Lecture notes on Formal Languages

Rick Nouwen

Version number: 1.020, February 4, 2021

Contents

1 Formal languages **1**
- 1.1 Strings 1
- 1.2 The Kleene star 2
- 1.3 Formal languages and decision problems 4
- 1.4 Computability 5
- 1.5 How many languages are there? 6
- 1.6 Formal versus natural language 7

2 Finite state automata **10**
- 2.1 Formal definition 11
- 2.2 Non-determinism 12
- 2.3 Acceptance 14
- 2.4 Determinism 15
- 2.5 Finite state transducers 17

3 Regular languages **19**
- 3.1 Regular languages and finite state automata 19
- 3.2 Closure properties 21
- 3.3 Non-regularity 23
- 3.4 The pumping lemma for regular languages 25

4 Formal grammars **28**
- 4.1 Formal definition 28
- 4.2 Derivation 29
- 4.3 Parse trees and ambiguity 30
- 4.4 Grammar equivalence 33
- 4.5 Regular grammars 34

5 Context-free languages **37**
- 5.1 Push-down automata 37
- 5.2 Context-free grammar 40
- 5.3 Chomsky Normal Form 41
- 5.4 Pumping lemma 43
- 5.5 Closure properties 46
- 5.6 Mirroring versus copying, and natural language 47

These lecture notes introduce the essential concepts underlying formal languages. They are meant as an overview that enables you to understand and work with formal languages. These lecture notes presuppose familiarity with set theory.

1 Formal languages

1.1 Strings

A set is the most basic mathematical method of describing collections. The only thing that matters to a set is the elements that it contains. There is no notion of order in a set, nor is there a possibility of repeated membership. Something is either in a set or not, nothing else is relevant. To illustrate, $\{1, 1\}$ is a very odd way of writing down the set $\{1\}$. There is no difference between $\{1, 2\}$ and $\{2, 1\}$. The set $\{1, \{1\}\}$ only contains the number 1 once, because it has two elements: the number 1 and the set containing that number.

Order and multiplicity *does* matter in the pairs we form when we form a Cartesian product. For instance, $\{1, 2\} \times \{1, 2\}$ contains both $(1, 2)$ and $(2, 1)$ as an element. It also contains the pair $(1, 1)$. So, while two sets are equal if and only if they have the same elements, two ordered pairs are equal if and only if they have the same elements *on the same positions*.

A *string* is like an ordered pair, except that there are no restrictions on how many elements it contains. We usually write strings without any extra notation. So, we write 1 for the string just containing a single 1 and we write 111 for the string that has three positions, each of which contains 1. As with ordered pairs $ab \neq ba$ and $aa \neq a$.

Nota bene:

As always, we want mathematics to be grounded in set theory and so we would like ordered pairs and strings to correspond to sets. But how can we do this? How can we represent order in something that is fundamentally unordered? The Polish mathematician Kazimierz Kuratowski proposed a way to do this. He identified the ordered pair (x, y) as the set $\{\{x\}, \{x, y\}\}$. That is, the first element in an ordered pair is the element that occurs in all the sets, while the second element is the element that occurs in just one. (Exercise: check that this still makes sense when you have an ordered pair like $(1, 1)$.)
Like ordered pairs, we can also ground strings in sets. Set-theoretically, a string is a function from natural numbers (the positions in the string) to the elements that make up the strings. As we know, a function is a set of ordered pairs and is thus itself also grounded in set theory. (See above). Example: the string 3512JK corresponds to $\{(1, 3), (2, 5), (3, 1), (4, 2), (5, J), (6, K)\}$.

Strings can be extended indefinitely. Say we have some string containing just the number 3, repeated many times. We can form a different string by just appending another 3 to this string. This new string can be extended in the same way, etc. This means that there are infinitely many strings, even if we build strings just from a single element.

Strings can also have no elements. The empty string is written as ϵ. (Alternative notation for the empty string include: Λ, λ and e).

Strings can be concatenated. If α and β are strings, then $\alpha \frown \beta$ is the unique string such that the elements in β follow the elements in α, preserving the order of the elements in both strings. Concatenation is not commutative: for instance, $12 \frown 21 \neq 21 \frown 12$. But concatenation is associative: $12 \frown (21 \frown 12) = (12 \frown 21) \frown 12 = 122112$. The empty string acts as a so-called *identity element* for concatenation, which means that for any string φ, it holds that $\epsilon \frown \varphi = \varphi \frown \epsilon = \varphi$. (Compare: 0 is the identity element for addition. For example, $a + 0 = 0 + a = a$.)

As will become evident below, it is often handy to have a special notation for repetitions of symbols in a string. For instance, we will sometimes write 1^3 for the string 111, and so $12^2 3^3 4^4$ is short for 1223334444. The set $\{1^n | n > 0\}$ is the set of all strings that contain 1 or more 1s and nothing else.

Strings have huge importance for artificial intelligence. This is because many kinds of knowledge can be stored and represented as a string. For instance, any text, whether it is the content of a book, a web page or a governmental law, etc. is a string of letters, digits, spaces and punctuation. Similarly, any computer program can be represented as a string of letters, digits, spaces and punctuation. (Alternatively, a computer program can be seen as a string of binary digits.) Also, any image can be seen as a string of pixel values and a sound recording is a string of values that represent subsequent properties of an audio signal. In general, when computers perform tasks, they perform tasks on strings.

1.2 The Kleene star

Let X be some set. The set of all finite(!) strings made up only of elements in X is written as X^* (the Kleene closure of X). Here is a recursive definition:

Definition 1

Kleene closure: For any set A, the following holds:

- Base case: $\epsilon \in A^*$
- Recursive step: If $s \in A^*$ and $t \in A$, then $s \frown t \in A^*$
- Any element of A^* is either ϵ or the result of a finite number of applications of the recursive step

A similar, but ultimately more useful definition is the following:

Definition 2

Kleene closure: For any set A, $A^* = \bigcup\{A_i \mid i \geq 0 \text{ and } i \in \mathbb{N}\}$
Where:

- $A_0 = \{\epsilon\}$
- $A_i = \{\sigma \frown a | \sigma \in A_{i-1} \text{ and } a \in A\}$ for any $i \in \mathbb{N}$ such that $i > 0$

For example, $\{1\}^*$ is the set $\{\epsilon, 1, 11, 111, 1111, \ldots\}$. The set $\{1, 2\}^*$ corresponds to:

$$\{\epsilon, 1, 2, 11, 12, 21, 22, 111, 112, 121, \ldots\}$$

A special case is $\emptyset^*$. Let's see which set this is by applying the above definition. First of all, ϵ is in $\emptyset^*$, since the definition has it that ϵ is in any set that results from Kleene closure. Note then that $\emptyset^* \neq \emptyset$, since we have found a string that is included in the Kleene closure of the empty set. According to the definition, further strings in $\emptyset^*$ are now to be the result from concatenating a given string in that set with some element in the original set. Since $\emptyset$ has no elements, we end up with no further strings, and so: $\emptyset^* = \{\epsilon\}$.

What is special about $\emptyset^*$ is that it is the only Kleene closure that is finite. In particular:

Theorem 1

For any non-empty finite or countably infinite set X, X^* is countably infinite.

Proof

According to the second definition I gave for Kleene closure, X^* is the infinite union of a family $\{X_i | i \geq 0\}$.
First case: X is finite. Take any set X_i that is the set of strings of length i that can be built from the elements of X. For a set of cardinality c, the number of strings of length n that you can build from this set equals c^n. So, $|X_i| = |X|^i$. This means that each X_i is finite. We can thus enumerate all the strings, by just first enumerating all the strings of length 0, then the finite number of strings of length 1, then the finite number of strings of length 2, etc. This yields a countably infinite number of strings.
Second case: X is countably infinite. We know that X^* is the union of a countable infinity of sets X_i. Each of these sets contains a countable number of strings. To see this, first look at X_0, which is obviously countable, since $|X_0| = 1$. Next, X_1 is countably infinite, since it contains all and only the strings of length 1 made up of the countably infinite elements of X. All further sets X_i are the result of concatenating one of the elements in X to one of the elements in X_{i-1}. We can show that if X_{i-1} is countably infinite, then so is X_i by creating a table where the (countably infinite) columns represent elements of X (so, $X = \{x_1, x_2, x_3, \ldots\}$) and the (countably infinite) rows represent strings in X_{i-1} (which we take to be $\{s_1, s_2, s_3 \ldots\}$). We can enumerate the elements in this table using the enumeration strategy depicted in the table below. (This is similar to how we normally show that there is a countably infinite number of rational numbers.) This shows that X_i is countably infinite whenever X_{i-1} is. Since X_1 (and, in fact X_0) is countable, so are all sets X_i.

	x_1	x_2	x_3	x_4	x_5	$\ldots$
s_1	1	2	4	7	11	
s_1	3	5	8	12		
s_2	6	9	13			
s_3	10	14				
s_4	15					
s_5						
$\vdots$						

Now we need to prove that a countably infinite union of countable infinite sets is countably infinite. For ease of reference, let's name the elements of the individual subsets X_i that make up X^* as follows: $X_i = \{s_{i1}, s_{i2}, s_{i3}, \ldots\}$. As I've shown, all these sets X_i are countable and there are countably many of them. X^* is the union of all these sets, so the task is to show that we can enumerate all the elements of all these sets X_i. We can refer to the individual elements in this enumeration as s_{ij} and enumerate as follows.

	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$\ldots$
$i = 1$	1	2	4	7	11	
$i = 2$	3	5	8	12		
$i = 3$	6	9	13			
$i = 4$	10	14				
$i = 5$	15					
$i = 6$						
$\vdots$						

This means that we start with s_{11}, then s_{12}, then s_{21}, s_{13}, etc. This way, we will reach each element in each set X_i. Note that, we left out X_0, but since X_0 contains a finite number of elements (namely just ϵ) we can just add its contents to the enumeration.

1.3 Formal languages and decision problems

Formal languages are sets of strings. In particular, a formal language over X is a subset of X^*. If $L \subseteq X^*$, then we can call X the alphabet of formal language L.

Consider for example the set A, the set consisting of all numerical digits and all capital letters:

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \cup \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$$

The set A^* is the set of all possible finite combinations of elements in this set. A formal language is a particular subset of that set. For instance, Dutch postal codes consist of 4 numerical digits followed by 2 capital letters. So, the set of all Dutch postal codes is a formal language over alphabet A. Similarly, a company may make many kinds of models of a certain product and label them with alphanumeric combinations. These labels, too, would form a formal language over alphabet A.

These examples are rather trivial illustrations of what a formal language is. To understand the value of formal languages, it is important to understand the relation to computer science and in particular to one kind of task that computers can perform. So-called *decision problems* are problems that, given an input, ask for a binary decision to be made. Here are some examples of decision problems:

- given an input n, output "yes" if n is prime and "no" otherwise
- given a sequence of letters, output "yes" if the sequence is a sentence of English and "no" otherwise
- given a sentence in a logical language, output "yes" if the sentence is tautological and "no" otherwise
- given a sequence of keystrokes, output "yes" if the sequence is an admissible password and "no" otherwise

Decision problems divide inputs up into two classes: one kind of input will result in "yes", the rest will result in "no". Because of this, decision problems correspond to formal languages. A language L over Σ is a subset of Σ^* and, as such, this language represents a choice between those strings in Σ^* that are in L (the inputs resulting in "yes") and those strings in Σ^* that are not in L (the inputs resulting in "no"). Consider, for example, the final example of a decision problem, where the task is to decide whether the input is to be permitted as a password for the user. This task amounts to saying yes to the admissible combinations and no to the non-admissible ones. The set of admissible combinations can be seen as a formal language and, so, the decision problem reduces to computing this language. That is, performing this task is the same as performing the task of deciding whether or not a string is in the language or not.

Formally, decision problems are related to characteristic functions:

Definition 3

Say we are interested in some set of elements U. Let A be a subset of U. The *characteristic function* of A, $f_A : U \to \{0,1\}$ is defined as follows:

$$f_A(x) = \begin{cases} 1 & \text{whenever } x \in A \\ 0 & \text{otherwise} \end{cases}$$

Conversely, the language L_f corresponding to some characteristic function f is defined as:

$$L_f = \{x \mid f(x) = 1 \text{ and } x \in U\}$$

We can think of a decision problem corresponding to both a language and a chararcteristic function. Computing the function f can be equated with deciding on membership in L_f. While there are many tasks that do not correspond to solving a decision problem, very often such tasks can be *reduced* to a decision problem. This is why decision problems are central to the scientific study of computation. To illustrate, consider the task of simple arithmetic addition. This task is normally not represented as a decision problem, but rather as a problem that requires finding the right answer: for instance, *what* is the number that equals 1+4? However, being able to solve such problems entails that you can solve a related decision problem, namely the problem of how to distinguish correct answers from incorrect one. So, the task of addition as a decision problem amounts to the language that contains strings like "1+4=5", but not strings like "1+4=6".

1.4 Computability

Say, we want to start make computers perform all sorts of tasks for which we humans need intelligence. The naive way to go about this would just be to take one task at a time and work on that task until we have satisfactory performance by a computer. Without a theory of what it means for a computer to compute, however, we have no way of knowing whether the things we are attempting are possible, or in what way the individual tasks are related, or what the complexity is of the task we are looking at, etc.

Historically, formal languages are at the basis of theories of *computability*. This is the notion used to reason about which tasks computers can perform. It turns out that some functions are not computable. This doesn't mean that we simply haven't found of way of getting a computer to perform the task corresponding to that function, or that we didn't manage to build a computer powerful enough to do so. Rather, it means that we can prove that it is theoretically (and therefore also practically) impossible to compute these functions.

We know this because we do have a theory of computation. Alan Turing (1912-1954) developed an abstract model of computation, the Turing machine, and used it to reason about decision problems. The most famous example of something Turing proved to be impossible is the *halting problem*. The task in the halting problem is the following. We would like an algorithm that takes as input some computer program code and an input to feed to that computer program and the algorithm should decide for us whether the program will halt or run indefinitely. Take for instance the following two mini pseudo code programs:

```
def test(n): if (n>0) return 1 else return 0;
```

```
def oei(n): while (n>0) do n++;
```

It is easy to see that `test(2)` will halt. (It returns `1`). Also, it is easy to see that `oei(2)` will not halt.

Since 2>0, it will keep on adding 1 to n indefinitely. (n++ is short for assigning to n a value that is 1 higher than the current value.) The question is now whether we can think of a function halt that looks roughly as follows:

```
def halt(f,n): . . . . . . . . . return YES else return NO;
```

and which would output YES for halt(test,2), YES for halt(oei,0) and NO for halt(oei,2).

Let us assume that halt *is* a computable function and then show that this runs into a contradiction. If halt is computable, then we should also be able to compute the following function:

```
def barber(f): if (halt(f,f)=YES) then oei(2) else return NO
```

This is a program that takes a function f as input and then does the following. It uses halt to test whether f applied to itself halts. (That is, halt(f,f) gives YES if f(f) halts and NO if f(f) does not halt.) If f(f) does halt, then barber runs oei(2). As a consequence barber(f) will fail to halt whenever f(f) does halt. If f(f) does not halt, then barber(f) will return NO and, as such, halt.

Now we imagine running the following: barber(barber). What would be the result of running this? Well, barber plays the role of f here. So to see what happens, we need to know what the outcome of halt(barber, barber) is. This outcome will be YES if barber(barber) halts and NO if it does not. But now we get a contradiction. Let's say barber(barber) halts. Then halt(barber,barber) returns YES and barber will run oei(2) indefinitely and, so, barber(barber) will not halt. Let's then instead say that barber(barber) does not halt. In that case halt(barber,barber) does not return YES and as a result, barber will return NO. So, if barber(barber) does not halt, we are forced to conclude that it does halt, namely by outputing NO. Whatever we do, we run into a contradiction. Note that there is nothing odd about the barber function. It contains a normal if-then-else condition which tests the output of a function, runs another function in one case and returns something in the other case. The only reason we could have to doubt whether we could define barber is the fact that it makes use of halt. The contradictions we run into, therefore will have to do with our assumption that halt exists and, as such, we can conclude that this is a non-computable function.

1.5 How many languages are there?

The previous section illustrate how formal languages can help us understand computation, in particular its limits and complexity. All the programs test, oei, halt and barber are (or in the case of halt are intended to be) characteristic functions for decision problems. So, if we want to understand computation, we need to understand as much as possible about formal languages. A good start is to ask how many languages there are.

For any alphabet Σ, the set of languages over Σ is any subset of Σ^*. So, the class of languages over Σ, notation $\mathbb{L}(\Sigma)$, is simply $\wp(\Sigma^*)$. The simplest case is $\Sigma = \emptyset$, for which $\Sigma^* = \{\epsilon\}$ and $\mathbb{L}(\Sigma) = \{\emptyset, \{\epsilon\}\}$.

As we saw, for any non-empty countable Σ, Σ^* is countably infinite. So, for any such alphabet $\mathbb{L}(\Sigma)$ will be the powerset of a countably infinite set. As a consequence, there are *uncountably infinitely many* languages of any non-empty alphabet. This follows given the following theorem.

Theorem 2

If X is a countably infinite set, then $\wp(X)$ is an uncountably infinite set.

Proof

Let's say that $\wp(X)$ *is* countable. In that case we should be able to enumerate all its sets. The theorem will be proven by showing that this assumption is untenable.
Consider the following table. The columns of the table are the enumerated elements of $X = \{x_1, x_2, x_3, \ldots\}$. The rows are intended to be the enumerated elements of $\wp(X)$. That is, each row represents a set as a vector, where a 1 indicates that the element corresponding to the column is a member of that set and 0 indicates that it is not.

x_1	x_2	x_3	$\ldots$
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0
0	0	1	0
1	0	1	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$

So, the first row in this table is the set that contains none of the elements in X. That is, it's $\emptyset$. The second row is the set containing just x_1. The sixth represents $\{x_1, x_3\}$. Etc. Because we are systematically going through all the elements of X (the columns of the table), the rows should enumerate all the subsets of X, if there are countably many.
But now take the *diagonal* of the table, the values indicated in red. This yields a vector $D = (0, 0, 0, 0, \ldots)$. Say we take this vector and change all its values to the opposite value, so we write a 0 where it said 1 and a 1 where it said 0. We then get $V = (1, 1, 1, 1, \ldots)$. Since this is just a vector of 0s and 1s, it represents a subset of X. If $\wp(X)$ is countable, V should correspond to some row in the table. But notice that it couldn't possibly be a row. If it is a row, then the diagonal (the red line of numbers) will cross that row at some column c. But the value of V at c has the opposite value from the diagonal at c and, so, V cannot be in the table. It follows that $\wp(X)$ is not countable.

So, even if the alphabet is extremely simple, such as $\Sigma = \{1\}$, $\mathbb{L}(\Sigma)$ contains an uncountably infinite set of languages. In what follows we will try and understand certain interesting subclasses of this uncountable infinity, some of which *are* countable. Before we do so, we turn from formal languages to *natural* ones. In particular, we will have a brief look at the role that infinity plays in natural languages.

1.6 Formal versus natural language

In some ways, a *natural language* could not be more different from a formal one. Natural languages are *natural* in the sense that they emerged in the natural world, through human interaction, without any premeditated design choices. It is worthwile elaborating on the notion of a natural language a bit, because despite this huge opposition between the naturalness of natural languages and the artificial nature of formal languages, both natural and formal languages share some important features.

First, however, here is one particular way in which the natural character of natural language manifests itself. We have all acquired at least one natural language. In fact, worldwide most humans grow up acquiring multiple languages as their *native* language. With *acquire* I don't mean that you learn the language at school. Rather, it means that you learn to use your language(s) through interaction with other speakers of that language like your family, the children you play with, your neighbours, etc.

There is no explicit instruction involved. Nobody actively teaches you the meaning of words, or how to pronounce them, or how to string them together into a sentence. Your language ability simply emerges as a result of interacting with other humans. At school, you learn how to do certain things with your language, like how to write it down or how to read and understand text, but you do not need school to acquire your general linguistic abilities: by the time you start school at 4 or 5 years old, your native language is already almost fully developed. You know words, you know how to pronounce them, how to string them together into sentences. You recognise words when they are spoken. You know what sentences mean when you hear them. You can draw inferences from them and phrase these inferences in your language. Etc. Etc.

The ability to speak and understand your native language(s) is something quite remarkable. The linguist and philosopher Noam Chomsky identifies this remarkability in the fact that we have access to a body of *knowledge of language* of which we are not consciously aware that we possess it. This knowledge is often captured by the term *competence*. This competence is not completely visible in our everyday use of language, but we may access it through introspection. For instance, any native speaker of English will be able to decide that (1) is not a sentence of that language. (Linguists mark ungrammatical sentences with a "*". Note that this is fully unrelated to the Kleene star.)

(1) *Sleep furiously green ideas colourless

This is just word soup, a seemingly random sequence of English words. It is very easy to come to the judgment that this is not a sentence. You know how to make that judgment, because you have knowledge of language. Nobody taught you that this sentence is ungrammatical, it is just something that you have the ability to decide on, via your language competence. What's more, that same competence allows you to decide that the following sentence *is* grammatical.

(2) Colourless green ideas sleep furiously.

The sentence in (2), a famous example due to Chomsky, clearly makes no sense whatsoever. But that does not seem to matter for your ability to judge it as grammatical: Every speaker of English will judge (2) differently from (1). While neither makes sense, (2) is a sentence, but (1) is not. This illustrates the robustness of our knowledge of language. At the heart of our everyday linguistic functioning are abilities that we only become aware of through introspection.

Crucially, there is an in principle infinite number of combinations of words for which we are (in principle) able to decide whether they are grammatical or not. For instance, we know that (2) remains grammatical when we add a prepositional phrase:

(3) Colourless green ideas sleep furiously in my head.

Or several prepositional phrases:

(4) Colourless green ideas sleep furiously in a dream in my head.

Or think of the following, perhaps clearer example:

(5) The cat stood on a table in a room in a castle on a hill in a country where people wear hats with feathers from birds from a forrest near a lake ...

Any speaker of English can and will decide that (5) is grammatical. There are obvious practical reasons

why natural language sentences are never infinite, but our introspection tells us that they seem to allow for infinitely repeating patterns. As such, our linguistic competence is infinite. Crucially, we achieve this infinite potential through finite means, given that our brains are finite.

This link with infinity highlights the commonality between natural and formal languages. Our knowledge of a natural language involves the decision problem that asks to distinguish grammatical from ungrammatical sentences and, so, there is a formal language that corresponds to that decision problem. What is more, we are interested in understanding what finite computational means allow us humans to capture this infinite formal language.

In the examples above, I associate knowledge of language with the ability of deciding on the grammaticality of a sentence. But there are many different kinds of linguistic knowledge. Parallel to the examples above, there is a similar case to be made that you have knowledge of how individual words are built. If you are a speaker of English, you will know that you can add the prefix "re" to some verbs, to get another verb. For instance, "discover" becomes "rediscover", "invent" becomes "reinvent" etc. Adding "-y" or "-ion" to these turns the verb into a noun: "discovery", "rediscovery", "invention", "reinvention". If you are a native speaker of English, you were never instructed that this is how things work. This ability has simply emerged as part of your knowledge of language.

In summary, our linguistic abilities are a good example of a set of *human* abilities that involve deciding on membership in an infinite formal language. So, from the context of artificial intelligence, it would make sense to understand what kind of formal languages are part of our human linguistic competence. What is their complexity? How do they relate to formal languages we know more generally from our theory of computation? These are the kind of questions we will approach in what follows.

2 Finite state automata

How do we study decision problems and computability? Turing proposed to use abstract machines. In particular, he proposed a mathematical model of computation that we now call *the Turing machine*. A Turing Machine is a formal model that mimics an imaginary machine that consists of a tape that is segmented into an infinite number of cells, a head that can move along the tape and read the contents of a cell as well as write content to a cell, and a mechanism that controls the actions of the machine based only on what the head reads and the *state* the machine is in. We will encounter a Turing Machine later in these lecture notes. For now, however, we turn to a much simpler abstract model of computation, namely the finite state automaton (FSA). As we will see, the difference in complexity between a Turing Machine and an FSA has consequences for the kind languages that can be computed. In fact, we will see that by studying models of computation of differing complexity, we get insights about different classes of decision problems.

The task of an FSA is to receive an input string and to decide on whether to accept it or not. In other words, FSAs solve a particular decision problem for the input they receive. Acceptance means that the string is a member of the language corresponding to the decision problem the FSA is meant to solve. Not accepting a string means that the string does not belong to the language. An FSA is an automaton that can be in one of a finite number of possible states. At the moment the FSA receives input, it is in a dedicated start state. Every FSA has one or more acceptance states. A string is accepted whenever the FSA is in one such state when the input has been read completely.

A finite state automaton reads the input symbol by symbol. For each state the automaton can be in, there is a specification of what to do when a certain symbol is read. The possible actions are extremely limited, however. The only thing an automaton can do is either remain in the same state or transition to a different state before proceeding to read the next symbol.

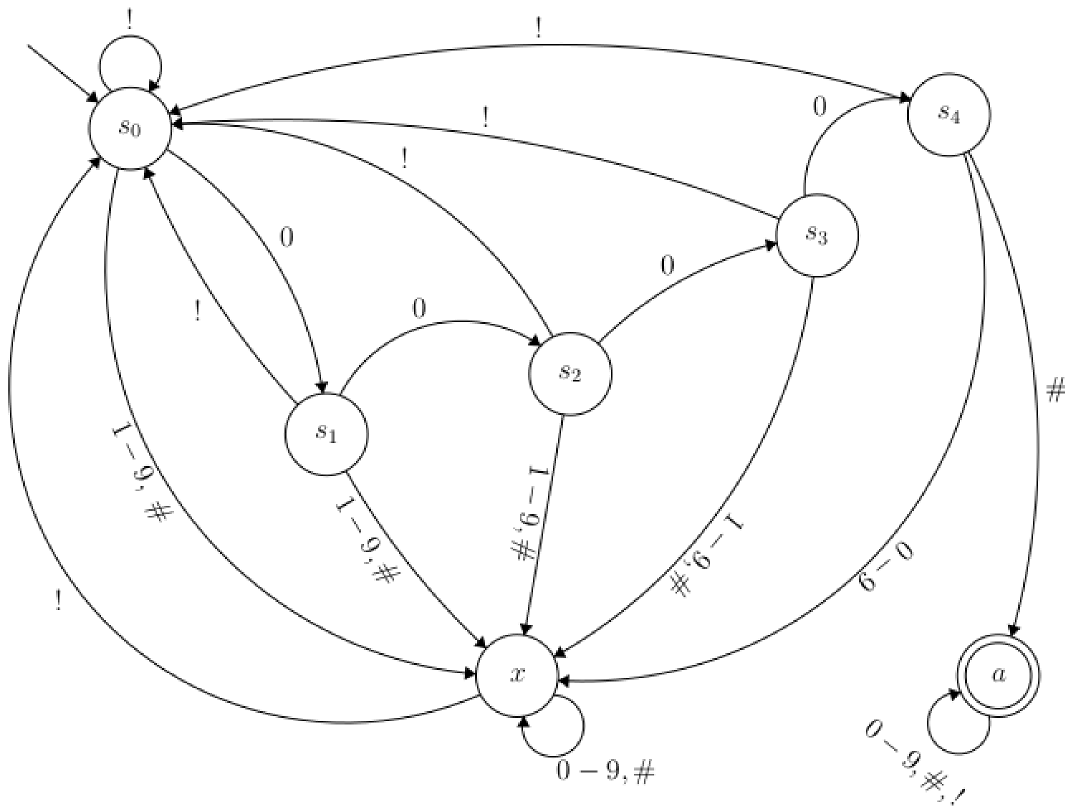
FSAs are abstract models, but there are examples of actual machines that resemble finite state automata. It is illustrative to start by looking at such a machine. Take, for instance, a digital lock on the door of a safe. The lock has two states: locked or unlocked. When it receives input, it is always in the locked state. It has a key pad that can record an input, namely a string of key presses. There is a key for each numerical digit, as well as for the symbols "!" and "#". The user can type in a code and send the code by pressing "#". If the user makes a mistake, he or she can type "!" to start completely from scratch. Let's say the code is "0000". We can now say that the unlocked state is an acceptance state and the locked state is not. The task of the digital lock is to accept strings like 0000#, 9!0000#, 0000!9!0000#, etc. and to not accept strings like 01234#, 0000, 0000!#, 0000000#, etc. That is, the safe unlocks only when a string of key presses occurs that correspond to the sending of the "0000" code.

Working locks like this actually exist. However, irrespective of how such real locks work, we can represent the task they perform as a formal language over alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \#, !\}$. The set of strings that unlock the safe is a subset of Σ^*. We can define a finite state automaton that computes this language in the way just described. Below, we will give a precise formal definition of FSAs, which will make it possible to define a particular automaton as a fully specified mathematical object. But before we turn to these formal definitions, we will look at an intuitive way of representing automata. It is often handy to represent models of computation graphically. For FSAs, we do this in the following way:

- states are circles, with the name of the state written in the circle
- acceptance states are indicated by a double line
- arrows indicate transitions between states triggered by the reading of a symbol, where we label the arrow with the responsible symbol

- the start state is indicated with an incoming arrow that is not connected to any other state

Here is an example of an automaton for the digital lock I just described:



As you can see, the automaton has 7 states. One of these is the start state (s_0) and one of them is an acceptance state (a). The only way this FSA will get into the acceptance state is by reading the symbol 0 four consecutive times, immediately followed by a "#". Such an input has the automaton transition from s_0 to s_1, s_2, s_3, s_4 and finally a. Anything else will either get the automaton into the state x or, if a ! is typed, back to the start state. From x, the only way to get further is to go back to s_0 by reading a "!".

2.1 Formal definition

Graphic depictions of automata are handy because they will help you to visualize what happens in an automaton given certain inputs. Any FSA, however, is first and foremost a formal object. That is, the automaton sketched here can not just be given graphically, it can also be defined in mathematical terms. To do so, let us formally define finite state automata:

Definition 4

A finite state automaton is a 5-tuple $\langle \Sigma, S, s, A, R \rangle$, such that:

- Σ is a finite set
- S is a finite set
- $s \in S$
- $A \subseteq S$
- $R \subseteq (S \times \Sigma) \times S$

Let's unpack this. Σ is the alphabet of the language that the automaton is to compute. So, each input is a string in Σ^*. S is the set of states that make up the automaton and s is its unique element that acts

as the start state. A is that subset of S of accepting states. Finally, R is where all the work happens. R is a relation between pairs of states and symbols (elements of $S \times \Sigma$) and states. It is often handy to represent R as a table, *the transition table* of the automaton, where the rows represent states, columns represent symbols and cells represent to which state the machine should transition when a certain symbol is read in a certain state.

To illustrate, here is a formal specification of the lock automaton that is graphically depicted above.

$$\langle\{0,1,2,3,4,5,6,7,8,9,!,\#\},\{a,x,s_0,s_1,s_2,s_3,s_4\},s_0,\{a\},R\rangle$$

where R is the relation given by the following table:

	0	1	2	3	4	5	6	7	8	9	!	#
s_0	s_1	x	x	x	x	x	x	x	x	x	s_0	x
s_1	s_2	x	x	x	x	x	x	x	x	x	s_0	x
s_2	s_3	x	x	x	x	x	x	x	x	x	s_0	x
s_3	s_4	x	x	x	x	x	x	x	x	x	s_0	x
s_4	x	x	x	x	x	x	x	x	x	x	s_0	a
x	x	x	x	x	x	x	x	x	x	x	s_0	x
a	a	a	a	a	a	a	a	a	a	a	a	a

Say this automaton receives the input 0!0000#. The table shows the states it will go through when reading this input. The automaton starts in s_0 and reads 0. As the top left-most cell says, this leads to a transition to s_1. The next symbol is ! which from s_1 leads back to s_0. The rest of the string will lead the machine through s_1, s_2, s_3, s_4 and, finally, a. Given that at the end of the string, the machine is in an acceptance state, the string is accepted.

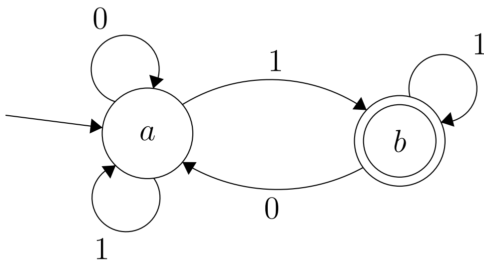
FSAs can be quite simple. Consider for instance the following:

$$\langle\{0,1\},\{s_0,s_1\},s_0,\{s_0\},\{((s_0,0),s_0),((s_0,1),s_1),((s_1,0),s_1),((s_1,1),s_0)\}\rangle$$

This automaton reads strings in $\{0,1\}^*$. It accepts only those that contain an even number of 1s (irrespective of the number of 0s it contains). (It also accepts string that contain no 1s, so I am assuming that 0 is even.) It does this by accepting any string it reads until it encounters a 1, which triggers a transition to the non-acceptance state s_1. From there, the only way to get back into an acceptance state is by once more reading a 1. So, every odd 1 that is read moves the machine into non-acceptance and only the next 1 moves it back to acceptance.

2.2 Non-determinism

In an FSA $\langle\Sigma,S,s,A,R\rangle$ it is the transition table R that regulates what happens when a particular symbol is encountered in a particular state. In the examples we have seen so far R was such that it unambiguously determines what happens in each possible situation. Such an automaton is called *deterministic*: given a string, there is a unique sequence of transitions that the automaton will go through. A *nondeterministic finite state automaton* (NFSA) is a finite state automaton that does not provide a unique sequence of transitions for each output. Here is a simple example of an NFSA.



This is an automaton over alphabet $\{0, 1\}$. It is nondeterministic because whenever a “1” is read in state a, there are two candidate transitions: either the automaton remains in state a, or it transitions into b. To distinguish between deterministic and nondeterministic automata, it is a good idea to have a closer look at the transition table component of FSAs.

As I wrote above, the transition table R is a relation, a subset of $(S \times \Sigma) \times S$. Recall first of all that any subset Z of a Cartesian product $X \times Y$ is a relation. Next, recall that such a relation Z is a function if and only if whenever both $(x, y) \in Z$ and $(x, y') \in Z$, then $y = y'$. In other words, a function $Z \subset X \times Y$ is a relation whenever each element in X is paired with at most 1 element in Y. The definition of a finite state automaton states that the transition specification R is a relation. An FSA is deterministic, whenever this transition relation is a function that maps pairs of states and symbols to states.

Definition 5

A deterministic finite state automaton is a 5-tuple $\langle \Sigma, S, s, A, R \rangle$ where

- Σ is a finite set
- S is a finite set
- $s \in S$
- $A \subseteq S$
- $R : (S \times \Sigma) \rightarrow S$

Nondetermistic automata can also be viewed as having a functional transition relation. However, in a non-deterministic machine, each state-symbol pair does not yield a unique state to transition to, but rather a set of states. This means we can define NFSAs as follows:

Definition 6

A nondeterministic finite state automaton is a 5-tuple $\langle \Sigma, S, s, A, R \rangle$ where

- Σ is a finite set
- S is a finite set
- $s \in S$
- $A \subseteq S$
- $R : (S \times \Sigma) \rightarrow \wp(S)$

For example, the NFSA given above can be formally represented as

$$\langle \{0, 1\}, \{a, b\}, a, \{b\}, \{((a, 0), \{a\}), ((a, 1), \{a, b\}), ((b, 1), \{b\}), ((b, 0), \{a\})\} \rangle$$

If we present a deterministic FSA with an input, it is easy to check whether the input is accepted or not, since the transition function fully specifies what to do with the input. (The next section formalises this.) Things are different for non-deterministic FSAs. There will be inputs that present us with a choice at certain points in reading the string. As such, we would need a strategy for navigating through such choices in order to be able to decide whether a string is recognised by an NFSAs. So, figuring out whether an NFSA accepts a string or not is much harder than figuring out

whether a deterministic automaton accepts a string. Given this, you may wonder why we bother discussing NFSAs in the first place. Well, first of all, an NFSA tends to be much smaller in size than a deterministic FSA that does the same thing. Second, there exist quite a few efficient algorithms that help us determine whether a string is accepted by a non-deterministic automaton or not. So, practically speaker, it is sometimes simply preferable to work with NFSAs.

In terms of expressive power, it turns out that the choice between deterministic and non-deterministic finite state automata is immaterial. For any language recognized by some deterministic FSA, there exists a non-deterministic FSA recognizing that same language, and vice versa.

Theorem 3

(i) Any non-deterministic FSA $\mathcal{N}$ is such that there exists a deterministic FSA $\mathcal{D}$ such that $\mathcal{L}(\mathcal{N}) = \mathcal{L}(\mathcal{D})$. Conversely, (ii) for any deterministic FSA $\mathcal{D}$, there exists a non-deterministic FSA $\mathcal{N}$, such that $\mathcal{L}(\mathcal{N}) = \mathcal{L}(\mathcal{D})$.

I will not provide the proof here, but hope to give you some intuition. Note first of all that (ii) is somewhat trivial. In fact, given definition 6, any deterministic automaton can be rewritten as a non-deterministic one simply by changing the transition function $R : (S \times \Sigma) \rightarrow S$ into a function $R' : (S \times \Sigma) \rightarrow \wp(S)$. We can do this as follows: R' is the function such that for any pair $(x, a) \in S \times \Sigma$: $R'((x, a)) = \{R(x, a)\}$. So, the nondeterministic version of the deterministic automaton is the automaton that maps each combination of a state and a symbol to the singleton set containing the state the deterministic version maps that combination to.

Part (ii) of the theorem is trickier and was first proved by Rabin and Scott in 1959. One way to see that (ii) is the case is the existence of reliable strategies to transform any nondeterministic automaton into a (usually considerably larger) deterministic one. This is outside the scope of the current course.

2.3 Acceptance

So far, we only have had an informal understanding of what happens when an automaton receives an input. We know that input strings are read symbol by symbol and that the transition function determines for each read symbol what to do, based on the state the FSA is in. The input is accepted if and only if the automaton is in an acceptance state after reading the final symbol. Here is a formal definition of acceptance.

Definition 7

A finite state automaton $\mathcal{M} = \langle \Sigma, S, s, A, R \rangle$ *accepts* an input string $w_1 \dots w_n$, if and only if there exists a *computation* $s_0, s_1, \dots, s_n$ such that:

- for all $0 \leq i \leq n$: $s_i \in S$.
- $s_0 = s$
- $s_n \in A$
- for all $1 \leq i \leq n$: either $s_i \in R((s_{i-1}, w_i))$ or $R((s_{i-1}, w_i)) = s_i$

Notation: We write $\mathcal{M} \rhd x$ whenever $\mathcal{M}$ accepts x and $\mathcal{M} \not\rhd x$ whenever it does not.

This says that a string gets accepted by an automaton whenever we can go through the string, symbol by symbol, and for each symbol we can find a transition in such a way that, if we start in the start state the machine will transition to an acceptance state after reading the final symbol. The last line in this definition is a bit complex. This is because I am assuming that $\mathcal{M}$ can be of two types. Either it is deterministic, in which case $R((s_{i-1}, w_i))$ will always point to at most one value, or it is nondeterministic in which case it will map to a set of states.

Here's an example. Consider the automaton

$$\mathcal{X} = \langle \{1\}, \{a, b, c, d\}, a, \{c\}, R_{\mathcal{X}} = \{((a, 1), b), ((b, 1), c), ((c, 1), d), ((d, 1), d)\} \rangle$$

This automaton accepts only a single string, namely "11". It rejects anything shorter or longer in $\{1\}^*$. The string "11" is accepted because there is a computation abc such that $R_{\mathcal{X}}((a, 1)) = b$ and $R_{\mathcal{X}}((b, 1)) = c$. There is no other string that has a computation with the required properties. For instance, the computation that goes with "111" is $abcd$, but $d \notin A$ and, so, the string is not accepted. Similarly, the string "1" is computed by "ab". Here, too, the final state, b, is not an acceptance state, so the string is not accepted.

We are now ready to finally connect finite state automata to formal languages. Given the definition of acceptance that I defined above, this is very straightforward. The language computed by an automaton is the set of strings that it accepts:

Definition 8

Let $\mathcal{M}$ be a finite state automaton. Its corresponding formal language is written as $\mathcal{L}(\mathcal{M})$ and is defined as:

$$\mathcal{L}(\mathcal{M}) = \{x \mid \mathcal{M} \rhd x\}$$

2.4 Determinism

Recall that a function $f \subseteq A \times B$ is a relation whenever for each $a \in A$ it is the case that $(a, x) \in f$ & $(a, y) \in f \Rightarrow x = y$. There exist relations that are functional in this sense, but which do not provide an output for each input in the domain. These functions are called *partial functions*: a function $f : A \to B$ is partial whenever there exists $x \in A$ such that there exists no $y \in B$ such that $(x, y) \in f$. That is, for some x, $f(x)$ will be undefined. If f is a partial function, we often write $f(x) \downarrow$ for $f(x)$ being defined and $f(x) \uparrow$ for f is undefined.

The definition of a deterministic FSA states that the transition table needs to be a function $R : (S \times \Sigma) \to S$. It does not state, however, whether R need to be a total function or whether it is partial. That is, we could define an automaton that fails to specificy transitions for some configurations the automaton can encounter. Take, for example, the following variation on $\mathcal{X}$:

$$\mathcal{X}' = \langle \{1\}, \{a, b, c, d\}, a, \{c\}, R_{\mathcal{X}'} = \{((a, 1), b), ((b, 1), c), ((c, 1), d)\} \rangle$$

$\mathcal{X}'$ differs from $\mathcal{X}$ only in that its transition function is not defined for $(d, 1)$. It turns out that this does not really matter so much for our definition of acceptance. Take for instance string "1111". We have both $X \not\rhd 1111$ and $\mathcal{X}' \not\rhd 1111$. The reason for not accepting the string, however, is different in both cases. For a string to be accepted by an FSA, two things have to be the case: (A) there has to be a computation for the string; (B) this computation needs to have certain properties (start in a starting state, end in an acceptance state). In $\mathcal{X}$, "1111" is not accepted because of (B): d is not an acceptance state. In $\mathcal{X}'$, "1111" is not accepted because of (A), the lack of a computation.

We can thus distinguish two kinds of determinism:

Definition 9

A finite state automaton $\langle \Sigma, S, s, A, R, \rangle$ is p-deterministic whenever:

- Σ is a finite set
- S is a finite set
- $s \in S$
- $A \subseteq S$
- $R : (S \times \Sigma) \to S$ and R is a partial function

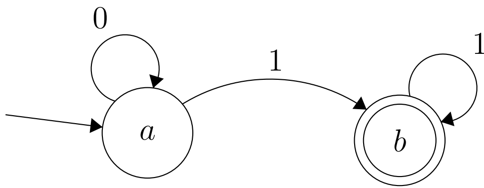
A finite state automaton is t-deterministic, when it is deterministic, but not p-deterministic.

There is a simple procedure that can turn a p-deterministic automaton into a deterministic FSA with a total transition function.

Let $\mathcal{A} = \Sigma, S, s, A, R\rangle$ be p-deterministic. Let σ be a state such that $\sigma \notin S$. The t-deterministic version of $\mathcal{A}$ is defined below. (Recall, that $\downarrow$ indicates that the function applied to its arguments is *defined* and $\uparrow$ indicates that it is not defined.)

$$\langle \Sigma, S \cup \{\sigma\}, s, A, R' \rangle$$
$$\text{where for any } (x, y) \in S \times (\Sigma \cup \{\sigma\}):$$
$$R'(x, y) = \begin{cases} R((x, y)) & \text{if } R((x, y)) \downarrow \\ \sigma & \text{if } R((x, y)) \uparrow \end{cases}$$

Take for example the following p-deterministic automaton. (It's p-deterministic since there is no defined transition for reading 0 in state b).



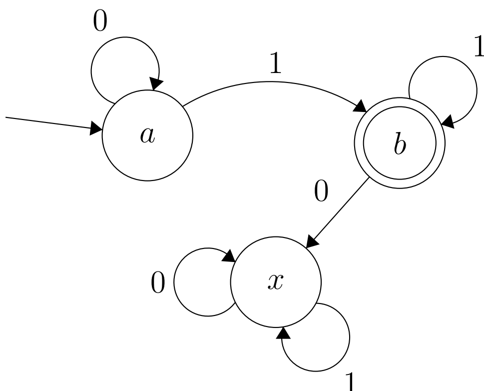
Assuming that the alphabet is $\{0, 1\}$, this is the automaton given by

$$\langle \{0, 1\}, \{a, b\}, a, \{b\}, \{((a, 0), a), ((a, 1), b), ((b, 1), 1)\} \rangle$$

To make this automaton t-deterministic, we add a state x and make sure that cases that are undefined trigger transition to x. So, the altered automaton is

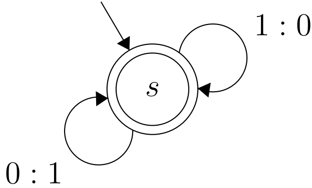
$$\langle \{0, 1\}, \{a, b, x\}, a, \{b\}, \{((a, 0), a), ((a, 1), b), ((b, 1), b), ((b, 0), x), ((x, 0), x), ((x, 1), x)\} \rangle$$

Graphically, this corresponds to:



2.5 Finite state transducers

A much used computational metaphor used for finite state automata is that they involve a control mechanism that sequentially reads input from a finite tape. The mechanism keeps track of where on the tape the automaton is and what state the automaton is. In this section, we move to an important variation on finite state automata, called *finite state transducers* or FSTs. Sticking with the metaphor, an FST is like an FSA but with two instead of one finite tape. The benefit of an FST is that we can think of the two tapes as playing different roles. In particular, one could see one of the tapes as providing an input and the other tape providing the output. Here is an example of a simple finite state transducer:



In graphical depictions of FSTs like this one, we use the colon as a special symbol that distinguishes the two tapes. We can for instance think of symbols to the left of ":" as part of the input and symbols to the right as part of the output. As such, this FST reads a string of arbitrary combinations of 1s and 0s and replaces it with string where every occurrence of a 1 is replace by a 0 and vice versa. So, this FST recognises input-output pairs like 010001:101110 and 000:111 and does not accept pairs like 00:01 or 111:0000.

Formally, a finite state transducer is more complex than a finite state automaton. This is because the FST distinguishes two (possibly distinct) alphabets, one for each side of the colon. Furthermore, there are two transition functions, also for both strings under consideration.

Definition 10

A finite state transducer is a 7-tuple $\langle \Sigma_1, \Sigma_2, S, s, A, R_1, R_2 \rangle$ where

- Σ_1 is a finite set
- Σ_2 is a finite set
- S is a finite set
- $s \in S$
- $A \subseteq S$
- $R_1 : (S \times \Sigma_1) \rightarrow S$
- $R_2 : (S \times \Sigma_1) \rightarrow \Sigma_2$

The above simple FST is given as:

$$\langle \{0, 1\}, \{0, 1\}, \{s\}, s, \{s\}, \{((s, 1), s), ((s, 0), s)\}, \{((s, 1), 0), ((s, 0), 1)\} \rangle$$

The input part of an FST accepts strings just like an FSA would. A string $w_1 \ldots w_n$ is accepted if there exists a computation $s_0 s_1 \ldots s_n$ such that: (i) s_0 is the start state, (ii) s_n is an acceptance state and (iii) for each $1 \leq s_i \leq n$: $R((s_{i-1}, w_i)) = s_i$. The following definition extends acceptance to include the output string. (Here, we once more use ":" as a special symbol to distinguish the input from the output).

Definition 11

A finite state transducer $\mathcal{T} = \langle \Sigma_1, \Sigma_2, S, s, A, R_1, R_2 \rangle$ *accepts* an input-output pair $w_1 \ldots w_n : o_1 \ldots o_n$ if and only if there exists a computation $s_0, s_1, \ldots, s_n$ such that:

- for all $0 \leq i \leq n : s_i \in S$
- $s_0 = s$
- $s_n \in A$
- for all $1 \leq i \leq n$: $R_1((s_{i-1}, w_i)) = s_i$
- for all $1 \leq i \leq n$: $R_2((s_{i-1}, w_i)) = o_i$

Notation: We write $\mathcal{T} \rhd x : y$ whenever $\mathcal{T}$ accepts $x : y$ and $\mathcal{T} \not\rhd x : y$ whenever it does not.

Practically, FSTs are used to *translate* or *manipulate* input strings. That is, they are employed when we need produce an output string that is based on the input string. Note, however, that the definition given here defines *acceptance* of both the input and the output, rather than just defining acceptance of the input and calculating the corresponding output. So, even though transducers can be seen as mapping an output to an input, they can be defined in terms of a decision problem. The problem of knowing which output goes with what input can be reduced to the problem of recognizing a set of input output pairs. Put differently, while FSAs correspond to formal languages, i.e. sets of strings, FSTs correspond to relations, sets of pairs of strings.

3 Regular languages

As we saw above, there are uncountably infinitely many languages, whenever the alphabet is non-empty. We can divide the set of languages up in interesting subclasses, however. We start with so-called regular languages: the class of languages recognized by finite state automata. To define such languages, we need the following operation:

Definition 12

Set concatenation: If L_1 and L_2 are two sets of strings, then $L_1 \cdot L_2 = \{l_1 \frown l_2 | l_1 \in L_1$ and $l_2 \in L_2\}$.

For example $\{1, 2\} \cdot \{1\} = \{11, 21\}$. Given this operation, we can provide a recursive definition of regular languages.

Definition 13

The set of regular language over Σ, notation $\mathcal{R}_\Sigma$, is recursively defined as follows:

- Base: $\emptyset \in \mathcal{R}_\Sigma$, $\{\epsilon\} \in \mathcal{R}_\Sigma$ and for each $\sigma \in \Sigma$: $\{\sigma\} \in \mathcal{R}_\Sigma$.
- Recursive step: If $L_1 \in \mathcal{R}_\Sigma$ and $L_2 \in \mathcal{R}_\Sigma$, then
  - $L_1 \cdot L_2 \in \mathcal{R}_\Sigma$,
  - $L_1 \cup L_2 \in \mathcal{R}_\Sigma$
  - $L_1^* \in \mathcal{R}_\Sigma$ and $L_2^* \in \mathcal{R}_\Sigma$
- Nothing else is in $\mathcal{R}_\Sigma$.

Take for example $\Sigma = \{0, 1\}$. The definition above now states that sets like $\{0\}$ and $\{1\}$ are regular languages and so are their concatenations $\{01\}$, $\{10\}$, $\{00\}$, $\{11\}$. Also, unions of these sets are regular, such as for instance $\{1, 01, 11\}$ or $\{0, 00\}$. Concatenations of such sets are also regular again: $\{10, 100, 010, 0100, 110, 1100\}$. Given the complexity of languages that one can build using just concatenation and union, you may wonder what the Kleene closure recursive step adds to the definition in regular languages. It matters in a small but important way. If we had used an alternative definition, one that did not include Kleene closure as a recursive step, then the definition would in fact describe an importantly different class of languages. Without Kleene, in any regular language strings of length exceeding 1 would result from a finite number of applications of the concatenation step. But that means that for any regular language, there is an upper bound to the length of the strings it contains. As such, all regular languages would be finite. The addition of the Kleene recursive step allows for infinite regular languages.

Note that $\Sigma^* = \{\epsilon\} \cup \Sigma \cup \Sigma \cdot \Sigma \cup \Sigma \cdot \Sigma \cdot \Sigma \cup \ldots$. This in turn may make you wonder why the definition includes concatenation as a recursive step. This is to include finite languages as regular. Without this step, we would only have $\emptyset$ and the singleton languages as finite regular languages.

3.1 Regular languages and finite state automata

Theorem 4

The class of languages for which there is some finite state automaton that computes that language is exactly the class of regular languages.

I will not prove this theorem here, but I will sketch the proof for one part of the theorem, which will hopefully make things more intuitive.

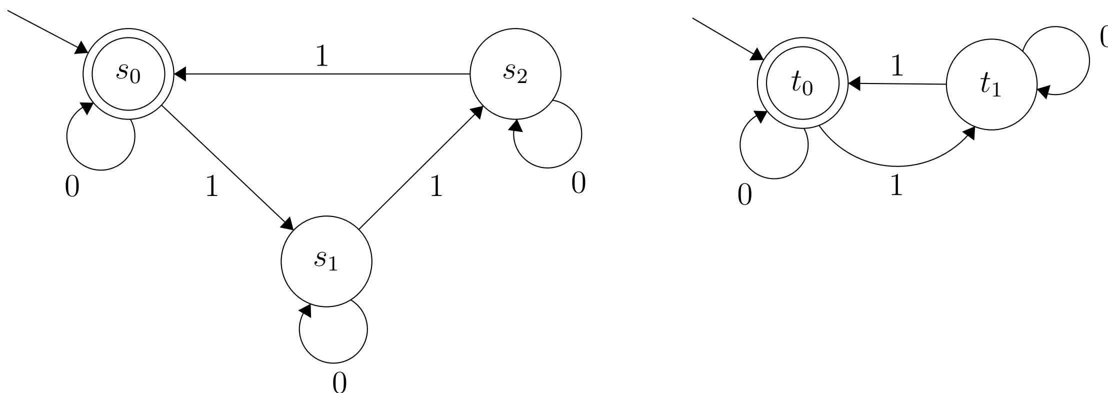
Theorem 5

For each regular language L, there is a finite state automaton $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = L$.

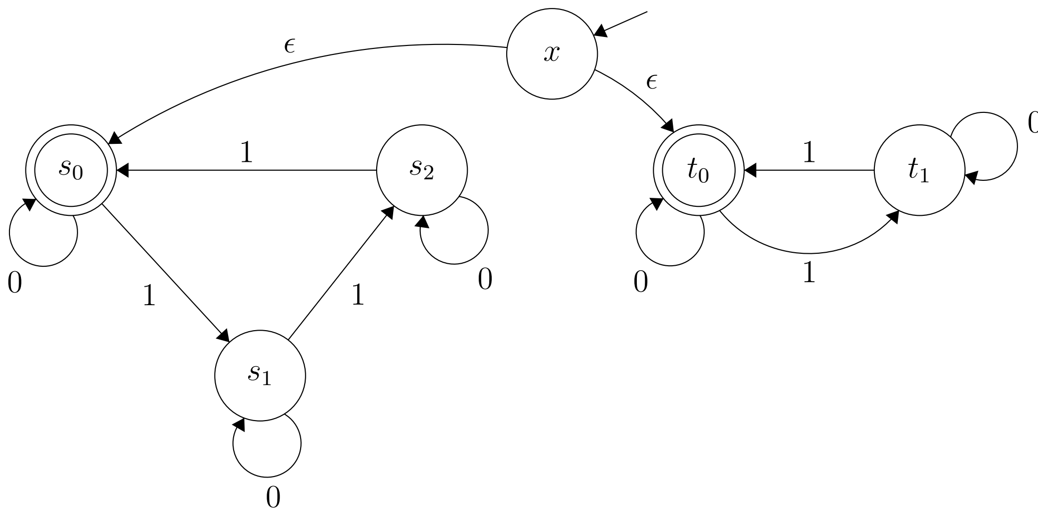
Proof

Recall that there are three basic cases of regular languages and three possible recursive steps. What we need to show is that all these cases can be handled by FSAs.
Base case 1, the regular language $\emptyset$: Just take an FSA without an accepting state: $\langle \Sigma, \{s\}, s, \emptyset, \emptyset \rangle$.
Base case 2, the regular language $\{\epsilon\}$: This is a minimal variation on base case 1: $\langle \Sigma, \{s\}, s, \{s\}, \emptyset \rangle$.
Base case 3, regular languages $\{\sigma\}$ for $\sigma \in \Sigma$: $\langle \Sigma, \{s, t\}, s, \{t\}, \{((s, \sigma), t)\}$
Recursive steps: Say we have $\mathcal{L}(\mathcal{M}_1) = L_1$ and $\mathcal{L}(\mathcal{M}_2) = L_2$. According to the definition of regular languages, whenever L_1 and L_2 is regular, then so is $L_1 \cdot L_2$, $L_1 \cup L_2$, L_1^* and L_2^*. So, for the current proof we need to show that there is an FSA $\mathcal{M}_{1 \cdot 2}$ such that $\mathcal{L}(\mathcal{M}_{1 \cdot 2}) = L_1 \cdot L_2$, there is an FSA $\mathcal{M}_{1 \cup 2}$ such that $\mathcal{L}(\mathcal{M}_{1 \cup 2}) = L_1 \cup L_2$ and there is an FSA $\mathcal{M}_{1*}$ such that $\mathcal{L}(\mathcal{M}_{1*}) = L_1^*$. The following procedures will deliver these FSA. Let $\mathcal{M}_1 = \langle \Sigma, S_1, s_1, A_1, R_1 \rangle$ and $\mathcal{M}_2 = \langle \Sigma, S_2, s_2, A_2, R_2 \rangle$.
Recursive step 1, concatenation: $\mathcal{M}_{1 \cdot 2} = \langle \Sigma, S_1 \cup S_2, s_1, A_2, R_1 \cup R_2 \cup \{((a, \epsilon), s_2) | a \in A_1\}\rangle$
Recursive step 2, union: $\mathcal{M}_{1 \cup 2} = \langle \Sigma, S_1 \cup S_2 \cup \{x\}, x, A_1 \cup A_2, R_1 \cup R_2 \cup \{((x, \epsilon), s_1), ((x, \epsilon), s_2)\}\rangle$.
Recursive step 3, Kleene closure: $\mathcal{M}_{1*} = \langle \Sigma, S_1 \cup \{x\}, x, A_1 \cup \{x\}, R_1 \cup \{((a, \epsilon), s_1) | a \in A_1\} \cup \{((x, \epsilon), s_1)\}\rangle$

I illustrate recursive step 3 with an example. First consider X, the language over $\{0, 1\}$ that contains strings of arbitrary combinations of 1s and 0s as long as the number of 1s is zero or a multiple of 3, and the language Y over $\{0, 1\}$ which is similar except that all strings are such that the number of 1s is zero or a multiple of 2. Here are two automato that recognise these languages.



The union of the two languages, $Z = X \cup Y$, is the language that contains arbitrary combinations of 0s and 1s as long as the number of 1s is either zero, a multiple of 3 or a multiple of 2. So, Z contains strings like 00001010001 and 000100001 but not 0001010001011. To build an automaton for Z we can use the recipe for recursive step 2 from the proof idea above:



3.2 Closure properties

We say that a set is closed under a certain operation whenever performing the operation on members of the set results in another member of the set. For instance, the natural numbers are closed under squaring. This is because whenever $x \in \mathbb{N}$, then x^2 is also in $\mathbb{N}$. The natural numbers are not closed under the square root operation. For instance, $\sqrt{2} \notin \mathbb{N}$.

Given the definition of regular languages, we already know that they are closed under union, concatenation and Kleene star. It can be important to know of such closure properties. This is because it may help us understand relations between decision problems. For instance, if we know that a certain problem can be seen as the concatenation of two problems solvable by finite state automata, then we know we can solve this problem using an FSA, too.

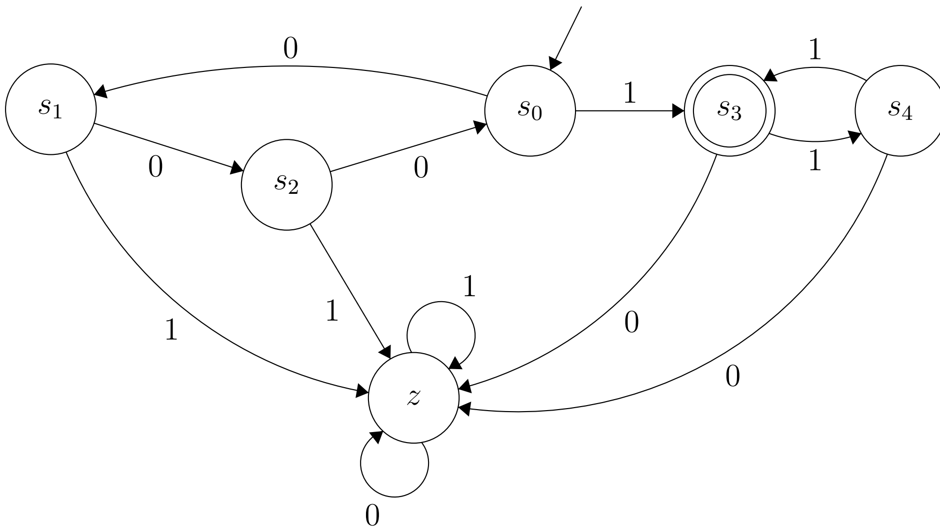
Beyond the three cases that follow directly from the definition of regular languages, this class has further closure properties. Here, I only discuss one of them, namely the case of *complementation*. Say that L_Σ is some language over Σ. The **complement** of L_Σ is set of strings in Σ^* that are not in L. So: $\overline{L_\Sigma} = \Sigma^* \setminus L$. There is an easy procedure that allows you to construct an FSA for the complement of a language on the basis of an automaton for the original language.

Definition 14

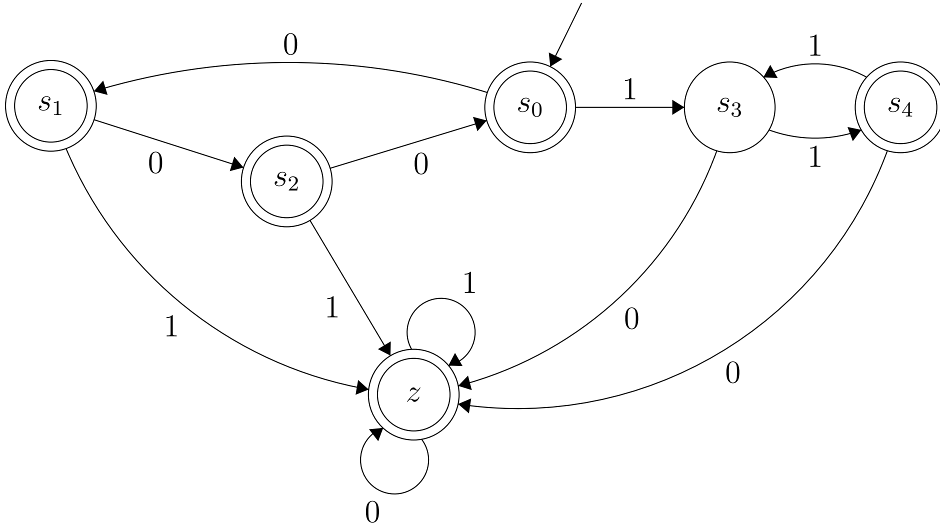
Let $\mathcal{M} = \langle \Sigma, S, s, A, R \rangle$ be t-deterministic. The FSA $\overline{\mathcal{M}}$ is defined as:

$$\overline{\mathcal{M}} = \langle \Sigma, S, s, S \setminus A, R \rangle$$

Take, for example, the following FSA, which recognizes $\{0^n 1^m \mid n \text{ is even, and } m \text{ is odd}\}$. (Note that 0 is an even number, so the FSA also accepts string like 1 or 111).

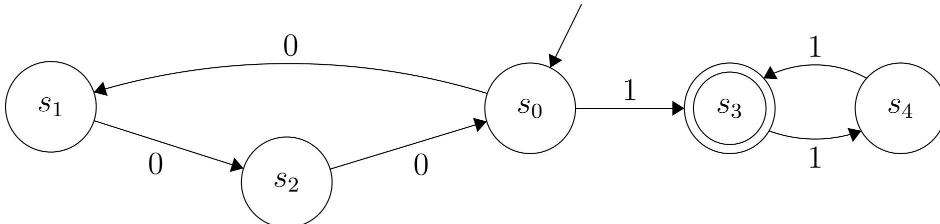


As per definition 14, we can take the complement of this FSA to recognize the language that contains strings that are not a sequence of an even number of 0s followed by an odd number of 1s. This complement language contains strings like 01 and 0011, but also strings like 10100. The complement automaton is simply the following:

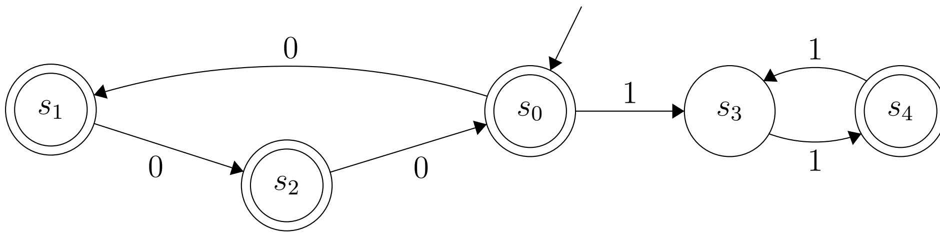


Note that this FSA just accepts anything as long as it doesn't finish reading the input in s_3, because that would result in a string that belongs to the original language.

It is important that complementation takes place with a t-deterministic automaton. Consider the following p-deterministic FSA.



This FSA also corresponds to $\{0^n 1^m \mid n \text{ is even, and } m \text{ is odd}\}$. For instance, it fails to accept 000001 because there is no computation for that string that results in an acceptance state. But if we were take the complement of this FSA, we'd get the following:



Crucially, this is not the complement of $\{0^n1^m \mid n \text{ is even, and } m \text{ is odd}\}$. For instance, it fails to accept strings like 01 or 1000.

Theorem 6

Whenever $\mathcal{M}$ is t-deterministic, then $\mathcal{L}(\overline{\mathcal{M}}) = \overline{\mathcal{L}(\mathcal{M})}$.

Proof

Say that $\mathcal{M} = \langle \Sigma, S, s_0, A, R \rangle$. The assumption is that $\mathcal{M}$ is t-deterministic. That means that for any string $w_1 \ldots w_n$ in Σ^* there is a computation $s_0 \ldots s_n$, such that for any w_i, $((s_{i-1}, w_i), s_i) \in R$. There are two kinds of computations of this kind: if $s_n \in A$, then this is the computation of an accepted string and if $s_n \notin A$, then it is the computation of a string that is not accepted. But this immediately means that we can swap accepted and non-accepted strings by just swapping acceptance states with states that are not acceptance. It follows from the definition of acceptance and $\mathcal{L}(\ )$ that $\mathcal{L}(\overline{\mathcal{M}}) = \overline{\mathcal{L}(\mathcal{M})}$.

Now everything is in place to prove that regular languages are closed under complementation.

Theorem 7

If L_Σ is a regular language over Σ, then so is $\overline{L_\Sigma}$.

Proof

Assume that L is regular. Since L is regular, there exists a t-deterministic FSA $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = L$. Given theorem 6, we know that $\mathcal{L}(\overline{\mathcal{M}}) = \overline{L}$. And so there exists an FSA corresponding to $\overline{L}$, namely $\overline{\mathcal{M}}$. Given theorem 4, this means that $\overline{L}$ must be regular.

3.3 Non-regularity

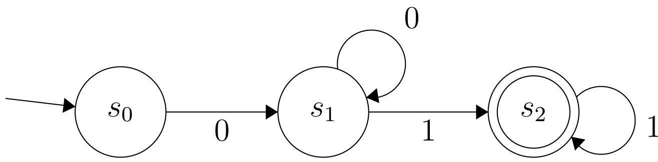
We've defined regular languages and we have seen that these are exactly the languages for which we can build a finite state automaton. The original abstract model of computing put forward by Alan Turing was a model that is quite a bit more complex than the FSAs we've discussed so far. One crucial difference (but by not means the only difference) is that a Turing Machine can not only read the input string, it can also write symbols and revisit what it has written down at a later stage in the computation. That is, a Turing Machine has a memory mechanism, while a finite state automaton does not. This difference matters. Turing Machines can compute a proper superset of the formal languages that FSAs can. Here is a classical example of a language that is **not regular**, a language for which there is no finite state automaton.

$$\{0^n1^n \mid n \geq 1\}$$

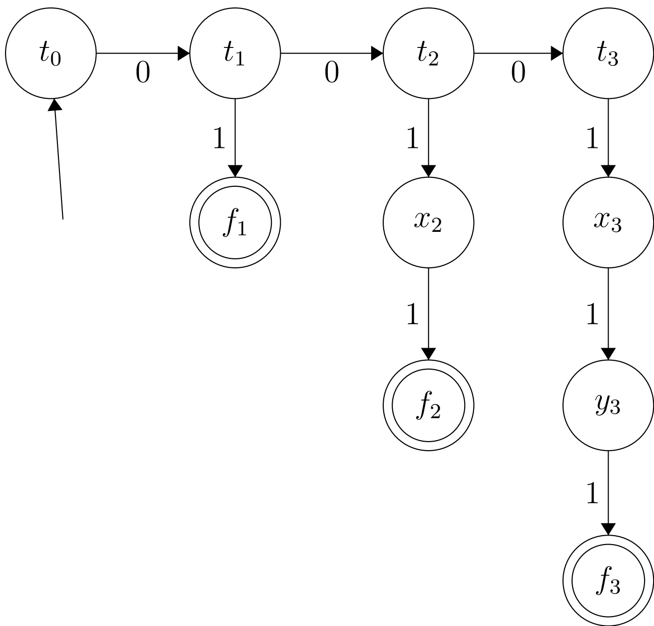
This language is the infinite sets of sequences of 0s and 1s, where all the 1s follow all the 0s and there are exactly as many 0s and 1s. In the next section, I discuss a proof of why this is not a regular

language, but before we turn to this proof, it is important to understand the intuition behind why there can't be an FSA that corresponds to this language. To do this let's just try and find a corresponding FSA and see the trouble such an attempt meets on the way. So, we are looking for finite state automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \{0^n 1^n \mid n \geq 1\}$.

Take an arbitrary string in the language, say 00001111. Say $\mathcal{A}$ accepts this string, and say that part of the acceptance is a computation $s_0 s_1 s_1 s_1 s_1$ corresponding to 0000. What would the rest of the computation look like? Well, after reading the four 0s, $\mathcal{A}$ will read the first 1 and it would then go to a new state s_2. For the language $\{0^n 1^n \mid n \geq 1\}$, it is crucial however that the automaton will now somehow "remember" that it has read four 0s. But there is no way it can remember this. As soon as the machine transitions from s_1 to s_2 all information about what it has encountered so far is gone. For instance, the following FSA is not the $\mathcal{A}$ we are looking for. It corresponds to $\{0^n 1^m \mid n \geq 1 \text{ and } m \geq 1\}$, which is a proper superset of $\{0^n 1^n \mid n \geq 1\}$. That is, it does not just include strings like 0011 and 00001111, but also strings like 001111 and 000011.



The only way an FSA can have some sort of memory is to have a different state for each number of 0s that has been read. For example:



This is an automaton that accepts 01, 0011, and 000111, but not strings like 011 or 001. Every time it is in a state where n 0s have been read, there will be n consecutive states that can be transitioned to by reading a 1. Only the last of these is an acceptance state. This trick works for $\{01, 0011, 000111\}$ and we could naturally extend the automaton to handle similar strings. Unfortunately, we cannot use it for $\{0^n 1^n \mid n \geq 1\}$. This is because if we want to extent this automaton to accept the countably infinite number of string in that language we will need an infinite number of states. For starters, we will need a countably infinite number of states like t_0, t_1, t_2, t_3, etc. Of course if we included such an infinite number of states, we would no longer have a *finite* state automaton.

The fact that not all languages are regular shows that the finite state automaton is a limited model of computation. It also shows that there is a distinguished class of decision problems, namely those corresponding to regular languages, for which this limited model suffices. That is, it can be important to know whether a certain problem is regular or not, because this will tell us the complexity of the

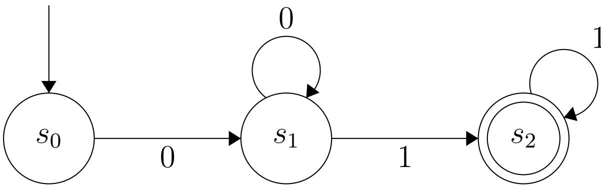
computational mechanism we need to use.

As we will see, there are multiple classes of formal languages, which form a hierarchy of increasing complexity. For now, however, we stick with the distinction between those languages that are and those that are not regular.

Proving that a language is regular is relatively easy. All you need to do is provide a finite state automaton for the language. For instance, say I want to prove that $\{0^n1^m \mid n \geq 1 \text{ and } m \geq 1\}$ is regular. I could present you with the (p-deterministic) automaton

$$\mathcal{R} = \langle\{0,1\},\{s_0,s_1,s_2\},s_0,\{s_2\},\{((s_0,0),s_1),((s_1,0),s_1),((s_1,1),s_2),((s_2,1),s_2)\}$$

which looks like this:



If I can now show that $\mathcal{L}(\mathcal{R}) = \{0^n1^m \mid n \geq 1 \text{ and } m \geq 1\}$, then this is proof that this language is regular. But this is easy to prove. All the computations that lead to accepted strings are of the form s_0 followed by at least one s_1 and at least one s_2. This is only possible if the string has at least one 0, followed by at least one 1.

We now turn to the formal proof of non-regularity.

3.4 The pumping lemma for regular languages

Above, we saw an important difference between languages like $\{01, 0011, 000111\}$ and $\{0^n1^n \mid n \geq 1\}$. The former is regular (witness the FSA we gave for it), while the latter is not. The upshot is that as long as we are not dealing with an infinite language, FSAs suffice.

Theorem 8

Any finite language is regular.

Proof

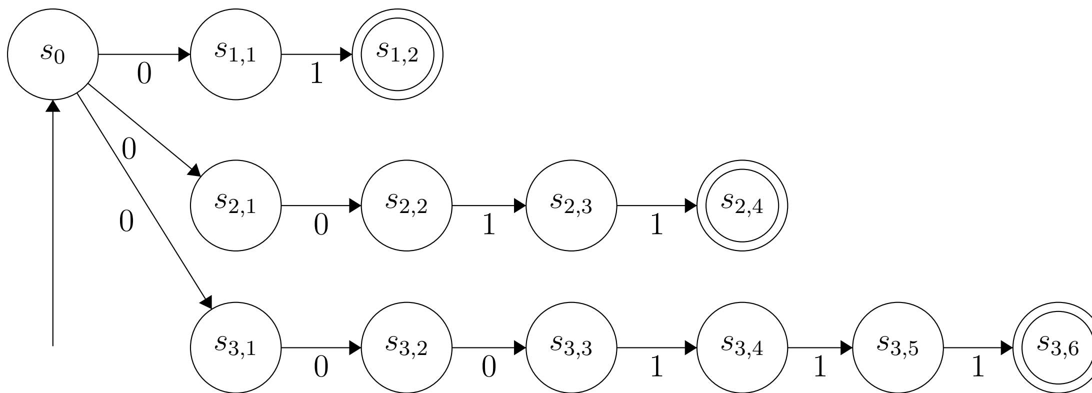
Let L be some finite set of strings over alphabet Σ. To prove that L is regular, we need to provide a finite state automaton for it. Let $L = \{x_1, \ldots x_n\}$. So, there are n strings in L. Let's say that each string $x_i \in L$ can be written as $x_{i,1} \ldots x_{i,k}$. For the whole language to be accepted, for every string x_i, we'll need a computation $s_0 \ldots s_k$, with s_0 the start state, s_k an accepting state and a transition $((s_{t-1}, x_{i,t-1}), s_t)$ for every $1 \leq t \leq k$. This means we can provide the FSA by just taking the set of all these transitions and the set of all corresponding states: $\mathcal{M} = \langle\Sigma, S, s_0, A, R\rangle$ such that:

- $S = \{s_0\} \cup \{s_{1,w} \mid 1 \leq w \leq |x_1|\} \cup \ldots \cup \{s_{n,w} \mid 1 \leq w \leq |x_n|\}$
- $A = \{s_{v,w} \mid x_v \in L \text{ and } w = |x_v|\}$
- $R = \{((s_{v,w-1}, x_{v,w-1}), s_{v,w}) \mid 1 \leq v \leq n \text{ and } 1 \leq w \leq |x_v|\}$

Since $\mathcal{L}(\mathcal{M}) = L$ it follows that L is regular. Since we took an arbitrary finite language, it follows that every finite language is regular.

For example, take the $\{01, 0011, 000111\}$ language. Here, the n in the proof is 3 and we have $x_1 = 01$, $x_2 = 0011$ and $x_3 = 000111$. (Any other order would do equally well.) These strings are 2, 4 and 6

symbols long. This means we need 2+4+6+1=13 states, one for each symbol plus a start state. If we follow the procedure in the proof, we end up with the following FSA.



Note that there are of course simpler finite state automata for this language. (I gave one earlier). But that is besides the point. By following this procedure we are guaranteed to provide an FSA for each finite language, thus proving that finite languages are regular. Providing a simpler FSA would only reiterate that point.

The above proof exploits the fact that finite state automata can include any finite number of states. Given that an FSA has a *finite* number of states, this means that FSAs for infinite languages can accept strings that are longer than the number of states in the FSA. This in turn means that the computation of some accepted string in an infinite language must involve the same state more than once. (This is sometimes referred to as the *pigeon whole principle*. If you have m pigeons in $n < m$ pigeon wholes, then some pigeon wholes will have to contain multiple pigeons.) In other words, FSAs for infinite languages must contain a *loop*.

It turns out that loops in finite state automata have a very particular feature. Say that we have some FSA and there is a computation of string x which starts in the start state and ends in some state s_i. Let's say that when the FSA subsequently reads string y, it enters a loop. That is, the first symbol of y triggers a transition to s_j and the sequence of transitions brought about by the rest of the symbols in y ends once more in s_j. Finally, let's assume that from there the FSA can read string z, transitioning from s_j to s_k and subsequently to some states ending in some accepting state s_f. So, the FSA accepts the string xyz and the computation that goes with this string is $s_0 \dots s_i s_j \dots s_j s_k \dots s_f$. Because this acceptance has a loop along its path of computation, it follows that there will be similar accepting computations when the loop is traversed less than one or more than one times. In other words, this FSA will also accept xz, $xyyz$, $xyyyz$, etc. In fact the set $\{xy^n z \mid n \geq 0\}$ will be a subset of the language corresponding to this automaton. This is the guiding intuition behind the *pumping lemma*.

The pumping lemma for regular language is an incredibly useful lemma, for it provides us with a method of proof for **non-regularity**. This may seem unintuitive, but notice that the pumping lemma give us a property that all regular languages have. This is not to say that all *non*-regular languages *lack* this property. In other words, if you find a language with the pumping property described by the lemma, this does not mean that this language is regular. However, if you find a language that does not have the property, then you know it can't be regular, since all regular languages have the property.

The pumping lemma for regular languages

If L is a regular language, then there is a number $p \in \mathbb{N}$, the so-called *pumping length*, such that every string $\sigma \in L$ such that $|\sigma| \geq p$ can be devided into $\sigma = xyz$ where:

- for each $n \geq 0$: $xy^n z \in L$
- $|y| > 0$
- $|xy| \leq p$

Proof

We know that L is regular, so there exists an automaton $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = L$. Let us assume that $\mathcal{M}$ has p states. Now take $\sigma \in L$ such that $\sigma = x_1 \ldots x_n$ with $n \geq p$. Since σ is accepted, we have a computation $c_1 c_2 \ldots c_p c_{p+1} \ldots c_n c_{n+1}$ such that $((c_i, x_i), c_{i+1})$ is a transition in $\mathcal{M}$, c_1 is the start state and c_{n+1} is an acceptance state. Since $n \geq p$, there are more states in this computation than there are states in $\mathcal{M}$. This means that there has to be some v, w such that $v \neq w$ and $c_v = c_w$. Let's call this state s. This means that the computation will look like $c_1 c_2 \ldots c_{v-1} s \ldots s c_{w+1} \ldots c_n c_{n+1}$. Let's call the string computed by $c_1 c_2 \ldots c_{v-1} s$, x, the string computed by $s c_{w+1} \ldots c_n c_{n+1}$, z and the string computed by $s \ldots s$, y. By assumption, $xyz \in L$. Because $s \ldots s$ is a loop, it follows that $xy^n z \in L$ for any $n \geq 0$. Given that $v \neq w$ in the original computation of σ, it follows that $|y| > 0$. Because there are only p unique states, $v < p$ and $w < p + 1$. It follows from this that $|xy| \leq p$.

A mnenomic for applying the pumping lemma

At the risk of being pedantic, the following dumbs the structure of how to apply the pumping lemma down to something extremely simple. Consider the following theorem.

The tail lemma

Every cow has a tail

Let's say I have the task of finding cows. My problem is that I am very bad at identifying animals. I encounter three: a cat, a cow and a frog. The tail lemma won't help me with the cat or a cow. All I know is that every cow has a tail. Both these animals could therefore be a cow, but they could also both be other animals that have a tail (as the cat indeed is). The lemma does help me with the frog. The frog does not have a tail and therefore cannot be a cow.

Here is an example of the pumping lemma in action. Consider the standard example of a non-regular language $L = \{0^n 1^n \mid n \geq 1\}$. Assume that L is regular. Then the pumping lemma should hold for L. So, there should be some p such that any string that is at least p can be "pumped" in the way described by the lemma. Take an arbitrary string that is long enough: $0^p 1^p$ (length $2 \times p$). We should now be able to divide this string into x, y and z such that $|xy| \leq p$, $|y| > 0$. Since $|xy| \leq p$, it follows that both x and y only contain 0s. In particular $y = 0^k$ for $1 \leq k \leq p$. Given the pumping lemma, it now follows that $xy^l z \in L$. This runs into a contradiction. Take for instance, $xy^2 z$. We know that xyz contains p 0s and p 1s. We also know that y contains k 0s and $k > 0$. It follows that $xy^2 z$ contains $p + k$ 0s and p 1s. Since $p + k > p$, it follows that $xy^2 z \notin L$. This contradicts the assumption that L was a language for which the pumping lemma holds. It follows that L is not regular.

4 Formal grammars

The models of computation we looked at so far were *automata*, abstract machines that involve state transitions on the basis of an input that is being read. In this section, we move to another model of computation, namely a *formal grammar*. Later we will see how grammars are related to automata. In particular, we will discuss different kinds of formal grammar and compare them to various kinds of automata beyond the class of finite state automata that we discussed so far.

4.1 Formal definition

Definition 15

A **formal grammar** is a 4-tuple $\langle \Sigma, N, S, P \rangle$ such that:

- Σ is a finite set of terminal symbols, i.e. the alphabet of the language corresponding to the grammar
- N is a finite set of non-terminal symbols
- $S \in N$ is the so-called start symbol
- $P \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ is a finite set of productions (or *production rules*)

It is common to use capital letters for non-terminals and lower-case letters for terminals. Here is an example of a formal grammar:

$$\mathcal{X} = \langle \{1\}, \{S\}, S, \{(S, 1), (S, 1S)\} \rangle$$

This grammar has one non-terminal symbol S and two productions. We often write production as rules, using $\rightarrow$. For instance, the productions of the above grammar look like:

$$\begin{array}{lcl} S & \rightarrow & 1 \\ S & \rightarrow & 1S \end{array}$$

When linguists are not concerned with matters of formal language theory, they often represent a formal grammar by just giving the production rules. For instance, you may encounter a "grammar" like the following:

$$\mathcal{Y} = \left\{ \begin{array}{r} \text{NP} \rightarrow \text{N (PP)} \\ \text{DP} \rightarrow \text{D NP} \\ \text{PP} \rightarrow \text{P DP} \\ \text{N} \rightarrow \text{box | table | diamond | room | house | sister | thief} \\ \text{P} \rightarrow \text{in | on | of} \\ \text{D} \rightarrow \text{the | a} \end{array} \right.$$

Note first of all that from these production rules alone, you will be unable to identify what grammar is meant. This is because there is no indication of what the start node is. The language corresponding to the intended grammar will be quite different when "PP" is the start node compared to when "DP" is the start symbol. In that sense, an informal grammatical representation like this corresponds to a set of grammars and a set of languages, one for each choice of the start symbol (DP, NP, PP). For instance, this grammar derives DPs like the following:

(6) a. a diamond

b. the table of the thief
c. a diamond in the box on a table in a room in the house of the sister of the thief of the diamond in the box

and PPs like:

(7) a. in a diamond
b. of the table of the thief
c. of a diamond in the box on a table in a room in the house of the sister of the thief of the diamond in the box

The production rules above illustrate some common notational conventions that are handy to know. First of all, one rule indicates the optionality of certain symbols by putting them in brackets. That is,

NP → N (PP)

is short for two rules, namely:

NP → N
NP → N PP

A different form of disjunction for production rules is the pipe symbol "|". This indicates alternative right-hand sides for the same left-hand side. So:

D → the | a

is short for:

D → the
D → a

4.2 Derivation

Derivation is the formal notion that concerns how a certain string is recognized/produced by a grammar. This notion is comparable to the notion of computation that I introduced for finite state automata. It is defined as follows:

Definition 16

Let $\mathcal{G} = \langle \Sigma, N, S, P \rangle$ and let α, β, γ, $\delta \in (N \cup \Sigma)^*$. If $(\alpha, \beta) \in P$ then we say that $\delta\alpha\gamma$ **directly derives** $\delta\beta\gamma$, which we write as: $\delta\alpha\gamma \Rightarrow_{\mathcal{G}} \delta\beta\gamma$. We say that α **derives** β, which we write $\alpha \Rightarrow^* \beta$, whenever:

- $\alpha \Rightarrow_{\mathcal{G}} \beta$ (α directly derives β), or
- there exists a non-empty sequence $x_1 \ldots x_n$ such that $\alpha \Rightarrow_{\mathcal{G}} x_1 \Rightarrow_{\mathcal{G}} x_n \Rightarrow_{\mathcal{G}} \ldots \Rightarrow_{\mathcal{G}} x_n \Rightarrow_{\mathcal{G}} \beta$

I will drop the subscript indicating the grammar if it is clear for which grammar derivation is being discussed.

For example, the grammar $\mathcal{X} = \langle \{1\}, \{S\}, S, \{(S, 1), (S, 1S)\} \rangle$ we gave above yields $S \Rightarrow^* 1$, since $S \Rightarrow 1$, since $(S, 1)$ is in the productions of this grammar. (That is, here we can take $\delta = \gamma = \epsilon$.) We also have $S \Rightarrow^* 111$, since $S \Rightarrow 1S \Rightarrow 11S \Rightarrow 111$. There are three direct derivation in this indirect derivation. First, $S \Rightarrow 1S$ because $S \rightarrow 1S$ is a production rule. (I.e. $\delta = \gamma = \epsilon$). Second, $1S \Rightarrow 11S$ because $(S, 1S)$ is a production rule. (Here, $\alpha = S$, $\beta = 1S$, $\delta = 1$, $\gamma = \epsilon$.) Finally, $11S \Rightarrow 111$ because $(S, 1)$ is a production rule. (So we take $\alpha = S$, $\beta = 1$, $\delta = 11$ and γ is empty.)

Another example: if we assume that the startsymbol of grammar $\mathcal{Y}$ is DP, then it has a derivation for the string "the box on the table", which goes as follows.

DP ⇒ D NP ⇒ the NP ⇒ the N PP ⇒ the box PP ⇒ the box P DP ⇒ the box on DP ⇒ the box on D NP ⇒ the box on the NP ⇒ the box on the N ⇒ the box on the table

Note that in this derivation, I consistently replace the left-most non-terminal symbol in accordance to some production rule. Such a derivation is called *the left-most derivation.* The *right-most derivation* consistently replaces the right-most non-terminal and looks like this:

DP ⇒ D NP ⇒ D N PP ⇒ D N P DP ⇒ D N P D NP ⇒ D N P D N ⇒ D N P D table ⇒ D N P the table ⇒ D N on the table ⇒ D box on the table ⇒ the box on the table

There are also derivations that are neither left-most nor right-most. For example:

DP ⇒ D NP ⇒ D N PP ⇒ the N PP ⇒ the N P DP ⇒ the box P DP ⇒ the box P D NP ⇒ the box P D N ⇒ the box on D N ⇒ the box on the N ⇒ the box on the table

It is important to note that for all these derivations, there is a sense in which they came about as by magic. We just happened to consistently pick the right production rule to get to the string we were after. For instance, say that we attempt a right-most derivation starting, as before, with the step DP ⇒ D NP. Given that our focus is on the right-most non-terminal, we need to find a production rule for NP. Let's say we pick NP⇒ N. Then our derivation becomes DP ⇒ D NP ⇒ D N. Now, we have no way to continue the derivation to derive the string "the box on the table", since both D and N are only rewritable as terminal symbols.

What matters for now is simply whether or not there exists a derivation for a certain string. If one exists, then the string is part of the language corresponding to the grammar. Actually *finding* such a derivation (that is, *proving* that the string is in the language) is a different and practical matter to which we turn later.

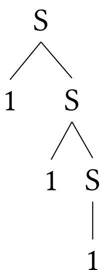
Definition 17

The language corresponding to a grammar $\mathcal{G} = \langle \Sigma, N, S, P \rangle$, written $\mathcal{L}(\mathcal{G})$ is defined as follows:

$$\mathcal{L}(\mathcal{G}) \;=\; \{\sigma \mid S \Rightarrow^*_{\mathcal{G}} \sigma\}$$

4.3 Parse trees and ambiguity

To visualise a derivation from a grammar, it is sometimes handy to draw a corresponding tree structure, where the leaves of the tree (read from left to right) make up the righthand side of the derivation and the root of the tree the lefthand side. For instance, the following tree represents a derivation of $S \Rightarrow 111$.



These structures are called *parse trees.* They are not just the graphic representation of the derivation of a string. They are formal object in their own right:

Definition 18

A **parse tree** is a pair (P, C), where

- P is a symbol, the *parent node*, and
- C is an ordered sequence of symbols and parse trees, the *children*

Nodes that are not trees are called *leaves.*

The following procedure produces parse trees from derivations. Whenever we have a direct derivation $N \Rightarrow x_1 \ldots x_n$, we build the parse tree $(N, (x_1, \ldots, x_n))$. The next step in the derivation could now rewrite x_i. For instance $x_1 \ldots x_i \ldots x_n \Rightarrow x_1 \ldots y_k \ldots y_k \ldots x_n$. We adjust the parse tree accordingly by replacing x_i by the tree $(x_i, (y_1, \ldots, y_k))$, yielding the tree $(N, (x_1, \ldots, (x_i, (y_1, \ldots, y_k)), \ldots, x_n))$. Once we have done this for every part of the derivation we have the parse tree corresponding to that derivation.

For instance, the tree just above definition 18 corresponds to the derivation $S \Rightarrow 1S \rightarrow 11S \rightarrow 111$ for grammar $\mathcal{X} = \langle\{1\}, \{S\}, S, \{(S, 1), (S, 1S)\}\rangle$. I illustrate this using the following table, where the left-hand side shows the unfolding of the tree, step by step in the derivation and the right-hand side the production rule that was used in the corresponding derivation step.

$(S, (1, S))$	$S \rightarrow 1S$
$(S, (1, (S, (1, S))))$	$S \rightarrow 1S$
$(S, (1, (S, (1, (S, 1)))))$	$S \rightarrow 1$

Note that a string will have multiple derivations in a grammar. For starters, for each left-most derivation there will be a right-most derivation. However, this does not have any impact on the parse tree. I illustrate this with $\mathcal{Y}$, repeated here, and the DP "the box on the table":

$$\mathcal{Y} = \begin{cases} \text{NP} \rightarrow \text{N (PP)} \\ \text{DP} \rightarrow \text{D NP} \\ \text{PP} \rightarrow \text{P DP} \\ \text{N} \rightarrow \text{box} \mid \text{table} \mid \text{diamond} \mid \text{room} \mid \text{house} \mid \text{sister} \mid \text{thief} \\ \text{P} \rightarrow \text{in} \mid \text{on} \mid \text{of} \\ \text{D} \rightarrow \text{the} \mid \text{a} \end{cases}$$

left-most derivation of "the box on the table"	
$(DP, (D, NP))$	$DP \rightarrow D\ NP$
$(DP, ((D, \text{the}), NP))$	$D \rightarrow \text{the}$
$(DP, ((D, \text{the}), (NP, (N, PP))))$	$NP \rightarrow N\ PP$
$(DP, ((D, \text{the}), (NP, ((N, \text{box}), PP))))$	$N \rightarrow \text{box}$
$(DP, ((D, \text{the}), (NP, ((N, \text{box}), (PP, (PDP))))))$	$PP \rightarrow P\ DP$
$(DP, ((D, \text{the}), (NP, ((N, \text{box}), (PP, ((P, \text{on}), DP))))))$	$P \rightarrow \text{on}$
$(DP, ((D, \text{the}), (NP, ((N, \text{box}), (PP, ((P, \text{on}), (DP, (DNP))))))))$	$DP \rightarrow D\ NP$
$(DP, ((D, \text{the}), (NP, ((N, \text{box}), (PP, ((P, \text{on}), (DP, ((D, \text{the}), NP))))))))$	$D \rightarrow \text{the}$
$(DP, ((D, \text{the}), (NP, ((N, \text{box}), (PP, ((P, \text{on}), (DP, ((D, \text{the}), (NP, N)))))))))$	$NP \rightarrow N$
$(DP, ((D, \text{the}), (NP, ((N, \text{box}), (PP, ((P, \text{on}), (DP, ((D, \text{the}), (NP, (N, \text{table}))))))))))$	$N \rightarrow \text{table}$

The right-most derivation of that same string looks as follows, and results in the same tree:

right-most derivation of "the box on the table"	
$(DP, (D, NP))$	$DP \to D\ NP$
$(DP, (D, (NP, (N, PP))))$	$NP \to N\ PP$
$(DP, (D, (NP, (N, (PP, (P, DP))))))$	$PP \to P\ DP$
$(DP, (D, (NP, (N, (PP, (P, (DP, (D, NP))))))))$	$DP \to D\ NP$
$(DP, (D, (NP, (N, (PP, (P, (DP, (D, (NP, N)))))))))$	$NP \to N$
$(DP, (D, (NP, (N, (PP, (P, (DP, (D, (NP, (N, \text{table}))))))))))$	$N \to$ table
$(DP, (D, (NP, (N, (PP, (P, (DP, ((D, \text{the}), (NP, (N, \text{table}))))))))))$	$D \to$ the
$(DP, (D, (NP, (N, (PP, ((P, \text{on}), (DP, ((D, \text{the}), (NP, (N, \text{table}))))))))))$	$P \to$ on
$(DP, (D, (NP, ((N, \text{box}), (PP, ((P, \text{on}), (DP, ((D, \text{the}), (NP, (N, \text{table}))))))))))$	$N \to$ box
$(DP, ((D, \text{the}), (NP, ((N, \text{box}), (PP, ((P, \text{on}), (DP, ((D, \text{the}), (NP, (N, \text{table}))))))))))$	$D \to$ the

These tables illustrate that left- or right-most derivation is immaterial to the resulting parse tree. We can capture this intuition in the notion of *derivation-similarity*.

Definition 19

The **trace** of a derivation is the sequence of production rules (elements in used by the derivation. (This is the right column in the tables above).

Two derivations are **similar** if their derivation traces are of equal length and contain the same productions. (That is, they are simply a reordering of one-another).

The left- and right-most derivation of "the box on the table" above are *derivation-similar*. They involve the same steps, yet in a different order. Try and convince yourself that each left-most derivation will have a derivation-similar right-most derivation. Note as well that whenever two derivations are similar, they will have the same parse tree.

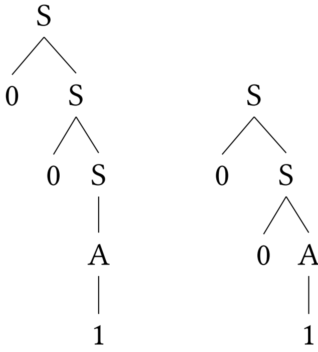
Given all this, we can turn things around. A parse tree is a representation of a class of derivations that are similar. Technically, a parse tree corresponds to an *equivalence class* of derivations. This is because derivation-similarity is an equivalence relation (it is transitive, reflexive and symmetric). As such, given a grammar and given some string we can construct the set of derivations of the string in that grammar that are similar to one-another. This is an equivalence class. For that class of derivations, there is exactly one parse tree.

Importantly, two derivations of the same string (and given the same grammar) are not always similar. Sometimes, the same string may have multiple parse trees. In that case, the grammar is called *ambiguous*.

Here is an example of a grammar that is ambiguous:

$$\langle \{0, 1\}, \{S, A\}, S, \{(S, 0), (S, 0S), (S, A), (S, 0A), (A, 1)\} \rangle$$

This grammar derives strings like 0, 00, 001, 00001, etc. Ambiguity arises when the string contains a 1. For instance, 001 can be derived in two ways: $S \Rightarrow 0S \Rightarrow 00S \Rightarrow 00A \Rightarrow 001$ or $S \Rightarrow 0S \Rightarrow 00A \Rightarrow 001$. The corresponding parse trees are:



These trees are constructed from the derivations as illustrated in the tables below. (As before, the tables show each step in the derivation. On the left of the table the tree is unfolding step by step; on the right I indicate the production that was used at the corresponding step in the derivation.)

$S \Rightarrow 0S \Rightarrow 00S \Rightarrow 00A \Rightarrow 001$	
$(S, (0, S))$	$S \rightarrow 0S$
$(S, (0, (S, (0, S))))$	$S \rightarrow 0S$
$(S, (0, (S, (0, (S, A)))))$	$S \rightarrow A$
$(S, (0, (S, (0, (S, (A, 1))))))$	$A \rightarrow 1$

$S \Rightarrow 0S \Rightarrow 00A \Rightarrow 001$	
$(S, (0, S))$	$S \rightarrow 0S$
$(S, (0, (S, (0, A))))$	$S \rightarrow 0A$
$(S, (0, (S, (0, (A, 1)))))$	$A \rightarrow 1$

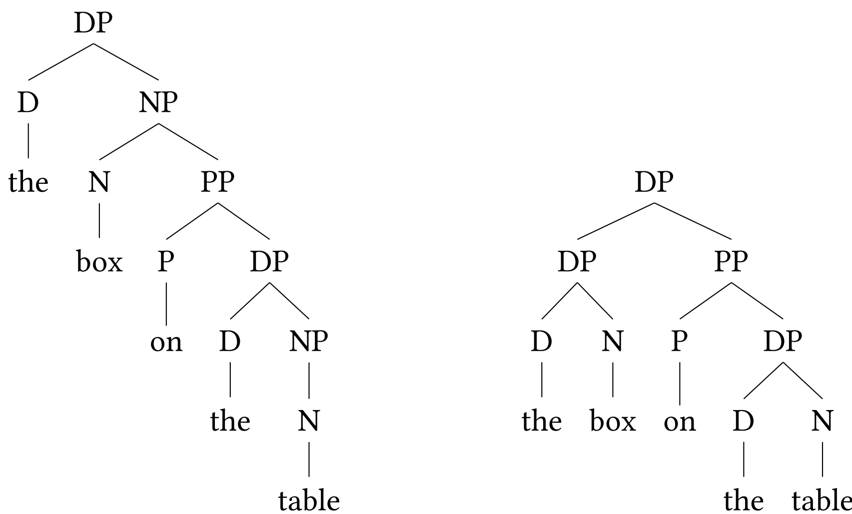
From the formal perspective we developed here, ambiguity is the notion that a grammar has two distinct parse trees for the same string. Given the link between parse trees and derivation similarity (a parse tree corresponds to the equivalence class of derivation-similar derivations), we can also see ambiguity as the phenomenon within a grammar where the same string has two distinct left-most derivations. Or, similarly, a grammar is ambiguous whenever there's a string that has two distinct right-most derivations.

4.4 Grammar equivalence

Consider the following two grammars. The left one is a repeat of the grammar we saw above. The grammar on the right is only slightly different.

NP → N (PP)	DP → DP PP
DP → D NP	DP → D N
PP → P DP	PP → P DP
N → box | table | diamond | room	N → box | table | diamond | room
| house | sister | thief	| house | sister | thief
P → in | on | of	P → in | on | of
D → the | a	D → the | a

Let us assume that the start symbol for both grammars is DP. Both grammars then derive the same language. This language contains strings like "a diamond", "the room of the sister", "the sister of the sister of the thief", etc. Note, however, that the grammars differ in how these strings are derived. In particular, the grammars differ in the parse trees that go with strings. Take, "the box on the table". Here are the two parse trees corresponding to the derivations of this string in the two grammars.



This shows that even though the two grammars are equivalent in the sense that they correspond to the same language, they differ in the *structures* they assign to these strings.

Definition 20

Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two formal grammars. $\mathcal{G}_1$ and $\mathcal{G}_2$ are **weakly equivalent** whenever $\mathcal{L}(\mathcal{G}_1) = \mathcal{L}(\mathcal{G}_2)$. $\mathcal{G}_1$ and $\mathcal{G}_2$ are **strongly equivalent** when they are weakly equivalent and they generate the same parse trees, given some renaming (if needed) of non-terminal symbols.

The above two grammars are *weakly* but not *strongly* equivalent.

4.5 Regular grammars

It should be intuitively clear that there are connections between formal grammars and finite state automata. Most importantly, they both can compute infinite languages using finite means. A formal grammar is just an abstract model of computation, just like an automaton is. So, for instance, the grammar $\langle\{1\}, \{S\}, S, \{(S, 1), (S, 1S)\}\rangle$ that we started this chapter with, is equivalent to the finite state automaton $\mathcal{Z}$, below, in the sense that they correspond to the same formal language, namely $\{1^n \mid n > 0\}$.

$$\mathcal{Z} = \langle\{1\}, \{s_0, s_1\}, s_0, \{s_1\}, \{((s_0, 1), s_1), ((s_1, 1), s_1)\}\rangle$$

As we will see below, formal grammars are much more expressive than finite state automata. However, there is a certain class of formal grammars that is equivalent to FSAs in the sense that the languages that these grammars can compute are exactly the regular languages. This is the class of *regular grammars*.

Definition 21

Let $\mathcal{G} = \langle\Sigma, N, S, P\rangle$ be a formal grammar. $\mathcal{G}$ is a *left linear grammar* whenever for every $(\alpha, \beta) \in P$ it is the case that $\alpha \in N$ and either $\beta \in \Sigma^*$ or $\beta = xX$ with $x \in \Sigma^*$ and $X \in N$. $\mathcal{G}$ is a *right linear grammar* whenever for every $(\alpha, \beta) \in P$ it is the case that $\alpha \in N$ and either $\beta \in \Sigma^*$ or $\beta = Xx$ with $x \in \Sigma^*$ and $X \in N$. A grammar is *regular* when it is either left or right linear.

This says that left linear grammars are grammars where all productions rules take one of two forms: either a non-terminal mapping to a terminal symbol, or a non-terminal mapping to a terminal symbol followed by a non-terminal one. Right linear grammars are similar, but with the order of terminal and non-terminal swapped in the second type of rule. The simple grammar we saw above,

$\langle\{1\}, \{S\}, S, \{(S,1), (S,1S)\}\rangle$, is left-linear. There is a right linear grammar that derives exactly the same language, namely:

$$\langle\{1\}, \{S\}, S\{(S,1), (S,S1)\}\rangle$$

These grammars are weakly equivalent to each other. In fact, for each left-linear grammar there is a weakly equivalent right-linear grammar, and vice versa. Given this, the class of languages that can be generated from left-linear grammars is the same as the class of languages that can be generated from right-linear grammar. These are **the regular languages**. Importantly, these are exactly the languages that finite state automata can recognize.

Theorem 9

If $\mathcal{M}$ is a finite state automaton, then there exists a regular grammar $\mathcal{G}$ such that $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{M})$. If $\mathcal{G}$ is a regular grammar, then there exists a finite state automaton $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{G})$. As a result, the set of languages computable with regular grammars is the set of regular languages.

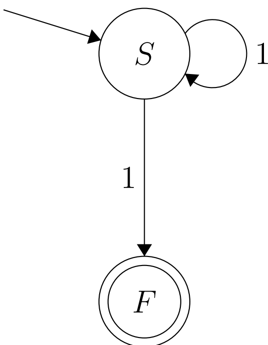
I will sketch the proof by going through a procedure to turn a regular grammar into a finite state automaton and vice versa. Say we have (left-linear) regular grammar $\mathcal{G} = \langle\Sigma, N, S, P\rangle$. The corresponding FSA is $\mathcal{M} = \langle\Sigma, N \cup \{F\}, S, \{F\}, R\rangle$, with R:

$$R = \{((X,x),Y) \mid (X,xY) \in P\} \quad \cup \quad \{((X,x),F) \mid (X,x) \in P, X \in N, x \in \Sigma\}$$

For example, applying this procedure to $\langle\{1\}, \{S\}, S, \{(S,1), (S,1S)\}\rangle$, we get:

$$\langle\{1\}, \{S,F\}, S, \{F\}, \{((S,1),F), ((S,1),S)\}\rangle$$

More graphically:



To "translate" an FSA into a corresponding regular grammar, we can do the following. Say, $\mathcal{M}$ is an FSA of the form $\langle\Sigma, S, s, A, R\rangle$. The corresponding left-linear regular grammar is $\mathcal{G} = \langle\Sigma, S, s, P\rangle$ with P:

$$P = \{(X,\epsilon) \mid X \in A\} \quad \cup \quad \{(X,xY) \mid ((X,x),Y) \in R\}$$

If we apply this to the automaton we gave for the language $\{1^n | n > 0\}$, we get

$$\langle\{1\}, \{S,F\}, S, \{(F,\epsilon), (S,1F), (S,1S)\} \quad \text{or graphically:} \quad \begin{array}{lcl} S & \rightarrow & 1S \\ S & \rightarrow & 1F \\ F & \rightarrow & \epsilon \end{array}$$

Although this grammar looks different from the original grammar we gave for this language, it is easy to verify that it corresponds to exactly the same language.

The real proof that regular grammars correspond to regular languages involves showing that these two procedures (translating regular grammars into FSAs and vice versa) work across the board.

5 Context-free languages

In chapter 3 we saw an example of a non-regular language: $\{0^n1^n \mid n \in \mathbb{N}\}$. Being non-regular, this is a language for which no finite state automaton and no regular grammar exists. So what mechanisms do we have to compute languages like these?

5.1 Push-down automata

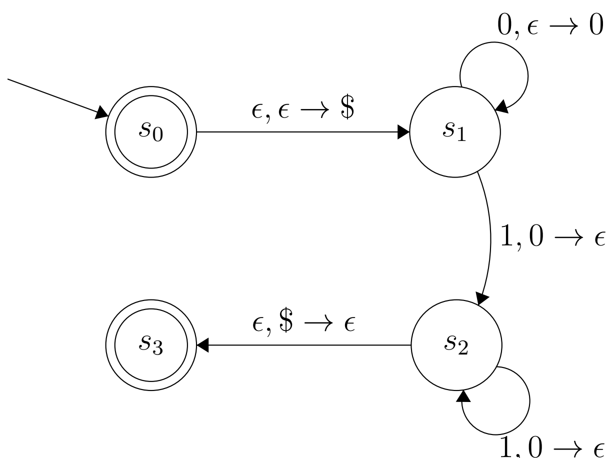
As we observed earlier, in order to deal with a language like $\{0^n1^n \mid n \in \mathbb{N}\}$ we need a computational mechanism that has some capacity of remembering things. For instance, the number of 0s that an automaton has encountered in the input. This is why finite state automata are unsuitable - they simply have no notion of memory whatsoever. For these kinds of languages, we therefore turn to a different kind of automaton called a *push-down automaton* or PDA.

A PDA is like an FSA, except that it comes with a so-called *stack*. A stack is a special kind of string that can only be accessed at one end. You can compare it to a pile of books viewed from the top, so you can only see the top book. Also, if you want to add a book, you can only add it at the top and if you want to take a book from it, you can only remove the top-most book. This is called the *last in first out* principle.

A stack has two operations defined on it. First of all, "pop" takes the last symbol in the stack (if there is any) and removes it. Second, "push" takes the stack and a symbol and concatenates them, so that the pushed symbol is now the top of the stack.

A push down automaton uses a stack to keep track of what happens when it reads an input string. The only difference between a finite state automaton and a push down automaton is that apart from reading a symbol from the string, state transitions additionally potentially involve popping and pushing a string to and from the stack. As such, the stack is the external memory of the automaton.

Here's an example. The following (graphical depiction of a) PDA corresponds to the language $\{0^n1^n \mid n \in \mathbb{N}\}$.



This automaton works as follows. Each transition is labeled with three things, notated: $a, b \to c$. Here, a is what is read from the input, b is what is popped from the stack and c is what is pushed to the stack. So, at the start this automaton reads nothing and pops nothing. It simply pushes "\$" to the stack and transitions to s_1. The "\$" symbol is simply to have a marker in the stack of where the reading of input started. Once in s_1, if it reads a 0, then it pops nothing, and pushes a 0 to the stack. This way, for each 0 that is read, there is a 0 on the stack. Once a 1 is read, it pops the top 0 from the stack and pushes nothing, transitioning to s_2. There, it keeps on popping 0s for each 1 that is read. The automaton transitions to the acceptance state s_3 if there is nothing left to read and "\$" can be

popped from the stack. That is, the automaton only accepts if after reading all the 1s, there are no more 0s on the stack. As you can verify yourself, this is only possible if the number of 0s and 1s is exactly the same.

The computation of a string can be captured in a table that keeps track of the current state and the current stack for each symbol read from the input. For instance, for the input 000111:

read	state	stack
	s_0	ϵ
ϵ	s_1	\$
0	s_1	\$0
0	s_1	\$00
0	s_1	\$000
1	s_2	\$00
1	s_2	\$0
1	s_2	\$
ϵ	s_3	ϵ

Formally, a pushdown automaton looks a lot like a finite state automaton, except with the addition of the stack and the added complexity of the transitions:

Definition 22

A finite state automaton is a 6-tuple $\langle \Sigma, \Gamma, S, s, A, R \rangle$, such that:

- Σ is a finite set (the language alphabet)
- Γ is a finite set (the stack alphabet)
- S is a finite set of states
- $s \in S$, the start state
- $A \subseteq S$, the acceptance states
- $R \subseteq (S \times \Sigma^* \times \Gamma^*) \times (S \times \Gamma^*)$, the transition relation

Let's unpack the transition relation. It is a relation between triples consisting of a state, a string of symbols of the alphabet of the language and a string of symbols of the stack alphabet on the one hand and pairs consisting of a state and a string of symbols from the stack language on the other. So, given a state, a symbol (or string) that is being read from the input string and a symbol (or string) to be popped from the top of the stack, there is a transition to a new state and a string of stack symbols is pushed to the stack.

The PDA I gave for $\{0^n 1^n \mid n \in \mathbb{N}\}$ is given as follows:

$$\mathcal{A} = \langle \{0, 1\}, \{\$, 0, 1\}, \{s_0, s_1, s_2, s_3\}, s_0, \{s_0, s_3\}, R \rangle$$

where R is represented as:

$$\begin{array}{rcl} (s_0, \epsilon, \epsilon) & \rightarrow & (s_1, \$) \\ (s_1, 0, \epsilon) & \rightarrow & (s_1, 0) \\ (s_1, 1, 0) & \rightarrow & (s_2, \epsilon) \\ (s_2, 1, 0) & \rightarrow & (s_2, \epsilon) \\ (s_2, \epsilon, \$) & \rightarrow & (s_3, \epsilon) \end{array}$$

which is a more convenient representation for the set $\{((s_0, (\epsilon, \epsilon)), (s_1, \$)), ((s_1, (0, \epsilon)), (s_1, 0)), ((s_1, (1, 0)), (s_2, \epsilon)), ((s_2, (1, 0)), (s_2, \epsilon)), ((s_2, (\epsilon, \$)), (s_3, \epsilon))\}$.

Definition 23

Let $\mathcal{P} = \langle \Sigma, \Gamma, S, s, A, R \rangle$ be a push-down automaton. Let $x \in \Sigma$ and $\sigma \in \Sigma^*$. (So, $x\sigma$ is also a member of Σ^*.) Let $t, t' \in S$ and $u, v, w \in \Gamma^*$. A **computation step** for $\mathcal{P}$ is defined as the relation $\vdash_{\mathcal{P}}$, which is defined as:

$(x\sigma, t, vu) \vdash_{\mathcal{P}} (\sigma, t', wu)$ if and only if $((t, (x, v)), (t', w)) \in R$.

A **computation** for $\mathcal{P}$ is a sequence of computation steps.

The relation $\vdash^*_{\mathcal{P}}$, the "computes" relation for push down automaton $\mathcal{P}$ is the reflexive and transitive closure of $\vdash_{\mathcal{P}}$. That is, it is the smallest relation such that $\vdash_{\mathcal{P}} \subseteq \vdash^*_{\mathcal{P}}$ and such that it is reflexive and transitive.

The triples in these computation steps represent situations the push-down automaton can be in. For instance $(s_1, 0111, 100)$ is a situation of an automaton in state s_1, which still needs to read 0111 (so 0 is the next symbol it reads), where the stack has 100 in it (so 1 is the symbol that could potentially be popped).

Here is an example of an application of these definitions for the PDA $\mathcal{A}$ that I gave for $\{0^n 1^n \mid n \in \mathbb{N}\}$.

$$\begin{aligned}&(s_0, 000111, \epsilon) \vdash_{\mathcal{A}} (s_1, 000111, \$) \vdash_{\mathcal{A}} (s_1, 00111, \$0) \vdash_{\mathcal{A}} (s_1, 0111, \$00)\\ &\vdash_{\mathcal{A}} (s_1, 111, \$000) \vdash_{\mathcal{A}} (s_2, 11, \$00) \vdash_{\mathcal{A}} (s_2, 1, \$0) \vdash_{\mathcal{A}} (s_2, \epsilon, \$) \vdash_{\mathcal{A}} (s_3, \epsilon, \epsilon)\end{aligned}$$

As a consequence:

$$(s_0, 000111, \epsilon) \vdash^*_{\mathcal{A}} (s_3, \epsilon, \epsilon)$$

Given the definition of the 'computes' relation, we can now define acceptance for push-down automata:

Definition 24

Let $\mathcal{P} = \langle \Sigma, \Gamma, S, s, A, r \rangle$ be a push-down automaton. $\mathcal{P}$ accepts $\sigma \in \Sigma^*$ if and only if for some $f \in A$:

$$(s, \sigma, \epsilon) \vdash^*_{\mathcal{P}} (f, \epsilon, \epsilon)$$

That is, a string is accepted if we can find a computation that starts in the start state with an empty stack and ends in an acceptance state with an empty stack (and the whole string read).

As we did with acceptance for finite state automata, the language corresponding to the automaton is simply the set of accepted strings.

Definition 25

If $\mathcal{P} = \langle \Sigma, \Gamma, S, s, A, r \rangle$ is a push-down automaton, then:

$$\mathcal{L}(\mathcal{P}) \;=\; \{\sigma \mid \text{there exists a } f \in A : (s, \sigma, \epsilon) \vdash^*_{\mathcal{P}} (f, \epsilon, \epsilon)\}$$

5.2 Context-free grammar

Before, we saw that there is a formal grammar counterpart to finite state automaton, namely regular (that is, either left-linear or right-linear) grammars. The formal grammar equivalent of a push-down automaton is a context-free grammar.

Definition 26

Let $\mathcal{G} = \langle \Sigma, N, S, P \rangle$ be a formal grammar. $\mathcal{G}$ is a **context-free grammar** (CFG) if and only if $P \subseteq N \times (\Sigma \cup N)^*$. That is, a context-free grammar is a formal grammar where all production rules have a single non-terminal symbol on the left-hand side and a string of made up of terminal and non-terminal symbols on the right-hand side.

Note, first of all, that every regular grammar is context-free. Productions like $S \to S1$ are left-linear, but they also fall within what is allowed to qualify as context-free. CFGs allow for much more than linear productions, though. For instance, $S \to 1S1$ is a typical production rule that qualifies as context-free but not linear/regular.

Here is a context-free grammar for $\{0^n1^n \mid n \in \mathbb{N}\}$:

$$\langle \{0,1\}, \{S\}, S, \left\{ \begin{array}{c} S \to \epsilon \\ S \to 0S1 \end{array} \right\} \rangle$$

CFGs and PDAs correspond to exactly the same class of languages, the *context-free languages.* I will not give the proof here, but to get the intuition, here's a procedure to construct a PDA that is equivalent to some CFG.

Let $\mathcal{C} = \langle \Sigma, N, S, P \rangle$ be a CFG. The corresponding PDA $\mathcal{A}$ will be the sextuple:

$$\langle \Sigma, \Sigma \cup N, \{s_0, s_1\}, s_0, \{s_1\}, R \rangle$$

(So, the stack alphabet is the set of terminal and non-terminal symbols of the CFG and there are just two states, one a start state and the other an acceptance state.) The transitions R are constructed as follows: First of all, R contains the transition $(s_0, \epsilon, \epsilon) \to (s_1, S)$. This is a transition to the acceptance state by reading nothing and pushing the start non-terminal symbol to the stack. Every rule in P is converted into a transition in R. If $X \to \sigma$ is a production, then the corresponding transition in R is $(s_1, \epsilon, X) \to (s_1, \sigma)$. This transition reads nothing, but pops the non-terminal X from the stack and pushes the right-hand side of the production to it. Finally, for each terminal symbol x, we add a transition $(s_1, x, x) \to (s_1, \epsilon)$.

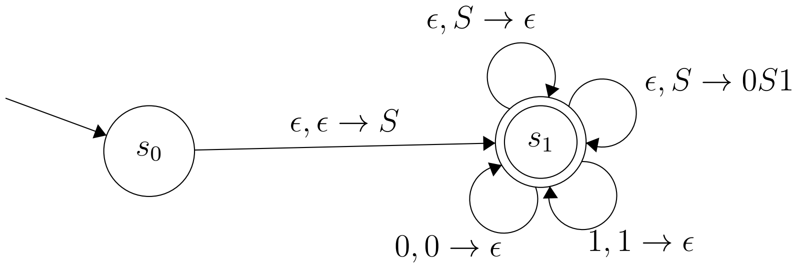
To see how this works, let us take the above CFG for $\{0^n1^n \mid n \in \mathbb{N}\}$. Given this CFG, the corresponding PDA will look like:

$$\langle \{0,1\}, \{0,1,S\}, \{s_0, s_1\}, s_0, \{s_1\}, R \rangle$$

Following the above instructions, R will amount to:

$$\begin{array}{c} (s_0, \epsilon, \epsilon) \to (s_1, S) \\ (s_1, \epsilon, S) \to (s_1, \epsilon) \\ (s_1, \epsilon, S) \to (s_1, 0S1) \\ (s_1, 0, 0) \to (s_1, \epsilon) \\ (s_1, 1, 1) \to (s_1, \epsilon) \end{array}$$

Graphically, this PDA looks as follows:



If we take this PDA and try to find a computation that proves acceptance of 000111, we find:

$$(s_0, 000111, \epsilon) \vdash (s_1, 000111, S) \vdash (s_1, 000111, 0S1) \vdash (s_1, 00111, S1)$$
$$\vdash (s_1, 00111, 0S11) \vdash (s_1, 0111, S11) \vdash (s_1, 0111, 0S111) \vdash (s_1, 111, S111)$$
$$\vdash (s_1, 111, 111) \vdash (s_1, 11, 11) \vdash (s_1, 1, 1) \vdash (s_1, \epsilon, \epsilon)$$

This PDA is quite different from the one I gave earlier, but – as you may verify – it accepts exactly the same language.

5.3 Chomsky Normal Form

Compared to linear grammars, the productions of context-free grammar can become very complicated. For instance, $S \to AaBbCcDdEe$ is a valid context-free production. Such rules complicate parsing quite a bit. For this and various other reasons, some of which we will see below, it is a good idea to impose some stricter contraints on the shape of context-free productions. This is the so-called *Chomsky Normal Form* (CNF). The rules are as follows:

- At most one production contains the empty string, namely a production which maps the start symbol to ϵ.
- All other productions take one of two forms:

$A \to BC$	a non-terminal mapped to two non-terminals
$A \to a$	a non-terminal mapped to an element of the alphabet

- Finally, the start symbol is not allowed to occur on the right-hand side.

These restriction would only make sense if they are harmless with respect to the class of languages that can be expressed. Indeed, CNF context-free grammars correspond to exactly the same class of languages as non-CNF CFGs (and PDAs) correspond to. So, we can obtain simpler rules and parse trees, without compromising on expressivity.

Part of how we know that CNF CFGs correspond to non-CNF CFGs is because there is an algorithm that can tranform each non-CNF CFGs into a (weakly) equivalent grammar in Chomsky normal form. It is important to know this algorithm, so you can always produce a CNF on the basis of some context-free grammar.

Say, $G = \langle \Sigma, N, S, R \rangle$ is a context-free grammar. If we want to turn it into Chomsky normal form grammar, this grammar will take the following form: $G' = \langle \Sigma, N_c, S_c, R_c \rangle$. Here, S_c is an entirely new non-terminal symbol. That is, $S_c \notin N$. Obviously, we do require that $S_c \in N_c$. We now consider R and change the productions in several ways until we have a CNF.

The first kind of change is adding a rule $S_c \to S$. This simply connects the old start symbol to the new one, in order to make sure the start symbol doesn't occur on the right-hand side of any rule.

Note that this is not a production that is accordance to the CNF rules, but ignore this for now, we will deal with that problem in a later step.

The second change we need to consider is to get rid of any rules containing the empty string. So, if R contains a rule $X \rightarrow \epsilon$, we remove this rule and make sure that this omission doesn't have consequences elsewhere. For instance, if some other rule looks like $Y \rightarrow AXb$, then we need to double this rule to account for the option that X is empty. So, the new production set will not just have $Y \rightarrow AXb$, but also $Y \rightarrow Ab$. This way the new grammar does all that the old one did, without making use of ϵ. Note that these rules are not yet CNF, but that will be fixed later.

The third manipulation that will bring us closer to CNF is to remove rules that have a single non-terminal on the right-hand side. A rule like $X \rightarrow Y$ can be removed by looking at rules with Y on the left-hand side and replacing Y in $X \rightarrow Y$ with that right-hand side. For instance, if $Y \rightarrow y$, then we can replace $X \rightarrow Y$ with $X \rightarrow y$.

The next change is to deal with rules that have more than two symbols on the right-hand side. Say, we have $X \rightarrow \alpha_1\alpha_2\alpha_3\ldots\alpha_n$ where α_i could be both terminal and non-terminal symbols. We can replace this with binary branching rules as follows.

$$\begin{array}{rcl} X & \rightarrow & \alpha_1 X_1 \\ X_1 & \rightarrow & \alpha_2 X_2 \\ X_2 & \rightarrow & \alpha_3 X_3 \\ \vdots & & \vdots \\ X_n & \rightarrow & \alpha_n \end{array}$$

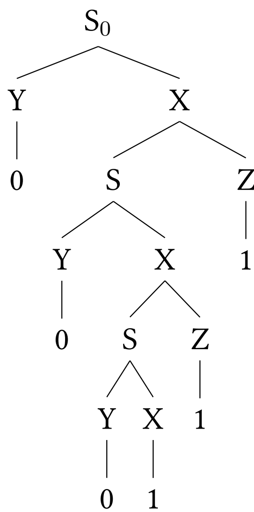
Note that we need to make sure that $X_1, \ldots X_n \notin N$. These have to be fresh non-terminal symbols that have not been used elsewhere in the grammar. For all but the final rule it is the case that these are only CNF productions if the α_i is a non-terminal. If it is not, then the next step deal with this. The final rule is only CNF if α_n is a terminal. If it is not, our earlier strategy for dealing with rules with just a single non-terminal on the right hand side will take care of it.

The final kind of non-CNF production we need to be able to deal with are rules where there are two elements on the right-hand side, but one of them is a terminal. This is easy. A rule like $X \rightarrow zY$ can be replaced by the combination $X \rightarrow ZY$ and $Z \rightarrow z$. (Similarly, for when the terminal is right of the non-terminal.)

This is all we need to transform any non-CNF CFG into a CFG that does adhere to the strict constraints of Chomsky normal form. Here's an example, the CFG for $\{0^n1^n \mid n \geq 0\}$ that I gave earlier:

$S \rightarrow \epsilon$	$S_0 \rightarrow S$	$S_0 \rightarrow S$	$S_0 \rightarrow S$	$S_0 \rightarrow \epsilon$	$S_0 \rightarrow \epsilon$
$S \rightarrow 0S1$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	$S_0 \rightarrow YX$	$S_0 \rightarrow YX$
	$S \rightarrow 0S1$	$S \rightarrow 0X$	$S \rightarrow YX$	$S \rightarrow \epsilon$	$S \rightarrow YX$
		$X \rightarrow S1$	$Y \rightarrow 0$	$S \rightarrow YX$	$Y \rightarrow 0$
			$X \rightarrow SZ$	$Y \rightarrow 0$	$X \rightarrow SZ$
			$Z \rightarrow 1$	$X \rightarrow SZ$	$X \rightarrow 1$
				$Z \rightarrow 1$	$Z \rightarrow 1$
the original grammar	adding new start symbol	reducing the right hand sides > 2	removing terminals from the right-hand sides	removing productions with one non-terminal on right-hand side	removing productions (other than the start symbol) that involve the empty string

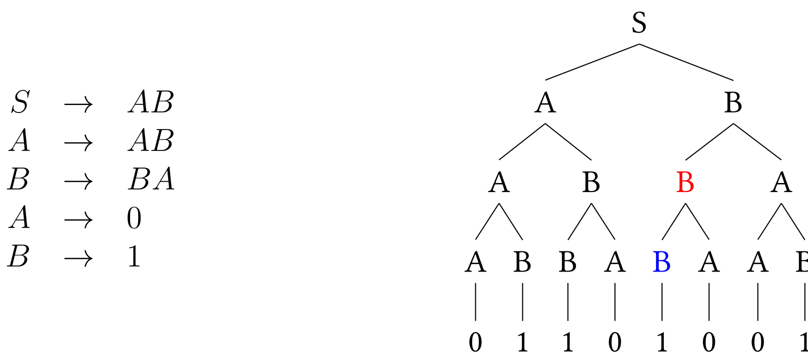
Here is an example parse tree corresponding to the derivation of the string 000111. The table next to the parse tree shows the left-most derivation corresponding to this parse tree.



left-most derivation of 000111	
$(S_0, (Y, X))$	$S_0 \to Y\ X$
$(S_0, ((Y, 0), X))$	$Y \to 0$
$(S_0, ((Y, 0), (X, (S, Z))))$	$X \to S\ Z$
$(S_0, ((Y, 0), (X, ((S, (Y, X)), Z))))$	$S \to X\ Z$
$(S_0, ((Y, 0), (X, ((S, ((Y, 0), X)), Z))))$	$Y \to 0$
$(S_0, ((Y, 0), (X, ((S, ((Y, 0), (X, (S, Z)))), Z))))$	$X \to S\ Z$
$(S_0, ((Y, 0), (X, ((S, ((Y, 0), (X, ((S, (Y, X)), Z)))), Z))))$	$S \to Y\ X$
$(S_0, ((Y, 0), (X, ((S, ((Y, 0), (X, ((S, ((Y, 0), X)), Z)))), Z))))$	$Y \to 0$
$(S_0, ((Y, 0), (X, ((S, ((Y, 0), (X, ((S, ((Y, 0), (X, 1))), Z)))), Z))))$	$X \to 1$
$(S_0, ((Y, 0), (X, ((S, ((Y, 0), (X, ((S, ((Y, 0), (X, 1))), (Z, 1))))), Z))))$	$Z \to 1$
$(S_0, ((Y, 0), (X, ((S, ((Y, 0), (X, ((S, ((Y, 0), (X, 1))), (Z, 1))))), (Z, 1)))))$	$Z \to 1$

5.4 Pumping lemma

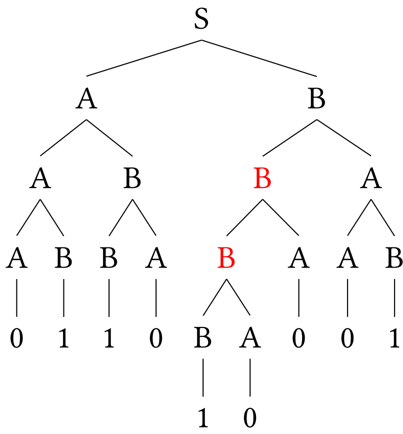
Consider the following CFG in Chomsky normal form. The start symbol is S. Next to the grammar is an example parse tree, corresponding to the derivation of the string 01101001. (Ignore the colours for now.)



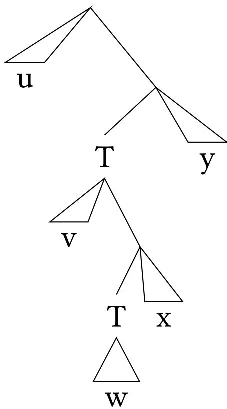
Now note the following. Since the grammar is in CNF, each parse tree will have at most binary branching at each node. So, the width of a tree at each level n of branching is at most 2^n. (The root node is level 0, so the tree has width $2^0 = 1$ there. The next level down is the first branching, level 1 and has width $2^1 = 2$. Etc.) Given the relation between the width and height of the tree we can deduce from the size of the string what the height of the tree will minimally be. Since the leaves of the tree will be arrived at via unary branching (as per Chomsky normal form), a tree of n branching levels will have at most 2^n symbols in the string it derived. The tree above branches on three levels and does so maximally and, so, its number of leaves is 2^3. Reversely, if we didn't have the tree for this string, we could still conclude that the height of the tree (measured in branchings) will be at least 3.

The grammar above only has 3 non-terminals. Since the height of the tree is 3, there will be a path from the root node to a leaf that contains 4 non-terminals, this entails that there must be paths from the root node (S) to a leaf that contains the same non-terminal more than once, as it indeed does. Given that this repetition was possible once, it must be the case that it is possible more times, and it must be the case that the repetition could have been omitted. That is, such repetitions show that stuff can be pumped, just like we discovered for regular languages. Let's illustrate this for the tree above.

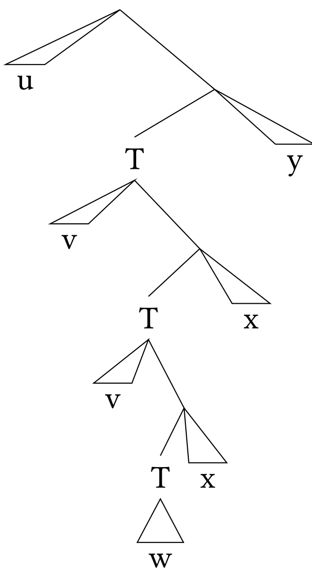
Consider the B node that is coloured red in the tree above. That node illustrates a derivation $B \Rightarrow^* 10$. But this means we should be able to replace the blue node B with a copy of the red node B. So, the following should also be a parse tree for this grammar.



The idea behind the pumping lemma for context-free languages generalises this idea of repeatability. Consider the following abstract parse tree for a string uvwxy, from an unknown CNF context-free grammar. (The triangular shapes represent unknown sub-trees that derive the (sub)strings u, v, w, x and y.)



Given this tree, we know that uvwxy belongs to the language. But we also know that T derives vwx. As a result, we know that we can replace the lowest node T with the parse tree for $T \Rightarrow^*$vwz, resulting in a parse tree for uv^{2}wx^{2}y.



Clearly, we can repeat this as many times as we want. Also, we could replace the sub-tree rooting in the top-most T with the small tree for $T \Rightarrow$ w, which would result in the string uwy. That is, given the fact that we saw a parse tree for uvwxy with two occurrences of T, we know that for all $i \geq 0$, the string uv^{i}wx^{i}y is in the language.

All this is the intuition behind the pumping lemma for context-free languages. Here's the formal statement of the lemma:

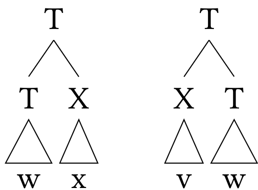
Theorem 10

Every infinite context-free language L is such that there exists a number m, such that for every string $\sigma \in L$ where $|\sigma| \geq m$, σ can divided up as $uvwxy$ and the following hold:

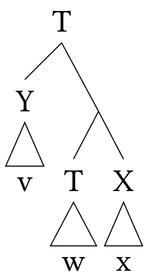
1. $|vwx| \leq m$
2. $|vx| \geq 1$
3. For every $i \geq 0$: $uv^iwx^iy \in L$

As we saw above, as soon as we find a path in a parse tree that contains a repetition of non-terminal, we have evidence of other strings in the language. As soon as we have a subtree rooted in non-terminal T that contains a node T deeper in that subtree, we can replace that node in the subtree with the subtree itself. This holds of paths of arbitrary length, as long as they exceed the pumping length. However, to make the lemma easy to use, we can focus on the repetition that is as close to the bottom of the tree as possible. That is, if there is a path longer than m, then for some non-terminal T, there will be a *minimal* subtree rooted in T with exactly one other occurrence of T deeper in that subtree. This subtree cannot be longer than m. If it was longer, then it must contain some other repetition in it - and it wouldn't be minimal. This is why the lemma can state that $|vwx| \leq m$.

The second condition in the lemma $|vx| \geq 1$ holds because the closest the repeated non-terminals are in the subtree that derives vwx is one node apart. As the following trees illustrate, that would result in either v or x being empty.



As soon as the repeated nodes are further apart neither v nor x can be empty:



Because we know there is a repetition, the two trees where the T nodes are one node apart are the shortest the string vwx can be. In other words, at most one of v or x can be empty, but not both.

I won't prove the lemma here, but hope the intuition behind the lemma sketched above suffice to understand how it works. As with the pumping lemma for regular languages, the typical application of the pumping lemma is only useful to prove that a language does *not* belong to the class. That is, we apply the above lemma if we want to prove that a language isn't context-free. If we encounter a language L and prove using the pumping lemma for regular languages that it is not regular, the next step could be to prove either that it is context-free (by providing a PDA or CFG for it) or that it is not context-free by applying the above lemma. As we saw above, all regular languages are context-free

(any linear grammar is context-free), so once we know that a language is not context-free, then we also know that it is not regular.

Consider the language $L = \{a^n b^n c^n \mid n \geq 0\}$. Notice first (informally) that this language is not regular. We won't be able to construct an FSA for this language, for exactly the same reasons as we won't be able to construct an FSA for the language $\{a^n b^n \mid n \geq 0\}$. But while this latter language is context-free, L is neither regular nor context-free. We can show this by applying the pumping lemma. Let's assume that L *is* context-free and that the pumping length is n. Then consider $\sigma = a^n b^n c^n$. Since $|\sigma| > n$ the properties mentioned in the lemma should now hold for some decomposition $\sigma = uvwxy$. The first property is that $|vwx| \leq n$. Given this, vwx either contains no as or it contains no cs. (If vwx contains both, it should also contain n bs, which makes it longer than n.) It follows from this that when we pump $uv^i wx^i y$ we end up with strings that either contain more cs than as or more as than cs. These resulting strings are not part of the language, which contradicts our assumption. This proves that L is not context-free.

5.5 Closure properties

As before with regular languages, it is interesting to consider closure properties of context-free languages. This may in some cases help us deduce the complexity of a language from the complexity of other languages.

Context-free languages are closed under union, concatenation and Kleene star. It is relatively easy to see why, using context-free grammars. Say we have two CFGs, $G_1 = \langle \Sigma_1, N_1, S_1, P_1 \rangle$ and $G_2 = \langle \Sigma_2, N_2, S_2, P_2 \rangle$. Assume that $N_1 \cap N_2 = \emptyset$. (Note, that we can always rename the non-terminals of a grammar and get a strongly equivalent grammar in return. In other words, this assumption is not essential to the proof.) It is now very easy to build context-free grammars that correspond to $\mathcal{L}(G_1) \cup \mathcal{L}(G_2)$, $\mathcal{L}(G_1) \cdot \mathcal{L}(G_2)$ and $\mathcal{L}(G_1)^*$. Since we know that $\mathcal{L}(G_1)$ and $\mathcal{L}(G_2)$ are context-free (we have CFGs for them), it will follow that context-free languages are closed under union, concatenation and Kleene star.

To build CFGs for the language obtained by union or concatenation, all we need to do is construct a new grammar $\langle \Sigma_1 \cup \Sigma_2, N_1 \cup N_2 \cup \{S\}, S, P \rangle$, where $S \notin N_1 \cup N_2$ and P is as follows:

$$
\begin{array}{ll}
P = P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\} & \text{union} \\
P = P_1 \cup P_2 \cup \{S \rightarrow S_1\ S_2\} & \text{concatenation}
\end{array}
$$

To obtain $(\mathcal{L}(G_1))^*$ we can take the grammar $\langle \Sigma_1, N_1 \cup \{S\}, S, P_1 \cup \{S \rightarrow SS_1, S \rightarrow \epsilon\} \rangle$.

Here's an example. Consider the following two CFGs with start symbol S_1 and S_2, respectively.

$$
\begin{array}{lllllll}
S_1 & \rightarrow & 1\ S_1\ 0 & \qquad & S_2 & \rightarrow & 1\ S_2\ 0 \\
S_1 & \rightarrow & 10 & \qquad & S_2 & \rightarrow & 110
\end{array}
$$

These correspond to $\{1^n 0^n \mid n \geq 1\}$ and $\{1^{n+1} 0^n \mid n \geq 1\}$, respectively. To obtain the union of these languages, we can simply construct the following grammar.

$$
\begin{array}{lll}
S & \rightarrow & S_1 \\
S & \rightarrow & S_2 \\
S_1 & \rightarrow & 1\ S_1\ 0 \\
S_1 & \rightarrow & 10 \\
S_2 & \rightarrow & 1\ S_2\ 0 \\
S_2 & \rightarrow & 110
\end{array}
$$

To obtain the concatenation of the two languages we construct the following:

$$\begin{array}{lcl} S & \rightarrow & S_1 S_2 \\ S_1 & \rightarrow & 1\ S_1\ 0 \\ S_1 & \rightarrow & 10 \\ S_2 & \rightarrow & 1\ S_2\ 0 \\ S_2 & \rightarrow & 110 \end{array}$$

Finally, if we want to construct a grammar for $\{1^n 0^n \mid n \geq 1\}^*$, which is the language $\{\epsilon, 1^n 0^n, 1^n 0^n 1^m 0^m, 1^n 0^n 1^m 0^m 1^k 0^k \ldots | n \geq 1, m \geq 1, k \geq 1 \ldots\}$, the following grammar will do:

$$\begin{array}{lcl} S & \rightarrow & S S_1 \\ S & \rightarrow & \epsilon \\ S_1 & \rightarrow & 1\ S_1\ 0 \\ S_1 & \rightarrow & 10 \end{array}$$

For regular languages, we showed that the complement of any such language is regular as well. This is not the case for context-free languages.

Theorem 11

Context-free languages are **not closed under complementation**.

Proof

Take $L_1 = \{1^i 2^i 3^j \mid i \geq 0 \text{ and } j \geq 0\}$ and $L_2 = \{1^i 2^j 3^j \mid i \geq 0 \text{ and } j \geq 0\}$. Both are context-free. (Check this by providing a CFG for the languages.) Let us assume that context-free languages *are* closed under complementation. Then $\overline{L_1}$ and $\overline{L_2}$ are context-free. Given the fact that context-free language are closed under union, $\overline{L_1} \cap \overline{L_2}$ is also context free. Call this language L. $\overline{L_1}$ is the language that has sequences of 1s followed by sequences of 2s, then sequences of 3s in such a way that the numer of 1s and 2s is not equal. $\overline{L_2}$ is similar except that the number of 2s and 3s are not equal. So, L is the language with strings like this with either (or both) the number of 1s and 2s or the number of 2s and 3s not being equal. Now take the complement of L: the language of strings with 1s followed by 2s and 3s, where the number of 1s and 2s are equal and the number of 2s and 3s are equal. So, $\overline{L} = \{1^i 2^i 3^i \mid i \geq 0\}$. Since L is context-free and the assumption is that complements of context-free languages are context-free, then we have to conclude that $\overline{L}$ is context-free. But it clearly is not. In fact, $\overline{L}$ was our prime example of a language that is *not* context-free. This shows that our assumption that context-free languages are closed under complementation must be false.

Theorem 12

Context-free languages are **not closed under intersection**.

Proof

Once more, take $L_1 = \{1^i 2^i 3^j \mid i \geq 0 \text{ and } j \geq 0\}$ and $L_2 = \{1^i 2^j 3^j \mid i \geq 0 \text{ and } j \geq 0\}$. Note that $L_1 \cap L_2 = \{1^i 2^i 3^i \mid i \geq 0\}$, which is not context-free.

5.6 Mirroring versus copying, and natural language

Consider the languages M and C, respectively the mirror language and the copy language:

$$M = \{ss^R \mid s \in \{0,1\}^*\}$$
$$C = \{ss \mid s \in \{0,1\}^*\}$$

Here, the operation $\cdot^R$ returns the reverse of the string is operates on. So, if $s = 01$, then $ss^R = 0110$. So, M contains all string so that the first half of the string is a string in $\{0,1\}^*$ and the second half is the reverse of this string. The copy language takes any string that is made up of two consecutive exact copies of a string over the alphabet $\{0,1\}$.

Note that neither of these languages is regular. (The proofs are a good exercise). Language M is context-free, which can be proved easily by providing the following context-free grammar with start symbol S.

$$S \to 0S0$$
$$S \to 1S1$$
$$S \to 11$$
$$S \to 00$$
$$S \to \epsilon$$

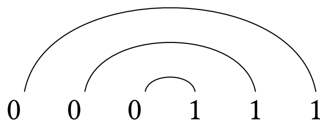
The copy language, on the other hand, is not context-free. To prove this, we walk through an application of the pumping lemma. If C were context-free, then there must be a pumping length n. Let's then consider the string $0^n1^n0^n1^n \in C$. We decompose this string into $uvwxy$ such that $|vwx| < n$. It follows that vwx is one of two types: (i) it contains only 1s or only 0s; (ii) it is made up of a series of 1s followed by a sequences of 0s or made up of a series of 0s followed by a series of 1s. In case (i) pumping would increase the number of 1s or 0s in one part of the string without adjusting the 1s and 0s elsewhere. The result is not an element of C. Note that in case (ii) v and x will contain fewer than n 1s or 0s. Also, either u will contain n 0s or y will contain n 1s (or both). So, the situation looks like one of these (the red vertical line is there to help you spot where the coping takes place):

$$0^n \; 1^{n-j} \; \underbrace{1^j \mid 0^k}_{vwx} \; 0^{n-k} \; 1^n \qquad 0^{n-j} \; \underbrace{0^j \; 1^k}_{vwx} \; 1^{n-k} \mid 0^n \; 1^n \qquad 0^n \; 1^n \mid 0^{n-j} \; \underbrace{0^j \; 1^k}_{vwx} \; 1^{n-k}$$

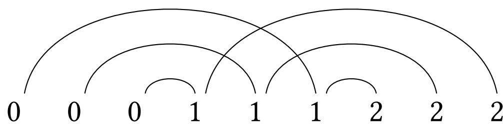
In the right-most situation, we cannot pump. The number of 0s and 1s left of the copy line is n, but if we pump there will be more 0s and/or 1s right of the copy line. The same holds for the situation in the middle. Here, there are n 0s and 1s on the right-hand side of the copy line and so if we pump we'll also loose the copying property. So, the only interesting case is the left-most one. It is instructive to go through all the possible ways vwx could be carved up. Let's first consider the case that v is empty. Then there are two possibilities. First, x contains only 0s. If this is the case, then pumping will result in a string with more 0s right of the copy line than on the left. Second, x contains all 0s but also some 1s. If we then remove x, the resulting string should still be in the language, but it cannot be, because the number of 0s left of the copy line will now exceed those right of the line (and vice versa for the number of 1s). We can go through similar options for the case where x is empty, with the same result. Finally, we consider the case where v and x are both not empty. Now v will contain at least one 1 and x will contain at least one 0. So, if we pump, we will increase the 1s on the left hand side of the copy line and we will increase the 0s on the right hand side of the copy line. The resulting string will not be in the language. This exhausts our options and proves that c is not context-free.

A crucial difference between mirroring and copying is that the former but not the latter can be achieved with **center-embedding**. Center-embedding is recursion that is nested, in the sense that the recursive step takes place within a string, rather than at the edge of that string. This is why $\{0^n1^n \mid n \geq 0\}$ is context-free: once we have a string in that language, we can build a new string

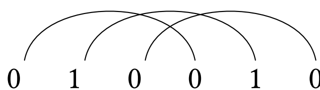
but splitting it in half and *nesting* 01 between the two halves. The way these strings are built causes a very typical dependency, illustrated here:



The language $\{0^n1^n2^n \mid n \geq 0\}$ is not context-free (as we saw earlier). This is because the strings simply cannot be built by having nested dependencies. There will always be crossed dependencies. For instance,



Similarly, the copy language also cannot be constructed using center embedding. Here it is very clear what the dependencies are, since the symbols in the copied string have to be repeated in the second half of the string, symbol by symbol. For instance, the following string is in C:



There's been considerable discussion in the linguistic literature on whether natural language is ever not context-free. That is, can we find natural languages where we see phenomena that involve recursion that creates crossed, not just nested dependencies. Notice, first, that English has clear examples of center embedding. Consider the following sentence consisting of a determiner (the), a noun (fish) and a verb (smelled). Schematically, you could thus see this sentence as a sequence DNV.

(8) The fish smelled. DNV

These sentences can be nested as followes:

(9) The fish the cat ate smelled DNDNVV

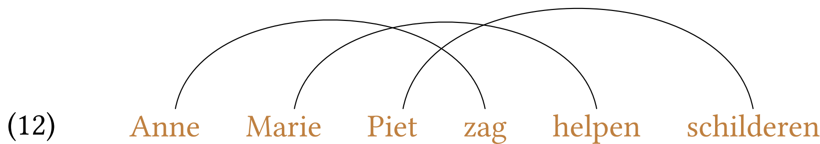
(10) The fish the cat my father loves ate smelled DNDNDNVVV

For a while, Dutch was seen as a candidate language that shows crossing dependencies (Huybregts, Utrecht working papers in Linguistics, 1979). Consider the following embedded clause, for instance:

(11) ... dat Anne Marie Piet zag helpen schilderen.
that Anne Marie Piet saw help paint.
"*...that Anne saw that Marie was helping Piet paint.*"

This sentence contains the pattern Name Name Name Verb Verb Verb. Crucially, however, the first name is the subject of the first verb, the second one of the second verb etc. So, this is clearly a case of crossing dependencies.

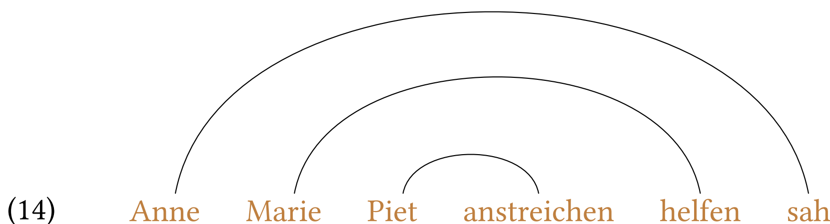
(12)



Note that the same sentence in German is nested and not crossed:

(13) ... daß Anne Marie Piet anstreichen helfen sah.
that Anne Marie Piet paint help saw.
"*...that Anne saw that Marie was helping Piet paint.*"

(14)



The problem with Huybregts' argument was that from a purely syntactic point of view, Dutch still looks context-free (Gazdar & Pullum, Linguistics & Philosophy, 1982). It simply contains the pattern Name^n Verb^n, which can be generated using center-embedding. A counter-argument may be that even if Dutch syntax is context-free, it seems that Dutch semantics cannot be that. There exists, however, a language very close to Dutch which displays crossing dependencies clearly in syntax. In Schwiizerdütsch (Swiss German), sentences close to the above Dutch one exist, with the same word order, but with the addition of case marking, which allowed Shieber (Linguistics & Philosophy, 1985) to strengthen Huybregts' argument to a purely syntactic one.