let us assume F on $\lceil F \rceil$ terminates. By definition of F, this happens only if H applied to $\lceil F \rceil \lceil F \rceil$ says *no*. But by definition of H, this means that F on $\lceil F \rceil$ loops, contradicting the assumption.

Let us then assume that F on input $\lceil F \rceil$ loops. This happens only if H applied to $\lceil F \rceil \lceil F \rceil$ says *yes*. But this means that F on $\lceil F \rceil$ terminates, again contradicting the assumption.

We reach a contradiction on both possible behaviours of F: the machine F cannot exist. We must conclude that our assumption that there is a TM H that decides L_{halt} is false. We say L_{halt} is undecidable.

We have shown that there is no Turing machine that can decide whether other Turing machines halt. Is this a specific shortcoming of TMs or a universal limitation of all computing models? Maybe we could find a more powerful programming language that overcomes this problem? It turns out that all computational formalisms (i.e., programming languages) that have actually been implemented are equal in power and can be simulated by each other. Beside TMs, all modern programming languages, the λ-calculus, μ-recursive functions and many others were proved to be equivalent.

The statement that all models of computability are equivalent is called the *Church-Turing thesis* because it was first formulated by Alonzo Church and Alan Turing in the 1930s. This is discussed further in section 12.6.

11.4 Recursive and recursively enumerable sets

When we work with Turing Machines, there is a distinction between a language being accepted or decided.

A machine M *accepts* a language L if, whenever we run M on an input w, the computation terminates in an accepting state if and only if $w \in L$. We say that M *decides* L if it accepts it and always terminates.

The difference is that a machine that just accepts a language but doesn't decide it may run forever on some words that do not belong to the language.

If that happens, we have to wait forever to discover whether the word is in the language or not.

Definition 11.3 *A language L is* recursively enumerable *if it is accepted by a Turing Machine.*

The terminology comes from a different characterization of such languages. We can define a *recursively enumerable set* as a collection of values that can be produced by a total algorithm. Whenever we have a computable function $f : \mathbb{N} \to A$ from the natural numbers to some set A, we say that the range of the function, $U = \{a \in A \mid \exists n : \mathbb{N}, f(n) = a\}$ is recursively enumerable. We can also write $U = \{f(n) \mid n \in \mathbb{N}\}$. The idea is that we can enumerate all elements of U by computing $f(0)$, $f(1)$, $f(2)$, etc. In the case of languages, the set A is Σ^*. We can prove that being a recursively enumerable subset of Σ^* is equivalent to be accepted by a Turing Machine.

Definition 11.4 *A language L is* recursive *if there is a Turing machine that accepts it and always terminate.*

In general, a *recursive set* is a subset of some set A for which we can effectively determine membership. Whenever we have a computable function $g : A \to \mathsf{Bool}$ from some set A to Booleans, we say that the preimage of true, $\{a \in A \mid g(a) = \mathsf{true}\}$ is recursive.

Theorem 11.5 *A subset $U \subseteq A$ is recursive if and only if both U and its complement $\overline{U}$ are recursively enumerable.*

Proof. We won't see a formal proof using Turing Machines, but I'll outline the idea.

- In one direction, assume that U is recursive. So we have a function $g : A \to \mathsf{Bool}$ that decides it.

  We can construct $f : \mathbb{N} \to A$ by using some enumeration of all of A (we assume it is computably countable, otherwise it won't make sense to talk about algorithms on it). When computing the values of f we keep an index n; we go through all the elements of A one by one, whenever we find an a such that $g(a) = \mathsf{true}$, we set $f(n) = a$ and we increase n by one, now looking for the next element of A satisfying g. In this way we construct an f that generates all elements of U, so we proved that U is recursively enumerable.

  In a similar way we can show that $\overline{U}$ is recursively enumerable by systematically searching for the elements $a \in A$ for which $g(a) = \mathsf{false}$.

- In the other direction, assume that both U and $\overline{U}$ are recursively enumerable. We must prove that U is recursive.

  Saying that U is recursively enumerable means that there is a computable function $f_1 : \mathbb{N} \to A$ such that $U = \{f_1(a) \mid n : \mathbb{N}\}$. Saying that $\overline{U}$ is also recursively enumerable means that there is a computable function $f_2 : \mathbb{N} \to A$ such that $\overline{U} = \{f_2(a) \mid n : \mathbb{N}\}$.

  From f_1 and f_2 we can construct a function $g : A \to \mathsf{Bool}$ that decides U. For a given element $a \in A$, we can search whether it belongs to U by running the two functions f_1 and f_2 repeatedly in parallel:

– run f_1 on 0, if $f_1(0) = a$ then we know that $a \in U$ and we terminate giving the answer $g(a) = \mathsf{true}$;

– run f_2 on 0, if $f_2(0) = a$ then we know that $a \in \overline{U}$ and we terminate giving the answer $g(a) = \mathsf{false}$;

– run f_1 on 1, if $f_1(1) = a$ then we terminate with answer $g(a) = \mathsf{true}$;

– run f_2 on 1, if $f_2(1) = a$ then we terminate with answer $g(a) = \mathsf{false}$;

– run f_1 on 2, if $f_1(2) = a$ then we terminate with answer $g(a) = \mathsf{true}$;

– run f_2 on 2, if $f_2(2) = a$ then we terminate with answer $g(a) = \mathsf{false}$;

– continue until you find an n such that either $f_1(n) = a$ or $f_2(n) = a$.

Because every a belongs to either U or $\overline{U}$, we know for sure that this process will always terminate and produce the correct answer.

We have constructed a function g that decides U, so U is recursive.

□

In the previous section we proved that the Halting Problem is undecidable, therefore the set $L_{\text{halt}} = \{\ulcorner M \urcorner w \mid M \text{ halts on input } w\}$ is not recursive.

It is easy to see that it is recursively enumerable: we just have to run M and see if it terminates.

There are many other problems that are semi-decidable but not decidable, that is, they can be expressed by a recursively enumerable language that is not recursive. We can prove this by reducing the Halting Problem (or any other already known undecidable problem) to them.

Definition 11.6 *A language L_1 is reducible to a language L_2 if there is a Turing Machine M such that:*

- *M terminates on every input;*
- *If we run M on an input w, the computation terminates with a word v on the tape such that $w \in L_1$ if and only if $v \in L_2$.*

So L_1 is reducible to L_2 if we can translate (by a computable function) every question of membership of L_1 to an equivalent membership question of L_2.

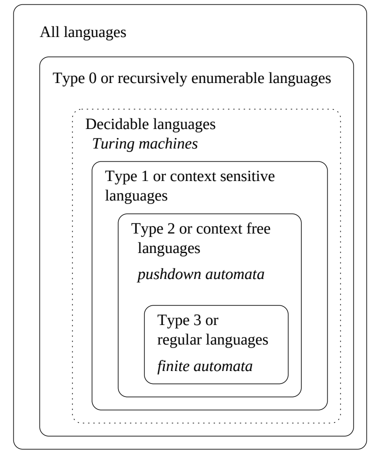
Theorem 11.7 *If L_{halt} is reducible to some language L, then L is not decidable.*

Proof. Suppose, towards a contradiction, that L is recursive. Then we have a Turing Machine M that decides L. We also know that there is a Turing Machine M' that translates instances of the Halting Problem to L. But then, by composing M' and M, we would be able to construct a machine H that decides the halting problem. But we know this is impossible. Therefore our assumption that L is recursive must be false. □

In general, if we know that a set V is not recursive and have another set U that we suspect is also not recursive, we can prove this fact by reducing V to U. Be very careful in the direction of the reduction: We reduce the problem that we already know to be undecidable to the one for which we want to prove undecidability.

11.5 Back to Chomsky

At the end of the course we should have another look at the Chomsky hierarchy, which classifies languages based on sublasses of grammars, or equivalently by different types of automata that recognize them



We have worked our way from the bottom to the top of the hierarchy: starting with finite automata, computation with fixed amount of memory via pushdown automata (finite automata with a stack), to Turing machines (finite automata with a tape). Correspondigly we have introduced different grammatical formalisms: regular expressions, context-free grammars and grammars.

Note that at each level there are languages that are on the next level but not on the previous: $\{a^n b^n \mid n \in \mathbb{N}\}$ is Type 2 but not Type 3, $\{a^n b^n c^n\}$ is Type 1 but not Type 2, and the Halting problem is Type 0 but not Type 1.

We could have gone the other way: starting with Turing machines and grammars and then introducing restrictions: Turing machines that only use their tapes as a stack describe Type 2 languages; Turing machines that never use the tape apart for reading the input describe Type 3 languages. Similarly, we have seen that context-free grammars restricted in specific ways describe precisely the regular languages (section 7.3).

Chomsky introduced his herarchy as a classification of grammars; the relation to automata was only observed a bit later. This may be the reason why he introduced the Type-1 level, which is not so interesting from an automata point of view (unless you are into computational complexity; i.e., resource use, here linear use of memory). It is also the reason why on the other hand the decidable languages do not constitute a level: there is no corresponding grammatical formalism (we can even prove this).

11.6 Exercises

Exercise 11.1

Consider the Turing Machine defined formally by:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F) \quad \text{where} \quad \begin{aligned} Q &= \{q_0, q_1, q_2, q_3, q_4, q_5\} \\ F &= \{q_5\} \\ \Sigma &= \{a, b\} \\ \Gamma &= \{a, b, X, Y, \sqcup\} \end{aligned}$$

with the transition function defined by:

$$\begin{array}{ll} \delta(q_0, a) = (q_3, X, L) & \\ \delta(q_0, x) = (q_0, x, R) & \text{for } x \in \{b, X, Y\} \\ \delta(q_0, \sqcup) = (q_2, \sqcup, L) & \\ \delta(q_1, b) = (q_4, Y, L) & \\ \delta(q_1, x) = (q_1, x, R) & \text{for } x \in \{a, X, Y\} \\ \delta(q_2, \sqcup) = (q_5, \sqcup, R) & \\ \delta(q_2, x) = (q_2, x, L) & \text{for } x \in \{X, Y\} \\ \delta(q_3, \sqcup) = (q_1, \sqcup, R) & \\ \delta(q_3, x) = (q_3, x, L) & \text{for } x \in \{a, b, X, Y\} \\ \delta(q_4, \sqcup) = (q_0, \sqcup, R) & \\ \delta(q_4, x) = (q_4, x, L) & \text{for } x \in \{a, b, X, Y\} \end{array}$$

1. Draw M graphically as a transition diagram.
2. Write down the sequence of instantaneous descriptions when starting the machine M with input *baab*. Is this word accepted or rejected?
3. Write down the sequence of instantaneous descriptions when starting the machine M with input *aba*. Is this word accepted or rejected?
4. What is the language accepted by M?

Exercise 11.2

Construct a Turing Machine that, when run on any word in $\{a, b\}^*$, rearranges the symbols so all the as come before all the bs. For example:

$$(\epsilon, q_0, aababba) \vdash^* (aaaabbb, q_f, \epsilon) \qquad (\epsilon, q_0, babbbaa) \vdash^* (aaabbbb, q_f, \epsilon)$$

[Hint: Your machine could look for pairs of symbols in the order '*ba*' and swap them to '*ab*'; keep repeating this until there are no such pairs left.]

1. Draw the machine as a transition diagram.
2. Give a formal definition of the machine, defining the transition function.

Exercise 11.3

Suppose that you have the following information about ten formal problems (expressed as languages) $P_1, P_2, \ldots P_{10}$:

- The Halting Problem is reducible to P_1;
- P_2 is reducible to P_1;
- P_1 is reducible to P_5;
- The complement of P_3 is recursively enumerable;
- P_4 is recursively enumerable;
- P_4 is reducible to the normalization problem for λ-calculus;
- P_4 is reducible to P_5;
- P_6 is not recursively enumerable;
- P_7 is recursive;
- P_7 is reducible to P_6;
- P_7 is reducible to P_8;
- The complement of P_8 is not recursively enumerable;
- The normalization problem for λ-calculus is reducible to P_9;
- P_{10} is reducible to P_3;
- P_{10} is reducible to P_4.

Note that the normalization problem for λ-calculus (see section 12) is *undecidable* Given this information, which of the languages P_1 to P_{10} are:

1. Undecidable?
2. Recursively enumerable?
3. Recursive?

Note that the same language may be in more than one of these three classes. Some others may be in none. Justify your answers by explaining how your conclusions follow from the given information.

12 λ-Calculus

In traditional imperative programming (like C, Java, Python), we have a clear distinction between *programs*, that are sequences of instructions that are executed sequentially, and *data*, that are values given in input, stored in memory, manipulated during computation and returned as output. Programs and data are distinct and are kept separated. Programs are not modified during computation. (It is in theory possible to do it, because programs are stored in memory like any other data. However, it is difficult and dangerous.)

Functional Programming is a different paradigm of computation in which there is no distinction between programs and data. Both are represented by terms/expressions belonging to the same language. Computation consists in the reduction of terms to normal form. That includes terms that represent functions, that is, the programs themselves.

The pure realization of this idea is the λ-calculus. It is a pure theory of functions with only one kind of objects: λ-terms. They represent both data structures and programs.

The main idea is the definition of functions by *abstraction*. For example, we may define a function f on numbers by saying that $f(x) = x^2 + 3$. By this we mean that any argument to the function, represented by the variable x, is squared and added to 3. The use of variables is different from imperative programming: x is just a place-holder to denote any possible value, while in imperative programming variables represent memory locations containing values that can be modified.

We can specify the function f alternatively with the *mapping* notation:

$$x \xmapsto{f} x^2 + 3.$$

This is written in λ-*notation* as: $f = \lambda x.x^2 + 3$. (In the functional programming language Haskell, it is `\x -> x^2+3`.)

While abstraction is the operation to define a new function, computing it on a specific argument is called *application*. We indicate it simply by juxtaposition:

$$f\,5 = (\lambda x.x^2 + 3)\,5 \rightsquigarrow 5^2 + 3 \rightsquigarrow^* 28.$$

As the example shows, the application of a λ-abstraction to an argument is computed by replacing the abstraction variable with the argument. This is called β-*reduction* and it is the basic computation step of λ-calculus.

The λ-notation is convenient to define functions, but you may think that the actual computation work is done by the operations used in the body of the abstraction: squaring and adding 3. However, the λ-calculus is *a theory of pure functions*: terms are constructed using only abstraction an application, there are no other basic operations. At first, this looks like a rather useless system: no numbers, no arithmetic operations, no data structures, no programming primitives. The surprising fact is that we don't really need them. We don't need numbers (5, 3) and we don't need operations ($-^2$, $+$). They all can be defined as purely functional constructions, built using only abstraction and application!

12.1 Syntax of λ-calculus

The language of λ-calculus is extremely simple, we start with variables and construct terms using only abstraction and computation. It's BNF definition is

as follows (assume x, y, z range over a given infinite set of variable names).

$$\begin{array}{rcll} t & := & x \mid y \mid z \mid \cdots & \text{variable names} \\ & \mid & \lambda x.t & \text{abstraction} \\ & \mid & t\,t & \text{application.} \end{array}$$

The β-reduction relation on term is defined, for every pair of terms t_1 and t_2 as:

$$(\lambda x.t_1)\,t_2 \rightsquigarrow_\beta t_1[x := t_2].$$

The left-hand side means: in t_1 substitute all occurrences of variable x with the term t_2. Substitution is actually quite tricky and its precise definition is a bit more complex that replacing every occurrences of x with t_2. One has to be careful to manage variable occurrences properly.

We need some intermediate concept. The first is α-equivalence and it says that, because the variable in an abstraction is just a place holder for an argument, the names of abstracted variables does not matter. For example, the simplest function we can define is the identity $\mathsf{id} := \lambda x.x$ which takes an argument x and returns it unchanged. Clearly, if we use a different variable name, $\lambda y.y$, we get exactly the same function. We say that the two terms (and any using different variables) are α-equivalent:

$$\lambda x.x =_\alpha \lambda y.y =_\alpha \lambda z.z =_\alpha \cdots.$$

We are free to change the name of the abstracted variable any way we like. However, we have to be careful to avoid *variable capture*. If the body of the abstraction contains other variables than the abstracted one, we can't change the name to those:

$$\lambda x.y\,(x\,z) =_\alpha \lambda w.y\,(w\,z) \neq_\alpha \lambda y.y\,(y\,z) \neq_\alpha \lambda z.y\,(z\,z).$$

The reason for this restriction is that changing the name of the abstracted variable from x to either y or z in this example would capture the occurrence of that variable which was *free* in the original term (not bound by a λ-abstraction).

Formally, we define set $\mathsf{FV}(t)$ of the variables that occur free in the term t, by recursion on the structure of t:

$$\begin{array}{l} \mathsf{FV}(x) = \{x\} \\ \mathsf{FV}(\lambda x.t) = \mathsf{FV}(t) \setminus \{x\} \\ \mathsf{FV}(t_1\,t_2) = \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2). \end{array}$$

Another way in which a variable can be incorrectly captured is when we perform a substitution that puts a term under an abstraction that may bind some of its variables:

$$(\lambda x.\lambda y.x\,y)\,(y\,z) \rightsquigarrow_\beta (\lambda y.x\,y)[x := (y\,z)] \neq \lambda y.(y\,z)\,y.$$

If we replace x with $(y\,z)$ in this way, the occurrence of the variable y (which was free before performing the β-reduction) become bound. This is incorrect. We should rename the abstraction variable before performing the substitution:

$$(\lambda y.x\,y)[x := (y\,z)] =_\alpha (\lambda w.x\,w)[x := (y\,z)] = \lambda w.(y\,z)\,w.$$

To avoid problems with variable capture, we adopt the *Barendregt variable convention*: before performing substitution (or any other operation on terms), change the names of the abstracted variables so they are different from the free variables and from each other.

With this convention, we can give a precise definition of substitution by recursion on the structure of terms:

$$\begin{aligned} x[x := t_2] &= t_2 \\ y[x := t_2] &= y \qquad \text{if } y \neq x \\ (\lambda y.t_1)[x := t_2] &= \lambda y.t_1[x := t_2] \\ (t_0\, t_1)[x := t_2] &= t_0[x := t_2]\, t_1[x := t_2]. \end{aligned}$$

In the third case, the variable convention ensures that the variable y and the bound variables in t_2 have already been renamed so they avoid captures. In more traditional formulations, one would add the requirements: "provided that $y \neq x$ and y doesn't occur free in t_2".

A part from some complication about substitution, the λ-calculus is extremely simple. It seems at first surprising that we can actually do any serious computation with it at all. But it turns out that all computable functions can be represented by λ-terms. We see some simple function in this section and we will discover how to represent data structures in the next.

For convenience, we use some conventions that allow us to save on parentheses.

- λ-abstraction associates to the right, so we write $\lambda x.\lambda y.x$ for $\lambda x.(\lambda y.x)$;
- Application associates to the left, so we write $(t_1\, t_2\, t_3)$ for $((t_1\, t_2)\, t_3)$;
- We can use a single λ symbol followed by several variables to mean consecutive abstractions, so we write $\lambda x\, y.x$ for $\lambda x.\lambda y.x$.

Here are three very simple functions implemented as λ-terms:

- The identity function $\mathsf{id} := \lambda x.x$. When applied to an argument it simply returns it unchanged:

  $$\mathsf{id}\, t = (\lambda x.x)\, t \rightsquigarrow x[x := t] = t.$$

- The first projection function $\lambda x.\lambda y.x$. When applied to two arguments, it returns the first:

  $$(\lambda x.\lambda y.x)\, t_1\, t_2 \rightsquigarrow (\lambda y.x)[x := t_1]\, t_2 = (\lambda y.t_1)\, t_2 \rightsquigarrow t_1[y := t_2] = t_1.$$

  Remember, in reading this reduction sequence, that we are adopting the variable convention, so the variable y doesn't occur free in t_1.

- The second projection function $\lambda x.\lambda y.x$. When applied to two arguments, it returns the second:

  $$(\lambda x.\lambda y.y)\, t_1\, t_2 \rightsquigarrow (\lambda y.y)[x := t_1]\, t_2 = (\lambda y.y)\, t_2 \rightsquigarrow y[y := t_2] = t_2.$$

  We can also say that the second projection is the function that, when applied to an argument t_1, returns the identity function $\lambda y.y$.

12.2 Church numerals

So far we have seen only some very basic functions that only return some of their arguments unchanged. How can we define more interesting computations? And first of all, how can we represent values and data structures? It is in fact possible to represent any kind of data buy some λ-term.

Let's start by representing natural numbers. Their encodings in λ-calculus are called *Church Numerals*:

$$\begin{aligned} \overline{0} &:= \lambda f.\lambda x.x \\ \overline{1} &:= \lambda f.\lambda x.f\, x \\ \overline{2} &:= \lambda f.\lambda x.f\,(f\, x) \\ \overline{3} &:= \lambda f.\lambda x.f\,(f\,(f\, x)) \\ &\cdots. \end{aligned}$$

A numeral $\overline{n}$ is a function that takes two arguments, denoted by the variables f and x, and applies f sequentially n times to x.

What is important is that we assign to every number a distinct λ-term in a uniform way. We must choose our representation so it is easy to represent arithmetic operations. The idea of Church numerals is nicely conceptual: numbers are objects that we use to count things, so we can define them as the counters of repeated application of a function.

Let's see if this representation is convenient from the programming point of view: can we define basic operations on it?

Let's start with the successor function, that increases a number by one:

$$\mathsf{succ} := \lambda n.\lambda f.\lambda x.f\,(n\, f\, x).$$

Let's test if it works on an example: if we apply it to $\overline{2}$ we should get $\overline{3}$:

$$\begin{aligned} \mathsf{succ}\,\overline{2} &= (\lambda n.\lambda f.\lambda x.f\,(n\, f\, x))\,\overline{2} \\ &\rightsquigarrow \lambda f.\lambda x.f\,(\overline{2}\, f\, x) = \lambda f.\lambda x.f\,((\lambda f.\lambda x.f\,(f\, x))\, f\, x) \\ &\rightsquigarrow \lambda f.\lambda x.f\,((\lambda x.f\,(f\, x))[f := f]\, x) = \lambda f.\lambda x.f\,((\lambda x.f\,(f\, x))\, x) \\ &\rightsquigarrow \lambda f.\lambda x.f\,((f\,(f\, x))[x := x]) = \lambda f.\lambda x.f\,(f\,(f\, x)) = \overline{3}. \end{aligned}$$

We have explicitly marked the substitutions in this reduction sequence: they are both trivial, substituting f with itself and x with itself. From now on, we'll do the substitutions on the fly, without marking them.

Other arithmetic operations can be defined by simple terms:

$$\begin{aligned} \mathsf{plus} &:= \lambda m.\lambda n.\lambda f.\lambda x.m\, f\,(n\, f\, x) \\ \mathsf{mult} &:= \lambda m.\lambda n.\lambda f.m\,(n\, f) \\ \mathsf{exp} &:= \lambda m.\lambda n.n\, m. \end{aligned}$$

Verify by yourself that these terms correctly implement the addition, multiplication and exponentiation functions. For example:

$$\begin{aligned} \mathsf{plus}\,\overline{2}\,\overline{3} &\rightsquigarrow^* \overline{5} \\ \mathsf{mult}\,\overline{2}\,\overline{3} &\rightsquigarrow^* \overline{6} \\ \mathsf{exp}\,\overline{2}\,\overline{3} &\rightsquigarrow^* \overline{8} \\ \mathsf{exp}\,\overline{2}\,\overline{3} &\rightsquigarrow^* \overline{9}. \end{aligned}$$

Exercise. Define a term isZero that tests if a Church numeral is $\overline{0}$ or a successor. It should have the following reduction behaviour:

$$\mathsf{isZero}\,\overline{0} \rightsquigarrow^* \mathsf{true}; \qquad \mathsf{isZero}\,(\mathsf{succ}\,t) \rightsquigarrow^* \mathsf{false} \quad \text{for every term } t.$$

Surprisingly, two other basic functions are much more difficult to define: the predecessor and the (cut-off) subtraction functions. Try to define two terms pred and minus such that:

$$\begin{array}{ll} \mathsf{pred}\,(\mathsf{succ}\,\overline{n}) \rightsquigarrow^* \overline{n} & \\ \mathsf{pred}\,\overline{0} \rightsquigarrow^* \overline{0} & \\ \mathsf{minus}\,\overline{n}\,\overline{m} \rightsquigarrow^* \overline{n-m} & \text{if } n \geq m \\ \mathsf{minus}\,\overline{n}\,\overline{m} \rightsquigarrow^* \overline{0} & \text{if } n < m. \end{array}$$

12.3 Other data structures

Other data types can be encoded in the λ-calculus.

Booleans For truth values we may choose the first and second projects that we defined earlier:

$$\begin{array}{l} \mathsf{true} := \lambda x.\lambda y.x \\ \mathsf{false} := \lambda x.\lambda y.y. \end{array}$$

We must show how to compute the logical operators. For example, conjunction can be defined as follows:

$$\mathsf{and} := \lambda a.\lambda b.a\,b\,\mathsf{false}.$$

Let's verify that it give the correct results when applied to Boolean values:

$$\begin{array}{ll} \mathsf{and\,true\,true} & = (\lambda a.\lambda b.a\,b\,\mathsf{false})\,\mathsf{true\,true} \\ & \rightsquigarrow^* \mathsf{true\,true\,false} = (\lambda x.\lambda y.x)\,\mathsf{true\,false} \rightsquigarrow^* \mathsf{true} \\ \mathsf{and\,true\,false} & = (\lambda a.\lambda b.a\,b\,\mathsf{false})\,\mathsf{true\,false} \\ & \rightsquigarrow^* \mathsf{true\,false\,false} = (\lambda x.\lambda y.x)\,\mathsf{false\,false} \rightsquigarrow^* \mathsf{false} \\ \mathsf{and\,false}\,t & = (\lambda a.\lambda b.a\,b\,\mathsf{false})\,\mathsf{false}\,t \\ & \rightsquigarrow^* \mathsf{false}\,t\,\mathsf{false} = (\lambda x.\lambda y.y)\,t\,\mathsf{false} \rightsquigarrow^* \mathsf{false} \end{array}$$

All the other logical operators could be defined if we had a conditional construct *if-then-else*. In fact this can be defined very easily:

$$\mathsf{if} := \lambda b.\lambda u.\lambda v.b\,u\,v.$$

Let's verify that it has the correct computational behaviour:

$$\begin{array}{ll} \mathsf{if\,true}\,t_1\,t_2 & = (\lambda b.\lambda u.\lambda v.b\,u\,v)\,\mathsf{true}\,t_1\,t_2 \\ & \rightsquigarrow^* \mathsf{true}\,t_1\,t_2 = (\lambda x.\lambda y.x)\,t_1\,t_2 \rightsquigarrow^* t_1 \\ \mathsf{if\,false}\,t_1\,t_2 & = (\lambda b.\lambda u.\lambda v.b\,u\,v)\,\mathsf{false}\,t_1\,t_2 \\ & \rightsquigarrow^* \mathsf{false}\,t_1\,t_2 = (\lambda x.\lambda y.y)\,t_1\,t_2 \rightsquigarrow^* t_2. \end{array}$$

Then we can define all logical connectives as conditionals, for example:

$$\mathsf{and} := \lambda a.\lambda b.\mathsf{if}\,a\,b\,\mathsf{false}, \quad \mathsf{or} := \lambda a.\lambda b.\mathsf{if}\,a\,\mathsf{true}\,b, \quad \mathsf{not} := \lambda a.\mathsf{if}\,a\,\mathsf{false\,true}.$$

As a curiosity (just a feature of the encoding, don't read anything deeply philosophical in it) notice that *false is the true identity!*

$$\mathsf{true\,id} \rightsquigarrow \mathsf{false}$$

This is also a good example in the managing of variables: $\mathsf{true} = \lambda x.\lambda y.x$ and $\mathsf{id} = \lambda x.x$. If we follow the variable convention, we shouldn't use x twice, so let's rename it in the identity: $\mathsf{id} = \lambda z.z$. Then

$$\mathsf{true\,id} = (\lambda x.\lambda y.x)\,(\lambda z.z) \rightsquigarrow \lambda y.\lambda z.z = \lambda x.\lambda y.y = \mathsf{false}$$

where at the end we're free to rename the bound variables according to α-conversion.

Tuples Pairs of λ-terms can be encoded by a single term: If t_1 and t_2 are terms, we define the encoding of the pair as

$$\langle t_1, t_2 \rangle := \lambda x.x\,t_1\,t_2.$$

First and second projections are obtained by applying a pair to the familiar projections (or truth values) that we have already seen:

$$\begin{array}{l} \mathsf{fst}\,p = p\,(\lambda x.\lambda y.x) \\ \mathsf{snd}\,p = p\,(\lambda x.\lambda y.y) \end{array}$$

We can verify that they have the correct reduction behaviour:

$$\begin{array}{l} \mathsf{fst}\,\langle t_1, t_2 \rangle = \langle t_1, t_2 \rangle\,(\lambda x.\lambda y.x) = (\lambda x.x\,t_1\,t_2)\,(\lambda x.\lambda y.x) \rightsquigarrow (\lambda x.\lambda y.x)\,t_1\,t_2 \rightsquigarrow^* t_1, \\ \mathsf{snd}\,\langle t_1, t_2 \rangle = \langle t_1, t_2 \rangle\,(\lambda x.\lambda y.y) = (\lambda x.x\,t_1\,t_2)\,(\lambda x.\lambda y.y) \rightsquigarrow (\lambda x.\lambda y.y)\,t_1\,t_2 \rightsquigarrow^* t_2. \end{array}$$

Triples and longer tuples may be encoded as repeated pairs, for example $\langle t_1, t_2, t_3 \rangle := \langle t_1, \langle t_2, t_3 \rangle \rangle$, or directly using the same idea as for pairs: $\langle t_1, t_2, t_3 \rangle := \lambda x.x\,t_1\,t_2\,t_3$.

12.4 Confluence

A λ-term may contain several redexes. We have the choice of which one to reduce first. When we make one step of β-reduction, some of the redexes that were there in the beginning may disappear, some may be duplicated into many copies, some new ones may be created. Although we say that β-reduction is a "simplification" of the term, in the sense that we eliminate a pair of consecutive abstraction and application by immediately performing the associated substitution, the resulting reduced term is not always simpler. It may actually be much longer and more complicated.

Therefore it is not obvious that, if we choose different redexes to simplify, we will eventually get the same result. It is also not clear whether the reduction of a term will eventually terminate.

The first property is anyway true. But there are indeed terms whose reduction does not terminate.

Theorem 12.1 (Confluence) *Given any λ-term t, if t_1 and t_2 are two reducts of it, that is $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$; then there exists a common reduct t_3 such that $t_1 \rightsquigarrow^* t_3$ and $t_2 \rightsquigarrow^* t_3$.*

Proof. TO BE COMPLETED $\square$

Definition 12.2 *A* normal form *is a λ-term that doesn't contain any redexes. A term* weakly normalizes *if there is a sequence of reduction steps that ends in a normal form. A term* strongly normalizes *if any sequence of reduction steps eventually ends in a normal form.*

There exist terms that do not normalize. The most famous one is a very short expression that reduces to itself:

$$\begin{aligned}\omega &:= (\lambda x.x\,x)\,(\lambda x.x\,x)\\ &\rightsquigarrow (x\,x)[x := \lambda x.x\,x] = (\lambda x.x\,x)\,(\lambda x.x\,x) = \omega\\ &\rightsquigarrow \omega \rightsquigarrow \cdots.\end{aligned}$$

There are also terms that grow without bound when we reduce them, for example:

$$\begin{aligned}(\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x) &\rightsquigarrow (\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x)\\ &\rightsquigarrow (\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x)\,(\lambda x.x\,x\,x)\\ &\rightsquigarrow \cdots.\end{aligned}$$

Here is an example of a term that weakly normalizes but doesn't strongly normalize:

$$(\lambda x.\lambda y.x)\,(\lambda z.z)\,\omega.$$

This terms applies the first projection function to to arguments. If we immediately reduce the application of the projection, the argument ω disappear and we are left with the identity function, which is a normal form:

$$(\lambda x.\lambda y.x)\,(\lambda z.z)\,\omega \rightsquigarrow \lambda z.z.$$

However, if we try to reduce the redex inside the second argument, we don't make any progress and we could continue reducing it forever.

12.5 Recursion

We have seen how to define some basic arithmetic functions as λ-terms: addition plus, multiplication mult, exponentiation exp. With a little effort you may be able to define the predecessor function and subtraction. what about (whole) division, maybe using the Euclid algorithm?

If we want to use the λ-calculus as a complete programming language, there should be a way to define any computable function. This is indeed possible. In fact there is a single λ-term, called *the* Y *combinator*, that allows us to use unrestricted recursion:

$$\mathsf{Y} := \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)).$$

This definition is inspired by the non-normalizing term ω. In fact we have that $\mathsf{Y}\,\mathsf{id} \rightsquigarrow^* \omega$.

The most striking property of Y is that it computes a *fixed point* for every term F:

$$\mathsf{Y}\,F \rightsquigarrow^* F\,(\mathsf{Y}\,F).$$

This is not exactly true: to be precise, $\mathsf{Y}\,F \rightsquigarrow (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)) =: \mathsf{fix}_F$ and $\mathsf{fix}_F \rightsquigarrow^* F\,(\mathsf{fix}_F)$. If we keep reducing, we get an infinite sequence of applications of F:

$$\mathsf{fix}_F \rightsquigarrow^* F\,\mathsf{fix}_F \rightsquigarrow^* F\,(F\,\mathsf{fix}_F) \rightsquigarrow^* F\,(F\,(F\,\mathsf{fix}_F)) \rightsquigarrow^* \cdots$$

To define a recursive function, we just need to encode a single step of it as a term F and then use Y to iterate that step as many times as it is necessary to get a result.

Here is, for example, the definition of the factorial:

$$\begin{aligned}&\mathsf{fact}_{\mathsf{step}} := \lambda f.\lambda n.\mathsf{if}\,(\mathsf{isZero}\,n)\,\overline{1}\,(\mathsf{mult}\,n\,(f\,(\mathsf{pred}\,n)))\\ &\mathsf{fact} := \mathsf{Y}\,\mathsf{fact}_{\mathsf{step}}.\end{aligned}$$

12.6 The universality of λ-calculus

The λ-calculus was invented to answer the question: "What does it mean that a problem is effectively solvable?" Up to that point the notion of a question being answerable by an precise method was left to intuition. Throughout the history of mathematics many difficult problems were posed. When someone claimed to have a solution, expert mathematicians would analyze it and come to an informed option about whether it was correct and precise.

But at the beginning of the 20th century, David Hilbert had formulated the challenge of defining exactly what we mean by a precise method: can we give a mathematically rigorous definition of what an effective procedure is? Both Alonzo Church and Alan Turing worked towards a realization of Hilbert's challenge and they both came up with a fully satisfying solution.

But their two theories looked completely different. Which one would be the "true" answer to Hilbert's question? It turned out that, beyond their formal difference, Turing machines and λ-terms are equally expressive. The same set of computable functions can be implemented in both. Turing himself proved this when he discovered Church's work and added a sketch of the proof as an Appendix to his article about computable numbers.

Around the same time, other models of computation were proposed. Kurt Gödel formulated the notion of μ-*recursive* function: any function on the natural numbers defined by certain forms of recursive definition. Stephen Kleene, a student of Church, proved that these functions are exactly those definable in the λ-calculus and went on the develop a rich theory of computation based on them.

Another of Church's students, J. Barkley Rosser, was the first to clearly formulate the notion that these three definitions were equivalent realizations of the informal notion of an effective computation method: "All three definitions are equivalent, so it does not matter which one is used."

Later, when the first digital computers were constructed, the Hungarian-American mathematician John von Neumann proposed a model that is more realistic and describes the actual architecture of real computers: the *register machine*. This notion was also equivalent to the previous ones. Many other different characterizations have been proposed since and they all turned out to be equivalent.

The statement that every effective definition of computable processes is equivalent to the λ-calculus (or to Turing machines or to μ-recursive functions and so on) is known as *the Church-Turing Thesis* (Stephen Kleene was the one who actually first formulated and named it).

The λ-calculus is not just a mathematical abstraction. All functional programming languages are based on (a typed version of) it. Lisp, ML, Haskell, OCaml are the most well-known. Functional programming has been so successful that traditional imperative languages are starting to introduce function abstraction as a basic feature in their definition. For example JavaScript and Python contain first-class functions as part of their definition.

12.7 Exercises

Exercise 12.1

Consider the following λ-terms: $\mathsf{nand\text{-}pair}$ is a function on pairs of Booleans, $\mathsf{nand\text{-}fun}$ a function from Church Numerals to pairs of Booleans:

$$\begin{aligned}\mathsf{nand\text{-}pair} &= \lambda p.\langle \mathsf{not}\,(\mathsf{and}\,(p\,\mathsf{true})\,(p\,\mathsf{false})), p\,\mathsf{true}\rangle \\ \mathsf{nand\text{-}fun} &= \lambda n.n\,\mathsf{nand\text{-}pair}\,\langle \mathsf{false}, \mathsf{false}\rangle\end{aligned}$$

1. What values do the following terms reduce to?

$$\begin{array}{ll}\mathsf{nand\text{-}pair}\,\langle \mathsf{true},\mathsf{true}\rangle \rightsquigarrow^* ? & \mathsf{nand\text{-}pair}\,\langle \mathsf{false},\mathsf{true}\rangle \rightsquigarrow^* ? \\ \mathsf{nand\text{-}pair}\,\langle \mathsf{true},\mathsf{false}\rangle \rightsquigarrow^* ? & \mathsf{nand\text{-}pair}\,\langle \mathsf{false},\mathsf{false}\rangle \rightsquigarrow^* ?\end{array}$$

2. Show the steps of reduction in the computation of $(\mathsf{nand\text{-}fun}\,\overline{4})$. [You can use the previous reductions and those from the lecture notes as single steps.]
3. Give an informal definition of what $\mathsf{nand\text{-}fun}$ does: for which numbers n does $(\mathsf{nand\text{-}fun}\,\overline{n}) \rightsquigarrow^* \langle \mathsf{true},\mathsf{true}\rangle$?

Exercise 12.2

Write a λ-term that implements the following function:

$$\begin{aligned}&\mathsf{thrFib} : \mathbb{N} \to \mathbb{N} \\ &\mathsf{thrFib}\,0 = 0 \\ &\mathsf{thrFib}\,1 = 0 \\ &\mathsf{thrFib}\,2 = 1 \\ &\mathsf{thrFib}\,(n+3) = \mathsf{thrFib}\,n + \mathsf{thrFib}\,(n+1) + \mathsf{thrFib}\,(n+2)\end{aligned}$$

[Use the same idea that was used in the lecture to define the Fibonacci numbers: first write an auxiliary function that returns a triple of numbers and then extract the first.]

13 Algorithmic Complexity

The undecidability of the Halting Problem shows that there are tasks that are impossible to accomplish using any computing machine. This is an intrinsic limitation of computation. But even for some problems for which algorithmic solutions are known, it may be impossible to compute them effectively. In some cases, the search for a solution is so complex that it would take a time longer than the age of the universe to get the answer. We can always hope for dramatic improvements in the power and speed of computers, but some tasks seem to have an essentially intractable complexity.

In talking about the time complexity of algorithms, we want to abstract away from implementation details. The exact number of steps needed to compute a result may depend on the model of computation: the same algorithm, realized as a Turing Machine or as a λ-term, will result in different number of head operations of the machine and reduction steps of the λ-term. The exact time of the computation will depend on the specific machine on which it runs: even the exact same program in the exact same language will take different times on different computers.

But beyond these differing details, the *complexity classes* of algorithms are quite constant across different models of computation and different computing devices. A complexity class characterizes algorithms that have a relation between size of the input and time of computation that can be expressed by certain functions. Although they are traditionally formulated in terms of Turing Machines, we can show that realizations in different paradigms have similar behaviour.

The most important complexity classes are $\mathcal{P}$ and $\mathcal{NP}$. $\mathcal{P}$ stands for *Polynomial Complexity*: A problem is in $\mathcal{P}$ if it can be solved by a Turing Machine whose running time is at most a polynomial in the size of the input. $\mathcal{NP}$ stands for *Non-deterministic Polynomial Complexity*: A problem is in $\mathcal{NP}$ if it can be solved by a Non-deterministic Turing Machine (NTM) whose running time is at most a polynomial in the size of the input. Remember that an NTM may have overlapping transitions: in certain configurations, the machine may have several allowed steps. In every run, it will randomly choose which transition to apply. A word is accepted if there exists one possible computation (among all allowed by the non-deterministic choices) that reaches an accepting state.

There are two alternative and equivalent ways of seeing the class $\mathcal{NP}$. A problem is in $\mathcal{NP}$ if it is solvable in polynomial time by a *parallel computer* which is allowed to spawn several parallel computations. Each computation is independent and must terminate in polynomial time; it is sufficient that the solution is found by one of the computations. Although the running time is polynomial, there may be an exponential blow-up of the number of parallel computations that need to run simultaneously.

The second alternative explanation of the class $\mathcal{NP}$ is that a problem is in it if we can verify solutions in polynomial time. There is an algorithm that, when given as input an instance of the problem and a potential solution, terminates in polynomial time and gives a positive answer if and only if the solution is correct.

Intuitively, it seems obvious that a problem that belongs to $\mathcal{NP}$ doesn't necessarily belong to $\mathcal{P}$: being able to find a solution in polynomial time by using non-deterministic or parallel computations, or being able to verify a solution in polynomial time, doesn't mean that we can generate a solution deterministically in polynomial time. There may be an exponential number of non-deterministic or parallel computations and an exponential number of potential solutions to verify.

However, until now nobody managed to prove that $\mathcal{P} \neq \mathcal{NP}$. It is the most famous open problem in theoretical computer science. There are strong reasons to believe that the two classes are distinct. There is a subclass of $\mathcal{NP}$, called $\mathcal{NP}$-complete, that contains problems that are in a sense universal with respect to this issue. A problem is in $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and every other problem in $\mathcal{NP}$ can be reduced to it in polynomial time. This means that if somebody finds a polynomial-time algorithm to solve one of these problems, then automatically all $\mathcal{NP}$ problems will be solvable in polynomial time. To date, thousands of different problems from disparate branches of computer science and mathematics have been proved to be $\mathcal{NP}$-complete. It seems highly unlikely that we could solve them all at once with the same algorithm.

13.1 The Satisfiability Problem

The first problem to be proved $\mathcal{NP}$-complete was Boolean Satisfiability (in short SAT). An instance of the problem is a propositional formula, that is, an expression that uses variables, the propositional connectives for conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$), and parentheses. An example is the formula $(x_1 \vee \neg x_2) \wedge \neg x_1$.

Solving an instance of SAT means determining if there is an assignment of truth values to the variables that makes the formula true. In the example above the assignment $[x_1 \mapsto \mathsf{false}, x_2 \mapsto \mathsf{false}]$ is such a solution. Now consider the following more complicated formula in three variables:

$$f = (\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_2) \wedge \neg(x_3 \wedge \neg x_1) \wedge (x_1 \wedge \neg x_2 \vee x_2 \wedge \neg x_3).$$

There is no assignment of truth values to x_1, x_2, x_3 that makes f true. To check this, we must try all potential solutions:

$$\begin{array}{ll}
[x_1 \mapsto \mathsf{true}, x_2 \mapsto \mathsf{true}, x_3 \mapsto \mathsf{true}] & \Longrightarrow f \mapsto \mathsf{false} \\
[x_1 \mapsto \mathsf{true}, x_2 \mapsto \mathsf{true}, x_3 \mapsto \mathsf{false}] & \Longrightarrow f \mapsto \mathsf{false} \\
[x_1 \mapsto \mathsf{true}, x_2 \mapsto \mathsf{false}, x_3 \mapsto \mathsf{true}] & \Longrightarrow f \mapsto \mathsf{false} \\
[x_1 \mapsto \mathsf{true}, x_2 \mapsto \mathsf{false}, x_3 \mapsto \mathsf{false}] & \Longrightarrow f \mapsto \mathsf{false} \\
[x_1 \mapsto \mathsf{false}, x_2 \mapsto \mathsf{true}, x_3 \mapsto \mathsf{true}] & \Longrightarrow f \mapsto \mathsf{false} \\
[x_1 \mapsto \mathsf{false}, x_2 \mapsto \mathsf{true}, x_3 \mapsto \mathsf{false}] & \Longrightarrow f \mapsto \mathsf{false} \\
[x_1 \mapsto \mathsf{false}, x_2 \mapsto \mathsf{false}, x_3 \mapsto \mathsf{true}] & \Longrightarrow f \mapsto \mathsf{false} \\
[x_1 \mapsto \mathsf{false}, x_2 \mapsto \mathsf{false}, x_3 \mapsto \mathsf{false}] & \Longrightarrow f \mapsto \mathsf{false}
\end{array}$$

In general, to look for a solution or to verify that there is no solution, we need to check all the possible assignments of truth values to the variables. If there are n variables in the formulas, then we must check 2^n assignments. Therefore checking them all takes a time that is exponential in the number of variables, so exponential in the side of the input. On the other hand, it is straightforward to verify whether an assignment gives a solution or not.

13.2 Time Complexity

Let's define time complexity exactly using Turing Machines.

Definition 13.1 *A Turing Machine M is said to have* time complexity f, *a function $\mathbb{N} \to \mathbb{N}$, if, whenever we run M on an input of size n, it will take at most $f(n)$ steps to terminate.*

The *number of steps* is measured by counting the movement of the reading/writing head of M, as given by the next-state relation $\underset{M}{\vdash}$.

This is a *worst case* measure of the running time of the machine: It is possible that M will terminate in less than n steps on some (or all) inputs. So f provides just an upper bound. We are interested in getting the answer within a certain time constraint.

As we know, any "problem" can be expressed precisely as a language recognition task. Each instance of the problem is specified by giving a description in the form of a word/list of symbols. So defining complexity classes for problems is the same as defining them for languages.

Definition 13.2 *A language L is in the class $\mathcal{P}$ if there is a Turing Machine M that decides L and has a time complexity expressible by a polynomial function.*

In the previous two definitions, we were talking about *deterministic* Turing machines. In any given instantaneous description of M, the transition function determines uniquely the step to be performed.

We are also going to consider *non-deterministic* Turing machines. These may have several possible transitions in the same instantaneous description. The transition function has the type:

$$\delta \in Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$$

So, if M is in state q and the head is reading a symbol x, the transition $\delta(q, x)$ is a set of triples, each one specifying a different action. For example, if $\delta(q, x) = \{(q_1, y, \mathrm{L}), (q_2, z, \mathrm{R})\}$, then the machine may either write a y, move left and go into state q_1 or write a z, move right and go into state q_2. When we run the machine, in these situation it will randomly choose which of the several possible steps to perform. We say that such a machine accepts a word if there is one run (among the many possible) that leads to an accepting state.

Definition 13.3 *A language L is in the class $\mathcal{NP}$ if there is a non-deterministic Turing Machine M that decides L and has a time complexity expressible by a polynomial function. In this case the polynomial bound must apply to all possible runs of M on a given input word.*

Equivalently, we can express membership in the class $\mathcal{NP}$ using deterministic Turing machines that take as input both a word and a certificate that proves that the word belongs to L. A certificate can be any string of symbols that the machine may interpret as evidence.

Theorem 13.4 *A language L is in the class $\mathcal{NP}$ if there is a deterministic Turing Machine M and a polynomial function f with the following properties:*

- *When we run M on a tape containing two inputs w and v (separated by a blank), it always terminates in a number of steps smaller than $f(|w|)$;*
- *If $w \in L$, then there exists some v such that M gives a positive answer when run on w and v;*

- *If $w \notin L$, then for every v, M always gives a negative answer when run on w and v.*

A string v such that M gives a positive answer when run on w and v is a *certificate* or proof that w is in L. If you think of L as representing some problem and w as an instance of it, then v can be thought of as a solution for w.

For example, given an instance of SAT, a certificate is an assignment of truth values to the variables that makes the formula true. We can define a Turing Machine with polynomial time complexity that, when run on a formula and an assignment of values to the variables, determines whether the formula computes to true under that assignment.

Theorem 13.5 SAT *is in $\mathcal{NP}$.*

13.3 $\mathcal{NP}$-completeness

Among the $\mathcal{NP}$ problems there are some that are *universal* in the sense that all $\mathcal{NP}$ problems can be reduced to them in polynomial time.

Definition 13.6 *A language L_0 is in $\mathcal{NP}$-complete if it is $\mathcal{NP}$ and, for every language L_1 in $\mathcal{NP}$ there exists a Turing Machine M that runs in polynomial time, with the following property:*

When we run M on a word w_1, it will terminate with the tape containing a world w_0 such that $w_1 \in L_1$ if and only if $w_0 \in L_0$.

We say that L_1 is reduced to L_0 in polynomial time.

Seeing L_0 and L_1 as encodings of problems P_0 and P_1, the definition says that we can turn every problem of P_1 into a problem of P_0 with the same solution.

The great turning point in the study of the time complexity of algorithms came with the discovery that there exist some $\mathcal{NP}$-complete problems.

Theorem 13.7 (Stephen Cook, 1971) SAT *is $\mathcal{NP}$-complete.*

Because Cook's Theorem, many other problems have been shown to be $\mathcal{NP}$-complete. They now run in the thousands. Here's a brief description of some famous ones.

THE TRAVELLING SALESMAN A salesman must visit n different towns. He has a table giving the distances between any pair of towns. He has a budget that he can use to buy petrol, which will allow him to travel a maximum length. Is there a route around all the towns with length shorter or equal to that allowed by the budget?

SUBSET SUM Given a set of (positive and negative) integers, is there a non-empty subset of it that sums up to zero? This is a special case of the *knapsack* problem that consists in maximizing the value of a set of objects that can fit into a knapsack with a limited capacity.

GRAPH COLOURING Given a graph and a fixed number of colours, is it possible to colour all the nodes of the graph so that two nodes of the same colour are not linked by an edge?

To this day, it is still unknown whether there are $\mathcal{NP}$ problems that are not in $\mathcal{P}$. All attempts to find a polynomial algorithm that solves any $\mathcal{NP}$-complete problem have failed. Also all attempts to prove that such an algorithm doesn't exist have failed.

If we were to discover such an algorithm, it would automatically give us a way to solve all $\mathcal{NP}$-complete problems in polynomial time: we can just reduce them all to the one we can solve. For this reason, most mathematicians and computer scientists think that it is impossible, but the question is still open and may be *the greatest mystery in computer science:*

$$\mathcal{P} = \mathcal{NP} \quad ?$$

13.4 Exercises

Exercise 13.1

This and the next exercise concerns programing a SAT solver in Haskell. The first exercise is to write a program that *checks* whether a proposed solution to an instance of SAT is correct. Use the following type definitions:

```
data SAT = Var Int | Not SAT | And SAT SAT | Or SAT SAT

type Assignment = [Bool]
```

`SAT` represents Boolean formulas with variables `Var 0`, `Var 1`, `Var 2`, `Var 3` and so on. An assignment is a list of Booleans, giving the values to some of the variables. For example `[True,False,False,True]` assigns the values: `Var 0 = True`, `Var 1 = False`, `Var 2 = False`, `Var 3 = True`. The other variables are left without value.

Write an evaluation function:

```
evaluate :: SAT -> Assignment -> Bool
```

It uses an assignment of values to variables to evaluate a formula. (If the formula contains variables with indices larger or equal to the length of the assignment, you can leave it undefined.)

Exercise 13.2

This exercise concerns writing a program that *decides* the solvability of an instance of SAT. We will break the task down into several steps.

1. Write a function that gives the highest index of a variable occurring in a formula:

   ```
   varNum :: SAT -> Int
   ```

2. Write a function that generates all the assignments for all variables up to a given index:

   ```
   allAssign :: Int -> [Assignment]
   ```

   For example `allAssign 2` should give all the possible assignments for variables `Var 0`, `Var 1` and `Var 2`:

```
allAssign 2 = [[True,True,True],   [True,True,False],
               [True,False,True],  [True,False,False],
               [False,True,True],  [False,True,False],
               [False,False,True], [False,False,False]]
```

3. Write a function that, for a given formula, verifies if there is one assignment on which the formula evaluates to `True`:

```
satisfiable :: SAT -> Bool
```

4. Define a function that actually returns the solution, if it exists:

```
solution :: SAT -> Maybe Assignment
```

Exercise 13.3

Discuss informally the complexity properties of the programs that you wrote.

1. Give an informal description of the steps of computation of `evaluate` and explain why it runs in polynomial time on the length of the input. Use this to argue that SAT is in the class $\mathcal{NP}$.
2. Give an informal description of the steps of computation of `satisfiable` and explain why it runs in exponential time on the length of the input. *(If you managed to write a polynomial-time program, congratulations: you solved $\mathcal{P} = \mathcal{NP}$!)*
3. Explain what it means that SAT is $\mathcal{NP}$-complete and what relevance this has for the $\mathcal{P} = \mathcal{NP}$ question.

A Model Answers to Exercises

Answer to Exercise 2.1

1. The language of all words over the alphabet $\{3, 5, 7, 9\}$ of length at least one and at most two.
2. $L = \{3, 5, 7, 9, 33, 35, 37, 39, 53, 55, 57, 59, 73, 75, 77, 79, 93, 95, 97, 99\}$
3. $|L_1| = \sum_{i=m}^{n} |\Sigma_1|^i$
(Note that the "big sigma" here is the standard arithmetic sum operator.) While the answer above is fine, note that this is just a geometric series, for which the sum easily can be stated in closed form. See e.g. `http://en.wikipedia.org/wiki/Geometric_series`, or note the following. Assuming $r \neq 1$:
$$(\sum_{i=m}^{n} r^i) + r^{n+1} = r^m + \sum_{i=m+1}^{n+1} r^i = r^m + r\sum_{i=m}^{n} r^i$$
Thus:
$$(\sum_{i=m}^{n} r^i)(1-r) = r^m - r^{n+1}$$
giving
$$\sum_{i=m}^{n} r^i = \frac{r^m - r^{n+1}}{1-r}$$
Finally, substituting $|\Sigma_1|$ for r, we conclude
$$|L_1| = \frac{|\Sigma_1|^m - |\Sigma_1|^{n+1}}{1 - |\Sigma_1|}$$
when $|\Sigma_1| \neq 1$. If $|\Sigma_1| = 1$, we have $|L_1| = n - m + 1$.
Strictly speaking, we should also note that neither of the above formulations cover the case when $\Sigma_1 = \emptyset$ and $m = 0$. This is because 0^0 is undefined. However, $\emptyset^0 = \{\epsilon\}$, which means $|L_1| = 1$ (for any $n \geq 0$). But this is a subtle technical point, and it is often the case that authors insist that an alphabet be a *non-empty* finite set (e.g. [HMU3]), making it kind of a moot point as well. Again, the simple answer $|L_1| = \sum_{i=m}^{n} |\Sigma_1|^i$ or something equivalent is fine.
4. $|L_1| = \sum_{i=3}^{7} |\Sigma|^i = \sum_{i=3}^{7} 4^i = 64 + 256 + 1024 + 4096 + 16384 = 21824$
or
$|L_1| = \frac{4^3 - 4^{7+1}}{1-4} = \frac{64-65536}{-3} = 21824$

Answer to Exercise 2.2

1. $L_3 = \{\epsilon, b, ac\} \cup \{a, b, ca\} = \{\epsilon, a, b, ac, ca\}$
2. $L_4 = \{\epsilon, b, ac\}\{\epsilon\}(\{a, b, ca\} \cap \{\epsilon, b, ac\}) = \{\epsilon, b, ac\}\{\epsilon\}\{b\} = \{\epsilon, b, ac\}\{b\} = \{b, bb, acb\}$
3. $L_5 = L_3\emptyset L_4 = \emptyset L_4 = \emptyset$

Answer to Exercise 2.3

1.

$$
\begin{aligned}
L_3 &= \{\epsilon, b, bb\} \cap \{a, ab, abc\} \\
&= \emptyset
\end{aligned}
$$

2.

$$
\begin{aligned}
L_4 &= (\{a, ab, abc\}\{\epsilon\}\{\epsilon, b, bb\}) \cap \Sigma^* \\
&= [\ \{\epsilon\} \text{ is the unit of concatenation of languages }] \\
& (\{a, ab, abc\}\{\epsilon, b, bb\}) \cap \Sigma^* \\
&= \{a, ab, abc, ab, abb, abcb, abb, abbb, abcbb\} \cap \Sigma^* \\
&= \{a, ab, abc, abb, abcb, abbb, abcbb\} \cap \Sigma^* \\
&= [\ \Sigma^* \text{ is all possible words over } \Sigma \text{ and thus the unit for intersection }] \\
& \{a, ab, abc, abb, abcb, abbb, abcbb\}
\end{aligned}
$$

3.

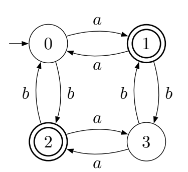
$$
\begin{aligned}
L_5 &= L_3\emptyset \cap L_4 \\
&= [\ \emptyset \text{ is the zero of concatenation of languages }] \\
& \emptyset \cap L_4 \\
&= \emptyset
\end{aligned}
$$

Answer to Exercise 2.4

$L = \{\epsilon, a, b, aa, ab, ba, bb, bc, aaa, aab, aba, abb, abc, baa, bab, bba, bbb, bbc, bca, bcb\}$

Answer to Exercise 3.1

1. DFA A



2.

w	$w \in L(A)$
ϵ	no
b	yes
$abaab$	yes
$bababbba$	no

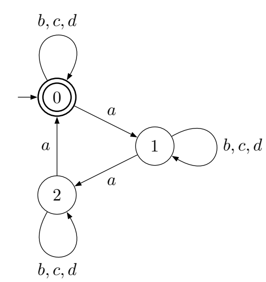
3.

$$
\begin{aligned}
\hat{\delta}_A(0, abba) &= \hat{\delta}_A(\delta_A(0, a), bba) && \text{def. } \hat{\delta}_A \\
&= \hat{\delta}_A(1, bba) && \text{because } \delta_A(0, a) = 1 \\
&= \hat{\delta}_A(\delta_A(1, b), ba) && \text{def. } \hat{\delta}_A \\
&= \hat{\delta}_A(3, ba) && \text{because } \delta_A(1, b) = 3 \\
&= \hat{\delta}_A(\delta_A(3, b), a) && \text{def. } \hat{\delta}_A \\
&= \hat{\delta}_A(1, a) && \text{because } \delta_A(3, b) = 1 \\
&= \hat{\delta}_A(\delta_A(1, a), \epsilon) && \text{def. } \hat{\delta}_A \\
&= \hat{\delta}_A(0, \epsilon) && \text{because } \delta_A(1, a) = 0 \\
&= 0 && \text{def. } \hat{\delta}_A
\end{aligned}
$$

4. $L(A)$ contains all words over $\{a, b\}$ in which the number of a's is even and the number of b's is odd, or vice versa. But that's the same as saying all the words over $\{a, b\}$ containing an odd number of symbols. Which in turn suggests there is a DFA with fewer states that accepts the same language. (Can you find it?)

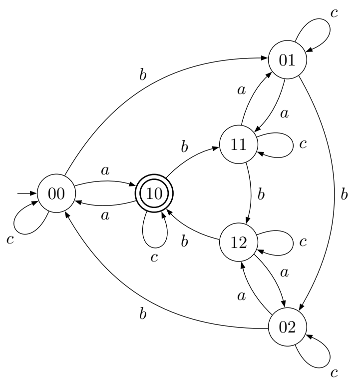
Answer to Exercise 3.2

We need to count the number of a's modulo 3, i.e. we need to keep track of whether the remainder when we divide the total number of a's seen so far by 3 is 0, 1, or 2. Thus we need 3 states. They are named 0, 1, and 2 below, to indicate said remainder. When any symbol other than a is read, the machine does not change state as the number of a's seen remain unchanged. 0 should be the accepting state because a remainder of 0 indicates that the number of a's seen is a multiple of 3. Note that 0 is a multiple of 3. Thus the empty string is accepted, and the accepting state is thus also the initial state.



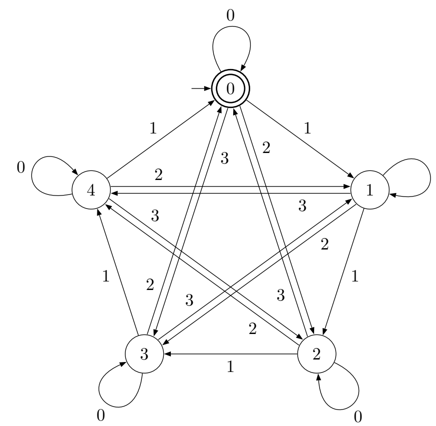
Answer to Exercise 3.3

We need to count the number of a's modulo 2 and the number of b's modulo 3, i.e. we need to keep track of whether the remainder when we divide the total number of a's seen so far by 2 is 0 or 1, and whether the remainder when we divide the total number of b's seen so far by 3 is 0, 1, or 2. Thus we need 6 states. We name the states 00, 01, 02, 10, 11, and 12 to indicate the number of a's modulo 2 and the number of b's modulo 3, respectively. (Of course, the names do not really matter, but it is helpful to be systematic and pick names that reflect the meaning of each state.) State 00 is the initial state, and state 10 the one and only accepting state.



Answer to Exercise 3.4

We need one state for each possible remainder when dividing by 5, i.e. 5 states. Let us label each state with the remainder in question. State 0 is thus both the initial and the only final state. We then just note that if the remainder when dividing the sum n of the symbols seen so far by 5 is r, and the next symbol is i, then the remainder of $n + i$ divided by 5 is just the remainder of $r + i$ divided by 5.



As the question does not specifically ask for a *transition diagram*, any *complete* representation of a DFA equivalent to the one above is OK:

- Transition diagram (like above)
- Transition table *with* initial and accepting states clearly indicated
- As a tuple $(Q, \Sigma, \delta, q_0, F)$ with all five components completely defined

However, an overly complicated solution, like too many states, is not acceptable.

For the sake of completeness, here is the transition table, with $\rightarrow$ indicating the start state and $*$ indicating the (in this case only) accepting state:

δ_D	0	1	2	3
$\rightarrow * 0$	0	1	2	3
1	1	2	3	4
2	2	3	4	0
3	3	4	0	1
4	4	0	1	2

And finally, mathematically as a tuple:

$$D = (Q_D, \Sigma_D, \delta_D, 0, F_D)$$

where

$$\begin{aligned} Q_D &= \{0, 1, 2, 3, 4\} \\ \Sigma_D &= \{0, 1, 2, 3\} \\ F_D &= \{0\} \end{aligned}$$

and the transition function δ_D, grouping by state, defined by:

$$\begin{aligned}
\delta_D(0,0) &= 0\\
\delta_D(0,1) &= 1\\
\delta_D(0,2) &= 2\\
\delta_D(0,3) &= 3\\
\delta_D(1,0) &= 1\\
\delta_D(1,1) &= 2\\
\delta_D(1,2) &= 3\\
\delta_D(1,3) &= 4\\
\delta_D(2,0) &= 2\\
\delta_D(2,1) &= 3\\
\delta_D(2,2) &= 4\\
\delta_D(2,3) &= 0\\
\delta_D(3,0) &= 3\\
\delta_D(3,1) &= 4\\
\delta_D(3,2) &= 0\\
\delta_D(3,3) &= 1\\
\delta_D(4,0) &= 4\\
\delta_D(4,1) &= 0\\
\delta_D(4,2) &= 1\\
\delta_D(4,3) &= 2
\end{aligned}$$

or, if you prefer to group by input symbols instead:

$$\begin{aligned}
\delta_D(0,0) &= 0\\
\delta_D(1,0) &= 1\\
\delta_D(2,0) &= 2\\
\delta_D(3,0) &= 3\\
\delta_D(4,0) &= 4\\
\delta_D(0,1) &= 1\\
\delta_D(1,1) &= 2\\
\delta_D(2,1) &= 3\\
\delta_D(3,1) &= 4\\
\delta_D(4,1) &= 0\\
\delta_D(0,2) &= 2\\
\delta_D(1,2) &= 3\\
\delta_D(2,2) &= 4\\
\delta_D(3,2) &= 0\\
\delta_D(4,2) &= 1\\
\delta_D(0,3) &= 3\\
\delta_D(1,3) &= 4\\
\delta_D(2,3) &= 0\\
\delta_D(3,3) &= 1\\
\delta_D(4,3) &= 2
\end{aligned}$$

Answer to Exercise 3.5

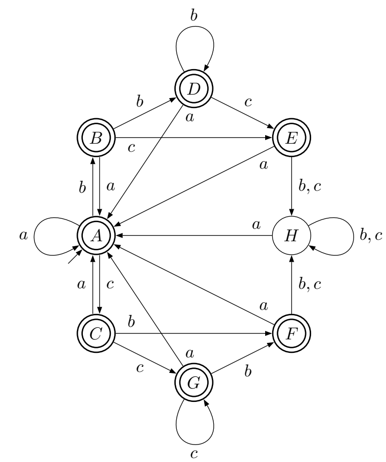
1. (a) $\epsilon \in L(A)$
   (b) $aaa \in L(A)$
   (c) $bbc \in L(A)$
   (d) $cbc \notin L(A)$
   (e) $abcacb \in L(A)$
2. Starting from $S_A = \{q_0, q_1, q_3\}$, the start state of $D(A)$, we compute $\hat{\delta}_A(S_A, x)$ for each $x \in \Sigma_A$. Whenever we encounter a state $P \subseteq Q_A$ of $D(A)$ that has not been considered before, we add P to the table and proceed to tabulate $\hat{\delta}_A(P, x)$ for each $x \in \Sigma_A$. We repeat the process until no new states are encountered. Finally, we identify the initial state ($\rightarrow$ to the left of the state) and all accepting states ($*$ to the left of the state). Note that a DFA state is accepting iff it contains at least one accepting NFA state (as this means it is *possible* to reach at least one accepting state on a given word, which means that word is considered to be in the language of the NFA).

	$\delta_{D(A)}$	a	b	c
$\rightarrow *$	$\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_3\} \cup \emptyset \cup \emptyset$ $= \{q_0, q_1, q_3\}$	$\{q_0\} \cup \{q_1\} \cup \{q_4\}$ $= \{q_0, q_1, q_4\}$	$\{q_0\} \cup \{q_2\} \cup \{q_3\}$ $= \{q_0, q_2, q_3\}$
$*$	$\{q_0, q_1, q_4\}$	$\{q_0, q_1, q_3\} \cup \emptyset \cup \emptyset$ $= \{q_0, q_1, q_3\}$	$\{q_0\} \cup \{q_1\} \cup \emptyset$ $= \{q_0, q_1\}$	$\{q_0\} \cup \{q_2\} \cup \emptyset$ $= \{q_0, q_2\}$
$*$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_3\} \cup \emptyset \cup \emptyset$ $= \{q_0, q_1, q_3\}$	$\{q_0\} \cup \emptyset \cup \{q_4\}$ $= \{q_0, q_4\}$	$\{q_0\} \cup \emptyset \cup \{q_3\}$ $= \{q_0, q_3\}$
$*$	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\} \cup \emptyset$ $= \{q_0, q_1, q_3\}$	$\{q_0\} \cup \{q_1\}$ $= \{q_0, q_1\}$	$\{q_0\} \cup \{q_2\}$ $= \{q_0, q_2\}$
$*$	$\{q_0, q_2\}$	$\{q_0, q_1, q_3\} \cup \emptyset$ $= \{q_0, q_1, q_3\}$	$\{q_0\} \cup \emptyset = \{q_0\}$	$\{q_0\} \cup \emptyset = \{q_0\}$
$*$	$\{q_0, q_4\}$	$\{q_0, q_1, q_3\} \cup \emptyset$ $= \{q_0, q_1, q_3\}$	$\{q_0\} \cup \emptyset = \{q_0\}$	$\{q_0\} \cup \emptyset = \{q_0\}$
$*$	$\{q_0, q_3\}$	$\{q_0, q_1, q_3\} \cup \emptyset$ $= \{q_0, q_1, q_3\}$	$\{q_0\} \cup \{q_4\}$ $= \{q_0, q_4\}$	$\{q_0\} \cup \{q_3\}$ $= \{q_0, q_3\}$
	$\{q_0\}$	$\{q_0, q_1, q_3\}$	$\{q_0\}$	$\{q_0\}$

(Note that we only needed to consider 8 states, a lot fewer than the $2^5 = 32$ possible states in this case. $32 - 8 = 24$ states are thus not reachable from the initial state.)
Giving simple names to the states resulting from the subset construction can facilitate drawing the transition diagram:

	$\delta_{D(A)}$	a	b	c
$\rightarrow *$	$\{q_0, q_1, q_3\} = A$	$\{q_0, q_1, q_3\} = A$	$\{q_0, q_1, q_4\} = B$	$\{q_0, q_2, q_3\} = C$
$*$	$\{q_0, q_1, q_4\} = B$	$\{q_0, q_1, q_3\} = A$	$\{q_0, q_1\} = D$	$\{q_0, q_2\} = E$
$*$	$\{q_0, q_2, q_3\} = C$	$\{q_0, q_1, q_3\} = A$	$\{q_0, q_4\} = F$	$\{q_0, q_3\} = G$
$*$	$\{q_0, q_1\} = D$	$\{q_0, q_1, q_3\} = A$	$\{q_0, q_1\} = D$	$\{q_0, q_2\} = E$
$*$	$\{q_0, q_2\} = E$	$\{q_0, q_1, q_3\} = A$	$\{q_0\} = H$	$\{q_0\} = H$
$*$	$\{q_0, q_4\} = F$	$\{q_0, q_1, q_3\} = A$	$\{q_0\} = H$	$\{q_0\} = H$
$*$	$\{q_0, q_3\} = G$	$\{q_0, q_1, q_3\} = A$	$\{q_0, q_4\} = F$	$\{q_0, q_3\} = G$
	$\{q_0\} = H$	$\{q_0, q_1, q_3\} = A$	$\{q_0\} = H$	$\{q_0\} = H$

3. We can now draw the transition diagram for $D(A)$:

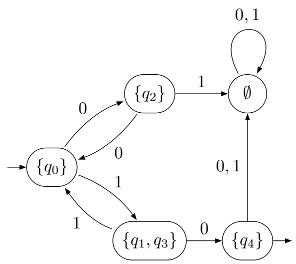


Answer to Exercise 3.6

1. Starting from $S_B = \{q_0\}$, the start state of $D(B)$, we compute $\hat{\delta}_B(S_B, x)$ for each $x \in \Sigma_B$. Whenever we encounter a state $P \subseteq Q_B$ of $D(B)$ that has not been considered before, we add P to the table and proceed to tabulate $\hat{\delta}_B(P, x)$ for each $x \in \Sigma_B$. We repeat the process until no new states are encountered. Finally, we identify the initial state ($\rightarrow$ to the left of the state) and all accepting states ($*$ to the left of the state). Note that a DFA state is accepting if it contains at least one accepting NFA state (as this means it is *possible* to reach at least one accepting state on a given word, which means that word is considered to be in the language of the NFA).

	$\delta_{D(B)}$	0	1
$\rightarrow$	$\{q_0\}$	$\{q_2\}$	$\{q_1, q_3\}$
	$\{q_2\}$	$\{q_0\}$	$\emptyset$
	$\{q_1, q_3\}$	$\emptyset \cup \{q_4\} = \{q_4\}$	$\{q_0\} \cup \emptyset = \{q_0\}$
	$\emptyset$	$\emptyset$	$\emptyset$
$*$	$\{q_4\}$	$\emptyset$	$\emptyset$

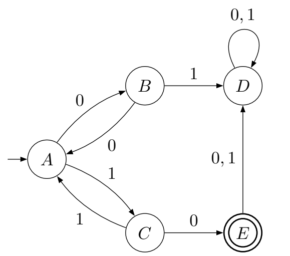
(In this case we only needed to consider 5 out of the $2^5 = 32$ possible states. $32 - 5 = 27$ states are thus not reachable from the initial state, and we do not need to worry about those.)
We can now draw the transition diagram for $D(B)$:



Accepting states have been marked by outgoing arrows in this case. That is an alternative to the double circle.

It is often convenient to give simple names to the states resulting from the subset construction as referring to the states by writing out the subsets in full can be a bit long-winded. These names can then be used when drawing the transition diagram:

	$\delta_{D(A)}$	0	1
$\rightarrow$	$\{q_0\} = A$	$\{q_2\}$	$\{q_1, q_3\}$
	$\{q_2\} = B$	$\{q_0\}$	$\emptyset$
	$\{q_1, q_3\} = C$	$\emptyset \cup \{q_4\} = \{q_4\}$	$\{q_0\} \cup \emptyset = \{q_0\}$
	$\emptyset = D$	$\emptyset$	$\emptyset$
$*$	$\{q_4\} = E$	$\emptyset$	$\emptyset$



Here the states were named using capital letters. But the choice is arbitrary, of course.

2. *General hint:* It is very easy to make simple mistakes when applying the subset construction. Thus, doing a sanity check on the result along the lines described here is always a good idea, even if you are not explicitly asked to, and even if you only do the check in your head!
   (a) The words 10 and 110010 are both accepted by the NFA B.
   The words ϵ and 1101 are both rejected by the NFA B.

(b)

word	*state sequence*	*last state*
10	A, C, E	accepting
110010	A, C, A, B, A, C, E	accepting
ϵ	A	not accepting
1101	A, C, A, B, D	not accepting

Thus the DFA $D(B)$ behaves like the NFA B at least for these four words, which should give us some reassurance as to the correctness of the answer.

Answer to Exercise 3.7

1. (a) $\epsilon \in L(C)$
   (b) $aa \in L(C)$
   (c) $bb \notin L(C)$
   (d) $abcabc \notin L(C)$
   (e) $abcabca \in L(C)$

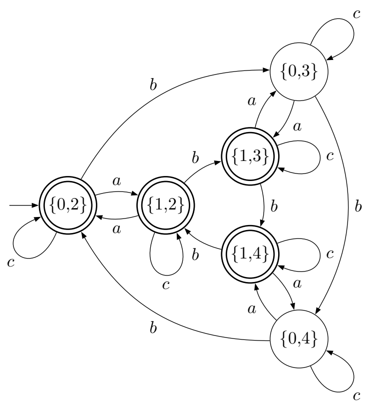
2. All words over Σ_C where the number of a's is odd or the number of b's is divisible by three (or both).

3. Starting from $S_C = \{0, 2\}$, the start state of $D(C)$, we tabulate $\delta_{D(C)}$ by computing the union of $\delta_C(q, x)$ over all q in a state of $D(C)$ for each symbol x in Σ_C, exploring new states as they emerge.

	$\delta_{D(C)}$	a	b	c
$\rightarrow *$	$\{0, 2\}$	$\{1\} \cup \{2\} = \{1, 2\}$	$\{0\} \cup \{3\} = \{0, 3\}$	$\{0\} \cup \{2\} = \{0, 2\}$
$*$	$\{1, 2\}$	$\{0\} \cup \{2\} = \{0, 2\}$	$\{1\} \cup \{3\} = \{1, 3\}$	$\{1\} \cup \{2\} = \{1, 2\}$
	$\{0, 3\}$	$\{1\} \cup \{3\} = \{1, 3\}$	$\{0\} \cup \{4\} = \{0, 4\}$	$\{0\} \cup \{3\} = \{0, 3\}$
$*$	$\{1, 3\}$	$\{0\} \cup \{3\} = \{0, 3\}$	$\{1\} \cup \{4\} = \{1, 4\}$	$\{1\} \cup \{3\} = \{1, 3\}$
	$\{0, 4\}$	$\{1\} \cup \{4\} = \{1, 4\}$	$\{0\} \cup \{2\} = \{0, 2\}$	$\{0\} \cup \{4\} = \{0, 4\}$
$*$	$\{1, 4\}$	$\{0\} \cup \{4\} = \{0, 4\}$	$\{1\} \cup \{2\} = \{1, 2\}$	$\{1\} \cup \{4\} = \{1, 4\}$

(Note that we only needed to consider 6 states. That is a lot fewer than the $2^5 = 32$ possible states in this case. $32 - 6 = 26$ states are thus not reachable from the initial state, and we do not need to worry about those.)

4. We can now draw the transition diagram for $D(C)$:



Note that $D(C)$ has the same structure as the DFA C above, except that this time each state corresponding to an odd number of a's *or* the number of b's being divisible by 3 is an accepting state.

Answer to Exercise 4.1

Note: these are not necessarily the only possibilities, nor necessarily the "simplest". But they are all fairly simple, and your answers should not be much more complicated.

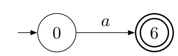
1. $(\mathbf{b}+\mathbf{c})^*\mathbf{a}(\mathbf{b}+\mathbf{c})^*$
2. $(\mathbf{a}+\mathbf{c})^*\mathbf{b}(\mathbf{a}+\mathbf{c})^*\mathbf{b}(\mathbf{a}+\mathbf{b}+\mathbf{c})^*$
3. $(\mathbf{a}+\mathbf{b})^*(\epsilon+\mathbf{c})(\mathbf{a}+\mathbf{b})^*(\epsilon+\mathbf{c})(\mathbf{a}+\mathbf{b})^*$
4. $(\mathbf{a}+\mathbf{b})^*(\mathbf{a}+\mathbf{c})^*$
5. $\mathbf{a}^*(\mathbf{b}\mathbf{a}^*\mathbf{c}+\mathbf{c}\mathbf{a}^*\mathbf{b})\mathbf{a}^*$
6. $\mathbf{c}^*(\mathbf{a}+\mathbf{b})((\mathbf{a}+\mathbf{b})\mathbf{c}^*(\mathbf{a}+\mathbf{b})+\mathbf{c})^*$
7. $(\mathbf{a}+\mathbf{b}+\mathbf{c})^*\mathbf{abba}(\mathbf{a}+\mathbf{b}+\mathbf{c})^*$

Answer to Exercise 4.2

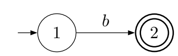
$$
\begin{array}{ll}
 & L((\mathbf{aa}+\epsilon\mathbf{b}^*\emptyset)(\mathbf{b}+\mathbf{c})) \\
= & \{L(EF)=L(E)L(F)\} \\
 & L(\mathbf{aa}+\epsilon\mathbf{b}^*\emptyset)L(\mathbf{b}+\mathbf{c}) \\
= & \{L(E+F)=L(E)\cup L(F)\}\ \text{(twice)} \\
 & (L(\mathbf{aa})\cup L(\epsilon\mathbf{b}^*\emptyset))(L(\mathbf{b})\cup L(\mathbf{c})) \\
= & \{L(EF)=L(E)L(F)\}\ \text{(three times)} \\
 & (L(\mathbf{a})L(\mathbf{a})\cup L(\epsilon)L(\mathbf{b}^*)L(\emptyset))(L(\mathbf{b})\cup L(\mathbf{c})) \\
= & \{L(E^*)=(L(E))^*\} \\
 & (L(\mathbf{a})L(\mathbf{a})\cup L(\epsilon)L(\mathbf{b})^*L(\emptyset))(L(\mathbf{b})\cup L(\mathbf{c})) \\
= & \{L(\mathbf{x})=\{x\},\ L(\emptyset)=\emptyset,\ L(\epsilon)=\{\epsilon\}\} \\
 & (\{a\}\{a\}\cup\{\epsilon\}\{b\}^*\emptyset)(\{b\}\cup\{c\}) \\
= & \{L\emptyset=\emptyset\ \text{(twice)}\} \\
 & (\{a\}\{a\}\cup\emptyset)(\{b\}\cup\{c\}) \\
= & \{\text{Set union}\} \\
 & \{a\}\{a\}\{b,c\} \\
= & \{\text{Concatenation of languages}\} \\
 & \{aab, aac\}
\end{array}
$$

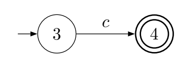
Answer to Exercise 4.3

Construct an NFA A for $(\mathbf{a}(\mathbf{b}+\mathbf{c}))^*$ according to the lecture notes. Start with the innermost subexpressions and then join the NFAs together step by step. (I have named the states according to how they will be named in the final NFA to make it easier to follow the derivation. It is OK to leave states unnamed to the end.) NFA for $\mathbf{a}$:

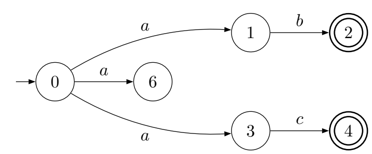


NFA for $\mathbf{b}+\mathbf{c}$

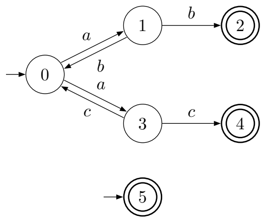




Join the above two NFAs to obtain an NFA for $\mathbf{a}(\mathbf{b}+\mathbf{c})$:



The last step is to carry out the construction corresponding to the $*$-operator. States 1 and 3 both immediately precede a final state, and we should thus add corresponding transition edges from those back to all start states. But there is only one start state, 0, so only one edge from each. Additionally, we must not forget to add an extra start state which is also final (here state 5) to ensure the NFA accepts ϵ. Finally, state 6 is manifestly now a "dead end" and can thus be eliminated:

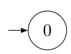


Note that the isolated state 5 also is part of the same automaton.

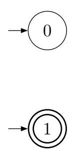
Answer to Exercise 4.4

States are named as they are introduced to make it easier to follow the construction, but it is OK to leave states unnamed to the end.

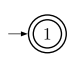
First construct an NFA for the subexpression $\emptyset$:



Then construct an NFA for the Kleene closure $\emptyset^*$. As there are no accepting states and thus no states immediately preceding any accepting state, there are no "loop edges" to add. But we do have to add an initial and accepting state to account for the fact that the automaton must accept the empty word, ϵ:

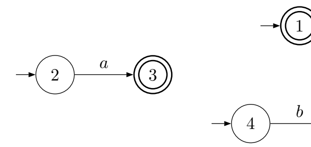


It is now clear that state 0 is a dead end, so we can eliminate it already now giving us the NFA

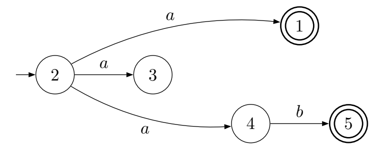


for the subexpression $\emptyset^*$. (This is exactly what we should expect as the expression is equivalent to ϵ.)

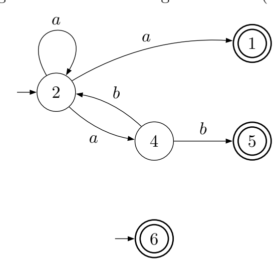
We can now consider the subexpression $(\mathbf{a}(\emptyset^* + \mathbf{b}))^*$. In addition to the NFA for $\emptyset^*$ constructed above, we first need NFAs for the atomic regular expressions $\mathbf{a}$ (to the left) and $\mathbf{b}$. We form an NFA for $\emptyset^* + \mathbf{b}$ by placing the NFAs for $\emptyset^*$ and $\mathbf{b}$ in parallel (to the right):



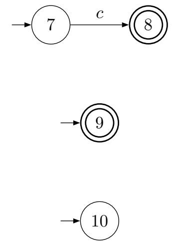
Join the above two NFAs to obtain an NFA for $\mathbf{a}(\emptyset^* + \mathbf{b})$, keeping in mind that the left DFA does not accept ϵ ("sub-case 1" in the lecture notes):



It is now clear state 3 is a dead end, so it can be removed prior to carrying out the construction for the Kleene closure (not forgetting one extra state for accepting ϵ) leaving us with the following NFA A for $(\mathbf{a}(\emptyset^* + \mathbf{b}))^*$:

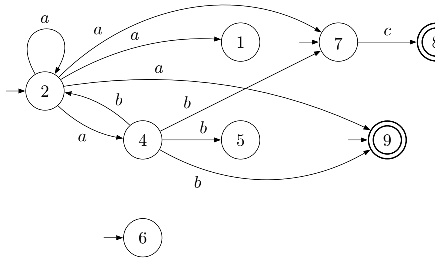


The NFA B for $(\mathbf{c} + \epsilon + \emptyset)$ is simply:

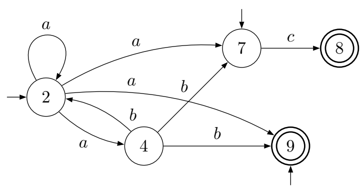


It's clear that state 10 will become a dead end, so it can be removed.

Now, joining A with B (less state 10), while keeping in mind that A *can* accept ϵ which means states 7 and 9 will remain initial states, gives:



It's now clear that states 1, 5, and 6 are all dead ends, so we can simplify by removing them and obtain the final NFA for $(\mathbf{a}(\emptyset^* + \mathbf{b}))^*(\mathbf{c} + \epsilon + \emptyset)$:



Note that there are *three* initial states: 2, 7, and 9. Finally, it is worth spending some time to ensure the final NFA makes sense by comparing the words it accepts with the words generated by the original regular expression.

Answer to Exercise 6.1

1. Straightforward, only the key steps are given. See below for an example of a complete proof. If n is the constant of the pumping lemma, consider a specific word $a^n b^m c^{n+m}$, where $m \geq 0$. This word clearly belongs to L_1, and it is sufficiently long to satisfy the requirements of the lemma. Once split into three parts according to the lemma, the middle part is going to be non-empty and it is going to consist of a's only. If the middle part is repeated, the resulting string is not going to have the right balance of a's, b's, and c's, and thus it is not going to belong to L_1.
2. Assume that L_2 is a regular language. Then, according to the pumping lemma for regular languages, there exists a constant n such that any word $w \in L_2$ that has length at least n ($|w| \geq n$) can be split into three parts, $w = xyz$, as follows:
   1. $y \neq \epsilon$
   2. $|xy| \leq n$
   3. $\forall k \in \mathbb{N} \,.\, xy^k z \in L_2$

   Consider the word[11] $w = a^{4n} b^{2n} c^n$. Clearly $w \in L_2$. Moreover $|w| = 7n \geq n$. The prerequisites of the lemma are thus fulfilled, and we know it is possible to split w into three parts x, y, and z satisfying the conditions of the lemma. Because our chosen word w starts with $4n$ a's, and because the combined length of the two first parts, x and y, is at most n, we know that x and y consists solely of a's. Thus $x = a^i$ and $y = a^j$ for some $i, j \in \mathbb{N}$ such that $i + j \leq n$. Furthermore, because y is not empty according to the lemma, we know that $j > 0$. z is whatever remains of w; i.e., $z = a^{(4n-i-j)} b^{2n} c^n$.
   Now consider words of the form $xy^k z$. According to the pumping lemma, these words belong to L_2 for any value of k. Let us consider a specific value for k, for example $k = 0$. The word $xy^0 z$ should belong to L. But $xy^0 z = xz = a^i a^{(4n-i-j)} b^{2n} c^n = a^{(4n-j)} b^{2n} c^n$. Because $j > 0$, it is now clear that there are *fewer* than twice as many a's as b's in this word, which thus *cannot* belong to L_2. We have reached a contradiction, and our assumption that L_2 is a regular language must be wrong. Thus L_2 is *not* a regular language, QED.

Answer to Exercise 7.1

1. (a)

$$\begin{array}{rcll} S & \underset{G}{\Rightarrow} & X & \text{by } S \to X \\ & \underset{G}{\Rightarrow} & \epsilon & \text{by } X \to \epsilon \end{array}$$

[11] Note how w is chosen: it is one specific word that obviously is a word in L_2, but that depends on the constant n in such a way that the length of w is at least n *whatever* n is. It is *completely wrong* to assume that n is some particular value! All we know is that n exists. Also, note that the structure of w was chosen to facilitate the proof. For example, a word $(aaaabbc)^n$ is also a word in L_2 with length at least n. But in this case, it is not going to be possible to show that *all* ways to divide the word into three parts that satisfy the constraints of the lemma are going to lead to a contradiction. By first establishing that n must be at least 7 (which can be done), we will see that a division of v into $x = \epsilon$, $y = aaaabbc$, and $z = (aaaabbc)^{n-1}$ always is a possibility, which in turn means that all words of the form $xy^k z$ *do* belong to L, and we do not get any contradiction.

or

$$\begin{array}{rcll} S & \underset{G}{\Rightarrow} & Y & \text{by } S \to Y \\ & \underset{G}{\Rightarrow} & \epsilon & \text{by } Y \to \epsilon \end{array}$$

When giving derivation sequences in a context-free grammar, it is normally *not* necessary to justify every single step as it is fairly obvious which production is being used. The justified derivation sequence above were just given for explanatory purposes. Thus, an answer like the following is perfectly OK too:

$$S \underset{G}{\Rightarrow} X \underset{G}{\Rightarrow} \epsilon$$

or even

$$S \Rightarrow X \Rightarrow \epsilon$$

as it is clear which grammar we are referring to from the context. However, to make it very clear how derivations work, we will give explicit justifications here.

(b)

$$\begin{array}{rcll} S & \underset{G}{\Rightarrow} & X & \text{by } S \to X \\ & \underset{G}{\Rightarrow} & aXb & \text{by } X \to aXb \\ & \underset{G}{\Rightarrow} & aaXbb & \text{by } X \to aXb \\ & \underset{G}{\Rightarrow} & aabb & \text{by } X \to \epsilon \end{array}$$

(c)

$$\begin{array}{rcll} S & \underset{G}{\Rightarrow} & Y & \text{by } S \to Y \\ & \underset{G}{\Rightarrow} & cYd & \text{by } Y \to cYd \\ & \underset{G}{\Rightarrow} & ccYdd & \text{by } Y \to cYd \\ & \underset{G}{\Rightarrow} & cccYddd & \text{by } Y \to cYd \\ & \underset{G}{\Rightarrow} & cccddd & \text{by } Y \to \epsilon \end{array}$$

2. No, $aaaddd \notin L(G)$. From the start symbol S, we can either derive X or Y. However, from X it is only possible to derive strings $a^n b^n$, while from Y it is only possible to derive strings $c^n d^n$, neither of which are words starting with a's and ending with d's.
3. $L(G) = \{a^n b^n \mid n \in \mathbb{N}\} \cup \{c^n d^n \mid n \in \mathbb{N}\}$

Answer to Exercise 7.2

The following is one possible grammar generating L:

$$\begin{array}{rcl} S & \to & XY \mid Z \\ X & \to & abXba \mid abX_1ba \\ X_1 & \to & bcX_1cb \mid bccb \\ Y & \to & dY \mid \epsilon \\ Z & \to & ddY \end{array}$$

S, X, X_1, Y, and Z are nonterminal symbols, S is the start symbol, and a, b, c, and d are terminal symbols.

(The following explanation is very detailed to make it easy to follow what is going on. An explanation of the key ideas is sufficient for full marks as long this explanation reflects a clear understanding of the construction.)

The grammar was constructed as follows. There are three constituent sets in the definition of the language L. Call them

$$\begin{array}{rcl} L_X & = & \{(ab)^m (bc)^n (cb)^n (ba)^m \mid m, n \geq 1\} \\ L_Y & = & \{d^n \mid n \geq 0\} \\ L_Z & = & \{d^n \mid n \geq 2\} \end{array}$$

Introduce a non-terminal for each, such that the set is generated by using that non-terminal as a start symbol; i.e. the nonterminal X corresponds to the set L_X etc. Then observe that L is the concatenation of the sets L_X and L_Y in union with L_Z; i.e. $L = L_X L_Y \cup L_Z$. This is captured by the two productions:

$$S \quad \to \quad XY \mid Z$$

The set L_Y, i.e. zero, one, or more d's, is described by the following recursive productions:

$$Y \quad \to \quad dY \mid \epsilon$$

The set L_Z is similar, except there has to be at least two d'd. We can obtain such a set by simply prefixing all words in the set L_Y by two d's. as follows:

$$Z \quad \to \quad ddY$$

(Obviously, there is nothing wrong by doing it from scratch, not "reusing" the productions for Y.)

Finally, as the words in L_X are strings with a balanced nesting of one pair of substrings (bc and cb) inside another (ab and ba), we need to introduce a "helper" non-terminal to deal with the inner nesting. Keeping in mind that each substring pair should occur at least once, we obtain:

$$\begin{array}{rcl} X & \to & abXba \mid abX_1ba \\ X_1 & \to & bcX_1cb \mid bccb \end{array}$$

Answer to Exercise 7.3

For $\phi = X$, the only possibility is $A = X$ and $\alpha = \gamma = \epsilon$. There are two possible production for X, meaning $\beta = aXb$ in one case and $\beta = ab$ in the other. For $\phi = XY$, we can either take $\alpha = \epsilon$, $A = X$, and $\gamma = Y$, or $\alpha = X$, $A = Y$, and $\gamma = \epsilon$. For the case where $A = X$ there are two possible productions: $X \to aXb$ and $X \to ab$, meaning that $\beta = aXb$ in one case and $\beta = ab$ in the other. For the case $A = Y$ there are also two possible productions, $Y \to bYc$ and $Y \to \epsilon$. A similar analysis can be made for $\phi = aXbYc$. For $\phi = cc$ we don't get any possibilities, because there is no nonterminal symbol in the word, and thus no way to chose a production in P. For $\theta = a$, finally, we note there is no production that yields a single a, and only one production that yields ϵ, $Y \to \epsilon$. Thus the only ways to directly derive a single a is from strings Ya and aY. We summarize in the following table:

$\phi = \alpha A \gamma$	α	A	γ	$A \to \beta$	$\theta = \alpha\beta\gamma$
X	ϵ	X	ϵ	$X \to aXb$	aXb
X	ϵ	X	ϵ	$X \to ab$	ab
XY	ϵ	X	Y	$X \to aXb$	$aXbY$
XY	ϵ	X	Y	$X \to ab$	abY
XY	X	Y	ϵ	$Y \to bYc$	$XbYc$
XY	X	Y	ϵ	$Y \to \epsilon$	X
$aXbYc$	a	X	bYc	$X \to aXb$	$aaXbbYc$
$aXbYc$	a	X	bYc	$X \to ab$	$aabbYc$
$aXbYc$	aXb	Y	c	$Y \to bYc$	$aXbbYcc$
$aXbYc$	aXb	Y	c	$Y \to \epsilon$	$aXbc$
Ya	ϵ	Y	a	$Y \to \epsilon$	a
aY	a	Y	ϵ	$Y \to \epsilon$	a

The pairs in question are thus

$$(X, aXb),\ (X, ab), (XY, aXbY),\ (XY, abY),\ (XY, XbYc),\ (XY, X),$$
$$(aXbYc, aaXbbYc),\ (aXbYc, aabbYc),\ (aXbYc, aXbbYcc),\ (aXbYc, aXbc),$$
$$(Ya, a),\ (aY, a)$$

Answer to Exercise 7.4

1. (a) Left-most derivation:

$$\begin{aligned} T &\Rightarrow F \Rightarrow P \Rightarrow (T) \Rightarrow (F) \Rightarrow (P) \Rightarrow (I) \Rightarrow (DI) \Rightarrow (7I) \Rightarrow (7DI) \\ &\Rightarrow (78I) \Rightarrow (78D) \Rightarrow (789) \end{aligned}$$

Sometimes left-most derivation steps are explicitly indicated by an *lm* subscript as follows:

$$T \underset{lm}{\Rightarrow} F \underset{lm}{\Rightarrow} P \underset{lm}{\Rightarrow} (T) \underset{lm}{\Rightarrow} \ldots \underset{lm}{\Rightarrow} (789)$$

Note that if the goal is just to derive a word, any derivation will do, not just a left-most one (unless you've been explicitly asked to provide a left-most derivation as here, of course). However, being systematic and sticking to e.g. left-most derivations where possible can help in avoiding mistakes. Anyway, to illustrate that there are other possibilities, here is a *right-most* derivation of the same word:

$$\begin{aligned} T &\underset{rm}{\Rightarrow} F \underset{rm}{\Rightarrow} P \underset{rm}{\Rightarrow} (T) \underset{rm}{\Rightarrow} (F) \underset{rm}{\Rightarrow} (P) \underset{rm}{\Rightarrow} (I) \underset{rm}{\Rightarrow} (DI) \\ &\underset{rm}{\Rightarrow} (DDI) \underset{rm}{\Rightarrow} (DDD) \underset{rm}{\Rightarrow} (DD9) \underset{rm}{\Rightarrow} (D89) \underset{rm}{\Rightarrow} (789) \end{aligned}$$

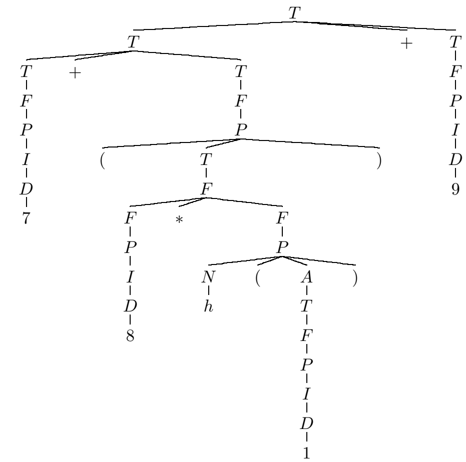
(b) Left-most derivation:

$$\begin{aligned} T &\Rightarrow T + T \Rightarrow F + T \Rightarrow P + T \Rightarrow I + T \\ &\Rightarrow D + T \Rightarrow 7 + T \Rightarrow 7 + F \Rightarrow 7 + F * F \\ &\Rightarrow 7 + P * F \Rightarrow 7 + N(A) * F \Rightarrow 7 + g(A) * F \Rightarrow 7 + g(T) * F \\ &\Rightarrow 7 + g(F) * F \Rightarrow 7 + g(F * F) * F \Rightarrow 7 + g(P * F) * F \\ &\Rightarrow 7 + g(I * F) * F \Rightarrow 7 + g(D * F) * F \Rightarrow 7 + g(3 * F) * F \\ &\Rightarrow 7 + g(3 * P) * F \Rightarrow 7 + g(3 * I) * F \Rightarrow 7 + g(3 * D) * F \\ &\Rightarrow 7 + g(3 * 5) * F \Rightarrow 7 + g(3 * 5) * P \\ &\Rightarrow 7 + g(3 * 5) * (T) \Rightarrow 7 + g(3 * 5) * (F) \Rightarrow 7 + g(3 * 5) * (P) \\ &\Rightarrow 7 + g(3 * 5) * (N(A)) \Rightarrow 7 + g(3 * 5) * (f(A)) \Rightarrow 7 + g(3 * 5) * (f()) \end{aligned}$$

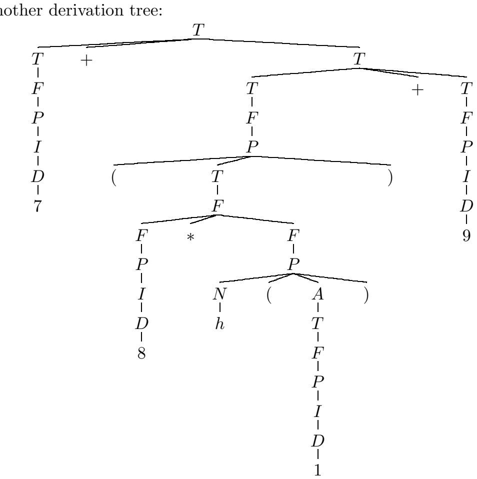
(c) It is not possible to derive $1 + 2 * 3)$. There are only two productions that introduce parentheses, and they always introduce them as balanced pairs. Thus it is not possible to derive a string with a single unmatched parenthesis as in this case.

(d) It is not possible to derive $1 + 7(9)$ because a number cannot be followed directly by a left (opening) parenthesis. The substring $7(9)$ must have been derived from T. There are only two productions that introduce parentheses. The production $P \to N(A)$ cannot have been used to derive $7(9)$ because, although P can be derived from T, 7 cannot be derived from N. The other possibility is the production $P \to (T)$. But, as numbers are only derivable via I, this would only be possible if a word IP can be derived form T. This in turn is not possible as the I ultimately has to be derived from a T, and inspection then shows that the only terminals that can follow an I are $+$, $*$, and).

2. Derivation tree:



3. Another derivation tree:



The fact that there are two different derivation trees for one word implies that the grammar is *ambiguous*.

4. Delete all old productions for A and add the following productions:

$$\begin{array}{rcl} A & \rightarrow & TL \mid \epsilon \\ L & \rightarrow & ,TL \mid \epsilon \end{array}$$

Here, L is a new nonterminal symbol, and "," is a new terminal symbol. The productions for L generate argument lists of zero, one or more arguments (any expression derivable from T), each preceded by a comma. The production $A \rightarrow TL$ thus generates argument lists of *one* or more arguments *separated* by commas, while the production $A \rightarrow \epsilon$ takes care of the case of zero arguments.

Answer to Exercise 8.1

The problem is that the grammar does not impart any associativity on the operators $+$ and $*$. Let us make them left-associative to address this. (Making them right-associative would also work, but we cannot make them non-associative as that would change the language; e.g. the example word $7 + (8 * h(1)) + 9$ would no longer belong to the language as non-associativity means we have to use explicit bracketing.) We make those operators left-associative by making the corresponding productions left-recursive:

$$\begin{array}{rcl} T & \rightarrow & T + F \mid F \\ F & \rightarrow & F * P \mid P \end{array}$$

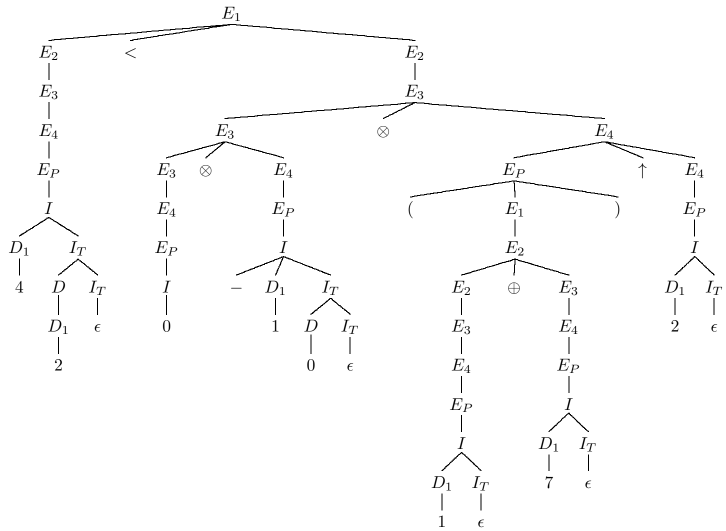
Answer to Exercise 8.2

1. The following CFG G_E is an unambiguous grammar satisfying the requirements. $G_E = (N, T, P, S)$ where:
   - $N = \{E_1, E_2, E_3, E_4, E_P, I, I_T, D, D_1\}$
   - $T = \{(,), <, \oplus, \otimes, \uparrow, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
   - P is given by:

$$\begin{array}{rcl} E_1 & \rightarrow & E_2 < E_2 \mid E_2 \\ E_2 & \rightarrow & E_2 \oplus E_3 \mid E_3 \\ E_3 & \rightarrow & E_3 \otimes E_4 \mid E_4 \\ E_4 & \rightarrow & E_P \uparrow E_4 \mid E_P \\ E_P & \rightarrow & I \mid (E_1) \\ I & \rightarrow & 0 \mid D_1 I_T \mid -D_1 I_T \\ I_T & \rightarrow & D I_T \mid \epsilon \\ D & \rightarrow & 0 \mid D_1 \\ D_1 & \rightarrow & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

   - $S = E_1$

2. Derivation tree for $42 < 0 \otimes -10 \otimes (1 \oplus 7) \uparrow 2$:



Answer to Exercise 8.3

First identify the nonterminals for which there are immediately left-recursive productions. Then group the productions for each such non-terminal into two groups: one where each RHS starts with the nonterminal in question (the immediately left-recursive terminals), and one where they don't:

$$
\begin{array}{lcl}
A & \to & A\alpha_1 \mid \ldots \mid A\alpha_m \\
A & \to & \beta_1 \mid \ldots \mid \beta_n
\end{array}
$$

In our case, there are immediately left-recursive productions for S and X in the given grammar. Grouping the productions as required yields:

- Grouping of productions for S:

$$
\begin{array}{lcl}
S & \to & Sa \\
S & \to & XbS \mid a
\end{array}
$$

- Grouping of productions for X:

$$
\begin{array}{lcl}
X & \to & XXX \mid XYY \\
X & \to & YYY \mid YYX
\end{array}
$$

- Remaining productions:

$$
Y \;\to\; cY \mid dY \mid e
$$

Then, continuing with the general example, the productions for a nonterminal A for which there are immediately left-recursive productions need to be replaced with new productions for A and productions for A', where A' is a new name, as follows:

$$
\begin{array}{lcl}
A & \to & \beta_1 A' \mid \ldots \mid \beta_n A' \\
A' & \to & \alpha_1 A' \mid \ldots \mid \alpha_m A' | \epsilon
\end{array}
$$

In our case, this transformation needs to be applied to the productions for the nonterminals S and X

$$
\begin{array}{lcl}
S & \to & XbSS' \mid aS' \\
S' & \to & aS' \mid \epsilon \\
X & \to & YYYX' \mid YYXX' \\
X' & \to & XXX' \mid YYX' \mid \epsilon \\
Y & \to & cY \mid dY \mid e
\end{array}
$$

Note in particular how the production $X \to XXX$ was transformed. The transformation rule specifies that the right-hand side should be split after the first X, meaning that the "α-part" is XX in this case. It might be confusing that this still starts with an X, but that is what the transformation rule says, and it is fairly easy to see why with a bit of thought. Then the XX is used to construct one of the right-recursive productions for the new nonterminal X'.

Answer to Exercise 10.1

1. $N_\epsilon = \{S, A, B\}$. A is nullable because $A \to \epsilon$ is a production. B is nullable because $B \to \epsilon$ is a production. S is nullable because $S \to ABB$ is a production and both A and B are nullable. C is not nullable because the RHSs of all productions for C include a terminal (c or d), meaning it is clear ϵ cannot be derived from C.
2. Keeping in mind which non-terminals are nullable, we obtain the following equations:

$$
\begin{array}{lcl}
\text{first}(A) & = & \text{first}(aA) \cup \text{first}(\epsilon) \\
 & = & \{a\} \cup \emptyset \\
 & = & \{a\}
\end{array}
$$

$$
\begin{array}{lcl}
\text{first}(B) & = & \text{first}(Bb) \cup \text{first}(\epsilon) \\
 & = & (\text{first}(B) \cup \text{first}(b)) \cup \emptyset \\
 & = & \text{first}(B) \cup \text{first}(b)
\end{array}
$$

$$
\begin{array}{lcl}
\text{first}(C) & = & \text{first}(cA) \cup \text{first}(d) \\
 & = & \text{first}(c) \cup \text{first}(d) \\
 & = & \{c\} \cup \{d\} \\
 & = & \{c, d\}
\end{array}
$$

The solutions of the equations for first(A) and first(C) are manifest. As to the equation for first(B), we need only observe that it has the form $X = X \cup Y$ and that there are no further constraints on first(B). The smallest solution to such an equation is simply $X = Y$, so first$(B) = \{b\}$. Now we can turn to setting up and solving the equation for first(S), again keeping in mind which non-terminals are nullable:

$$\begin{aligned}
\text{first}(S) &= \text{first}(ABB) \cup \text{first}(BBC) \cup \text{first}(CA)\\
&= (\text{first}(A) \cup \text{first}(B) \cup \text{first}(B) \cup \text{first}(\epsilon))\\
&\quad \cup (\text{first}(B) \cup \text{first}(B) \cup \text{first}(C))\\
&\quad \cup \text{first}(C)\\
&= (\text{first}(A) \cup \text{first}(B) \cup \emptyset)\\
&\quad \cup (\text{first}(B) \cup \text{first}(C))\\
&\quad \cup \text{first}(C)\\
&= (\{a\} \cup \{b\}) \cup (\{b\} \cup \{c,d\}) \cup (\{c,d\})\\
&= \{a,b,c,d\}
\end{aligned}$$

Thus, we obtained a solution directly.

3. Note: very detailed account below for clarity. It is sufficient to just state the constraints *according to the definitions* and then simplify.
Constraints for follow(S):

$$\{\$\} \subseteq \text{follow}(S)$$

Constraints for follow(A) from the productions where A occurs in the RHS, i.e.

$$\begin{aligned}
S &\to ABB\\
S &\to CA\\
A &\to aA
\end{aligned}$$

(note: nullable(BB) and nullable(ϵ)):

$$\begin{aligned}
\text{first}(BB) &\subseteq \text{follow}(A)\\
\text{follow}(S) &\subseteq \text{follow}(A)\\
\text{first}(\epsilon) &\subseteq \text{follow}(A)\\
\text{follow}(S) &\subseteq \text{follow}(A)\\
\text{first}(\epsilon) &\subseteq \text{follow}(A)\\
\text{follow}(A) &\subseteq \text{follow}(A)
\end{aligned}$$

Constraints for follow(B) from the productions where B occurs in the RHS, i.e.

$$\begin{aligned}
S &\to ABB\\
S &\to BBC\\
B &\to Bb
\end{aligned}$$

(note: nullable(B) and nullable(ϵ)):

$$\begin{aligned}
\text{first}(B) &\subseteq \text{follow}(B)\\
\text{follow}(S) &\subseteq \text{follow}(B)\\
\text{first}(\epsilon) &\subseteq \text{follow}(B)\\
\text{follow}(S) &\subseteq \text{follow}(B)\\
\text{first}(BC) &\subseteq \text{follow}(B)\\
\text{first}(C) &\subseteq \text{follow}(B)\\
\text{first}(b) &\subseteq \text{follow}(B)
\end{aligned}$$

Constraints for follow(C) from the productions where C occurs in the RHS, i.e.

$$\begin{aligned}
S &\to BBC\\
S &\to CA\\
C &\to cC
\end{aligned}$$

(note: nullable(A) and nullable(ϵ)):

$$\begin{aligned}
\text{first}(\epsilon) &\subseteq \text{follow}(C)\\
\text{follow}(S) &\subseteq \text{follow}(C)\\
\text{first}(A) &\subseteq \text{follow}(C)\\
\text{follow}(S) &\subseteq \text{follow}(C)\\
\text{first}(\epsilon) &\subseteq \text{follow}(C)\\
\text{follow}(C) &\subseteq \text{follow}(C)
\end{aligned}$$

Using

$$\begin{aligned}
\text{first}(\epsilon) &= \emptyset\\
\text{first}(A) &= \{a\}\\
\text{first}(B) &= \{b\}\\
\text{first}(C) &= \{c,d\}\\
\text{first}(BB) &= \text{first}(B) \cup \text{first}(B) \cup \text{first}(\epsilon)\\
&= \{b\} \cup \{b\} \cup \emptyset = \{b\}\\
\text{first}(BC) &= \text{first}(B) \cup \text{first}(C) \cup \emptyset\\
&= \{b\} \cup \{c,d\} = \{b,c,d\}
\end{aligned}$$

and eliminating trivial constraints yields:

$$
\begin{array}{rcl}
\{\$\} & \subseteq & \text{follow}(S) \\
\\
\{b\} & \subseteq & \text{follow}(A) \\
\text{follow}(S) & \subseteq & \text{follow}(A) \\
\\
\{b\} & \subseteq & \text{follow}(B) \\
\text{follow}(S) & \subseteq & \text{follow}(B) \\
\{b, c, d\} & \subseteq & \text{follow}(B) \\
\\
\text{follow}(S) & \subseteq & \text{follow}(C) \\
\{a\} & \subseteq & \text{follow}(C)
\end{array}
$$

This is equivalent to

$$
\begin{array}{rcl}
\{\$\} & \subseteq & \text{follow}(S) \\
\{b\} \cup \text{follow}(S) & \subseteq & \text{follow}(A) \\
\{b\} \cup \text{follow}(S) \cup \{b, c, d\} & \subseteq & \text{follow}(B) \\
\text{follow}(S) \cup \{a\} & \subseteq & \text{follow}(C)
\end{array}
$$

which can be further simplified to the final constraints:

$$
\begin{array}{rcl}
\{\$\} & \subseteq & \text{follow}(S) \\
\{b\} \cup \text{follow}(S) & \subseteq & \text{follow}(A) \\
\{b, c, d\} \cup \text{follow}(S) & \subseteq & \text{follow}(B) \\
\{a\} \cup \text{follow}(S) & \subseteq & \text{follow}(C)
\end{array}
$$

4. The smallest set satisfying the constraint for follow(S) is obviously just $\{\$\}$. Substituting this into the remaining constraints makes the smallest sets satisfying those obvious too. Thus:

$$
\begin{array}{rcl}
\text{follow}(S) & = & \{\$\} \\
\text{follow}(A) & = & \{b\} \cup \{\$\} = \{b, \$\} \\
\text{follow}(B) & = & \{b, c, d\} \cup \{\$\} = \{b, c, d, \$\} \\
\text{follow}(C) & = & \{a\} \cup \{\$\} = \{a, \$\}
\end{array}
$$

Answer to Exercise 10.2

1. $N_\epsilon = \{S, A, B, C\}$. A is nullable because $A \to \epsilon$ is a production. B is nullable because $B \to \epsilon$ is a production. C is nullable because $C \to \epsilon$ is a production. S is nullable because $S \to AB$ is a production and both A and B are nullable. D is not nullable because the right-hand sides of all productions for D include a terminal (d or e), meaning it is clear ϵ cannot be derived from D.

2. Keeping in mind which non-terminals are nullable, we obtain the following equations:

$$
\begin{array}{rcl}
\text{first}(A) & = & \text{first}(aA) \cup \text{first}(\epsilon) \\
& = & \{a\} \cup \emptyset \\
& = & \{a\} \\
\\
\text{first}(B) & = & \text{first}(BCDb) \cup \text{first}(\epsilon) \\
& = & (\text{first}(B) \cup \text{first}(CDb)) \cup \emptyset \\
& = & \text{first}(B) \cup (\text{first}(C) \cup \text{first}(Db)) \\
& = & \text{first}(B) \cup \text{first}(C) \cup (\text{first}(D) \cup \emptyset) \\
& = & \text{first}(B) \cup \text{first}(C) \cup \text{first}(D) \\
\\
\text{first}(C) & = & \text{first}(cD) \cup \text{first}(\epsilon) \\
& = & \{c\} \cup \emptyset \\
& = & \{c\} \\
\\
\text{first}(D) & = & \text{first}(dC) \cup \text{first}(e) \\
& = & \{d\} \cup \{e\} \\
& = & \{d, e\}
\end{array}
$$

The solutions of the equations for first(A), first(C), and first(D) are manifest. Recall that an equation of the form $X = X \cup Y$, in the absence of other constraints on X, simplifies to $X = Y$ when we are looking for the smallest solution. The equation for first(B) has the form $X = X \cup Y$ and there are no other constraints on first(B). The smallest solution is thus given by $\text{first}(B) = \text{first}(C) \cup \text{first}(D) = \{c\} \cup \{d, e\} = \{c, d, e\}$.
Now we can turn to setting up and solving the equation for first(S), again keeping in mind which non-termainals are nullable:

$$
\begin{array}{rcl}
\text{first}(S) & = & \text{first}(AS) \cup \text{first}(AB) \\
& = & (\text{first}(A) \cup \text{first}(S)) \cup (\text{first}(A) \cup \text{first}(B)) \\
& = & \text{first}(S) \cup \text{first}(A) \cup \text{first}(B) \\
& = & \text{first}(S) \cup \{a\} \cup \{c, d, e\} \\
& = & \text{first}(S) \cup \{a, c, d, e\}
\end{array}
$$

Again, an equatiom of the form $X = X \cup Y$, with no further constraints on first(S), meaning that the smallest solution is simply $\text{first}(S) = \{a, c, d, e\}$.

3. Note: very detailed account below for clarity. It is sufficient to just state the constraints *according to the definitions* and then simplify.
Constraints for follow(S). Note that S only appear in one RHS, of the production $S \to AS$, where it appears last; i.e. the string following S is just ϵ, and by definition we have nullable(ϵ). The constraints for S are

thus:

$$\begin{array}{rcl} \{\$\} & \subseteq & \text{follow}(S) \\ \text{first}(\epsilon) & \subseteq & \text{follow}(S) \\ \text{follow}(S) & \subseteq & \text{follow}(S) \end{array}$$

Constraints for follow(A) follow from the productions where A occurs in the RHS, i.e.

$$\begin{array}{rcl} S & \rightarrow & AS \\ S & \rightarrow & AB \\ A & \rightarrow & aA \end{array}$$

(note: nullable(S), nullable(B), and nullable(ϵ)):

$$\begin{array}{rcl} \text{first}(S) & \subseteq & \text{follow}(A) \\ \text{follow}(S) & \subseteq & \text{follow}(A) \\ \text{first}(B) & \subseteq & \text{follow}(A) \\ \text{follow}(S) & \subseteq & \text{follow}(A) \\ \text{first}(\epsilon) & \subseteq & \text{follow}(A) \\ \text{follow}(A) & \subseteq & \text{follow}(A) \end{array}$$

Constraints for follow(B) follow from the productions where B occurs in the RHS, i.e.

$$\begin{array}{rcl} S & \rightarrow & AB \\ B & \rightarrow & BCDb \end{array}$$

(note: nullable(ϵ), $\neg$nullable(CDb)):

$$\begin{array}{rcl} \text{first}(\epsilon) & \subseteq & \text{follow}(B) \\ \text{follow}(S) & \subseteq & \text{follow}(B) \\ \text{first}(CDb) & \subseteq & \text{follow}(B) \end{array}$$

Constraints for follow(C) follow from the productions where C occurs in the RHS, i.e.

$$\begin{array}{rcl} B & \rightarrow & BCDb \\ D & \rightarrow & dC \end{array}$$

(note: $\neg$nullable(Db) and nullable(ϵ)):

$$\begin{array}{rcl} \text{first}(Db) & \subseteq & \text{follow}(C) \\ \text{first}(\epsilon) & \subseteq & \text{follow}(C) \\ \text{follow}(D) & \subseteq & \text{follow}(C) \end{array}$$

Constraints for follow(D) follow from the productions where D occurs in the RHS, i.e.

$$\begin{array}{rcl} B & \rightarrow & BCDb \\ C & \rightarrow & cD \end{array}$$

(note: $\neg$nullable(b) and nullable(ϵ)):

$$\begin{array}{rcl} \text{first}(b) & \subseteq & \text{follow}(D) \\ \text{first}(\epsilon) & \subseteq & \text{follow}(D) \\ \text{follow}(C) & \subseteq & \text{follow}(D) \end{array}$$

Using

$$\begin{array}{rcl} \text{first}(\epsilon) & = & \emptyset \\ \text{first}(S) & = & \{a,c,d,e\} \\ \text{first}(B) & = & \{c,d,e\} \\ \text{first}(CDb) & = & \text{first}(C) \cup \text{first}(D) \\ & = & \{c\} \cup \{d,e\} = \{c,d,e\} \\ \text{first}(Db) & = & \text{first}(D) = \{d,e\} \\ \text{first}(b) & = & \{b\} \end{array}$$

and eliminating trivial constraints (of the types $\emptyset \subseteq X$ and $X \subseteq X$) yields:

$$\begin{array}{rcl} \{\$\} & \subseteq & \text{follow}(S) \\ \\ \{a,c,d,e\} & \subseteq & \text{follow}(A) \\ \text{follow}(S) & \subseteq & \text{follow}(A) \\ \{c,d,e\} & \subseteq & \text{follow}(A) \\ \\ \text{follow}(S) & \subseteq & \text{follow}(B) \\ \{c,d,e\} & \subseteq & \text{follow}(B) \\ \\ \{d,e\} & \subseteq & \text{follow}(C) \\ \text{follow}(D) & \subseteq & \text{follow}(C) \\ \{b\} & \subseteq & \text{follow}(D) \\ \text{follow}(C) & \subseteq & \text{follow}(D) \end{array}$$

Noting that follow(C) $\subseteq$ follow(D) $\wedge$ follow(D) $\subseteq$ follow(C) implies follow(C) = follow(D), this is equivalent to:

$$\begin{array}{rcl} \{\$\} & \subseteq & \text{follow}(S) \\ \{a,c,d,e\} \cup \text{follow}(S) \cup \{c,d,e\} & \subseteq & \text{follow}(A) \\ \text{follow}(S) \cup \{c,d,e\} & \subseteq & \text{follow}(B) \\ \{d,e\} \cup \{b\} & \subseteq & \text{follow}(C) = \text{follow}(D) \end{array}$$

which can be further simplified to the final constraints:

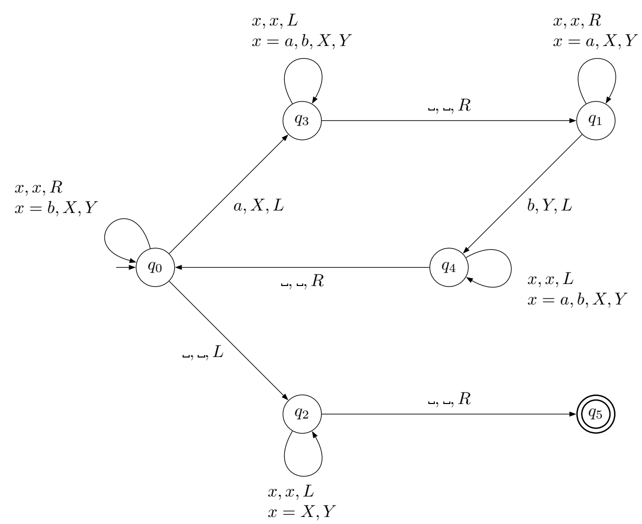
$$\begin{array}{rcl} \{\$\} & \subseteq & \text{follow}(S) \\ \{a,c,d,e\} \cup \text{follow}(S) & \subseteq & \text{follow}(A) \\ \{c,d,e\} \cup \text{follow}(S) & \subseteq & \text{follow}(B) \\ \{b,d,e\} & \subseteq & \text{follow}(C) = \text{follow}(D) \end{array}$$

4. The smallest set satisfying the constraint for follow(S) is obviously just $\{\$\}$. Substituting this into the remaining constraints makes the smallest sets satisfying those obvious too. Thus:

$$\begin{aligned}
\text{follow}(S) &= \{\$\} \\
\text{follow}(A) &= \{a, c, d, e\} \cup \{\$\} = \{a, c, d, e, \$\} \\
\text{follow}(B) &= \{c, d, e\} \cup \{\$\} = \{c, d, e, \$\} \\
\text{follow}(C) &= \{b, d, e\} \\
\text{follow}(D) &= \{b, d, e\}
\end{aligned}$$

Answer to Exercise 11.1

1. Diagram of Turing Machine M. The loops have a single set of symbols, representing several transitions. For example the loop around q_0 has the transition x, x, R with x being b, X, or Y. This stands for the three transitions b, b, R; X, X, R; and Y, Y, R.



2. Initially, the tape contains the word $baab$, is in state q_0 and the head points to the first b in the work. So the initial instantaneous description is $(\epsilon, q_0, baab)$. The computation then proceeds, according to the transition function, as follows:

$$\begin{aligned}
&(\epsilon, q_0, baab) \vdash (b, q_0, aab) \vdash (\epsilon, q_3, bXab) \vdash (\epsilon, q_3, ␣bXab) \vdash (\epsilon, q_1, bXab) \\
&\vdash (\epsilon, q_4, ␣YXab) \vdash (\epsilon, q_0, YXab) \vdash (Y, q_0, Xab) \vdash (YX, q_0, ab) \\
&\vdash (Y, q_3, XXb) \vdash (\epsilon, q_3, YXXb) \vdash (\epsilon, q_3, ␣YXXb) \vdash (\epsilon, q_1, YXXb) \\
&\vdash (Y, q_1, XXb) \vdash (YX, q_1, Xb) \vdash (YXX, q_1, b) \vdash (YX, q_4, XY) \\
&\vdash (Y, q_4, XXY) \vdash (\epsilon, q_4, YXXY) \vdash (\epsilon, q_4, ␣YXXY) \vdash (\epsilon, q_0, YXXY) \\
&\vdash (Y, q_0, XXY) \vdash (YX, q_0, XY) \vdash (YXX, q_0, Y) \vdash (YXXY, q_0, \epsilon) \\
&\vdash (YXX, q_2, Y) \vdash (YX, q_2, XY) \vdash (Y, q_2, XXY) \vdash (\epsilon, q_2, YXXY) \\
&\vdash (\epsilon, q_2, ␣YXXY) \vdash (\epsilon, q_5, YXXY)
\end{aligned}$$

We ended in the accepting state q_5, so the input word $baab$ is accepted.

3. Starting the machine in input aba, we obtain the following sequence of instantaneous descriptions.

$$\begin{aligned}
&(\epsilon, q_0, aba) \vdash (\epsilon, q_3, ␣Xba) \vdash (\epsilon, q_1, Xba) \vdash (X, q_1, ba) \vdash (\epsilon, q_4, XYa) \\
&\vdash (\epsilon, q_4, ␣XYa) \vdash (\epsilon, q_0, XYa) \vdash (X, q_0, Ya) \vdash (XY, q_0, a) \\
&\vdash (X, q_3, YX) \vdash (\epsilon, q_3, XYX) \vdash (\epsilon, q_3, ␣XYX) \vdash (\epsilon, q_1, XYX) \\
&\vdash (X, q_1, YX) \vdash (XY, q_1, X) \vdash (XYX, q_1, \epsilon)
\end{aligned}$$

The machine is in state q_1 and is reading a blank. There are no transitions for blank from q_1, so the machine stops. Because q_1 is not an accepting state, the input word aba is rejected.

4. This machine accepts the words in $\{a, b\}$ that contain the same number of as and bs. The language accepted by this Turing Machine is

$$L(M) = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$$

where I used the notation $\#_x(w)$ for the number of occurrences of the symbol x in the word w.
To see this, think about what happens in each of the states. You can characterize each state by performing some action:

q_0: Look for the next a, replace it with X and change to q_3. If you don't find any more as, change to q_2.

q_3: Go back to the beginning of the word and change to q_1.

q_1: Look for the next b, replace it with Y and change to q_4. If you don't find any more bs, stop and reject.

q_4: Go back to the beginning of the word and change to q_0 (start again).

q_2: Go through the word from right to left, checking that it contains only Xs and Ys (if it contains some a or b, stop and reject), then move to q_5.

q_5: Accept.

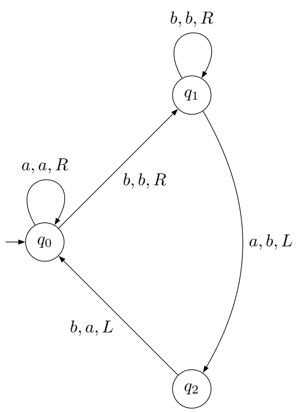
So the machine repeatedly searches for an a (state q_0) and replaces it with X, then searches for a b (state q_1) and replaces it with Y. It pairs them repeatedly; if it fails to pair an a with a b, it stops and reject. When it has finished all the as, it goes through the word again (state q_2)

checking that there are no bs left. In that case it knows that each a has been paired with a b with nothing left over; only in that case it moves to q_5 and accepts.

Answer to Exercise 11.2

There are several ways of implementing a Turing Machine that performs the computation required. The following is very simple, but does what is necessary.

1. This is the diagram illustrating the machine:



The machine starts at the beginning of the tape, skipping over the as in state q_0. As soon as it finds a b, it changes to q_1 and starts skipping over the bs. If it gets to the end of the word, it means that the symbols are in the correct order and it stops. But if it finds an a while in q_1, it knows that this a came after at least one b: we found a pair of symbols in the wrong order. We replace this a with a b and we replace the b that came before it with an a (state q_2). We are now in the position before this pair and we can return to q_0 and start looking for another wrong pair.

2. The machine is formally defined as follows:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, ␣, \emptyset) \quad \text{where} \quad \begin{aligned} Q &= \{q_0, q_1, q_2, \} \\ \Sigma &= \{a, b\} \\ \Gamma &= \{a, b, ␣\} \end{aligned}$$

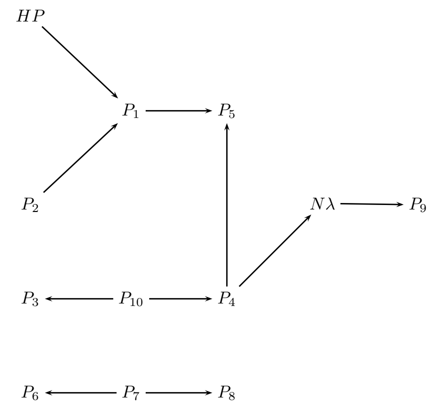
(The set of accepting states is empty, because the problem does not require that we accept words in a certain language.)
The transition function is defined by:

$$\begin{aligned} \delta(q_0, a) &= (q_0, a, R) \\ \delta(q_0, b) &= (q_1, b, R) \\ \delta(q_1, a) &= (q_2, b, L) \\ \delta(q_1, b) &= (q_1, b, R) \\ \delta(q_2, b) &= (q_0, a, L) \end{aligned}$$

(There is no transition from q_2 when reading an a. This never happens: every time we are in q_2 we are sure that we're reading a b.)

Answer to Exercise 11.3

Here is a diagram showing the reducibility relations between the languages (HP is the halting problem, $N\lambda$ is the normalization problem for λ-terms).



First of all, we already know some information directly: HP and $N\lambda$ are undecidable and the hypotheses tell us that P_7 is recursive, P_4 and the complement of P_3 is recursively enumerable, and P_6 and the complement of P_8 are not recursively enumerable.

A language is decidable (or recursive) if both it and its complement are recursively enumerable, so we can immediately conclude that P_6 and P_8 are undecidable.

Remember how we can use the knowledge that a problem A is reducible to another problem B to determine its reducibility properties. If B is decidable, we can use the reduction to decide A as well. Conversely, if A is undecidable, then B must be undecidable as well.

Similarly, if B is recursively enumerable, so is A. Conversely, if A is not recursively enumerable, neither is B.

These derivations also hold for complements: if the complement of B is recursively enumerable, that so is the complement of A.

As we know that HP and $N\lambda$ are undecidable, and they reduce to P_1 and P_9, respectively, we deduce that P_1 and P_9 are also undecidable. Now that we determined that P_1 is undecidable, because it reduces to P_5 we can deduce that P_5 is also undecidable.

Because P_4 is recursively enumerable and P_{10} is reducible to it, also P_{10} is recursively enumerable. In addition, P_{10} reduces to P_3, whose complement is recursively enumerable. This tells us that the complement of P_{10} is also recursively enumerable. We derived that both P_{10} and its complement are recursively enumerable, therefore P_{10} is recursive.

The rest of the given information does not add anything to our knowledge. For example, knowing that P_2 is reducible to the undecidable problem P_1 doesn't reveal anything about P_2.

1. Undecidable: P_1, P_5, P_6, P_8, P_9

2. Recursively enumerable: P_4, P_7, P_{10}

3. Recursive: P_7, P_{10}

Answer to Exercise 12.1

1. The function $\mathsf{nand\text{-}pair}$ takes a pair of Booleans p and returns a pair of Booleans whose first element is the negation of the conjunction of the elements of p, and whose second element is simply the first of p. (Remember that $p\,\mathsf{true}$ is the first projection of p and $p\,\mathsf{false}$ is the second projection.) So we have:

$$\begin{array}{ll}\mathsf{nand\text{-}pair}\,\langle\mathsf{true},\mathsf{true}\rangle \rightsquigarrow^* \langle\mathsf{false},\mathsf{true}\rangle & \mathsf{nand\text{-}pair}\,\langle\mathsf{false},\mathsf{true}\rangle \rightsquigarrow^* \langle\mathsf{true},\mathsf{false}\rangle \\ \mathsf{nand\text{-}pair}\,\langle\mathsf{true},\mathsf{false}\rangle \rightsquigarrow^* \langle\mathsf{true},\mathsf{true}\rangle & \mathsf{nand\text{-}pair}\,\langle\mathsf{false},\mathsf{false}\rangle \rightsquigarrow^* \langle\mathsf{true},\mathsf{false}\rangle\end{array}$$

2. Remember that Church numerals are just iterators, so the definition of $\mathsf{nand\text{-}fun}$ tells us to iterate n times the function $\mathsf{nand\text{-}pair}$ starting with the initial value $\langle\mathsf{false},\mathsf{false}\rangle$. Applied to the numeral $\overline{4} = \lambda f.\lambda x.f\,(f\,(f\,(f\,x)))$ it produces the following reduction steps:

$$\begin{array}{l}\mathsf{nand\text{-}fun}\,\overline{4} = (\lambda n.n\,\mathsf{nand\text{-}pair}\,\langle\mathsf{false},\mathsf{false}\rangle)\,\overline{4} \\ \rightsquigarrow \overline{4}\,\mathsf{nand\text{-}pair}\,\langle\mathsf{false},\mathsf{false}\rangle = (\lambda f.\lambda x.f\,(f\,(f\,(f\,x))))\,\mathsf{nand\text{-}pair}\,\langle\mathsf{false},\mathsf{false}\rangle \\ \rightsquigarrow^* \mathsf{nand\text{-}pair}\,(\mathsf{nand\text{-}pair}\,(\mathsf{nand\text{-}pair}\,(\mathsf{nand\text{-}pair}\,\langle\mathsf{false},\mathsf{false}\rangle))) \\ \rightsquigarrow^* \mathsf{nand\text{-}pair}\,(\mathsf{nand\text{-}pair}\,(\mathsf{nand\text{-}pair}\,\langle\mathsf{true},\mathsf{false}\rangle)) \\ \rightsquigarrow^* \mathsf{nand\text{-}pair}\,(\mathsf{nand\text{-}pair}\,\langle\mathsf{true},\mathsf{true}\rangle) \\ \rightsquigarrow^* \mathsf{nand\text{-}pair}\,\langle\mathsf{false},\mathsf{true}\rangle \\ \rightsquigarrow^* \langle\mathsf{true},\mathsf{false}\rangle\end{array}$$

3. As the previous calculation shows, $\mathsf{nand\text{-}fun}\,\overline{n}$ iterates n times $\mathsf{nand\text{-}pair}$, starting with the initial value $\langle\mathsf{false},\mathsf{false}\rangle$. So $\mathsf{nand\text{-}fun}\,\overline{0}$ will do zero iterations and just return this initial value. The sequence of values that the iteration goes through is obtained by repeating the steps of part (a). Here are the values obtained (with the iteration number written on top of the arrows):

$$\begin{array}{l}\langle\mathsf{false},\mathsf{false}\rangle \xrightarrow{1} \langle\mathsf{true},\mathsf{false}\rangle \xrightarrow{2} \langle\mathsf{true},\mathsf{true}\rangle \\ \xrightarrow{3} \langle\mathsf{false},\mathsf{true}\rangle \xrightarrow{4} \langle\mathsf{true},\mathsf{false}\rangle \xrightarrow{5} \langle\mathsf{true},\mathsf{true}\rangle \\ \xrightarrow{6} \langle\mathsf{false},\mathsf{true}\rangle \xrightarrow{7} \langle\mathsf{true},\mathsf{false}\rangle \xrightarrow{8} \langle\mathsf{true},\mathsf{true}\rangle \xrightarrow{9} \cdots\end{array}$$

We obtain $\langle\mathsf{true},\mathsf{true}\rangle$ for 2, 5 and 8. It is clear that there is a cycle that repeats every three steps, so we have $\mathsf{nand\text{-}fun}\,\overline{n} \rightsquigarrow^* \langle\mathsf{true},\mathsf{true}\rangle$ for those numbers n of the form $n = 2 + 3 * m$ for some m.

Answer to Exercise 12.2

To implement thrFib in the λ-calculus, we need to see it as the iteration of a function a number of times starting with a given value. As in the case of the Fibonacci function described in the lecture, we need to use an auxiliary function that returns three values at the same time (it was just two for Fibonacci). Then at each step we shift two of the values and add the sum of the three:

$$\langle n,m,k\rangle \longmapsto \langle m,k,n+m+k\rangle.$$

Informally, our auxiliary function is defined by

$$\begin{array}{l}\mathsf{thrFib_{aux}} : \mathbb{N} \to \mathbb{N}\times\mathbb{N}\times\mathbb{N} \\ \mathsf{thrFib_{aux}}\,0 = \langle 0,0,1\rangle \\ \mathsf{thrFib_{aux}}\,(n+1) = \langle m,k,n+m+k\rangle \quad \text{where } \langle n,m,k\rangle = \mathsf{thrFib_{aux}}\,n\end{array}$$

In the λ-calculus we use the encoding of triples as

$$\langle a,b,c\rangle = \lambda x.x\,a\,b\,c$$

With the projections defined by

$$\mathsf{fst} = \lambda t.t\,(\lambda x.\lambda y.\lambda z.x), \quad \mathsf{snd} = \lambda t.t\,(\lambda x.\lambda y.\lambda z.y), \quad \mathsf{trd} = \lambda t.t\,(\lambda x.\lambda y.\lambda z.z).$$

(You can also encode triples as repeated pairs. It is not important how you do it, as long as you can define projections.)

Using triples and their projections, we can define the auxiliary function in λ-calculus:

$$\begin{array}{l}\mathsf{thrFib_{aux}} = \lambda n.n\,\mathsf{step}\,\langle\overline{0},\overline{0},\overline{1}\rangle \\ \qquad \text{where } \mathsf{step} = \lambda t.\langle\mathsf{snd}\,t,\mathsf{trd}\,t,\mathsf{plus}\,(\mathsf{fst}\,t)\,(\mathsf{plus}\,(\mathsf{snd}\,t)\,(\mathsf{trd}\,t))\rangle\end{array}$$

Finally, the function we want will just take the first component of the result of the auxiliary function:

$$\mathsf{thrFib} = \lambda n.\mathsf{fst}\,(\mathsf{thrFib_{aux}}\,n).$$

Answer to Exercise 13.1

The function `evaluate` can be defined by recursion on the structure of the SAT formula, using just the projection from the assignment to determine the value on the variables.

```
evaluate :: SAT -> Assignment -> Bool
evaluate (Var i) a = a!!i
evaluate (Not x) a = not (evaluate x a)
evaluate (And x y) a = (evaluate x a) && (evaluate y a)
evaluate (Or x y) a = (evaluate x a) || (evaluate y a)
```

Note that in the variable case, if the index i is larger or equal to the length of the assignment list, we would get an error. But we are assuming that we evaluate expressions with assignments that guarantee values for all variables occurring in the expression.

Answer to Exercise 13.2

1. The function `varNum` is also defined by recursion on the structure of the expression, returning just the index for variables and the maximum of the recursive calls for conjunctions and disjunctions.

```
varNum :: SAT -> Int
varNum (Var n) = n
varNum (Not x) = varNum x
varNum (And x y) = max (varNum x) (varNum y)
varNum (Or x y) = max (varNum x) (varNum y)
```

2. Notice that the length of the assignment returned by `allAssign n` is $n + 1$ (it must contain all the possible values for variables `Var 0`, ..., `Var n`, that is the first $n + 1$ variables). So `allAssign 0` should return `[[True],[False]]`. We may use this as a base case. For the recursive case, we can just add either a `True` or a `False` in front of each of the list computed by the recursive call.

```
allAssign :: Int -> [Assignment]
allAssign 0 = [[True],[False]]
allAssign n = [b:bs | b  <- [True,False],
                      bs <- allAssign (n-1)]
```

(An alternative solution for the base case is `allAssign (-1) = [[]]`.)

3. Evaluate the formula using all the possible assignments for its variables. If at least one of them returns `True`, then the formula is satisfiable.

```
satisfiable :: SAT -> Bool
satisfiable x = or (map (evaluate x)
                        (allAssign (varNum x)))
```

4. From the list of all assignments, filter out those that return `True`. If this list is empty, then the formula is not satisfiable. If it is not empty, any of its elements (for example the head) is a solution.

```
solution :: SAT -> Maybe Assignment
solution x = if null bs then Nothing else Just (head bs)
```



```
  where bs = filter (evaluate x)
                    (allAssign (varNum x))
```

Answer to Exercise 13.3

1. The program `evaluate` traverses the whole structure of the input formula. For the `Not`, `And` and `Or` operators it simply calls itself on the subformulas and then does a simple operation. In the variable case it just looks up the value in the assignment, which is linear in the length of the assignment.
   So the number of operations executed at each step is at most linear in the length of the assignment. The number of steps is bound by the size of the formula. So the overall complexity is at most quadratic on the size of the input.
2. The function `satisfiable` applies `evaluate` to all the possible assignments of values to the variables. Each application of `evaluate` takes polynomial time, but there are exponentially many assignments that need to be tested; to be precise, for n variables there are 2^n assignments. Because we are performing the evaluation for each of these assignments, the overall complexity is also exponential.
3. The fact that SAT is $\mathcal{NP}$-complete means that every $\mathcal{NP}$ problem can be reduced to it in polynomial time. This means that if we ever find a polynomial time algorithm for SAT, we would automatically be able to solve all $\mathcal{NP}$ problems in polynomial time. This would prove that $\mathcal{NP} = \mathcal{P}$.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Ullman Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986. 90

[Cox07] Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). http://swtch.com/ rsc/regexp/regexp1.html, January 2007. 29

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle, Inc., Java SE 8 edition edition, 2015. 62

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition edition, 2001. 17, 44, 79, 103

[Hud99] Scott E. Hudson. Cup parser generator for java. http://www.cs.princeton.edu/∼appel/modern/java/CUP/, 1999. 99

[Mar01] Simon Marlow. Happy: The parser generator for haskell. http://www.haskell.org/happy/, 2001. 99

[Nil16] Henrik Nilsson. Compilers (G53CMP) — lecture notes. http://www.cs.nott.ac.uk/∼nhn/G53CMP, 2016. 62, 82

[Par05] Terence Parr. ANTLR: Another tool for language recognition. http://www.antlr.org/, 2005. 99