Course notes for

Computability and Complexity

Gordon J. Pace

Department of Computer Science and AI
University of Malta

February 2003

Contents

Chapter 1

Introduction

1.1 Syntax and Semantics

Throughout this course we will be talking about languages. A language is a collection of sequences of symbols. In English, for example, the symbols are English words, and the English language is a set of valid sentences including "The mouse ate the cheese". Prime numbers can also be seen as a language: the symbols being the digits from 0 to 9, and the language includes strings such as 23 and 53 but not 18 and 1001.

Consider the sentence "Mouse dinosaur the". The sentence is, indisputably, not part of the English language. We say that it grammatically incorrect. Now consider "Large fat animals eat frequently". This is unarguably correct. But now consider the sentence "Colourless green ideas sleep furiously". Is this a valid English sentence? It seems to be grammatically correct, following the same structure as the previous sentence. In contrast, however, the new sentence makes no sense. How can the two adjectives colourless and green be applied to the same object? And since when do ideas sleep, or have a colour if it comes to that? And sleeping furiously? Grammatically correct sentences may still be meaningless. We thus need to make an important distinction between the grammar of the language, and the actual meaning of the statement in the language. Technically, we refer to these as the *syntax* and *semantics* of the language respectively.

Syntax: 1. The study of the rules whereby words or other elements of sentence structure are combined to form grammatical sentences; 2. (Computer Science) The rules governing the formation of statements in a programming language.

Semantics: 1. The study or science of meaning in language; 2. The study of relationships between signs and symbols and what they represent; 3. The meaning or the interpretation of a word, sentence or other language form:

We're basically agreed; let's not quibble over semantics.

Source: ???

Can we construct a similar example in a computer language to show this dichotomy? Consider the following C program:

```
int main () { return(1);
```

This fails to satisfy the basic rules of syntax of the language, which insist that there must be a corresponding closing `}` for the one following the definition of the main function.

What about the following program?

```
int main () { return (1/0); }
```

Provided the compiler does not try to optimise the program, the program will compile successfully, but will give a run-time error when it tries to divide by zero. Clearly, this is a syntactically valid C program which is not semantically meaningful.

So how do we deal with languages? What is a valid sentence? How do we describe languages, and can we always write computer programs to tell us whether a given sentence is grammatically correct or not?

1.2 Describing Languages

Have you ever noticed that there is an infinite number of valid English sentences? It is easy to come up with a schema to create such an infinite collection. Consider the sentence: "One is considered to be an unlucky number". "One" can be replaced by any other number in the sentence without making it any less grammatically correct. Since there is an infinitude of numbers that can be described in English, we can construct an infinite number of sentences in this manner. Another schema to construct an infinite number of sentences is: "The house next to mine is red", "The house next to the house next to mine is red", "The house next to the one next to the one next to mine is red", etc ad infinitum.

But even if you never noticed the infinite nature of the English language, you have probably remarked on the finite size of our brains. It thus immediately follows that the brain cannot enumerate (list) all valid sentences, but must somehow compress the information describing the infinite language in a finite manner.

Although in this course we do not care how languages are internally represented in our brains, we would like to be able to identify tools to describe infinite languages using a finite description.

One way of describing a language is to use mathematical notation (which has the advantage of having an unambiguous and formal meaning). One such convenient notation is set comprehension:

$$E = \{a^n \mid \exists m : \mathbb{N} \cdot n = 2m\}$$

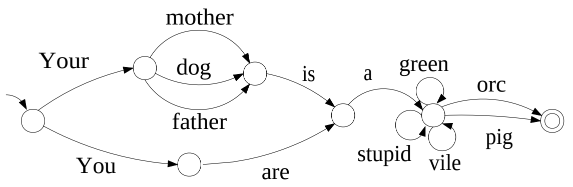
If we take the meaning of a^n to be "the symbol a repeated n times", then E is the language of all strings consisting of an even number of as.

Another way of describing languages is using so called "production rules". For instance, in English we would say that a noun phrase can be (amongst other things) a noun, or an adjectival phrase followed by a noun. We would write this as:

$$\begin{aligned} \langle NP \rangle &\rightarrow \langle N \rangle \mid \langle AP \rangle \langle N \rangle \mid \ldots \\ \langle N \rangle &\rightarrow \text{cat} \mid \text{mouse} \mid \text{dog} \mid \ldots \end{aligned}$$

Thus, if I were to ask you whether "cat" is a valid noun phrase, you would answer affirmatively, explaining that a noun phrase can be made up of just a noun (using the first rule), and that "cat" is a valid noun (using the second rule). Note that I have enclosed the grammatical classes in angle brackets to make it clear that they are not actual symbols in the language, but symbols used during the derivation and which will never appear in a valid sentence.

As a final example, a language may be described visually in terms of a diagram. The following diagram describes a language of insults:



The diagram can be used as in a game. Given a sentence, the diagram can be used to tell us whether it is a valid one or not. We start off with a marker on the circle marked with an incoming arrow (on the extreme left of the diagram). At every move we read the next word in the sentence. If there is an outgoing arrow from the currently marked circle with the word just read out, we move the marker to the destination of the arrow. If, at the end of the sentence, the marker lies on a double circle, the phrase is a valid one. If not, that is we end up on a normal circle, or halfway through the sentence we encounter a word with no corresponding outgoing arrow, then the phrase is incorrect. Thus, "Your mother is a vile green orc" is a valid insult, while "Your mother are a vile orc" and "You are a vile green" are not.

In this course we will be examining different means of describing languages. One obvious question is whether different mechanisms are equally powerful. For example, we can ask whether any set language expressed as a set comprehension can always be represented in a diagram similar to the above. The answer is, in fact, that it is not always possible — for certain languages, we will need a more sophisticated kind of diagram. This gives us an opportunity to classify languages, based upon the level of complexity of the production rules or diagrams required to describe them. Once we identify these classes, we can then analyse the complexity of the computer program required to accept that class of languages (a program which, given a sentence, tells you whether or not the string belongs to the language). How much memory is required? How many steps are required to tell whether a given sentence is in the language or not? It turns out that there is a class of languages for which it is impossible to write a program which always answers this membership question. This has important implications in computer science, since we will have identified a task impossible for a machine to perform. This leads us to a short tour of the area of computability, where we briefly study these limits.

However, not all that is computable is tractable. Imagine a language, where the most efficient program to give a yes/no answer, when given a sentence made up of n symbols, takes 2^n steps to calculate the answer. Even though the language is decidable, it is clearly not practically so. Even if we double the speed of computers, we can only analyse sentences which are one symbol longer — not a very practical improvement. In the final part of the course we will look into a class of languages (or, equivalently, programming tasks) which for which the best known algorithms are exponential, but for which, as yet, no one has managed to prove the non-existence of a more efficient algorithm solving the problem. This class includes a number of very important problems, and the question of whether a more efficient algorithm exists is regularly cited as one of the most important problems in mathematics and computer science today.

1.3 Revisiting Syntax vs Semantics

After this short digression in language representation, we can now begin to appreciate the syntax/semantics question slightly better. Syntax seems to be characterised by a set of rules which can be used to check the validity of sentences. For example, the production rules used to generate English sentences ignore semantics, allowing us to generate "Colourless green ideas sleep furiously" but not "Dog dinosaur the". Once these have been weeded out, a semantic model handling the meaning of the sentences can be used to identify which of the remaining sentences also make sense.

Consider the following grammar to generate sentences of the form $i + j = k$ where i, j and k are represented in unary as a repetition of the symbol 1 (eg 7 is represented as 1111111 or 1^7 in our shorthand notation).

$$\begin{array}{rcl} \langle S \rangle & \rightarrow & \langle N \rangle + \langle N \rangle = \langle N \rangle \\ \langle N \rangle & \rightarrow & 1\langle N \rangle \mid \varepsilon \end{array}$$

The rules say that a sum $\langle S \rangle$ is made up of three numbers $\langle N \rangle$ separated by the symbols + and =. A number $\langle N \rangle$ is just a (possibly empty) sequence of the symbol 1. Note that ε means "no symbol" since writing production rules like "$\langle N \rangle \rightarrow 1\langle N \rangle \mid$ " would be confusing.

Now we can argue that we have handled the basic language syntax, and it is now up to someone smarter to deal with the semantics — to filter out incorrect sums. However, the person we hand this task over to, comes back with another grammar:

$$\begin{array}{rcl} \langle S \rangle & \rightarrow & 1\langle S \rangle 1 \mid +\langle E \rangle \\ \langle E \rangle & \rightarrow & 1\langle E \rangle 1 \mid = \end{array}$$

First of all, look at $\langle E \rangle$. This generates a phrase of the form $1^n = 1^n$ by adding a 1 at the beginning and one at the end together. Using a similar technique, $\langle S \rangle$ adds more 1s at the end, and an equal number at the front, with the ones at the front followed by a + symbol.

Surprisingly, the semantics of the phrase have been expressed in terms of syntax. If this can be done for any semantic description (and, in fact, it turns out that any computable function can be written as a grammar), the distinction between syntax and semantics starts becoming rather blurred. However, the language complexity classes we will be identifying come to the rescue. Early on in the course, we will show that the second (semantically filtered) language is more complex than the first. While the first can be expressed in a diagram similar to the insult generator, the second cannot, requiring a more complex computation model. This gives us a motivation to separate the syntax of this language from its semantics.

1.4 Formalising Languages and Grammars

Notation: Given a set S, 2^S is the set of all subsets of S (called the power-set of S). The set of sequences over elements of S is written as S^*. The empty sequence, an element of S^* for any S, is written as ε.

1.4.1 Strings

Given a finite set Σ, we call $s \in \Sigma^*$ a *string over* Σ. Given two strings s, t we write the catenation of the strings simply by their juxtaposition: st. Catenation can be defined in terms of the basic operation of appending a single symbol to the start of a sequence:

$$\begin{array}{rcll} \varepsilon s & \overset{\text{df}}{=} & s & \\ (as)t & \overset{\text{df}}{=} & a(st) & \qquad \text{where } a \in \Sigma \end{array}$$

The length of a string s over Σ, written as $\sharp(s)$, is defined as follows:

$$\begin{array}{rcll} \sharp(\varepsilon) & \stackrel{\text{df}}{=} & 0 & \\ \sharp(as) & \stackrel{\text{df}}{=} & 1 + \sharp(s) & \text{where } a \in \Sigma \end{array}$$

Similarly, we can count the number of occurrences of a symbol a in a string s:

$$\begin{array}{rcl} \sharp_a(\varepsilon) & \stackrel{\text{df}}{=} & 0 \\ \sharp_a(bs) & \stackrel{\text{df}}{=} & \begin{cases} 1 + \sharp_a(s) & \text{if } a = b \\ \sharp_a(s) & \text{otherwise} \end{cases} \end{array}$$

We can also define repetition of a string for a particular number of times:

$$\begin{array}{rcl} s^0 & \stackrel{\text{df}}{=} & \varepsilon \\ s^{n+1} & \stackrel{\text{df}}{=} & ss^n \end{array}$$

Reversing a string can also be defined inductively:

$$\begin{array}{rcl} \varepsilon^R & \stackrel{\text{df}}{=} & \varepsilon \\ (as)^R & \stackrel{\text{df}}{=} & s^R a \end{array}$$

1.4.2 Languages

Given a finite set Σ, we say that L is a *language over* Σ if $L \subseteq \Sigma^*$. We call Σ the *alphabet of* L.

Since languages are nothing other than sets, we can talk about the intersection, union and set difference of two languages. The set complement of a language L (over alphabet Σ) is simply $L^c \stackrel{\text{df}}{=} \Sigma^* \setminus L$.

String operations can be lifted to work over languages in a straightforward manner:

$$\begin{array}{rcl} L_1 L_2 & \stackrel{\text{df}}{=} & \{s_1 s_2 \mid s_1 \in L_1,\ s_2 \in L_2\} \\ L^R & \stackrel{\text{df}}{=} & \{s^R \mid s \in L\} \end{array}$$

Language iteration is defined similar to string iteration:

$$\begin{array}{rcl} L^0 & \stackrel{\text{df}}{=} & \{\varepsilon\} \\ L^{n+1} & \stackrel{\text{df}}{=} & LL^n \end{array}$$

The *transitive closure* of a language L, written as L^+, is also a language consisting of any number of (non-zero) repetitions of L: $L^+ \overset{\text{df}}{=} \bigcup_{i=1}^{\infty} L^i$. The *reflexive, transitive closure* of a language L also includes zero repetitions: $L^+ \overset{\text{df}}{=} \bigcup_{i=0}^{\infty} L^i$.

1.4.3 Phrase Structure Grammars

A *phrase structure grammar* is based upon the use of production rules. It consists of a a quadruple $G = \langle \Sigma, N, I, P \rangle$ where Σ is the finite alphabet of the resulting language, sometimes called *the terminal symbols of G,* N is the set of extra internal symbols used in the grammar production rules, sometimes called *the non-terminal symbols of G,* I is the initial or start symbol ($I \in N$) from which all strings in the language must be derived, and P is the set of production rules $P \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$.

Example: Consider the grammar to produce correct sums:

$$\begin{array}{rcl} \langle S \rangle & \rightarrow & 1\langle S \rangle 1 \mid + \langle E \rangle \\ \langle E \rangle & \rightarrow & 1\langle E \rangle 1 \mid = \end{array}$$

This is formalised as the quadruple:

$\langle \{1, +, =\},\ \{S, E\},\ S,\ \{(S, 1S1),\ (S, +E),\ (E, 1E1),\ (E, =)\} \rangle$

We say that a string s' over $N \cup \Sigma$ can be derived from string s (over the same alphabet) in one step if s' is the result of applying a production rule to a substring of s. We write this as $s \Rrightarrow s'$:

$$s \Rrightarrow s' \quad \overset{\text{df}}{=} \quad \exists \alpha,\ t,\ t',\ \beta \cdot s = \alpha t \beta \wedge s' = \alpha t' \beta \wedge (t, t') \in P$$

We can generalise the number of steps in a string derivation:

$$\begin{array}{rcl} s \Rrightarrow^0 t & \overset{\text{df}}{=} & s = t \\ s \Rrightarrow^{n+1} t & \overset{\text{df}}{=} & \exists x \cdot s \Rrightarrow x \wedge x \Rrightarrow^n t \\ s \Rrightarrow^* t & \overset{\text{df}}{=} & \exists n : \mathbb{N} \cdot s \Rrightarrow^n t \end{array}$$

If $s \Rrightarrow^* s'$, we say that s' is *derivable from s.*

Given a grammar $G = \langle \Sigma, N, I, P \rangle$, we say that string $\alpha \in (N \cup \Sigma)^*$ is reachable if $I \Rrightarrow^* \alpha$. The set of all reachable strings is written as $\mathcal{R}(G)$:

$$\mathcal{R}(G) \quad \overset{\text{df}}{=} \quad \{\alpha \in (N \cup \Sigma)^* \mid I \Rrightarrow^* \alpha\}$$

The *language generated by grammar G,* written as $\mathcal{L}(G)$, is the set of all reachable strings consisting of just terminal symbols:

$$\mathcal{L}(G) \quad \overset{\text{df}}{=} \quad \mathcal{R}(G) \cap \Sigma^*$$

We sometimes also say that G *produces* $\mathcal{L}(G)$.

1.5 Proving Properties of Grammars

Formalising languages and grammars allows us to reason formally about them, proving properties and equivalence of grammars and languages. Although in this course we will not be seeing all that many proofs about particular languages, it is important to see a couple of proofs before we can start proving properties of *classes* of languages.

1.5.1 Correct Sums

Consider grammar G which produces only correct sums:

$$\begin{array}{rcl} S & \rightarrow & 1S1 \mid +E \\ E & \rightarrow & 1E1 \mid = \end{array}$$

where $\{S,\ E\}$ are non-terminals, $\{1,\ +,\ =\}$ terminals and S begin the initial symbol. We would like to prove that the language generated by G, is the language of all correct sums and nothing else:

$$\mathcal{L}(G) = \{1^n + 1^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$$

How can we prove this? We start off by splitting the proof into two parts: (a) $\mathcal{L}(G) \supseteq \{1^n{+}1^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$ and (b) $\mathcal{L}(G) \subseteq \{1^n{+}1^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$. Clearly, if we prove (a) and (b), we have proved the desired result.

Proof of (a): We are required to prove that all correct sums are generated by the grammar. Intuitively, why is this so? Well, starting from S, we can generate all strings of the form 1^nS1^n, from which we would argue that it follows that we can generate all strings of the form $1^n + 1^mE1^{n+m}$, from which we can conclude that all correct sums are derivable. Let us formalise these statments: (i) $\forall n : \mathbb{N} \cdot 1^nS1^n \in \mathcal{R}(G)$, (ii) $\forall n, m : \mathbb{N} \cdot 1^n + 1^mE1^{n+m} \in \mathcal{R}(G)$, and (iii) $foralln, m : \mathbb{N} \cdot 1^n + 1^mE1^{n+m} \in \mathcal{L}(G)$. Clearly, (a) then follows from (iii).

Proof of (a.i): We can prove this by induction on n.

Base case $n = 0$: We want to show that $1^0S1^0 \in \mathcal{R}(G)$.

By definition of $\Rightarrow\!\!>^0$
$S \Rightarrow\!\!>^0 S$
$\Leftrightarrow$ By definition of s^0, $S = 1^0S1^0$
$S \Rightarrow\!\!>^0 1^0S1^0$
$\Rightarrow$ definition of $\Rightarrow\!\!>^*$
$S \Rightarrow\!\!>^* 1^0S1^0$
$\Leftrightarrow$ definition of $\mathcal{R}(G)$
$1^0S1^0 \in \mathcal{R}(G)$

Inductive case: Assuming that $1^kS1^k \in \mathcal{R}(G)$, we want to prove that $1^{k+1}S1^{k+1} \in \mathcal{R}(G)$. From the inductive hypothesis, it follows that $S \Rightarrow\!\!>^* 1^kS1^k$ (from definition of $\mathcal{R}(G)$). Furthermore, by applying the production rule $S \rightarrow 1S1$, we can conclude that $1^kS1^k \Rightarrow\!\!> 1^{k+1}S1^{k+1}$. Hence, it follows that $S \Rightarrow\!\!>^* 1^{k+1}S1^{k+1}$, or equivalently $1^{k+1}S1^{k+1} \in \mathcal{R}(G)$.

This completes the inductive proof.

Proof of (a.ii): We can use induction once again, this time on m.

Base case $m = 0$: We would like to prove that $1^n + 1^0E1^{n+0} \in \mathcal{R}(G)$. But $1^n + 1^0E1^{n+0}$ is equal to $1^n + E1^n$. We have already proved that for any n, $1^nS1^n \in \mathcal{R}(G)$. Thus $S \Rightarrow\!\!>^* 1^nS1^n$. But by applying the production rule $S \rightarrow +E$ we know that $1^nS1^n \Rightarrow\!\!> 1^n + E1^n$, and thus $S \Rightarrow\!\!>^* 1^n + E1^n$. This is exactly what we sought out to prove: $1^n + E1^n \in \mathcal{R}(G)$.

Inductive case: Assuming that $1^n+1^kE1^{n+k} \in \mathcal{R}(G)$, we want to prove that $1^n+1^{k+1}E1^{n+k+1} \in \mathcal{R}(G)$. As before, the inductive hypothesis implies that $S \Rightarrow$ $>^* 1^n + 1^kE1^{n+k}$. But by applying the right production rule, $1^n + 1^kE1^{n+k} \Rightarrow$ $> 1^n + 1^{k+1}E1^{n+k+1}$, and thus $S \Rightarrow\!\!>^* 1^n + 1^{k+1}E1^{n+k+1}$, or equivalently $1^n + 1^{k+1}E1^{n+k+1} \in \mathcal{L}(G)$.

(a.ii) thus follows by induction.

Proof of (a.iii): This is quite easy. From (a.ii) we know that for all n and m, $S \Rightarrow\!\!>^* 1^n + 1^mE1^{n+m}$. But we also know that $1^n + 1^mE1^{n+m} \Rightarrow\!\!> 1^n + 1^m = 1^{n+m}$. Thus, $S \Rightarrow\!\!>^* 1^n+1^m = 1^{n+m}$, or $1^n+1^m = 1^{n+m} \in \mathcal{R}(G)$. Furthermore, since $1^n + 1^m = 1^{n+m} \in \Sigma^*$, $1^n + 1^m = 1^{n+m} \in \mathcal{L}(G)$.

This completes the proof of (a).

Proof of (b): (b) states that only correct sums can be generated, which is not so easy to prove. Why is it true? If we look at the grammar hard enough, we realise that the only strings we can generate from S are of the form 1^nS1^n, or $1^n + 1^mE1^{n+m}$, or $1^n + 1^m = 1^{n+m}$. Once we have made this observation, the proof follows.

Let $R = \{1^nS1^n,\ 1^n + 1^mE1^{n+m},\ 1^n + 1^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$. If we prove that $\mathcal{R}(G) \subseteq R$, then it follows that $\mathcal{L}(G) = \mathcal{R}(G) \cap \Sigma^* \subseteq R \cap \Sigma^* = \{1^n + 1^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$.

Consider $w \in \mathcal{R}(G)$. It follows that $S \Rightarrow\!\!>^* w$, or that for some n, $S \Rightarrow\!\!>^n w$. We can now prove the desired result by induction on n, the length of the derivation: for any n, $S \Rightarrow\!\!>^n w$ implies that $w \in R$.

Base case n=0: $S \Rightarrow\!\!>^0 w$ implies that w is just S. But S is 1^0S1^0 which is in

R.

Inductive case: Assuming that the statement is true for $n = k$, we want to prove it for $n = k + 1$: if $S \Rrightarrow^{k+1} w$, then $w \in R$. Since $S \Rrightarrow^{k+1} w$, we know that there exists w' such that $S \Rrightarrow^{k} w' \Rrightarrow w$. By the inductive hypothesis, we know that $w' \in R$. We now simply need to consider applying the production rules to statements in R and showing that the resulting string is still in R.

For example, if w' is of the form 1^nS1^n, we can only apply $S \to 1S1$ or $S \to +E$ resulting in w being either $1^{n+1}S1^{n+1}$ or $1^n + E1^n$, both of which are in R.

Similarly, we can show that this is true for the remaining cases.

This completes an outline of a proof of (b), which together with (a) allows us to conclude that $\mathcal{L}(G) = \{1^n{+}^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$.

1.5.2 Monkeying Around

A monkey sits in front of a bowl containing a mixture of black and white beans. Every so often, the monkey either pulls out two black beans out of the bowl, and places another white bean back into the bowl, or it simply takes two white beans out. If the bowl initially has 23 white beans and 24 black ones, what colour would the last bean left in the bowl be?

How can we solve the puzzle? One way of modelling what is going on is to give a grammar describing what the monkey is doing. Every production rule will correspond to the monkey manipulating the beans, and the string in the derivation will describe the state of the bowl.

Consider the following grammar:

$$\begin{array}{rcl} S & \to & w^{23} \diamond b^{24} \\ \diamond bb & \to & w \diamond \\ ww & \to & \varepsilon \end{array}$$

The diamond symbol is used to separate the black beans from the white ones, to make sure that when we add a white bean we add it in the right place. Obviously, this is not the only grammar which models the monkey behaviour. The following is another example which allows us to mix black with white beans:

$$\begin{array}{rcl} S & \to & w^{23}b^{24} \\ bb & \to & w \\ ww & \to & \varepsilon \\ bw & \to & wb \\ wb & \to & bw \end{array}$$

Note that the last two rules do not model the monkey behaviour, but allow for the beans to be sorted out to allow us to find two adjacent beans of the same colour to apply the other laws.

Modelling the grammar using a grammar does not solve the puzzle. However, it gives us a mathematical model in which we can verify that our proposed solution is correct. Now for the solution …

Look closely at the (first) grammar, and play around with a few derivations (or beans). Have you noticed something interesting about the number of black beans? Look at the possible rules: either we remove two black beans, or the number remains constant. Since we start with an even number of black beans, these rules guarantee that we will always have an even number of black beans left. Hence, we can never have one black bean left and thus, if we end up with one bean in the bowl, it has to be white! That sounds like a correct argument justifying our solution. However, it is nowhere near formalised into an actual proof. By using the grammar, we will prove the desired property in a more formal way. Note that even the proof I will present is far from formal. When a mathematician refers to a *formal proof* it is a proof which goes back to the basic axioms or proved theorems — no hand-waving arguments are allowed.

So what is our hypothesis? Any string which can be derived from S in grammar G (which uses the first set of production rules, and initial symbol S) has an even number of bs:

$$\forall s : (N \cup \Sigma)^* \cdot s \in \mathcal{R}(G) \Rightarrow \exists n : \mathbb{Z} \cdot \sharp_b(s) = 2n$$

Or equivalently:

$$\forall s : (N \cup \Sigma)^* \cdot S \Rrightarrow^* s \Rightarrow \exists n : \mathbb{Z} \cdot \sharp_b(s) = 2n$$

But if $S \Rrightarrow^* s$, there exists a natural number n such that $S \Rrightarrow^n s$. We can now prove the evenness property by induction on n.

Base case $n = 0$: If $S \Rrightarrow^0 s$, then, by definition of $\Rrightarrow^0$, $s = S$. But $\sharp_b(S) = 0$ which is even.

Inductive case: If we assume that after k steps we have an even number of black beans, can we prove that after yet another step we must have an even number of black beans? The inductive hypothesis is:

$$S \Rrightarrow^k s \Rightarrow \exists n : \mathbb{Z} \cdot \sharp_b(s) = 2n$$

And we require to prove that:

$$S \Rrightarrow^{k+1} s \Rightarrow \exists n : \mathbb{Z} \cdot \sharp_b(s) = 2n$$

But if $S \Rrightarrow^{k+1} s$, we can know that $S \Rrightarrow^k s' \Rrightarrow s$. Applying the inductive hypothesis, we know that $\exists n : \mathbb{N} \cdot \sharp_b(s') = 2n$.

Now consider $s' \Rrightarrow s$. We have only three possible rules we can apply:

- $S \to w^{23} \diamond b^{24}$: Using the result of exercise 9, we know that $\sharp_b(s) = 2n - \sharp_b(S) + \sharp_b(w^{23} \diamond b^{24}) = 2n + 24$. But $2n + 24 = 2(n + 12)$ which is thus also even.

- $\diamond bb \to \diamond w$: Again, $\sharp_b(s) = 2n - \sharp_b(\diamond bb) + \sharp_b(\diamond w) = 2n - 2$. But $2n - 2 = 2(n - 1)$ which is also even.

- $ww \rightarrow \varepsilon$: As before, $\sharp_b(s) = 2n - \sharp_b(ww) + \sharp_b(\varepsilon) = 2n$, which is also even.

Hence, in all cases, the inductive step can be proved, completing the proof.

1.6 Mathematical Tools

Since we will be presenting various proofs in this course, it is important to be familiar with recurring proof techniques. Needless to say, this section is far from comprehensive, and is intended to give but a taste of the major techniques used in the rest of the course.

1.6.1 Proof by Induction

Induction is one of the most important mathematical tools in computer science. The essence of induction is that: if individual steps do not change the situation, then no matter how hard we try, nothing can ever change. If by taking one step I will always remain on the same island, then no matter how many steps I take, I can never leave the island I started off from.

With natural numbers, a proof by induction proceeds by first identifying a variable on which we intend to perform induction. We then prove that the property is true of 0 (this is called the *base case* of induction). Then, we prove the equivalent of 'one step' (called the *inductive step:*) if the property holds for k, then it will hold for $k+1$. The assumption that the property holds for k is called the *inductive hypothesis.* If we manage to prove these two statements, then the property must hold for all natural numbers.

Here is an example of normal induction on natural numbers:

Example: Prove that $\sharp(s^n) = n\sharp(s)$.

We prove this by induction on n.

Base case $n = 0$*:* $\sharp(s^0)$ is equal to (by definition of s^0) $\sharp(\varepsilon)$, which is 0 by definition of the length function. But $0 = 0\sharp(s)$ which is what we wanted to prove.

Inductive case: The inductive hypothesis is that the property holds for k: $\sharp(s^k) = k\sharp(s)$.

$\quad\ \sharp(s^{k+1})$
$=$ by definition of string exponentiation
$\quad\ \sharp(ss^k)$
$=$ using the law: $\sharp(st) = \sharp(s) + \sharp(t)$ from exercise 6
$\quad\ \sharp(s) + \sharp(s^k)$
$=$ using inductive hypothesis
$\quad\ \sharp(s) + k\sharp(s)$
$=$ arithmetic
$\quad\ (k+1)\sharp(s)$

Hence the inductive step also holds, and thus the property is true.

Induction can be applied to all sorts of mathematical structures, not just natural numbers. In our case, string induction is another tool we will regularly need. The idea is that we prove the property of ε, then prove that, if the property holds for a string s, then it must also hold for a string as (a is a single symbol in the alphabet). From these , it then follows that the property holds for all strings.

Example: Prove that for any string $\sharp(s) = \sharp(s^R)$.

We can prove this by string induction on s.

Base case $s = \varepsilon$:

$$
\begin{array}{ll}
 & \sharp(\varepsilon) \\
= & \text{by definition of reverse of the empty string} \\
 & \sharp(\varepsilon^R)
\end{array}
$$

Inductive case: The inductive hypothesis is $\sharp(s) = \sharp(s^R)$. We want to prove that $\sharp(as) = \sharp((as)^R)$.

$$
\begin{array}{ll}
 & \sharp(as) \\
= & \text{by definition of length} \\
 & 1 + \sharp(s) \\
= & \text{by inductive hypothesis} \\
 & 1 + \sharp(s^R) \\
= & \text{arithmetic} \\
 & \sharp(s^R) + 1 \\
= & \text{by definition of length} \\
 & \sharp(s^R) + \sharp(a) \\
= & \text{using the law: } \sharp(st) = \sharp(s) + \sharp(t) \text{ from exercise 6} \\
 & \sharp(s^R a) \\
= & \text{definition of string reversal} \\
 & \sharp((as)^R)
\end{array}
$$

This completes the proof by induction.

Note that, strictly speaking, string induction is just an application of normal induction, and all string induction proofs can be redone using induction the length of the string. However, this is usually more complicated than by using string induction directly.

1.6.2 Proof by Construction

Proofs by construction also appear regularly in computer science. When trying to prove that an object with a certain property exists, we can sometimes come up with such an object. It is then simply a matter of proving that the properties hold to complete the proof.

Thus, if asked whether a grammar producing the language $\{1^m + 1^n = 1^{m+n}$ exists, we can reproduce the grammar given earlier and the proof given in section 1.5.1 to prove the statement.

Example: Given that there exists a grammar $G = \langle \Sigma, N, I, P \rangle$ such that

$\mathcal{L}(G) = L$, prove that there exists a grammar which produces the language $L \cup \{\varepsilon\}$.

The proof works by constructing the desired new grammar $G' = \langle \Sigma, N \cup \{I'\}, I', P \cup \{(I', \varepsilon), (I', I)\} \rangle$, where $I' \notin N$. We claim that $\mathcal{L}(G') = L \cup \{\varepsilon\}$ which we would need to prove to complete the proof.

1.6.3 Proof by Contradiction

Another frequently occurring proof technique in mathematics is proof by contradiction. Proofs by contradiction follow this pattern: If I want to prove a statement S, then I assume that S is not true, and from this assumption arrive to a contradiction. This means that there was something wrong with my assumption and S cannot be false. Hence, it has to be true.

Such proofs are particularly useful when we need to prove the non-existence of something. We start by assuming it exists, and then show that its existence leads to a contradiction.

Example: Given a grammar $G = \langle \Sigma, N, S, P \rangle$ such that $\mathcal{L}(G)$ is infinite, then for any number n, there must be a string whose derivation requires at least n steps.

We first assume that this is not true. Thus all strings can be derived by a sequence of at most $n-1$ steps. Now, every production rule can either increase or decrease the length of a string to which it is applied. Let α be the largest possible increase in length (such an α exists since we have only a finite number of production rules). In $n-1$ steps, no string can ever be longer than $1+\alpha(n-1)$. Thus all strings derivable in the language are bounded above by this length. Furthermore let the number of terminal symbols in Σ be β. Thus, we can only find 1 zero-length string, β different 1-letter strings, β^2 different 2-letter strings $\ldots \beta^m$ m-letter strings. The total number of strings of length not larger than $1+\alpha n$ is thus:

$$\sum_{i=0}^{1+\alpha n} \beta^i$$

which is finite. But we know that the language is infinite, which means that our assumption that all strings can be derived in at most $n-1$ steps leads to a contradiction. Hence, a string with a derivation length of at least n must exist.

1.6.4 The Pigeon-Hole Principle

Rather than a proof technique, the pigeon-hole principle is a tool to reason about counting. The principle can be formulated in various ways, the most common being: *Given $n+1$ objects to be put in n holes, there will be a hole with at least two objects inside.* Although this may sound self-evident, this is an extremely effective tool, which we will use throughout the course.

Example: Given a grammar $G = \langle \Sigma, N, S, P \rangle$ such that $\mathcal{L}(G)$ is infinite, then there must be a string whose derivation repeats some symbol in $\Sigma \cup N$.

Let n be the number of symbols $|\Sigma \cup N|$. From the previous example, we know that there is a string which requires a derivation of at least $n + 2$ steps. Since the empty string can only appear once in the derivation (otherwise we just skip the part in between and get a shorter derivation) and thus every other string in the derivation includes at least one symbol, there are no less than $n+1$ symbols in total. But the grammar has only n distinct symbols, and thus, by the pigeon hole principle there must be at least one which repeats.

1.7 Exercises

1. *(Easy)* Give one sentence which can be generated by the insult generator which is does not conform to the rules of the English language. Give an alternative machine which corrects the problem.

2. *(Moderate)* Modify the grammar accepting correct sums to accept sentences of the form $1^m + 1^n = 1^i + 1^j$ where $m + n = i + j$.

3. *(Easy)* Write a grammar to accept boolean expressions consisting of constants *true* and *false*, operators $\neg$ (not) and $\wedge$ (and) and bracketing. For example, a valid sentence in the language is: $\mathit{true} \wedge \neg(\mathit{false} \wedge \mathit{true})$.

4. *(Moderate-Difficult)* Write a set of production rules which, starting from a valid boolean expression would reduce it to *true* or *false*. For example, one rule would be $\neg\mathit{true} \rightarrow \mathit{false}$. Pay careful attention to bracketed expressions. What would happen if you reverse the production rules (ie replace $\alpha \rightarrow \beta$ with $\beta \rightarrow \alpha$)? Use this insight to write a grammar which generates sentences of the form $e = f$ where both e and f are boolean expressions and are equal if evaluated (ie $\mathit{true} = \neg\mathit{false}$ will be in the language but not $\mathit{true} = \mathit{false}$).

5. *(Difficult)* Write a grammar to accept correct multiplications: $\{1^m * 1^n = 1^{mn} \mid m, n \in \mathbb{N}\}$.

6. *(Easy)* Prove the following laws about strings:

$$\begin{array}{rcl} r\varepsilon & = & r \\ (st)^R & = & t^R s^R \\ \sharp(st) & = & \sharp(s) + \sharp(t) \\ \sharp_a(st) & = & \sharp_a(s) + \sharp_a(t) \end{array}$$

7. *(Easy)* Prove the following laws about languages:

$$\begin{array}{rcl} L^+ & = & L^*L \\ L^* & = & L^+ \cup \{\varepsilon\} \end{array}$$

8. *(Moderate)* Prove the following law about grammars: $x \Rrightarrow^* y \wedge y \Rrightarrow^* z$ implies that $x \Rrightarrow^* z$.

9. *(Moderate)* Prove the following law about grammars: $x \Rrightarrow y$ using the production rule $s \rightarrow t$ (x, y, s, t all being strings) implies that $\sharp_a(y) = \sharp_a(x) - \sharp_a(s) + \sharp_a(t)$.

10. *(Moderate)* Prove that there exists a grammar which produces the language $\{a^{2n} \mid n \in \mathbb{N}\}$.

Chapter 2

Regular Languages

In the introductory chapter, we have seen how we can define languages in terms of grammars using production rules. However, phrase structure grammars can be too general for certain applications. For example, it is not clear how to write a program which decides whether a given string is in the language generated by a grammar. We will start by looking at restricted grammars and then slowly allow more and more complex ones until we can discuss general phrase structure grammars.

2.1 Regular Grammars

The first class of restricted grammars are so called *regular grammars.* The idea it that we would like to be able to scan a string from left to right, deciding whether or not it is in the language in a straightforward manner.

Definition: A *regular grammar* is a phrase-structure grammar $\langle \Sigma, N, I, P \rangle$ such that every production rule is in the form $A \to aB$, $A \to a$ or $A \to \varepsilon$ (where $A, B \in N$, $a \in \Sigma$). In other words, $P \subseteq N \times (\{\varepsilon\} \cup \Sigma N \cup \Sigma)$.

Note that with this constraint, we have at most one non-terminal symbol in the string during a derivation, which simplifies things considerably. Furthermore, the non-terminal always appears at the end of the string.

We say that a regular grammar G is *non-deterministic* if there are two rules of the form $A \to aB$ and $A \to aC$ (for the same A and a) or two rules $A \to a$ and $A \to aB$ (for the same a and A). A regular grammar which is not non-deterministic, is obviously called *deterministic.*

Given a string $s \in \Sigma^*$ and deterministic grammar G, it is straightforward to check whether $s \in \mathcal{L}(G)$. The idea is simply to scan string s left to right, one symbol at a time, and choose the corresponding production depending on the current non-terminal and next symbol in s. Note that thanks to determinism, we never have more than one choice we can make. In fact, it is not much more complicated to generalise this algorithm to work for non-deterministic grammars. However, we will see that there is a much more straightforward way of answering the membership question in the next section.

Definition: A language L is said to be regular, if there exists a regular grammar G which generates L: $L = \mathcal{L}(G)$.

We have thus characterised a class of languages based on the grammar that can generate it. We will later show that not all languages are regular.

Definition: A phrase structure grammar $G = \langle \Sigma, N, I, P \rangle$ is called an *extended regular grammar* if all production rules are of the form $N \times (N \cup \Sigma \cup \Sigma N \cup \{\varepsilon\})$.

In other words, we have also permitted rules of the form $A \to B$. In fact, every language described by an extended regular grammar can also be generated by a standard regular grammar (see next theorem). However, we will be able to construct certain grammars more easily using these extra rules.

Theorem: Extended regular grammars are just as expressive as regular grammars.

Proof: We would like to prove that if $G = \langle \Sigma, N, I, P \rangle$ is an extended regular grammar, then there exists a regular grammar $G' = \langle \Sigma, N', I', P' \rangle$ such that $\mathcal{L}(G) = \mathcal{L}(G')$.

Given a set of extended regular grammar production rules P, we will define $\overline{P}$ to be the 'good' productions $P \backslash \{A \to B \mid A \to B \in P\}$.

Furthermore, given a non-terminal A, we define $\overline{\varepsilon}(A)$ to be the non-terminals which can be generated from A (with no other symbols): $\{B \in N \mid A \Rrightarrow^* B\}$. This is computable (see exercise 7).

We can now define G' as follows:

$$\begin{aligned} N' &\overset{\text{df}}{=} N \\ I' &\overset{\text{df}}{=} I \\ P' &\overset{\text{df}}{=} \{A \to \alpha \mid \exists B \in \overline{\varepsilon}(A) \cdot B \to \alpha \in \overline{P}\} \end{aligned}$$

In other words, we throw away all 'bad' transitions, but if $A \Rrightarrow^* B$ and $B \to \alpha$, we add $A \to \alpha$.

The proof that this construction works would follow from the fact that every new transition can be simulated in the old system, and vice-versa.

□

2.2 Finite State Automata

In the first chapter, we also used a simple type of automaton in the shape of a diagram to describe languages. We can rather easily formalise these diagrams:

Definition: A *finite state automaton* is a tuple $M = \langle \Sigma, Q, q_0, F, T \rangle$, where Σ is a finite set of symbols (the alphabet of the automaton), Q is a finite set of states (the 'names' of the circles in the diagram), q_0 is the state (or circle) with an incoming arrow (the initial or start state) $q_0 \in Q$, F is the set of double circled states (the final states) $F \subseteq Q$ and T is the transition relation of the automaton, telling us where to go from a given state, with a given symbol $T \subseteq Q \times \Sigma \times Q$.

As in the case of grammars, we say that a finite state automaton is non-deterministic if, by reading a symbol a at some state q, we have more than one transition we can follow: $\exists q, q', q'' \in Q, a \in \Sigma \cdot (q, a, q') \in T \wedge (q, a, q'') \in T \wedge q' \neq q''$.

We can now define the 'game' we used to determine whether or not a string is in the language recognised by the diagram mathematically.

We say that we can go from state q to state q' accepting symbol a, if (q, a, q') is in the transition relation. We write this as $q \overset{a}{\Rrightarrow}_1 q' \overset{\text{df}}{=} (q, a, q') \in T$.

We can generalise this to work for longer derivations, over any strings:

$$\begin{aligned} q \overset{\varepsilon}{\Rrightarrow} q' &\overset{\text{df}}{=} q = q' \\ q \overset{as}{\Rrightarrow} q' &\overset{\text{df}}{=} \exists q'' \in Q \cdot q \overset{a}{\Rrightarrow}_1 q'' \wedge q'' \overset{s}{\Rrightarrow} q' \end{aligned}$$

Thus, in a finite state automaton M, we write that $q \overset{s}{\Rrightarrow} q'$ if there is a path from q to q' following string s. Note that there be more than one path due to non-determinism, and thus $q \overset{s}{\Rrightarrow} q'$ does not mean that there may not also be some other state q'' such that $q \overset{s}{\Rrightarrow} q''$.

Using this relation, we can now define what the language generated by an automaton is.

Definition: The language generated by automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$ (written $\mathcal{L}(M)$) is a subset of Σ^*. A string is in this language if it can take us from the start state q_0 to some final state: $\mathcal{L}(M) \overset{\text{df}}{=} \{s \in \Sigma^* \mid \exists q_F \in F \cdot q_0 \overset{s}{\Rrightarrow} q_F\}$

As in the case of regular grammars, it is rather straightforward to decide whether a string is accepted (is in the language generated) by a deterministic finite state automaton.

```
state := q0;
while not(end-of-string(s)) and (state != NULL) do
   a := get-next-symbol(s);
   state := next-state(state, a);
done
accepted := state != NULL and element-of(state, final);
```

With non-deterministic automata, it is slightly more complex, but the following theorem gives us a way of testing whether a string is accepted by a non-deterministic automaton using the algorithm for deterministic automata.

Theorem: For every non-deterministic automaton M, there exists a deterministic automaton M' such that $\mathcal{L}(M) = \mathcal{L}(M')$.

In other words, we are showing that non-deterministic automata are not more expressive than deterministic ones.

Construction: The proof of this theorem is a constructive one. Given $M = \langle \Sigma, Q, s_0, F, T \rangle$, we show how to construct deterministic $M' = \langle \Sigma', Q', s'_0, F', T' \rangle$ such that both accept the same language. In fact, the proof of equivalence is tedious and not very illuminating, and we will be leaving it out. Consult a textbook if you are interested in seeing how the proof would proceed.

Before anything else, note that the new automaton will accept strings over the same alphabet, and thus $\Sigma' \stackrel{\text{df}}{=} \Sigma$.

The idea behind the construction is the following: In a non-deterministic automaton, we may have a choice of which state to go to. We will encode such a choice by enriching the states to be able to say 'I can be in any one of these states'. Q', the states of M' will be 2^Q, where a set of states $\{q_1, \ldots q_n\}$ means that if I were playing the game on the original non-deterministic automaton, I could have chosen a path which leads to any of the states q_1 to q_n.

The initial state q'_0 is easy to construct: I can only start in state q_0, represented in the new automaton by $\{q_0\}$.

What about the final states F'? In non-deterministic M, we accept a string s, if there exists at least one path from the initial state to a final state. Thus, given a state $q' \in Q'$, we will be able to stop if there is a final state of Q in q' (remember that q' is a state of M' and is thus a subset of Q): $F' \stackrel{\text{df}}{=} \{q' \in Q' \mid q' \cap F \neq \emptyset\}$.

Finally, we come to the transition relation T'. If we lie in a state $q' = \{q_1, \ldots q_n\}$ in Q', it means that we may be in any state q_i. Following a symbol a we will be able to go to any new state q'_i such that $(q_i, a, q'_i) \in T$. We will thus have all transitions of the form $(q', a, \{q_2 \mid \exists q_1 \in q' \cdot (q_1, a, q_2) \in T\})$:

$$T' \stackrel{\text{df}}{=} \{(q', a, \{q_2 \mid \exists q_1 \in q' \cdot (q_1, a, q_2) \in T\}) \mid q' \in Q',\ a \in \Sigma\}$$

Note that this language is deterministic, since for every q', a pair, we generate exactly one successor.

□

Since the proof is constructive (we have not only shown that a deterministic automaton exists, but we have actually given instructions on how to construct it), we can now test membership of a string in the language accepted by a finite state automaton by first constructing an equivalent deterministic automaton and then applying the algorithm.

Note that, if n is the number of states in an automaton M, the number of states after making M deterministic is 2^n (the size of the power set of the original states). A 10 state automaton would end up with more than 1000 states while a 20 state automaton would end up with over a 1,000,000 states! Furthermore, it has been shown that we cannot avoid this exponential explosion. It is thus not very efficient to determinise automata, since the size of the resources required increases dramatically.

But how does this apply to regular languages? The following theorem states that regular grammars and finite state machines are equally expressive. In other words, for every regular grammar there is a finite state automaton which recognises the same language and vice versa. The proof is also constructive, allowing us to construct an equivalent finite state automaton from any regular grammar, and then apply the algorithm we have written to determine whether a given string lies in the language generated by the grammar or not.

Lemma: For every finite state automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$, there exists a regular grammar $G = \langle \Sigma, N, I, P \rangle$, such that $\mathcal{L}(M) = \mathcal{L}(G)$

Construction: The idea is to use the states of the automaton to remember the current non-terminal in the derivation string. The set of non-terminal symbols will just be the states of the automaton: $N \overset{\text{df}}{=} Q$. The start symbol, is now just the initial state: $I \overset{\text{df}}{=} q_0$.

The production rules are the most complex part. For every transition in the automaton (q, a, q'), we will introduce the production rule $q \rightarrow aq'$. What about termination? For every state in $q \in F$, we should add a transition to stop the derivation $q \rightarrow \varepsilon$:

$$P \overset{\text{df}}{=} \{q \rightarrow aq' \mid (q, a, q') \in T\} \cup \{q \rightarrow \varepsilon \mid q \in F\}$$

□

Lemma: For every regular grammar $G = \langle \Sigma, N, I, P \rangle$, there exists a finite state automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$ such that $\mathcal{L}(G) = \mathcal{L}(M)$.

Construction: This is more complicated that the opposite conversion. Clearly, we can use the same trick of taking the non-terminal symbols to be the states of automaton M, but we will add a new state τ, the use of which will be explained in a minute: $Q \overset{\text{df}}{=} N \cup \{\tau\}$. The initial state is just the start symbol $q_0 \overset{\text{df}}{=} I$.

Now consider production rules of the form $A \rightarrow aB$. These are easy to translate into (A, a, B). What about termination transitions of the form $A \rightarrow a$? State τ now comes into play. We will take τ to be a final state, and add the transition (A, a, τ). Finally, this leaves just epsilon transitions: $A \rightarrow \varepsilon$. In this case, we just add A to the set of final states.

$$\begin{aligned} F &\overset{\text{df}}{=} \{\tau\} \cup \{q \in N \mid q \rightarrow \varepsilon \in P\} \\ T &\overset{\text{df}}{=} \{(q, a, \tau) \mid q \rightarrow a \in P\} \cup \{(q, a, q') \mid q \rightarrow aq' \in P\} \end{aligned}$$

□

Theorem: Finite state automata and regular grammars are equally expressive.

Proof: This follows immediately from the previous two lemmata.

□

2.3 Closure

An important mathematical notion is that of *closure.* A set S is said to be *closed under a mathematical operation op,* if by applying *op* to any element of S always yields an element of S: $\forall x \in S \cdot op(x) \in S$.

Obviously this can be generalised for operators taking more than one parameter in the obvious way. For example, in the binary operator case, we say that S is closed under $\odot$ if $\forall x, y \in S \cdot x \odot y \in S$.

What is so important about closure, you may ask. First of all, we might want closure to ensure that the operator is well defined. Secondly note that S would usually be a set we would like to study in detail. Be it the set of natural numbers, the primes, planar graphs or regular languages, we have usually proved various properties of S. By proving the closure of S under a set of operators, we have an 'invariance' property: no matter how many times we apply the operators, we always end up with an object in S. Thus the result will always satisfy the properties we have proved of the elements of S.

Regular languages have various applications, including parsing, formal verification and hardware design to mention but a few. A number of operators recur in these and other applications of regular languages: taking the union of two languages, the catenation of two languages, iterating a language and, for certain applications, taking the intersection of two languages and the complement of a language. We will prove that the class of regular languages is well behaved under these applications, in the sense that if we apply the operators to regular languages, we always end up with a result which is itself a regular language. Furthermore, we give constructive proofs, which tell us exactly how to calculate the resulting language. This means that, for instance, when we use regular languages to describe temporal properties of systems, the union of two properties is itself a temporal property. Furthermore, thanks to the constructive proof, if we have an algorithm which allows us to verify a property specified as a regular language, we automatically know how to verify a property corresponding to the union of two such others using the same algorithm.

2.3.1 Language Union

Theorem: Regular languages are closed under language union.

Construction: Let L_1 and L_2 be regular languages. We would like to show that $L_1 \cup L_2$ is also regular. Since L_1 is a regular language, there exists a regular grammar $G_1 = \langle \Sigma_1, N_1, I_1, T_1 \rangle$ which produces L_1. Similarly, let $G_2 = \langle \Sigma_2, N_2, I_2, T_2 \rangle$ be a regular grammar producing L_2. We would now like to construct a regular grammar $G = \langle \Sigma, N, I, T \rangle$ such that $L_1 \cup L_2 = \mathcal{L}(G)$.

The alphabet Σ is easy to construct. It is simply the union of the two base alphabets: $\Sigma_1 \cup \Sigma_2$. We now assume that N_1 and N_2 are disjoint. If not, we can always rename the non-terminal symbols of one of them to make sure that they are disjoint.

The trick is now to introduce a new start symbol I which can emulate either I_1 or I_2. This can be easily done using an extended regular grammar.

$$\begin{array}{rcl}
\Sigma & \overset{\text{df}}{=} & \Sigma_1 \cup \Sigma_2 \\
N & \overset{\text{df}}{=} & N_1 \cup N_2 \cup \{I\} \\
P & \overset{\text{df}}{=} & P_1 \cup P_2 \cup \{I \rightarrow I_1,\ I \rightarrow I_2\}
\end{array}$$

Since we have previously shown that for every extended regular grammar, there exists a regular grammar with the same language, this construction is sufficient.

To prove that the new grammar produces exactly the union of the original two grammars, we would have two cases (i) take a string in the language of the new grammar. The derivation of the string must start with one of the two new production rules, and then follow rules only from the production rules of one grammar (this would be proved by induction on the length of the derivation) and is thus in the language of one of the grammars; (ii) without loss of generality, take a string in L_1. This means that there exists a derivation of the string in the first grammar. But we know that $I \Rrightarrow I_1$ and hence there exists a derivation of the string in the new grammar.

□

One of the important things to note is that the size of the resulting grammar is just the sum of the sizes of the grammars we started off from (plus a constant), which means that the algorithm is linear, and thus not expensive.

2.3.2 Language Complement

Theorem: Regular languages are closed under language complement.

Construction: Let L be a regular language. We would like to show that the language complement of L^c ($\Sigma^* \backslash L$) is also regular. Since L is a regular language, there exists a regular grammar which produces L. But, for every regular grammar, there exists a finite state automaton which produces the same language. Furthermore, we can also construct a deterministic automaton recognising the same language. Let $M = \langle \Sigma, Q, q_0, F, T \rangle$ be such an automaton.

Note that since M is deterministic, for every state and input pair (q, a), there is at most one q' such that $(q, a, q') \in T$. We will start by augmenting M to make it *total* — for every state and input pair (q, a), there will always exist exactly one q' such that $(q, a, q') \in T$. The trick is to add a dummy state Δ and a transition (q, a, Δ) if there was no q' such that $(q, a, q') \in T$. Furthermore, once we fall inside Δ, we can never escape, guaranteed by adding (Δ, a, Δ) for every a to T. Let $M' = \langle \Sigma, Q', q_0, F, T' \rangle$ be the resulting automaton.

Note that M' is still deterministic, and accepts the same language as M. If we run the membership algorithm we gave on M', we can never fail to match a state and input in the transition relation. In other words, for any string

s, we can follow it all the way through the automaton without getting stuck. Furthermore, $\mathcal{L}(M) = \mathcal{L}(M')$.

Now consider the automaton M^c, exactly like M', except that we now take $Q' \setminus F$ to be the set of final states. If a string s is accepted by M^c, it means that s ends up in a final state of M^c, which is not a final state of M'. Hence M' would not accept s. Conversely, if s is not accepted by M^c, s would lead us to a non-final state of M^c (since M^c, like M' is total), which was a final state of M'. Hence, M^c accepts the complement of M' which, in turn accepts L.

□

What about the complexity? It is sufficient to not that we have to produce a deterministic finite state automaton. Recall that determinising a finite state automaton can be exponential. Hence, this algorithm is not very useful in practice. In fact, it is known that complementation of regular languages is a hard problem, and that we cannot do any better than this (modulo a constant factor, of course).

2.3.3 Language Intersection

Theorem: Regular languages are closed under language intersection.

Proof: Given two regular languages L_1 and L_2, we know that $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$. But we know that the complement of of regular language is itself regular, as is the union of two regular languages. Therefore, $(L_1^c \cup L_2^c)^c$ is a regular language, implying that $L_1 \cap L_2$ is regular.

□

Since we use language complementation, the size of the resulting automaton is at least exponential with respect to the size of the original (double exponential, in fact).

2.3.4 Language Catenation

Theorem: Regular languages are closed under language catenation.

Construction: Let L_1 and L_2 be regular languages. We would like to show that $L_1 L_2$ is also regular. Since L_1 is a regular language, there exists a regular grammar $G_1 = \langle \Sigma_1, N_1, I_1, T_1 \rangle$ which produces L_1. Similarly, let $G_2 = \langle \Sigma_2, N_2, I_2, T_2 \rangle$ be a regular grammar producing L_2. We would now like to construct a regular grammar $G = \langle \Sigma, N, I, T \rangle$ such that $L_1 L_2 = \mathcal{L}(G)$. This is what we require to be able to conclude that the catenation of the two regular languages is itself a regular language.

As before, let us assume that N_1 and N_2 are disjoint. What we want to do is to start G_2 whenever G_1 'terminates'. When does G_1 'terminate'? It can do so with a production rule of the form $A \rightarrow a$ or one of the form $A \rightarrow \varepsilon$. In the first case, we just replace $A \rightarrow a$ with $A \rightarrow a I_2$. For the second case, we use an extended regular grammar rule: $A \rightarrow I_2$.

$$\begin{array}{rcl}\Sigma & \stackrel{\text{df}}{=} & \Sigma_1 \cup \Sigma_2 \\ I & \stackrel{\text{df}}{=} & I_1 \\ N & \stackrel{\text{df}}{=} & N_1 \cup N_2 \\ P & \stackrel{\text{df}}{=} & P_2 \cup \{A \to I_2 \mid A \to \varepsilon \in P_1\} \cup \{A \to aI_2 \mid A \to a \in P_1\} \cup \{A \to aB \mid A \to aB \in P_1\}\end{array}$$

□

Note that the number of resulting non-terminals is just the sum of the non-terminals in the original languages. Similarly, the number of resulting production rules is the sum of the production rules in the original two languages.

2.3.5 Language Iteration

Theorem: Regular languages are closed under language iteration (L^+ and L^*).

Construction: Let L be a regular language. By definition, there exists a regular grammar G such that $\mathcal{L}(G) = L$. For L^+, we use a trick similar to the one we used in language catenation — for every 'termination' rule, we add rules to 'restart' the grammar:

$$\begin{array}{rcl}\Sigma' & \stackrel{\text{df}}{=} & \Sigma \\ I' & \stackrel{\text{df}}{=} & I \\ N' & \stackrel{\text{df}}{=} & N \\ P' & \stackrel{\text{df}}{=} & P \cup \{A \to I \mid A \to \varepsilon \in P\} \cup \{A \to aI \mid A \to a \in P\}\end{array}$$

To construct a grammar recognising L^*, we use the law: $L^* = L^+ \cup \{\varepsilon\}$. Let G' be the grammar recognising L^+. We can define G'' recognising L^* as follows:

$$\begin{array}{rcl}\Sigma'' & \stackrel{\text{df}}{=} & \Sigma' \\ N'' & \stackrel{\text{df}}{=} & N' \cup \{I''\} \\ P'' & \stackrel{\text{df}}{=} & P \cup \{I'' \to \varepsilon,\ I'' \to I'\}\end{array}$$

□

To recognise L^+ we are, at most, doubling the number of production rules, with another extra non-terminal, and two production rules to recognise L^*.

2.4 To Be Or Not To Be a Regular Language

If we start off with regular languages all the standard operators produce just other regular languages! So, this leads to a natural question: are there languages which are not regular? To answer this question we need to look closer at regular languages and their recognisers. Where does the memory of a finite state automaton lie? It can be found in the state we are currently in. There are no other memory locations the automaton may use to store information. Thus, given an automaton with n states, the automaton can only differentiate between n different situations. Are there any languages which require us to differentiate between an unbounded number of situations? Such languages would be impossible to represent as finite state automata.

2.4.1 The Pumping Lemma

Consider a finite state automaton M accepting a language L. Consider the acceptance path of s in M, s being a *very* long string in L. Since s is very long, we will pass through many states, and since the states are finite we will eventually have to repeat the same state:

$$q_0 \underbrace{\overbrace{\overset{a_1}{\Rightarrow} q_2 \ldots \overset{a_m}{\Rightarrow}}^{s_1} q \overbrace{\overset{a_{m+1}}{\Rightarrow} \ldots \overset{a_l}{\Rightarrow}}^{s_2} q \overbrace{\overset{a_{l+1}}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow}}^{s_3}}_{s} q_n$$

Thus, s could be split into three parts $s = s_1 s_2 s_3$. s_1 can take us from q_0 to a state q, s_2 can take us from q back to itself, and finally s_3 can take us from q to a final state q_n. We can thus repeat s_2 as many times as we want in the middle, and $s_1 s_2^{300} s_3$, $s_1 s_2^7 s_3$ and $s_1 s_2^{438} s_3$, must all be in L.

This means that the finite number of states in an automaton implies a certain repetition (regularity) in the languages we can produce. We will then use this result to prove that certain not-so-repetitive languages cannot be regular.

Lemma: Given an n state finite state automaton M, then for any string $s \in \mathcal{L}(M)$ such that $\sharp(s) \geq n$, there exist s_1, s_2, s_3 such that $s = s_1 s_2 s_3$, $\sharp(s_1 s_2) \leq n$, $\sharp(s_2) \geq 1$ and $\forall k \in \mathbb{N} \cdot s_1 s_2^k s_3 \in \mathcal{L}(M)$.

Proof: Let $s \in \mathcal{L}(M)$, such that $\sharp(s) \geq n$. Since M recognises L, there is a path in M, starting from q_0 and ending in a final state taking as many steps as there are symbols in s to accept:

$$q_0 \overbrace{\overset{a_1}{\Rightarrow} q_2 \ldots \overset{a_m}{\Rightarrow}}^{s} q_m$$

Note that in this path, there are $\sharp(s) + 1$ states. But $\sharp(s) \geq n$ means that we have at least $n+1$ states. But using the pigeon-hole principle, we have n distinct states, and a chain of at least $n+1$ states, means that at the very latest in the first $n+1$ steps, a state must be repeated:

$$q_0 \overbrace{\overset{a_1}{\Rightarrow} q_2 \ldots \overset{a_k}{\Rightarrow}}^{s_1} q \overbrace{\overset{a_{k+1}}{\Rightarrow} \ldots \overset{a_l}{\Rightarrow}}^{s_2} q \overbrace{\overset{a_{l+1}}{\Rightarrow} \ldots \overset{a_m}{\Rightarrow}}^{s_3} q_m$$

This means that we can divide s into three parts s_1, s_2, s_3 such that $s = s_1 s_2 s_3$, where $\sharp(s_1 s_2) \leq n$ (we repeat no longer than after the first $n+1$ steps) and $\sharp(s_2) \geq 1$ (because the state is repeated with at least one symbol in between). Furthermore, $q_0 \overset{s_1}{\Rightarrow} q$, $q \overset{s_2}{\Rightarrow} q$ and $q \overset{s_3}{\Rightarrow} q_m$ where $q_m \in F$.

We now need to prove that $\forall i \in \mathbb{N} \cdot s_1 s_2^i s_3 \in L$. We will start by proving by induction that $\forall i \in \mathbb{N} \cdot q_0 \overset{s_1 s_2^i}{\Rrightarrow} q$.

Base case i=0: We want to prove that $q_0 \overset{s_1 s_2^0}{\Rrightarrow} q$. But $q_0 \overset{s_1}{\Rrightarrow} q$, and $s_1 s_2^0 = s_1$, and thus $q_0 \overset{s_1 s_2^0}{\Rrightarrow} q$.

Inductive case: Assuming that $q_0 \overset{s_1 s_2^i}{\Rrightarrow} q$, we want to prove that $q_0 \overset{s_1 s_2^{i+1}}{\Rrightarrow} q$. Since $q_0 \overset{s_1 s_2^i}{\Rrightarrow} q$ (inductive hypothesis) and $q \overset{s_2}{\Rrightarrow} q$ (see previous diagram), it follows from exercise 1 that $q_0 \overset{s_1 s_2^{i+1}}{\Rrightarrow} q$.

Hence, by induction, it follows that $\forall i \in \mathbb{N} \cdot q_0 \overset{s_1 s_2^i}{\Rrightarrow} q$. Furthermore, since $q \overset{s_3}{\Rrightarrow} q_m$, we can conclude that $\forall i \in \mathbb{N} \cdot q_0 \overset{s_1 s_2^i s_3}{\Rrightarrow} q_m$, and since $q_m \in F$, it follows that $\forall i \in \mathbb{N} \cdot s_1 s_2^i s_3 \in \mathcal{L}(M)$.

□

Theorem: *(The pumping lemma for regular languages)* For every regular language L, there exists a constant p, called the *pumping length* such that if $s \in L$ and $\sharp(s) \geq p$, then there exist s_1, s_2, s_3 such that $s = s_1 s_2 s_3$, $\sharp(s_1 s_2) \leq p$, $\sharp(s_2) \geq 1$ and $\forall n \in \mathbb{N} \cdot s_1 s_2^n s_3 \in L$.

Proof: Since L is a regular language, there exists finite state automaton M such that $\mathcal{L}(M) = L$. We take p to be the number of states in M, and the rest follows from the previous lemma.

□

2.4.2 Applications of the Pumping Lemma

This pumping lemma may seem to be out of scope. Here, we are trying to show that there are languages which are not regular, and we have started by proving a property which all regular languages must satisfy. How can we proceed to show the non-regularity of a given language? If we can show that a language does not satisfy the pumping lemma property, we know that it cannot be regular.

Example: Prove that $L \overset{\text{df}}{=} \{a^n b^n \mid n \in \mathbb{N}\}$ is not a regular language.

Let us assume that L is regular. Therefore, it must satisfy the pumping lemma for regular languages. Hence, for some p, we know that any string longer than p will start to loop internally.

Consider $s = a^p b^p \in L$. Clearly, $\sharp(s) \geq p$. Now, by the pumping lemma we know that there exist s_1, s_2 and s_3 such that $s_1 s_2 s_3 = a^p b^p$, $\sharp(s_1 s_2) \leq p$, $\sharp(s_2) \geq 1$ and $\forall n \in \mathbb{N} \cdot s_1 s_2^n s_3 \in L$.

Since $\sharp(s_1 s_2) \leq p$, we know that $s_1 s_2$ is just a repetition of as. From this, together with $\sharp(s_2) \geq 1$, it follows that s_2 is a non-empty repetition of as: $s_2 = a^i$ where $i \neq 0$. Also, $s_1 = a^j$ and $s_3 = a^{p-i-j} b^p$.

But by the pumping lemma, $s_1 s_2^2 s_3 \in L$, where $s_1 s_2^2 s_3 = a^j a^{2i} a^{p-i-j} b^p =$

$a^{p+i}b^p$. Since $i \neq 0$, we know that it is not in language L. Hence, we have reached a contradiction, and thus L cannot be regular.

□

Example: Prove that $L \overset{\text{df}}{=} \{ww \mid w \in \{a, b\}^*\}$ is not a regular language.

Assume that L is a regular language. We can apply the pumping lemma, which says that any string longer than some constant p will loop.

Consider string $s = a^p ba^p b$. Since $s \in L$ and $\sharp(s) \geq p$, we can apply the pumping lemma. Therefore there must exist a way of splitting s into three parts s_1, s_2 and s_3 such that $s_1 s_2 s_3 = a^p ba^p b$, $\sharp(s_1 s_2) \leq p$, $\sharp(s_2) \geq 1$ and $\forall n \in \mathbb{N} \cdot s_1 s_2^n s_3 \in L$.

As in the previous example, we can show that $s_1 = a^i$, $s_2 = a^j$ $(j \geq 1)$ and $s_3 = a^{p-i-j}ba^p b$. But the pumping lemma says that $s_1 s_2^2 s_3 \in L$, and $s_1 s_2^2 s_3 = a^i a^{2j} a^{p-i-j} ba^p b$ which can be simplified to $a^{p+j}ba^j b$ which is not in L since $j \geq 1$. Since this leads to a contradiction, L cannot be regular.

□

Example: Prove that $P = \{a^m \mid m \in \mathit{Primes}\}$ is not regular.

As before, assume that L is regular. Let p be the pumping length of L.

Let q be a prime number greater than $p + 1$. Now consider $a^q \in L$. Applying the pumping lemma, we know that there must exist a way of splitting a^q into three parts s_1, s_2 and s_3 such that $s_1 s_2 s_3 = a^q$, $\sharp(s_1 s_2) \leq p$, $\sharp(s_2) \geq 1$ and $\forall n \in \mathbb{N} \cdot s_1 s_2^n s_3 \in L$.

Therefore, we can take $s_1 = a^i$, $s_2 = a^j$, $s_3 = a^k$, with $i + j + k = q$ and $j \geq 1$. Furthermore, since $\sharp(s_1 s_2) \leq p$ and $\sharp(s) > p + 1$, we know that $k > 1$.

Now consider $s' = s_1 s_2^{i+k} s_3$, which the pumping lemma says should be in L. $s' = a^{i+j(i+k)+k} = a^{(i+k)(1+j)}$ But $i + k \neq 1$ (because $k > 1$), and $1 + j > 1$ (since $j > 0$). Hence, $s' = a^l$ such that l can be factorised into $(i+k)$ and $(1+j)$ and is thus not prime. $s' \notin L$ contradicts the pumping lemma, implying that L is not regular.

□

2.4.3 Play the Game

Note that all the proofs take the same form, and are played like a game. We start by assuming that we have a regular language.

The pumping lemma says that a pumping length p must exist. Since I have no control over the length, we assume that it is given to us by the 'other player'.

It is now our turn to construct a string s of length not shorter than p.

Again, it is the opponent's turn to split the string into three parts with certain restrictions on the length of the pieces.

Finally, it is up to us to choose a power to raise the second part of the string such that no matter how the opponent chopped the string, the result cannot be in the original language.

Also note that the pumping lemma can only be used to show that a language is not regular. All it says is that regular languages have a certain property. If I were to tell you that 'All mice are intelligent', and provide you with an example of something intelligent, you cannot conclude it is a mouse (it *may* be a mouse, but not necessarily — Einstein was intelligent, but was not a mouse). On the other hand, if I show you a stupid cow, the fact that it is not intelligent allows you to conclude that it is not a mouse. The pumping lemma follows this reasoning analogously.

2.5 Computability

For which interesting questions about regular languages can we write a program which gives us the answer automatically? We have already written a program which can tell us whether a given string is a member of a regular language (represented as an finite state automaton or regular grammar). We will now look at other questions about regular languages we might want answered.

2.5.1 Is the Language Empty?

Theorem: Given a finite state automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$, $\mathcal{L}(M) = \emptyset$ if and only if $\mathcal{L}(M)$ contains no string shorter than n, the number of states in Q.

Proof: Clearly, if $\mathcal{L}(M) = \emptyset$, then $\mathcal{L}(M)$ contains no string shorter than n. It is thus sufficient to prove that if $\mathcal{L}(M)$ contains no string shorter than n, then $\mathcal{L}(M)$ is empty.

The proof will follow by contradiction. Assume that $\mathcal{L}(M) \neq \emptyset$, and let $s \in \mathcal{L}(M)$ be a shortest string in the set. Since $\mathcal{L}(M)$ contains no string shorter than n, then $\sharp(s) \geq n$.

Consider the derivation of $s = a_1 a_2 \ldots a_m$: $\underbrace{q_0 \overset{a_1}{\Rrightarrow} q_1 \overset{a_2}{\Rrightarrow} \ldots \overset{a_n}{\Rrightarrow} q_n}_{n+1 \text{ states}} \overset{a_{n+1}}{\Rrightarrow} \ldots q_m$

Since the number of distinct states is n, using the pigeon-hole principle, we know that some state must repeat. Therefore, we can divide s into three parts $s = s_1 s_2 s_3$ ($s_2 \neq \varepsilon$) such that for some $q \in Q$ and $q_m \in F$:

$$q_0 \overset{s_1}{\Rrightarrow} q \overset{s_2}{\Rrightarrow} q \overset{s_3}{\Rrightarrow} q_m$$

But this means that $q_0 \overset{s_1 s_3}{\Rrightarrow} q_m$, and thus $s_1 s_2 \in \mathcal{L}(M)$. Furthermore, since $s_2 \neq \varepsilon$, $\sharp(s_1 s_2) < \sharp(s_1 s_2 s_3)$, which contradicts our assumption that s was the shortest string in $\mathcal{L}(M)$. Hence, our assumption that $\sharp(s) \geq n$ must be false, implying that there must be an s such that $\sharp(s) < n$.

Therefore, $\mathcal{L}(M) \neq \emptyset$ implies that there exists a string $s \in \mathcal{L}(M)$ such that $\sharp(s) < n$, or equivalently, if $\mathcal{L}(M)$ contains no string shorter than n, then $\mathcal{L}(M)$ is empty.

□

Since the alphabet is finite, we can enumerate the set of strings of length 0 to $n-1$, and for each string we can check whether or not the string lies in the language. Hence, regular language emptiness is computable.

2.5.2 Is the Language Infinite?

Theorem: Given a finite state automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$, $\mathcal{L}(M)$ is finite if and only if the $\mathcal{L}(M)$ contains no string s such that $n \leq \sharp(s) < 2n$ where n is the number of states in Q.

Proof: The proof is split into two parts: (i) if $\mathcal{L}(M)$ contains a string of length at least n, then $\mathcal{L}(M)$ must be infinite; (ii) if L is infinite, then there must exist a string s such that $n \leq \sharp(s) < 2n$. Clearly the 'only if' direction follows from the contrapositive of (i), while the 'if' direction is just the contrapositive of (ii).

Proof of (i): Using the pumping lemma, we have a regular language $\mathcal{L}(M)$, and a string s such that $\sharp(s) \geq n$. By the pumping lemma, we can split s into three $s = s_1 s_2 s_3$ such that for all i, $s_1 s_2^i s_3$ is in $\mathcal{L}(M)$. Since (also by the pumping lemma) $s_2 \neq \varepsilon$, all these strings are distinct, and hence this guarantees that $\mathcal{L}(M)$ is infinite.

Proof of (ii): Assume $\mathcal{L}(M)$ is infinite. The proof will proceed by contradiction, and we will thus start by assuming that there is no string s such that $n \leq \sharp(s) < 2n$. Since the language is infinite, there must be a string of length at least n. Call s one such shortest string. From our assumption, we know that $\sharp(s) \geq 2n$.

By the pumping lemma, $s = s_1 s_2 s_3$ such that $\sharp(s2) > 0$, $\sharp(s_1 s_2) < n$ and for all i, $s_1 s_2^i s_3 \in \mathcal{L}(M)$. Consider $s_1 s_2^0 s_3$ which must thus be in $\mathcal{L}(M)$. From the facts that $\sharp(s) \geq 2n$ and $\sharp(s_1 s_2) < n$, we know that $\sharp(s_3) \geq n$ and thus so is $\sharp(s_1 s_2^0 s_3) \geq n$. But string $s_1 s_3$ is (a) shorter than s (since $\sharp(s_2) \neq 0$), (b) longer than n. This contradicts that s is a shortest string of length at least n.

Therefore, there must exist a string s such that $n \leq \sharp(s) < 2n$.

$\square$

As before, we can enumerate all strings of length at least n, but less that $2n$, and for each one, check whether or not the string lies in the language. This gives us an algorithm to check whether a regular language is infinite or not.

2.5.3 Are Two Regular Languages Equal?

Finally, we treat the question of checking whether two regular languages are equal. Clearly, if we can answer the question whether $L_1 \subseteq L_2$, we can answer the equality question.

But in set theory, we know that $L_1 \subseteq L_2 \Leftrightarrow L_2^c \cap L_1 = \emptyset$. But we know how to construct the regular grammar/automaton describing the complement and intersection of two languages. Furthermore, we have just shown how to check whether a language described by a regular grammar or automaton can be checked for emptiness. We can thus check the equality of two regular languages.

2.6 Exercises

1. *(Moderate)* Prove that if $q_1 \overset{s_1}{\Rrightarrow} q_2$ and $q_2 \overset{s_2}{\Rrightarrow} q_3$, then $q_1 \overset{s_1 s_2}{\Rrightarrow} q_3$.

2. *(Moderate)* Prove that, given a regular grammar G, all strings reachable in G are of the form Σ^* or $\Sigma^* N$.

3. *(Easy)* Doubling a string s replaces each symbol a in s by aa:

$$\begin{aligned} 2(\varepsilon) &\overset{\text{df}}{=} \varepsilon \\ 2(as) &\overset{\text{df}}{=} aa\, 2(s) \end{aligned}$$

   As usual, doubling a language can be defined in terms of this operator $2(L) \overset{\text{df}}{=} \{2(s) \mid s \in L\}$.

   Give a construction to show that regular languages are closed under doubling.

4. *(Moderate)* Give a construction to show that regular languages are closed under language reversal.

5. *(Easy)* Prove that regular languages are closed under language (set) difference. Discuss the complexity of the constructed result.

6. *(Difficult)* Given a string $s \in \Sigma^*$, and symbol $a \in \Sigma$, we define *the hiding of a in s* as follows:

$$\begin{aligned} \varepsilon \dagger a &\overset{\text{df}}{=} \varepsilon \\ (bs) \dagger a &\overset{\text{df}}{=} \begin{cases} s \dagger a & \text{if } b = a \\ b(s \dagger a) & \text{otherwise} \end{cases} \end{aligned}$$

   This can be generalised to work over whole languages: $L \dagger a \overset{\text{df}}{=} \{s \dagger a \mid s \in L\}$.

   Give a construction to show that regular languages are closed under hiding.

7. Give an algorithm to calculate $\overline{\varepsilon}(A)$ on a regular grammar. What is the space and time complexity of the algorithm?

8. *(Easy-Moderate)* Prove that the following languages are not regular:

   - $\{w \mid w \in \{a, b\}^*,\ w = w^R\}$
   - $\{a^{n^2} \mid n \in \mathbb{N}\}$
   - $\{1^n + 1^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$

9. *(Difficult)* The linear repetition of a string s repeats the symbols of s in an increasing fashion:

$$\nearrow (a_1 a_2 \ldots a_n) \quad \overset{\text{df}}{=} \quad a_1^1 a_2^2 \ldots a_n^n$$

Similarly, the linear repetition of a language L is defined in terms of this string operator:

$$\nearrow (L) \quad \overset{\text{df}}{=} \quad \{\nearrow (s) \mid s \in L\}$$

Prove that regular languages are not closed under linear repetition.

Hint: Use can use the language $L = \{(ab)^n \mid n \in \mathbb{N}\}$ and show that $(\nearrow (L))^R$ is not regular.

10. *(Easy)* Write the algorithms to check for language emptiness and language finitude, based on the solutions given in this chapter.

11. *(Easy)* Prove that all finite languages are regular.

12. *(Moderate)* An engineer decides to use a finite state automaton to generate test inputs for his program. Each symbol represents a keypress from the user. The program requests three numbers (written in base 10) from the user (each followed by the pressing of the return key).

    (a) Give a finite state automaton which generates all numbers even numbers.

    (b) Construct another automaton to generate all numbers divisible by 3. (**Hint:** In base 10, a number is divisible by 3 if and only if the sum of its digits is itself divisible by 3).

    (c) Hence or otherwise, discuss how the engineer can generate an automaton to recognise all numbers divisible by 6.

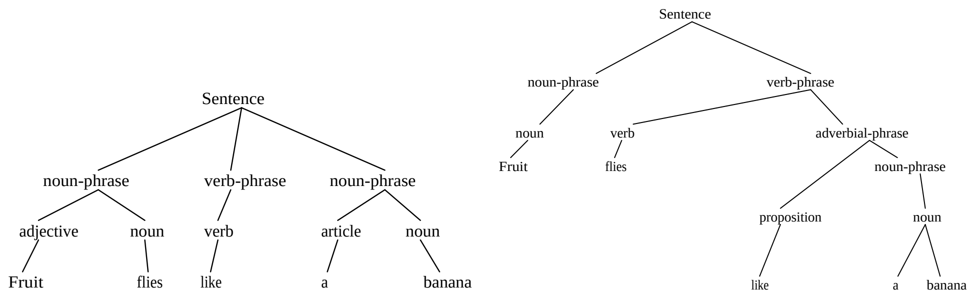
    (d) How would she then combine all these to construct the test pattern automaton which recognises a sequence of three numbers: the first being divisible by 2, the second by 3, and the third by 6?

Chapter 3

Context-Free Languages

We have seen that certain useful languages cannot be recognised by regular grammars. Consider a programming language, such as C. One very basic syntactic constraint is that brackets in mathematical expressions must match — for each opening '(', the expression must have a corresponding ')'. However, using the pumping lemma, it is easy to prove that this is impossible to recognise using a regular grammar. We thus need to identify a larger class of grammars which can be used to recognise a wider range of languages, and which we can still recognise in a reasonably efficient manner.

In natural language processing, one way of showing the analysis of a sentence is through the use of a syntax tree. Consider the sentence 'Fruit flies like a banana'. This can be interpreted in two different ways, as the following syntax, or parse trees show:



Can we produce the parse tree for any string for any kind of grammar? If we look at a parse tree, for it to be a tree, we should only be able to open up one node into a number of disjoint subnodes. Furthermore, intuitively, we can only have a 'phrase-type' (as in noun phrase, verb clause, etc) split into parts. Once we hit on actual words in the language, the tree will not be opened any further.

3.1 Context-Free Grammars

This seems to be a reasonable extension to regular grammars which only allowed us to 'generate' strings from left to right. One important thing to note is that in these grammars a 'phrase-type' will always be able to produce the same set of sentences, wherever it occurs in a derivation. In other words, these grammars ignore context — the symbols occurring around a 'phrase-type'. We will formalise this notion and call the grammars *context-free grammars.*

Definition: A phrase structure grammar $G = \langle \Sigma, N, I, P \rangle$ is said to be *context-free,* if all productions in P are of the form $A \to \alpha$, where A is a non-terminal, and α is any string over terminal and non-terminal symbols: $P \subseteq N \times (N \cup \Sigma)^*$.

Before going any further, it is worthwhile asking whether context-free grammars are, in fact, more expressive than regular grammars.

Proposition: Not all languages generated by context-free grammars are regular.

Proof: In chapter 1, we saw how $L = \{1^m + 1^n = 1^{m+n} \mid m, n \in \mathbb{N}\}$ can be generated by the grammar with the following production rules:

$$\begin{array}{rcl} S & \to & 1S1 \mid +A \\ A & \to & 1A1 \mid = \end{array}$$

where S is the initial symbol. Note that this grammar is context-free. Furthermore, in your answer to exercise 8, you have shown that L is not regular[1].

$\square$

Definition: A language L, is said to a *context-free language* if there exists a context-free grammar G such that $\mathcal{L}(G) = L$.

Note that, since every regular grammar is a context-free grammar, all regular languages are also context-free languages. However, from the proposition we have just proved, certain context-free languages are not regular.

3.2 Pushdown Automata

Since finite state automata can only recognise regular languages, we can express languages using context-free grammars for which no corresponding finite state automaton exists. We would thus like to identify a class of automata which can recognise all context-free grammars.

What is it that finite state automata lack, when it comes to recognising context-free languages? If we look at a parse tree of a string in a context-free grammar, we can imagine the derivation being performed in steps such that in each of the steps we open the leftmost non-terminal according to a production rule.

[1]There is an undischarged assumption in this proof. I am assuming that you have done the exercises.

Thus, for example, consider the following grammar:

$$\begin{array}{rcl} S & \rightarrow & AB \\ A & \rightarrow & ab \mid a \\ B & \rightarrow & b \mid bb \mid aSa \end{array}$$

The string *abb* can be recognised in different ways, resulting in different parse trees. $S \Rrightarrow AB \Rrightarrow abB \Rrightarrow abb$ and $S \Rrightarrow AB \Rrightarrow Abb \Rrightarrow abb$ illustrate two ways in which *abb* can be derived. However, note that whatever derivations of a string, there is always a sequence in which we always open up the leftmost non-terminal: $S \Rrightarrow AB \Rrightarrow abB \Rrightarrow abb$. Using this technique, we can construct an automaton which, while remembering the still unprocessed right part of the intermediate string, accepts the input from left to right. Initially the machine would start with the start symbol written in its memory. The machine would then need the capability to read the first symbol in its memory, and (i) if it finds a terminal symbol, it expects the terminal symbol from the input string and (ii) if it finds a non-terminal symbol A, it replaces it non-deterministically with a string α such that $A \rightarrow \alpha$ is a production rule of the grammar. Since the memory we need is not necessarily bounded, a finite state automaton will not do.

Definition: A *pushdown automaton* is a tuple $\langle Q, \Sigma, \Gamma, q_0, F, T \rangle$ where Q is a finite set of states, Σ is the alphabet of the automaton, Γ is the set of symbols it can use to write in its memory, q_0 is the initial state ($q_0 \in Q$), F is the set of final states ($F \subseteq Q$) and T is the transition relation of the automaton. Recall that the machine reads at most the first symbol in its memory and writes something back at the start of its memory, possibly consuming an input symbol in the process: $T \subseteq Q \times (\Gamma \cup \{\perp\}) \times (\Sigma \cup \{\perp\}) \times Q \times \Gamma^*$. $\perp$ is used for transitions which do not consume the first memory symbol or the first input symbol.

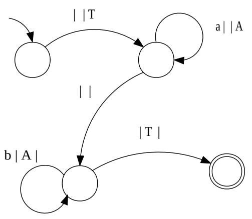
A transition $(q, x, a, q', s) \in T$ will be interpreted to mean: if I am in state q, with x at the head of my memory and a lies at the head of the input, then I will go to state q' and write s to the front of my memory. $(q, x, \perp, q', s) \in T$ is the same except that the input is left untouched and $(q, \perp, a, q', s)$ will add s to the front of the memory without consuming the first symbol remembered. These transitions will be written as $q/x \xrightarrow{a} q'/s$, $q/x \xrightarrow{\varepsilon} q'/s$ and $q/ \xrightarrow{a} q'/s$ respectively. We say that a string is accepted by a pushdown automaton, if starting from the initial state, and an empty memory, the pushdown automaton can consume all the input string ending up in one of its final states.

Example: Based on our intuitive interpretation of how pushdown automata work, the following automaton accepts the language $\{a^n b^n \mid n \in \mathbb{N}\}$:

$$\begin{array}{rcl} Q & \stackrel{\text{df}}{=} & \{q_0, q_1, q_2, q_3\} \\ \Gamma & \stackrel{\text{df}}{=} & \{A, T\} \end{array}$$

$$F \stackrel{\text{df}}{=} \{q_3\}$$
$$T \stackrel{\text{df}}{=} \{(q_0, \bot, \bot, q_1, T), (q_1, \bot, a, q_1, A), (q_1, \bot, \bot, q_2, \varepsilon), (q_2, A, b, q_2, \varepsilon), (q_2, T, b, q_3, \varepsilon)\}$$

We will depict pushdown automata graphically, representing a transition of the form $q/x \xrightarrow{a} q'/x'$ as an arrow between states q and q' labelled $a \mid x \mid x'$:



Definition: The *configuration* of a pushdown automaton $M = \langle Q, \Sigma, \Gamma, q_0, F, T\rangle$ is a pair of type $Q \times \Gamma^*$. We write $(q, xt) \overset{a}{\Rrightarrow} (q', s't)$ if $q/x \xrightarrow{a} q'/s' \in T$ (if the transition is of the form $(q, \bot, a, q', s')$, x is the empty string). Similarly, we write $(q, xt) \overset{\varepsilon}{\Rrightarrow} (q', s't)$ if $q/x \xrightarrow{\varepsilon} q'/s' \in T$ (as before, if the transition is of the form $(q, \bot, \bot, q', s')$, x is the empty string).

Iterated transitions can also be defined as usual:

$$\begin{aligned}
(q, s) \overset{t}{\Rrightarrow}^{0} (q', s') &\stackrel{\text{df}}{=} q = q' \wedge s = s' \wedge t = \varepsilon \\
(q, s) \overset{t}{\Rrightarrow}^{n+1} (q', s') &\stackrel{\text{df}}{=} \exists q'' : Q,\ s'' : \Gamma^*,\ t_1, t_2 : \Sigma^* \cdot \\
&\qquad (q, s) \overset{t_1}{\Rrightarrow} (q'', s'') \wedge (q'', s'') \overset{t_2}{\Rrightarrow}^{n} (q', t') \wedge t = t_1 t_2 \\
(q, s) \overset{t}{\Rrightarrow}^{*} (q', s') &\stackrel{\text{df}}{=} \exists n : \mathbb{N} \cdot (q, s) \overset{t}{\Rrightarrow}^{n} (q', s')
\end{aligned}$$

This allows us to define the language accepted by a pushdown automaton M:

$$\mathcal{L}(M) \stackrel{\text{df}}{=} \{s \in \Sigma^* \mid \exists t : \Gamma^*,\ q_F : F \cdot (q_0, \varepsilon) \overset{s}{\Rrightarrow} (q_F, t)\}$$

Lemma: For every context-free language L, there exists a pushdown automaton M which accepts L ($\mathcal{L}(M) = L$).

Construction: The construction we will use reflects perfectly the reasoning we used to come up with pushdown automata.

Given a context-free language L, there exists a context-free grammar $G = \langle N, \Sigma, I, P\rangle$ which produces L. We now need a pushdown automaton $M = \langle Q, \Sigma, \Gamma, q_0, F, T\rangle$ which also recognises L. M will start by pushing $I\Box$ onto the memory. $\Box$ is used so that we know when the stack has been emptied. It will pop values off the stack – if it finds a terminal symbol, it will try to match it with the input, while if it finds a non-terminal symbol, it will apply one of the production rules in P. When it finally finds $\Box$ on the stack, the automaton can terminate successfully:

$$
\begin{array}{rcl}
Q & \stackrel{\text{df}}{=} & \{\sigma_0,\ \sigma_1,\ \sigma_2\} \\
\Sigma_Q & \stackrel{\text{df}}{=} & \Sigma \\
\Gamma & \stackrel{\text{df}}{=} & N \cup \Sigma \cup \{\Box\} \\
q_0 & \stackrel{\text{df}}{=} & \sigma_0 \\
F & \stackrel{\text{df}}{=} & \{\sigma_2\} \\
T & \stackrel{\text{df}}{=} & \{\sigma_0/ \xrightarrow{\varepsilon} \sigma_1/I\Box\} \\
& \cup & \{\sigma_1/a \xrightarrow{a} \sigma_1/ \mid a \in \Sigma\} \\
& \cup & \{\sigma_1/A \xrightarrow{\varepsilon} \sigma_1/\alpha \mid A \to \alpha \in P\} \\
& \cup & \{\sigma_1/\Box \xrightarrow{\varepsilon} \sigma_2/\}
\end{array}
$$

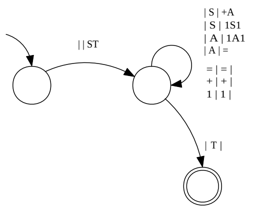
The proof of correctness is beyond the scope of this course, but it consists of two main parts (i) showing that for every string produced by a context-free grammar G, there exists a derivation of the string which always opens the leftmost non-terminal symbol and (ii) that for a derivation which always opens the leftmost non-terminal, there is a corresponding derivation in the pushdown automaton produced.

$\Box$

Example: Consider the context free grammar producing correct sums:

$$
\begin{array}{rcl}
S & \to & 1S1 \mid +A \\
A & \to & 1A1 \mid =
\end{array}
$$

The following pushdown automaton also accepts correct sums:



Lemma: The language accepted by a pushdown automaton is always context-free.

Construction: Let $M = \langle Q, \Sigma, \Gamma, q_0, F, T\rangle$ be the automaton we are given. We assume that M has the following three properties:

1. It has exactly one final state;
2. It empties the stack before terminating;
3. Transitions may add symbols to the memory, or remove a symbol from memory, but not both at the same time.

If M does not satisfy these properties, it can be modified to do so. To satisfy (1) it would suffice to add a new state q_F (which will become the only final state) and transitions of the form $q/ \xrightarrow{\varepsilon} q_F/$ for every final state q. In the case of (2), we would need to start by pushing a new symbol onto the stack $\Box$, and then replace each final state with a transition to a new state which empties the stack until it finds $\Box$, upon which it terminates. Finally, to satisfy (3) we may have to replace a transition $q/x \xrightarrow{a} q'/x_1x_2\dots x_n$ by a sequence $q/x \xrightarrow{a} q_1/$, $q_1/ \xrightarrow{\varepsilon} q_2/a_1$, $q_2/ \xrightarrow{\varepsilon} q_3/a_2 \dots q_n/ \xrightarrow{\varepsilon} q/a_n$.

Assuming that M satisfies these three properties, we would like to construct a grammar $G = \langle N, \Sigma, I, P\rangle$ which produces $\mathcal{L}(M)$.

The trick we now use is to construct a grammar with non-terminals $A_{p,q}$ for every pair of states $p, q \in Q$. Starting from $A_{p,q}$ we will be able to construct all the strings which take us from state p with an empty stack, to stack q with an empty stack. The start symbol of the grammar would thus be A_{q_0,q_F}, where q_F is the only final state of M.

Before we proceed, note that starting at a state p and with non-empty memory m, a string produced from $A_{p,q}$ will take us to q leaving m in the memory at the end.

What about the production rules in G?

- For every state q, we add the rule $A_{q,q} \rightarrow \varepsilon$;
- For every three states $p, q, r \in Q$, we add the rule $A_{p,r} \rightarrow A_{p,q}A_{q,r}$;
- For every producer-consumer pair of transitions $p/ \xrightarrow{a} q/x$ and $r/x \xrightarrow{b} s/$ we add the production $A_{p,s} \rightarrow aA_{q,r}b$.

The first two types of production rules are rather straightforward to understand. The third type is less obvious. The idea is that any derivation which changes the stack must add symbols to the stack at the beginning and remove them at the end. In between the two, the stack is left the same. Therefore, for every possible pair of transitions which act as a producer and consumer of the symbol, the machine can be left to its own devices as long as it leaves the stack unchanged.

$\Box$

Theorem: The class of languages accepted by pushdown automata is the same as the class of context-free languages.

Proof: This follows directly from the previous two lemmata.

$\Box$

3.3 Closure

As in the case of regular languages, we would like to analyse under what operators is the class of context-free languages closed.

Theorem: Context-free languages are closed under language union.

Construction: Let L_1 and L_2 be context-free languages. By definition, there exist context-free grammars $G_1 = \langle \Sigma_1, N_1, I_1, P_1 \rangle$ and $G_2 = \langle \Sigma_2, N_2, I_2, P_2 \rangle$ such that $\mathcal{L}(G_1) = L_1$ and $\mathcal{L}(G_2) = L_2$. We would now like to construct a grammar $G = \langle \Sigma, N, I, P \rangle$ such that $\mathcal{L}(G) = L_1 \cup L_2$.

The solution is identical to the one used with (extended) regular grammars. We assume that N_1 and N_2 are disjoint. We then add a new initial non-terminal I which can evolve to either I_1 or I_2:

$$\begin{aligned}
\Sigma &\stackrel{\text{df}}{=} \Sigma_1 \cup \Sigma_2 \\
N &\stackrel{\text{df}}{=} N_1 \cup N_2 \cup \{I\} \\
P &\stackrel{\text{df}}{=} P_1 \cup P_2 \cup \{I \to I_1,\ I \to I_2\}
\end{aligned}$$

Clearly, G is also a context-free language. The proof then follows practically identically to the one we would use for regular languages.

□

Theorem: Context-free languages are closed under language catenation.

Construction: As before, let L_1 and L_2 be context-free languages. Therefore, there must exist context-free grammars $G_1 = \langle \Sigma_1, N_1, I_1, P_1 \rangle$ and $G_2 = \langle \Sigma_2, N_2, I_2, P_2 \rangle$ such that $\mathcal{L}(G_1) = L_1$ and $\mathcal{L}(G_2) = L_2$. We would now like to construct a grammar $G = \langle \Sigma, N, I, P \rangle$ such that $\mathcal{L}(G) = L_1 L_2$.

The construction is rather straightforward. Assuming that N_1 and N_2 are disjoint, we add a new non-terminal I, which is taken to be the initial symbol of the new grammar. The production rules of G are simply:

$$P \stackrel{\text{df}}{=} \{I \to I_1 I_2\} \cup P_1 \cup P_2$$

Clearly, the new grammar is also context-free. To prove that the construct works, it suffices to prove that $\mathcal{R}(G) = \mathcal{R}(G_1)\mathcal{R}(G_2)$. The $\supseteq$ direction is easy to prove. The opposite direction can be proved using induction on the derivation length.

It then follows that $\mathcal{L}(G) = \mathcal{L}(G_1)\mathcal{L}(G_1)$.

□

Theorem: Context-free languages are closed under language iteration.

Construction: Assume that L is a context-free language, generated by context-free grammar $G = \langle \Sigma, N, I, P \rangle$. To recognise the language L^+ it suffices to add a new non-terminal I' (the initial symbol of the new grammar) and adding the two rules $I' \to I'I \mid I$, while to recognise L^*, we add the rules $I' \to I'I \mid \varepsilon$. Clearly, both new grammars are themselves context-free.

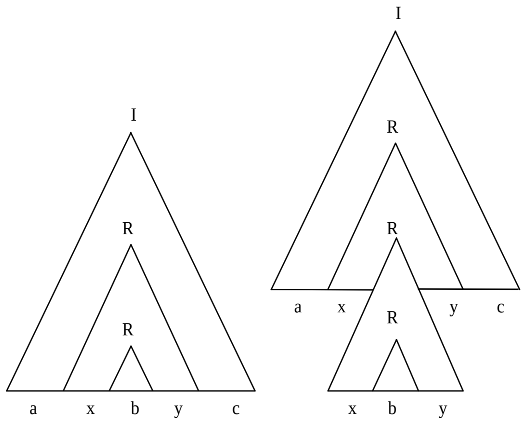
□

What about other language operators, such as language complement, intersection and language difference? It turns out that context-free languages are not closed under these operators. We will be able to prove this later in this chapter.

3.4 Pumping Lemma 2: The Revenge

Recall that the pumping lemma for regular languages depended on the fact that the finite state automata have only a finite set of configurations they can be in. This property no longer holds for pushdown automata. Can we still find a regularity that pushdown automata (or context-free grammars) must follow?

If we look at the parse tree of an string, we note that the tree must become deeper as the string chosen becomes longer. Since each node is labelled by a non-terminal symbol, we can choose a string long enough to ensure that some non-terminal repeats itself in a derivation path. Thus, a subtree would represent a derivation $A \Rrightarrow^{+} xAy$. If at least one of x and y is non-empty, we can arbitrarily repeat this derivation to repeat x and y as many times as we want:



How can we ensure that x and y are not both empty? If our grammar has production rules with at least two symbols on the right hand side, a derivation may only increase the length of the string. Therefore, $\sharp(xRy) > \sharp(R)$, implying that $\sharp(xy) > 0$.

But not all context-free grammars satisfy this property, or do they?

Lemma: For any context-free language L, there exists a context-free grammar $G = \langle \Sigma, N, I, P \rangle$ such that $\mathcal{L}(G) = L \backslash \{\varepsilon\}$ and for every production rule $A \to \alpha \in P$, either $\alpha \in \Sigma$ or $\sharp(\alpha) \geq 2$.

Construction: The first step is to get rid of epsilon productions. We can do this by removing every epsilon production $A \to \varepsilon$ and for every rule which includes an A on the right hand side, we replicate the rule with and without every occurrence of A eg $X \to aAbA$ would be transformed into the rules $X \to aAbA \mid abA \mid aAb \mid ab$. The resulting grammar can be proved to generate the same language as the original grammar apart from the empty string.

Given an epsilon-free grammar, we can then follow the same strategy as in the case of regular grammars to get rid of rules of the form $A \to B$.

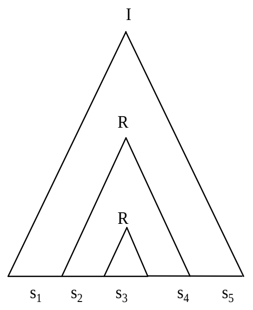
The resulting grammar now satisfies the property required in the lemma. Note that this construction is computable.

□

Theorem: *(The Pumping Lemma for context-free languages)* For every context-free language L, there exists a constant p, called the *pumping length of* L such that if $s \in L$ and $\sharp(s) \geq p$, then there exist s_1, s_2, s_3, s_4 and s_5 such that $s = s_1 s_2 s_3 s_4 s_5$, $\sharp(s_2 s_3 s_4) < p$, $\sharp(s_2 s_4) > 0$ and $\forall n \in \mathbb{N} \cdot s_1 s_2^n s_3 s_4^n s_5 \in L$.

Proof: Assume L is infinite (if it is not, we can choose p to be a number greater that the maximum length of strings in L, making the theorem vacuously true). Since L is a context-free language, we know that there exists a context-free grammar which generates the language. Furthermore, using the previous lemma, there exists a context-free grammar $G = \langle \Sigma, N, I, P \rangle$, such that all productions in P produce at least two symbols, and $\mathcal{L}(G) = L \setminus \{\varepsilon\}$. Let w be the maximum length of the right-hand side of rules in P.

Consider the parse tree of a string of length n. Since every rule produces at most w subnodes, the tree has depth at least $\log_w n$. We will choose the pumping length to be $w^{|N|+2}$: n satisfies $n \geq w^{|N|+2}$. Therefore, the depth of the tree is at least $\log_w(w^{|N|+1}) = |N| + 2$. Take the path from the root of the tree to any terminal leaf. The path is of length $|N| + 2$ and all but one (the last) of the internal nodes is a non-terminal. Using the pigeonhole principle, at least one non-terminal must be repeated along the path:



Therefore, $I \Rrightarrow^* s_1 R s_5$, $R \Rrightarrow^* s_2 R s_4$ and $R \Rrightarrow^* s_3$. From these it follows that for all i, $I \Rrightarrow^* s_1 s_2^i s_3 s_4^i s_5$. Furthermore, assume that there are no other repetitions of non-terminals along any path in subtree with the top R as root apart from these two Rs (if there are any other repetitions, we would simply take a lower subtree).

Since every production rule has at least two symbols (or a single terminal) on the right-hand side, and we have no epsilon production rules, $s_2 s_4 \neq \varepsilon$. Furthermore, how can we prove that $\sharp(s_2 s_3 s_4) < p$? Using a similar argument to one we have used earlier in the proof, if $\sharp(s_2 s_3 s_4) \geq p$, it follows that there must be another repeating instance of a non-terminal, which contradicts our assumption that there are no further repetitions of non-terminals in this subtree.

□

As in the case of regular languages, we can only use the pumping lemma to prove that a language is *not* context-free. We will illustrate this via a couple of examples.

Example: Prove that $L = \{a^i b^i c^i \mid i \in \mathbb{N}\}$ is not context-free.

Assume that L is context-free. Therefore, there exists a pumping length p such that for any string of length p or longer, we can split it into five parts $s_1 s_2 s_3 s_4 s_5$ such that $s_2 s_4 \neq \varepsilon$, $\sharp(s_2 s_3 s_4) < p$ and for all i, $s_1 s_2^i s_3 s_4^i s_5 \in L$.

Consider the string $s = a^p b^p c^p \in L$. Clearly, $\sharp(s) > p$. Therefore, we can split it into five parts, $s = s_1 s_2 s_3 s_4 s_5$, satisfying the pumping lemma properties. Therefore, $s' = s_1 s_2^2 s_3 s_4^2 s_5 \in L$. However, since $\sharp(s_2 s_3 s_4) < p$, it follows that $s_2 s_3 s_4$ can consist of at most two distinct symbols (from a, b, c). Furthermore, $s_2 s_4 \neq \varepsilon$, and therefore, we in s' we have increased the number of instances of one or two symbols in s. Therefore, the number of as, bs and cs in s' are not equal and thus cannot be in L.

This leads to a contradiction, and therefore, L cannot be context-free.

□

Example: Show that the language $Z = \{s \in \{L, R, U, D\}^* \mid \sharp_L(s) = \sharp_R(s),\ \sharp_U(s) = \sharp_D(s)\}$ is not context-free.

Assume it is, and let p be its pumping length. Consider the string $L^p U^p R^p D^p$. Clearly, the string is in Z and is longer than p. By the pumping lemma, we can split s into five, $s = s_1 s_2 s_3 s_4 s_5$ of s, with $s' = s_2 s_3 s_4$ not longer than p. Therefore, s' must contain at least one symbol, but may not contain both L and R, and neither both U and D. Therefore, $s_1 s_2^2 s_3 s_4^2 s_5$ no longer has a balance of L and R symbols or U and D symbols and is thus not in language Z contradicting the pumping lemma. Therefore, L is not context-free.

□

3.5 Back to Closure

We promised to come back to the question of whether context-free languages are closed under some operators we have not yet treated: language intersection, language complement and language difference.

Theorem: Context-free languages are not closed under language intersection.

Proof: It is sufficient to give a counter-example. Consider the languages $L_1 = \{a^i b^j c^k \mid i, j, k \in \mathbb{N},\ i = j\}$ and $L_2 = \{a^i b^j c^k \mid i, j, k \in \mathbb{N},\ j = k\}$. Both can be generated by context-free grammars. For instance, the first language can be generated by a grammar with the following production rules:

$$\begin{array}{rcl} S & \rightarrow & XC \\ C & \rightarrow & cC \mid \varepsilon \\ X & \rightarrow & aXb \mid \varepsilon \end{array}$$

Therefore, both languages are context-free. Now consider their intersection:

$$\begin{aligned} L_1 \cap L_2 &= \{a^i b^j c^k \mid i,j,k \in \mathbb{N},\ i=j,\ i=k\} \\ &= \{a^i b^i c^i \mid i \in \mathbb{N}\} \end{aligned}$$

But we have already proved that this language is not context-free. Therefore, context-free languages are not closed under intersection.

□

From this proof, it follows that context-free are not closed under language complement and set difference:

Theorem: Context-free languages are not closed under language complement and difference.

Proof: Let us assume context-free languages are closed under set complement. Therefore, given two context-free languages L_1 and L_2, it follows that $(L_1^c \cup L_2^c)^c$ is also a context-free language. But this is equal to $L_1 \cap L_2$ which would mean that context-free languages are closed under intersection. Therefore, context-free languages cannot be closed under language complement.

Now assume that context-free languages are closed under set difference. Given an alphabet Σ, the language Σ^* is clearly context-free:

$$\{S \to \varepsilon\} \cup \{S \to aS \mid a \in \Sigma\}$$

But, by definition, $L^c = \Sigma^* \setminus L$. Therefore, context-free languages would also be closed under language complement. This means that this class of languages is not closed under language difference.

□

3.6 Computability

As in the case of regular languages, we will see whether certain questions about context-free languages are computable.

3.6.1 The Membership Question

Can we design an algorithm to decide whether a string s is contained in the language generated by a context-free grammar?

Given a context-free grammar, recall that we can obtain an alternative context-free grammar, which generates the same strings as the first (modulo the empty string), but in which all production rules produce at least two symbols except possibly for rules which produce a single terminal.

At this stage, we note an interesting property of string derivations in this grammar: The derivation of a string of length n can take at most $2n-1$ steps.

Consider a derivation of a string of length n. Clearly, we can only apply at most n rules of the form $A \to a$ (because we have no rules to get rid of terminal symbols). Every other rule would increase the length of the string.

Assume the derivation is of length $2n$ or longer. Therefore, there must be at least n length increasing productions applied, resulting in a string of length, at least $n+1$ (we start off with a string of length 1, and add at least one symbol n times). But the type of rules we have ensure that a string cannot get shorter in a derivation (see exercise 5). Therefore, there is no way in which we can derive the original string.

Back to the membership question, if we are asked whether ε is in the original question, we can easily decide this (exercise 4). So assume that we are given a non-empty string s. We can search all possible derivations of length $2n-1$ or shorter — there are at most $(\sum_{i=0}^{2n-1} |P|^i)$ of them. If we generate s, then we can answer affirmatively, and if none of them results in s, we know that the string is not in the language.

We can actually do much better than this in terms of efficiency. However, we just want to prove that this problem is computable: there exists an algorithm which decides it.

3.6.2 Is the Language Empty?

Theorem: If L is a context-free language, with pumping length p, L is empty if and only if L contains no string of length less that p.

Proof: Clearly, if L is empty, it contains no strings of length less than p.

Now assume that L has no string shorter than p. Let s be a shortest string in the language. Since all strings in the language are if length at least p, we can apply the pumping lemma to s: for some decomposition $s = s_1s_2s_3s_4s_5$ such that $s_2s_4 \neq \varepsilon$, and $s' = s_1s_3s_5 \in L$. But $\sharp(s') < \sharp(s)$, contradicting our assumption that there is no string in L shorter than s.

$\square$

As in the case of regular languages, we can calculate length p, and loop through all strings up to length p applying the membership test to check this property. Note that this algorithm is very inefficient, and much more efficient checks for practical applications.

3.6.3 Is the Language Infinite?

Theorem: If L is a context-free language, with pumping length p, L is infinite if and only if L contains a string s such that $2p \geq \sharp(s) > p$.

Proof: Assume that such a string s exists. By the pumping lemma, since $\sharp(s) > p$, we can split s into five $s_1s_2s_3s_4s_5$ such that $s_1s_2^is_3s_4^is_5 \in L$ for all i. Furthermore, since $s_2s_4 \neq \varepsilon$, all such strings are distinct, and thus L is infinite. This completes the 'only if' part of the proof.

Now assume that L is infinite. We would like to prove that there exists a string s such that $2p \geq \sharp(s) > p$. Assume none exist. Since L is infinite, we can choose a string s to be a shortest one of length more then $2p$. Again, by the pumping lemma, we can split s into five $s_1s_2s_3s_4s_5$ such that $s_1s_2^is_3s_4^is_5 \in L$ for all i. Therefore, $s_1s_3s_5 \in L$. From the pumping lemma we also know that $s_2s_4 \neq \varepsilon$ and than $\sharp(s_2s_3s_4) < p$. It thus follows that $\sharp(s_1s_3s_5) < \sharp(s)$. Now, since s was chosen to be the shortest string longer than $2p$, $\sharp(s_1s_3s_5) \leq 2p$. But from the fact that $\sharp(s_2s_3s_4) < p$ and that $\sharp(s) > 2p$, we know that $\sharp(s_1s_3s_5) > p$, which contradicts the assumption that no string of length between p and $2p$.

□

3.6.4 Equality

Note that since context-free languages are not closed under language difference, we cannot check for equality of two context-free languages L_1, L_2 by checking that both $L_1 \setminus L_2$ and $L_2 \setminus L_1$ are empty (as we did for regular languages). In fact, it turns out that writing an algorithm which decides the equality of two context-free grammars is impossible. Although we will not have the time to cover this result in the rest of the course, most of the necessary tools will be covered. Most books in the bibliography give the proof which you should be able to understand at the end of this course.

3.7 Exercises

1. *(Easy)* If $I \Rrightarrow^* s_1Rs_5$, $R \Rrightarrow^* s_2Rs_4$ and $R \Rrightarrow^* s_3$ prove that the following holds: $\forall i \in \mathbb{N} \cdot I \Rrightarrow^* s_1s_2^is_3s_4^is_5$.

2. *(Easy)* Show that $\{1^n - 1^m = 1^{n-m} \mid n, m \in \mathbb{N}\}$ is a context-free language.

3. *(Easy)* Formalise the constructions used in section 3.2 to obtain a push-down automaton with only one final state, and one which empties the stack before terminating.

4. *(Easy)* Give an algorithm which, given a context-free grammar G returns whether $\varepsilon \in \mathcal{L}(G)$.

5. *(Easy)* Given a grammar such that all transitions are of the form $A \rightarrow a$, with A being a non-terminal, and a a terminal, or $A \rightarrow \alpha$ with $\sharp(\alpha) > 1$, prove that $\alpha \Rrightarrow^* \beta \Rightarrow \sharp(\alpha) \leq \sharp(\beta)$.

6. *(Moderate)* Prove that the language $\{a^p \mid p \in \mathit{Primes}\}$ is not context-free.

7. *(Moderate)* Prove that the language $\{1^m * 1^n = 1^{mn} \mid m, n \in \mathbb{N}\}$ is not a context-free language.

8. *(Moderate)* Prove that the language $\{ww \mid w \in \{a, b\}^*\}$ is not a context-free language.

9. *(Moderate/Difficult)* A *n-symbol pushdown automaton* is a normal push-down automaton, but which has only n distinct stack symbols.

(a) Show that 1-symbol pushdown automata can recognise all regular languages.

(b) Show that 1-symbol pushdown automata can also recognise certain languages which are not regular.

Hint: Try using the language $\{a^n b^m \mid n, m \in \mathbb{N},\ n \geq m\}$.

(c) Show that 2-symbol automata can recognise exactly the class of context-free languages.

10. *(Difficult)* Show that the intersection of a regular language and a context-free language is always a context-free language.

11. *(Difficult)* Do you think that context-free languages are closed under language reversal? Justify your answer.

Chapter 4

Computation: A Historical Interlude

This story can be traced back a couple of thousand of years. Greek philosophers spent their days discussing and debating in their equivalent of the Maltese village square, the *zuntier.* As anyone who has tried discussing anything with a Mediterranean knows, Mediterraneans will defend their point of view using all sorts of arguments, logical and otherwise. Validity of an argument to establish the truth of the conclusion has thus always been one of the most hotly debated topics in the region[1]. 'How can one assess the validity of an argument?' was one of the primary questions asked by Greek philosophers.

Before proceeding any further, we must see what we mean by an 'argument'. It can be divided into two parts: underlying premises or hypotheses, and a sequence of 'simple' steps which lead from the the premises to the conclusion.

Let us look more closely at these two elements. Let us start with the premises. Clearly, for an argument to be valid, the premises themselves must be valid. If I start from the premise that the moon is made of green cheese, I should be able to argue that the moon smells bad, which may not be the case if, in fact, the moon is made of curried chicken. This leads to a vicious circle. Before I start an argument I have to prove the validity of the premises. Is there a way out of this? Well, if we can build arguments starting from no premises at all, we can build correct conclusions based uniquely on the validity of the argument steps, then we can escape. In practice however, we would simply be relegating the breaking off of the cycle using the argument steps which is not so helpful. What if we start with a collection of basic truths we can agree upon. If we agree that the statements 'The moon is made of green cheese' and 'Green cheese smells bad' are basic truths, then clearly we cannot disagree whether or not the moon smells bad.

[1]Take a look at Maltese politics, where the validity of the arguments used by the other party is put to question, and never the conclusions themselves.

Let us move on to the argument steps. Simply basing an argument on correct premises is not sufficient[2]. We thus need to agree on a set of valid rules which tell us how we can form valid new statements from known truths. As in the case of the basic statements, we need to agree on these rules. One such rule that the Greeks were familiar with was the *syllogism rule:* from the premises 'All objects of type X are of type Y' and 'x is of type X', we can conclude that 'x is of type Y'. The classic example is that if we can argue for (or accept the basic validity of) 'All men are mortal' and 'Socrates is a man', then we can conclude that 'Socrates is mortal'. Obviously, this rule only tells us how to construct conclusions starting with premises of the given forms. We cannot use this rule to conclude anything from 'All men are mortal' and 'Margaret Thatcher is not a man'. If we have a sufficiently rich set of correct rules, we should be able to argue for the validity of a statement or its converse. Applying this sort of reasoning on natural language arguments can be very difficult. However, Euclid used this approach on the field of geometry. Starting with a small set of simple truths upon which everyone agreed, and rules on how truths can be combined to produce deeper truths, he wrote thirteen books proving the validity of various theorems in geometry (for example that a triangle which has two equal angles must have two equal sides).

This approach of agreeing on a number of basic truths (called postulates or axioms), and rules on how to combine truths (called *rules of deduction* or *inference rules*) was established as the basis of mathematics as we know it. As opposed to science, where truth is based on repeated observations of the same phenomenon which strengthen but never contradict a given statement, mathematicians allow truths to be built only from axioms and inference rules[3]. To continue the story, we have to skip over a couple of millennia. Over these centuries, Europe was going through the Dark Ages. A few mathematicians cautiously applied this approach to some areas of mathematics, careful that they are not accused of heresy by the Catholic church. It was only in the Arab world, that Greek learning was developed. Whereas Greek mathematics was interested in establishing truths, Islamic mathematics concentrated on calculations. A notable development, relevant to our story was the development of the notion of algorithm. Around 800AD, Abu Ja'far Muhammad ibn Musa Al-Khwarizmi living in Baghdad, wrote a book giving step-by-step instructions on how to perform arithmetic using Hindu numerals (which used zero as a digit). The spirit of giving recipes to handle formal objects caught on, and today we call such recipes

[2]A lawyer once claimed that the opposing lawyer's observations were correct, but not his conclusion. When challenged how this could be, he told a story of how a 5 year old farmer's daughter once ran to her father shouting 'Quickly daddy! Daddy! I've just been in the hay barn where I saw John the farmhand and Alice the new milkmaid. She pulled her skirt up, and he lowered his breeches. If you do not rush to stop them, they will pee all over our hay!' You see, her observations were impeccably correct, even if her conclusion probably wasn't.

[3]One may justifiably ask whether the acceptance of the axioms and rules is based on observation. Clearly this may be the case, but a mathematical theorem goes beyond such observations. It says that if I live in a universe where my axioms and rules of inference are valid, then so must be the conclusion of the theorem. The validity of the axioms in our universe is of no relevance to the mathematician.

algorithms — a corruption of Al-Khwarizmi's name.

In the meantime, in the West, philosophers started asking related questions. One pertinent question was Descartes' question of whether we live in a mechanical universe working on clockwork. Given the current state of the universe and enough time, can I use a set of rules to predict the future? Descartes' interest was primarily free will – only the illusion of free will can exist in a clockwork universe. However, looking at this question from a mathematical viewpoint gives rise to a pertinent question: is there a set of axioms and rules of inference from which we can deduce all the truth and nothing but the truth?

In the early 1800s, a mathematician Carl Friedrich Gauss came up with an interesting observation based on Euclid's work. Euclid's work started by stating a number of postulates. Of these, the fifth so called *parallel postulate* was more complex than the rest. It has been presented in a number of equivalent ways, but more or less it states that given a point and a line, there is exactly one way in which we can draw a line parallel to the given line and passing through the given point. For centuries mathematicians wondered whether they could make do without this axiom by proving it in terms of the other axioms or at least reduce it to simpler axioms[4]. Various mathematicians tried and failed, starting from the Greek scientist Ptolemy who managed to come up with a wrong proof. Others managed to prove its equivalence to other statements such as 'The sum of the inner angles of a triangle is 180°'. Gauss wondered what would happen if we replace the axiom by its converse[5]. He could not reach any contradicting observations. However, in doing so he realised something interesting: a universe in which the sum of the inner angles of a triangle do not necessarily sum up to 180° can be consistent, and he developed the necessary mathematics to reason about it[6]. In other words, there are different universes in which Euclid's rules are too rigid, and we need to replace them by alternative rules. These alternatives were studied in depth by other mathematicians, notably János Bolyai, Nikolai Ivanovich Lobatchevsky and Bernhard Riemann who developed what is today known as non-Euclidean geometry.

These questions gave rise not only to modern physics (Einstein's universe is a non-Euclidian one, curved in higher dimensions just as the surface of a sphere is two dimensional, but curved from a third dimension point of view), but also to meta-mathematics. Mathematicians started asking how they can show a set of axioms and deduction rules to be consistent or sound (only true things follow

[4]Euclid himself seems not to have liked the postulate, avoiding its use whenever possible.

[5]He kept most of these thoughts private, and most of his results were only discovered after his death.

[6]If you are wondering what universe that would look like, consider drawing figures on the surface of a sphere. A 'straight' line joining two points would be the tracing of the shortest path between the points. On a sphere lines turn out to be arcs on circles whose centres coincide with the centre of the sphere. Imagine the Earth to be the sphere, and a triangle with the north pole as one of the points. The other two points both lie on the equator, distant a quarter of the circumference of the equator from each other. Now look at the angles subtended by the lines. Each of the angles turns out to be a right angle. Help! A triangle whose inner angles sum up to 270°! An interesting corollary is that if we were to try to confirm Euclid's laws on the Earth's surface we would find them to be incorrect.

from the rules) and complete (all true things follow from the rules). In the early 1900s, Bertrand Russell proved the inconsistency of a standard axiomatisation of formal set theory. This was a big blow to mathematicians who had hailed set theory as a perfect basis in terms of which to formalise the whole of mathematics.

Russell's Paradox

Bertrand Russell proved the inconsistency of a well-accepted axiomatization of set theory using a paradox now better known as *Russell's Paradox.* This is best illustrated using a metaphoric presentation that Russell himself used. John visited a remote village, where he discovered that the local barber enjoyed a monopoly. The only barber in town was known to shave exactly all those men who do not shave themselves. When he returned home, John was asked whether or not the barber shaved himself. John reasoned that if the barber shaved himself, then he would be one of the men who shaved themselves, and thus using the rule he was told, was not shaved by the barber. Therefore, he does not shave himself. This means that he is shaved by the barber, who is himself! What is wrong? Obviously, the person who gave John the information made a nonsensical statement.

Mathematically, Russell's paradox can be expressed quite succinctly in terms of set theory. First of all notice that a set can be an element of itself. The collection of all sets containing more than three objects is itself a set. How many elements does it contain? Clearly more than three. Therefore, it is an element of itself. On the other hand, the set of all red objects is not red, and therefore not an element of itself. Now, we can define the collections of all sets which do not contain themselves:

$$P \overset{\text{df}}{=} \{S \mid S \notin S\}$$

Now, is $P \in P$? If so, then P must satisfy the constraint given in the set comprehension: $P \notin P$. This leads to contradiction, implying that $P \notin P$. But if this is the case, the set comprehension constraint is satisfied, implying that $P \in P$, also leading to a contradiction. The only remaining possible source of problems is the way set theory allows us to define sets: we can define nonsensical sets, making this version of set theory inconsistent.

Various ways out of the contradiction were proposed and shown to be consistent. However, this incident shows the dangers of using an axiomatization which is not known to be sound.

In the same period, David Hilbert, a leading mathematician, presented 23 challenges to the mathematical community which he described as amongst the most salient at the time. The main driving force behind Hilbert's programme was that, to use his own words, 'in mathematics there is no ignorabimus' — in mathematics there is no such thing as 'we do not know'. Hilbert's *Entscheidungsproblem* asked whether mathematics was decidable. In other words, is there a surefire recipe which given a mathematical statement can tell us whether the statement is true or not. Hilbert and his audience were of the opinion that mathematics was sound, complete and decidable.

In the 1930s, a mathematician Kurt Gödel published a seminal paper 'On formally undecidable propositions in *Principia Mathematica* and related systems'.

Hilbert's Tenth Problem

Hilbert's tenth problem was to design an algorithm which decides whether a polynomial equation has integral solutions. A polynomial equation is an equation in which a number of variables, where a variable can only be raised to the power of a constant. Thus, both sides of the equation can be expressed as a finite sum of products of variables (possibly raised to a constant power) and constants. $x^3 + xy^2 = 4$ is a valid polynomial equation, while $2^x = \sin y$ is not. Note that Hilbert did not bother to ask whether an algorithm exists, but to devise the algorithm.

Before proceeding any further, it is important to agree on what an algorithm is. Formally, it is a step-by-step recipe which eventually always comes up with an answer. Consider the following program working on single variable equations:

```
boolean solve(equation :: integer -> boolean)
{ Given an equation as a function which given the value of the
single variable returns whether or not it is a solution }
begin
x := 0;
forever
if(equation(x) or equation(-x)) then return(YES);
x := x+1;
end
```

If the equation has a solution, this program always eventually answers affirmatively, but it will never answer negatively. Even though the program can be easily extended to work on equations over any number of variables, this is *not* an algorithm.

Note that Hilbert's problem can be expressed as a language problem. Let E be the set of all strings describing polynomial equations. We can define a language $Z \overset{\text{df}}{=} \{p \in E \mid p \text{ has an integer solution}\}$, and Hilbert's problem can be reexpressed as devising an algorithm to decide membership in Z.

Now imagine that we can write an algorithm `halt` which given a program P tells us whether or not P terminates. We would now have a solution to Hilbert's problem, because we can test whether `solve(e)` terminates (and therefore a solution exists) or not (implying that no solution exists). A solution to Hilbert's tenth problem would thus be `halt(solve(e))`. We'll be discussing `halt` again later in this course.

The main result, later dubbed as *Gödel's incompleteness theorem,* states that no mathematical axiomatization powerful enough to allow us to reason about addition and multiplication can be both sound and complete. In other words, if we are given axioms and rules from which we can only derive true statements about sums and products, then there must be statements which are true, but not provable. If we add more axioms and rules to allow us to prove all true statements, Gödel's result guarantees that we are also able to prove some false results using these rules! This sent shock waves all over the mathematical community. Mathematicians had difficulties accepting the result, but the proof stood their scrutiny. Down went two of Hilbert's hopes: soundness and completeness.

Prove Yourself
Gödel's result was based on a twist on the age old paradox of the Cretan saying that 'All Cretans are liars.' Whichever way you look at it, this leads to a contradiction. If they always lie, then the speaker must be lying and therefore they do not always lie. But if they always tell the truth, then the speaker must be a liar. In other words, a perfectly logical Cretan can never utter such a statement.
The problem arises because the sentence refers to its own truth value. Similarly, Gödel constructed a sentence referring to its own provability. Using arithmetic, he managed to encode the statement 'This sentence is not provable.' If the statement is true, then it is not provable and hence the axiomatic basis incomplete. If, on the other hand, the statement is false, then the sentence is provable, implying that the the axiomatic basis is not sound. Either way, we cannot win.

Gödel's Incompleteness Theorem and Fermat's Last Theorem
Gödel's theorem had mathematicians worried about their work. What if open problems that mathematicians had spent their whole lives trying to prove turned out to be unprovable? A number of mathematical fiction stories were written about mathematicians and Gödel's incompleteness theorem. Unfortunately, not all these books are mathematically sound. Take Apostolos Doxiadis' 'Uncle Petros and Goldbach's Conjecture.' The story revolves around a mathematician (the uncle Petros of the title) who has spent years trying to prove Goldbach's Conjecture. He comes across Gödel's work and his career crumbles as he worries that his life's quest may be unprovable. At one point, he tries to meet Gödel to ask him whether it is possible to prove the provability or otherwise of Goldbach's Conjecture. As you will see, either uncle Petros was not worth his salt as a mathematician, or the author did not look at the mathematical statements he made in sufficient detail!
One formulation of Goldbach's Conjecture is that every even number greater than 2 is the sum of two primes. Let us assume that that the truth or falsity of this conjecture is unprovable and that we can prove that this is the case. Now, either the conjecture is true, or it isn't. If it is not true, then we can find an even number which is not the sum of two primes. Clearly, this would constitute a proof that the conjecture is false. Therefore, if we have proved it to be unprovable, we have proved it to be true, and thus proved it! In other words, either the conjecture is provable, or it is unprovable and we can never know it. How this analysis could have escaped a mathematician is beyond belief . . .

In the 1936, Alan Turing published the shattering answer to Hilbert's third and final hope. His paper, entitled 'On Computable Numbers With an Application to the Entscheidungsproblem,' said that, no, mathematics is not decidable. Turing proposed a form of automaton which could perform calculations. This form of automaton is now known as a *Turing Machine.* The Turing machine model has an interesting property: it is powerful enough to be able to design a *universal machine* which, given the description of a Turing machine as input, can simulate it. In other words, Turing machines can be made self-referential (by passing an encoding of a machine as its own input), which you may recall was the key trick used by Russell and Gödel.

To Halt or not to Halt
Turing machines can be encoded and given as input to other Turing machines. If you think that this is weird and useless, remember that every time you write a program on your computer, you are using a similar device — your compiler is itself a program, which takes a program as input. Now Turing asked whether the Turing machine equivalent to the `halt` algorithm can be encoded. Such a machine would take a Turing machine M and its input i as input, and tell us whether M terminates with input i.
Imagine such a machine exists, reasoned Turing. Now I can design another machine D which takes a Turing machine M as input and, performs the following program:

$$D(M) \overset{\text{df}}{=} \texttt{if (halt(}M\texttt{,}M\texttt{)) then loop forever else terminate}$$

If M, given itself as input, terminates, then D will loop forever, otherwise, it will terminate. Now does $D(D)$ terminate? If it does, then `halt(`D`,`D`)` returns false meaning that $D(D)$ does not terminate. Conversely, if $D(D)$ does not terminate then it means that `halt(`D`,`D`)` returns true, implying that $D(D)$ does terminate. The only way out of the conundrum is to concede that `halt` cannot exist.

Another important paper appearing in 1936 was written by Alonzo Church, in which he presented another model of computation called the λ-calculus. Interestingly, Turing machines and the λ-calculus were shown to have the same expressive power. The *Church-Turing Thesis* states that Turing machines, the λ-calculus, are equivalent to our notion of what an algorithm is. Obviously, this cannot be mathematically expressed — the whole point of the thesis is that we are identifying our informal notion of algorithm with a precise mathematical one. No implementable model of computation more expressive than these has been identified since then. In particular, it can be shown that the expressive power of programming languages is equivalent to that of these models.

Later, in 1956, Noam Chomsky proposed phrase-structure grammars to describe languages. Interestingly, general grammars have also been shown to be equivalent to Turing machines. In other words, given a Turing machine, we can construct a grammar which generates a string if and only if the Turing machine returns *yes* when given the string as input. Conversely, given any any grammar, we construct an Turing machine which always answers *yes* when given a string in the language generated by the grammar.

Please be Reasonable
How are the notions of computation and languages related? Given a computational problem with a yes/no answer, we can pose the problem as a language membership one. Let L be the set of the input strings which return 'yes'. If we can design an automaton which decides whether an input is in L or not, we have shown that the problem is solvable.
One important property that 'reasonable' models must possess is that they have a finite description over a finite alphabet. This means that we can list all possible instances of the model in alphabetical order. Thus we can enumerate all machines which recognise languages. Similarly, we can list all strings over a particular alphabet. Now we can define a language which is not accepted by any of the machines. To make sure that the language we construct is different from the nth machine in the machine list, we include the nth string of the word list in the new language if and only if it is not in the language accepted by the nth machine. The conclusion? No reasonable model of computation is sufficiently powerful to accept all languages.

This terminates this brief informal tour of computability and language decidability. In the coming chapter, we will be exploring these issues more formally.

Chapter 5

Turing Machines

Turing machines are basically extended versions of pushdown automata. As with pushdown automata, Turing machines has a central 'processing' part which can be in one of a finite number of states and an infinite tape used for storage. However, there are a number of important differences between Turing machines and pushdown automata:

- unlike the pushdown automaton stack, the tape used by a Turing machine is infinite in both directions;
- Turing machines receive their input written on the tape which they use for storage;
- Turing machines control the head position to where reading and writing on the tape is performed;

Definition: A Turing machine is a 7-tuple $M = \langle K, \Sigma, \Gamma, q_0, q_Y, q_N, P \rangle$, where:

- K is a finite set of states;
- Σ is a finite set of input symbols
- Γ is the finite set of symbols which can appear on the machine tape. Amongst these is the special blank symbol $\Box$ and the input symbols Σ;
- $q_0 \in K$ is the initial state of the machine;
- $q_Y,\ q_N \in K$ are the final states denoting acceptance/rejection of the input;
- P is the 'program' of the machine. For every machine state and tape symbol combination, P tells us what the next state will be, what will be written on the tape at the current position, and whether the tape head is to be moved to the right, the left or left where it is.
  $$P : (K \setminus \{q_Y, q_N\}) \times \Sigma \to K \times \Sigma \times \{R, L, S\}$$

We are defining Turing machines as *acceptors* – machines which answer a yes or question. Usually Turing machines are presented as transducer machines, which given an input give an output. In the case of Turing machines, the output would be the text left on the tape at the end of a computation.

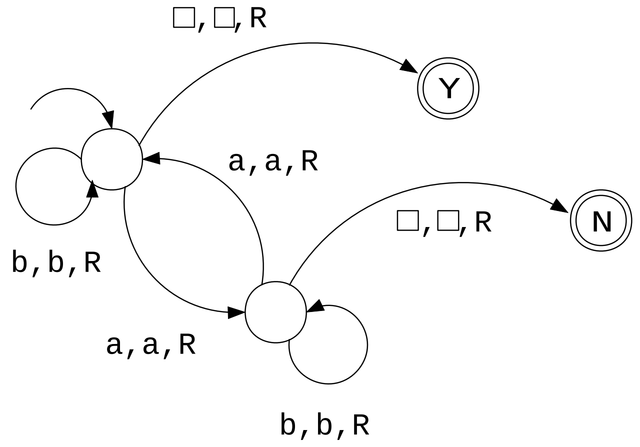
When describing a Turing machine, it is not very helpful just to list the program P. Thus, we usually use a graph to depict the machine, where every node represents a state, and directed edges are labelled with a triple (current symbol on tape, symbol to write on tape, direction in which to move) representing the program contents. Thus, there is a connecting arrow from state q to q' labelled (a, b, δ) if and only if $P(q, a) = (q', b, \delta)$. As usual, we mark the initial state by an incoming arrow, and final states by two concentric circles labelled by a Y or N.

Note that we always assume that the Turing machine starts with the head pointing at the leftmost symbol of the input string which cannot include the blank symbol $\Box$.

Example: Design a Turing machine which returns whether an input ranging over $\{a, b\}^*$ has an even number of as.

State	Read	Write	Next State	Move
q_0	a	a	q_1	R
q_0	b	b	q_0	R
q_0	$\Box$	$\Box$	q_Y	S
q_1	a	a	q_0	R
q_1	b	b	q_1	R
q_1	$\Box$	$\Box$	q_N	S

Graphically, this can be expressed as:



We have thus informally shown what it means for a Turing machine to accept a string. However, before we can prove anything about them, we need to formalize this notion.

To completely describe the state in which a Turing machine is, we need to know not only the state of the machine, but also what is written on the tape and where the tape head lies.

Definition: The configuration of a Turing machine, is a 3-tuple (q, i, t), where:

- $q \in K$ describes the state of the Turing machine.
- $i \in \mathbb{Z}$ describes the position of the tape head.
- $t \in \mathbb{Z} \to \Sigma$ is the tape contents.

Initially, the input is written on the tape and the tape head is placed on the first symbol of the input.

Definition: The *initial configuration* of a Turing machine M with input $x \in T^*$, written as $C_0^{M,x}$ (or simply C_0^x if the Turing machine we are referring to is obvious): $C_0^x \stackrel{\text{df}}{=} (q_0, 0, t_x)$, where t_x is defined as:

$$t_x(n) \quad \stackrel{\text{df}}{=} \quad \begin{cases} x!n & \text{if } 0 \leq n < \sharp(()x) \\ \Box & \text{otherwise} \end{cases}$$

Note that $s!n$ denotes the nth symbol in string s. Another notation we will use is $t =_n t'$ to mean than functions t and t' agree on all applications, except possibly for n:

$$t =_n t' \quad \stackrel{\text{df}}{=} \quad \forall i \in dom t \setminus \{n\} \cdot t(i) = t'(i)$$

Recall that we used automata configurations to define the life cycle of a computation:

Definition: A configuration $C = (q, i, t)$ is said to evolve in one step to $C' = (q', i', t')$ in a Turing machine $M = \langle K, \Sigma, q_0, q_Y, q_N, P\rangle$, written as $C \vdash_M C'$ (or $C \vdash C'$ if we are obviously referring to M), if:

- $P(q, t(i)) = (q', t'(i), S)$ and $i' = i$ and $t =_i t'$ or
- $P(q, t(i)) = (q', t'(i), R)$ and $i' = i + 1$ and $t =_i t'$ or
- $P(q, t(i)) = (q', t'(i), L)$ and $i' = i - 1$ and $t =_i t'$.

Definition: A configuration C is said to evolve to C' in Turing machine M ($C \vdash_M^* C'$) if either:

- $C = C'$ or ...
- There is a configuration C'' such that $C \vdash_M C''$ and $C'' \vdash_M^* C'$.

Definition: A configuration C is said to be a *diverging configuration* if it leads to an infinite behaviour sequence:

$$\infty_C \stackrel{\text{df}}{=} \exists C_0, C_1 \ldots \cdot C = C_0 \wedge \forall i \mathbb{N} \cdot C_i \vdash C_{i+1}$$

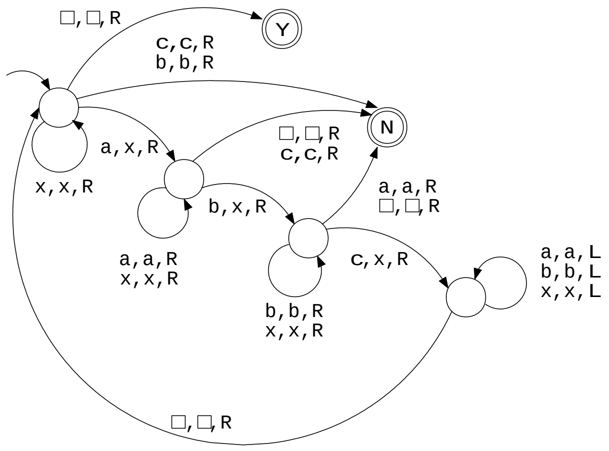
5.1 Decisions and Semi-Decisions

Definition: A language L is said to be *recognised* or *decided* by a Turing machine M if every string in L leads to q_Y, while every other string leads to q_N:

- $x \in L \Rightarrow \exists i, t \cdot C_0^x \vdash^* (q_Y, i, t)$
- $x \notin L \Rightarrow \exists i, t \cdot C_0^x \vdash^* (q_N, i, t)$

L is said to be *recursive* if we can devise a Turing machine which recognises it. We will use $\mathcal{L}_R$ to refer to the class of all such languages.

Example: The language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is a recursive language.



Note that there is an alternative way of interpreting the language accepted by a Turing machine:

Definition: A language L is said to be *semi-recognised* or *semi-decided* by a Turing machine M, if every string in L leads to q_Y.

$$x \in L \Leftrightarrow \exists i, t \cdot C_0^x \vdash^* (q_Y, i, t)$$

The rest of the strings may lead to q_N, to the Turing machine breaking (ending up in a configuration with no outgoing transition) or loop forever.

If a Turing machine semi-recognises language L, we say that L is *recursively enumerable.* We will use $\mathcal{L}_R$ to refer to the class of all such languages.

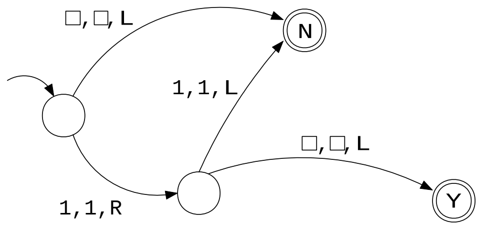
Example: Consider the following function:

$$f(n) \overset{\text{df}}{=} \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{otherwise} \end{cases}$$

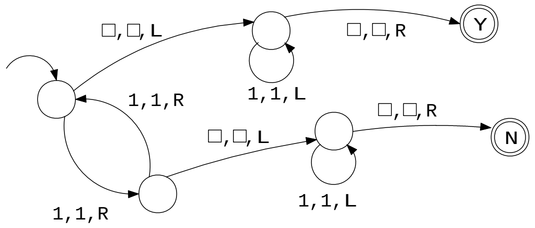
f is said to reduce a number n to 1 if after some number of applications of f to n, we end up with 1. For example, 3 is reduced to 1 in 7 steps: 3, 10, 5, 16, 8, 4, 2, 1. Let L be the set of numbers which can be reduced to 1 by f:

$$L \overset{\text{df}}{=} \{1^n \mid \exists i \in \mathbb{N} \cdot f^i(n) = 1\}$$

L can be shown to be recursively enumerable. We would first need to show that we can design Turing machines to decide whether the number on the tape is equal to 1. This is easy to do:

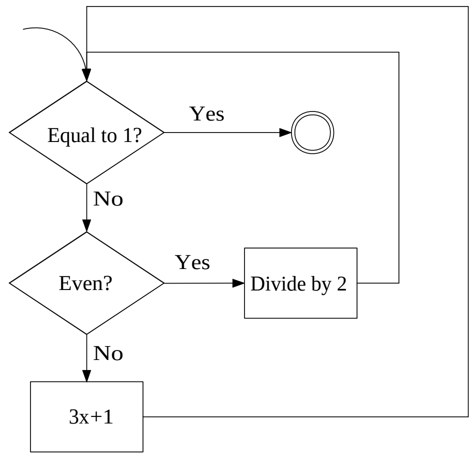


If the number is not equal to 1, we would need to decide whether the number is even or odd using a similar Turing machine to the one given in an earlier example, which recognised strings with an even number of as:



Depending on whether the number is even or odd, we need to use a machine which either divides the number by 2 or multiplies it by 3 and adds 1.

Finally, we just need to join the machines together to get a Turing machine which semi-recognises inputs which reduce to 1:



At this point in the course, it is still unclear as to whether the class of recursive and that of recursively enumerable languages are equivalent, and how they are related to regular and context-free languages. Let us start with an obvious comparison:

Proposition: All recursive languages are recursively enumerable: $\mathcal{L}_R \subseteq \mathcal{L}_E$.

Proof: The proof is quite straightforward. Let $L \in \mathcal{L}_R$. By definition, there exists a Turing machine such that $\forall x \in L \cdot \exists i, t \cdot C_0^x \vdash^* (q_Y, i, t)$ and $\forall x \notin L \cdot \exists i, t \cdot C_0^x \vdash^* (q_N, i, t)$.

Therefore we have a Turing machine for which $\forall x \in L \cdot \exists i, t \cdot C_0^x \vdash^* (q_Y, i, t)$, implying that $L \in \mathcal{L}_E$.

$\square$

We have already shown that there are non-context free languages which are recursive. The language we have shown to lie in this gap is $\{a^i b^i c^i \mid i \in \mathbb{N}\}$. We have shown that this language is not context-free using the pumping lemma, but that they is recursive by constructing a Turing machine which recognises it. But Turing machines, as it has already been noted, are just a jazzed up version of pushdown automata. In fact, given a pushdown automaton, it is possible to construct a Turing machine which emulates it.

Proposition: $\mathcal{L}_{cfl} \subset \mathcal{L}_R$.

The next natural question is whether Turing machines can semi-decide any language at all. In fact it is not the case.

Before we proceed, we will mention a mathematical technique called Gödel numbering. Given a sequence of numbers n_0, n_1, $\ldots n_m$, we can encode these as a single number $2^{n_0} * 3^{n_1} * \ldots * p_m^{n_m}$, where p_i is the ith prime number. This encoding has the property that we can uniquely identify the original number sequence. In other words, no two number sequences have the same Gödel encoding. Take 84, for example. The prime decomposition of 82 is $2^2 * 3^1 * 5^0 * 7^1$ and is thus the encoding of $(2, 1, 0, 1)$.

Theorem: There are languages which are not recursively enumerable.

Proof: Consider a Turing machine M over alphabet Σ. Enumerate the states $q_0, \ldots q_n$, and the alphabet symbols $s_0, \ldots s_m$. Similarly we encode the directions as d_0, d_1 and d_2. To specify M, we need to encode the program 5-tuples and the final states.

Each program can be seen as a 5-tuple of the form $(q_i, a_j, q_k, a_l, d_m)$. Using so called Gödel numbering, this can be represented as a number:

$$2^i 3^j 5^k 7^l 11^m$$

Hence, a finite sequence of 5-tuples, can be represented as a finite sequence of numbers $n_1, n_2, \ldots n_m$. This, in turn can also be encoded as an integer:

$$2^{n_1} 3^{n_2} \ldots p_m^{n_m}$$

where p_m is the mth prime number.

Similarly, the final states q_y and q_n can be encoded as an integer. Thus, the program and final states of M can be encoded as a positive integer, such that no two distinct Turing machines map to the same number. Hence we can put Turing machines in order of their Gödel encoding. Let the ordered machines be

M_1, M_2, etc, and the languages semi-decided by these machines to be L_1, L_2 etc.

We can also list strings over alphabet Σ lexicographically — shortest first, and in alphabetical order for strings of the same length. Let these strings be names s_1, s_2, etc.

Now consider the language $L \overset{\text{df}}{=} \{s_i \mid i \in \mathbb{N},\ s_i \notin L_i\}$. Clearly L is well-defined. If L is in $\mathcal{L}_E$, then L must be semi-recognised by a Turing machine M_i, and thus equal to L_i.

Now, is $s_i \in L$? Assume it is. Since $L = L_i$, this implies that $s_i \in L_i$ and therefore, by definition of L, $s_i \notin L$.

On the other hand, assume that $s_i \notin L$. Since $L = L_i$, this implies that $s_i \notin L_i$ and therefore, by definition of L, $s_i \in L$.

Therefore, assuming that L is semi-recognised by a Turing machine leads to a contradiction, and thus we have identified a language not in $\mathcal{L}_E$.

□

The technique used to prove this result can be easily adapted to be applied to any finitary description of a language. In particular, we can apply it to phrase structure grammars to prove that not even general grammars are universal language recognisers. The question naturally arises: are Turing machines and phrase structure grammars equally expressive?

This leads to the Church-Turing thesis: every effectively computable function is Turing-computable. According to this, there ought to be no mechanical means of establishing whether a string is in a language or not. This means, that if we can use a phrase structure grammar to algorithmically compute whether a string lies in a language or not, then phrase structure grammars cannot be more expressive than Turing machines. We will discuss this in more detail later.

5.2 More Turing Machines

Before proceeding any further, we will list a number of variations on the Turing machine model which do not affect the computing power of the resulting machines.

5.2.1 Extensions

According to the Church-Turing thesis, no reasonable extension to a Turing machine should increase its computing power.

Synchronized multi-tape Turing machines: These machines are basically a Turing machine but with more than one tape under its control. It is called synchronized, since all tape heads move together. Initially, the input is written on tape 1 and eventually it is also read off tape 1.

Clearly, the behaviour of any single tape machine can be emulated by such a machine. How can we emulate a k-synchoronised-tape machine on a normal Turing machine?

To prove this formally, we would need to formalise the concept of a k-tape machine, and its behaviour. This is beyond the scope of the course, and we will just give an informal account. However, we note that the program of a k-tape machine would be a function of type: $(K \setminus \{q_Y, q_N\}) \times \Gamma^k \to K \times \Gamma^k \times \{L, R, S\}$. We would need to match what is read on all k tapes, and write to all of them. However, one direction suffices, since the tape heads move in unison.

Now imagine creating a one tape machine with tape alphabet being: $\Sigma \cup \Gamma^k$. Clearly, this alphabet is still finite. Initially, we would scan the input, and replace every input symbol a with the symbol $\boxed{a, \Box, \Box, \ldots \Box}$ and move the head back to the first symbol. Every old production rule of the form $P_k(q, (a_0, \ldots, a_{k-1})) = (q', (a'_0, \ldots, a'_{k-1}), \delta)$ can be emulated with a new rule of the form $P_1(q, \boxed{a_0, \ldots, a_{k-1}}) = (q', \boxed{a'_0, \ldots, a'_{k-1}}, \delta)$.

What happens if the old machine uses part of the original tape beyond what was originally covered by the input? The machine would fail since we have no rules from single symbols (except for the initial rewriting of the input). To solve this problem, we add a new rule: $P_1(q, \Box) = (q', \boxed{\Box, \ldots, \Box}, S)$.

Multi-tape Turing machines: A multi-tape Turing machine is a Turing machine which has under its control a (finite) number of infinite tapes. Program instructions now contain, not just one head movement instruction, but as many as there are tapes. Initially, the input is placed on tape 1 and all heads start off in the first position of all tapes. The output is also read off tape 1.

This is very similar to the synchronised head multi-tape machine. In contrast however, the k tape heads now move independently. The program returns not just one direction, but k directions — one for each tape. If we use the previous encoding of k-tapes to one, a problem arises: We do not know where to find the heads of the tapes. We solve this problem by marking the head position on each of the tapes — the tape alphabet is now enriched to $\Sigma \times (\Gamma \cup \overline{\Gamma})^k$. $\overline{\Gamma}$ is just an overlined version of each symbol, and on each tape we will ensure that we have exactly one overlined symbol, denoting where the head on that tape lies. As before, the new alphabet is finite.

Similar to what we did before, we start by overwriting every input symbol a with $\boxed{\overline{a}, \Box, \ldots, \Box}$.

What about transitions? We will make sure that after every transition we will leave the head on the leftmost position occupied by the old k heads. Now, given a transition of the form $P_k(q, (a_0, \ldots a_{k-1})) = (q', (a'_0, \ldots, a'_{k-1}), (\delta_1, \ldots \delta_{k-1}))$, we can replace it with a sequence of transitions which travels to the right and checks what lies underneath each head, and if it matches the $(a_0, \ldots a_{k-1})$ as desired, it overwrites them with $(a'_0, \ldots a'_{k-1})$ and moves the heads as instructed. Finally, it leaves the head on the leftmost head once again. Moving any head onto a blank symbol would invoke a new transition breaking the symbol into the new one representing k blanks, with the head on the right tape.

Off-line Turing machines: These machines have two tapes: one is read-only and used to give the input, while the other is read-write.

This is just a restricted version of multi-tape machines, where we always write on the first tape what we have just read. Since we can emulate multi-tape machines, we can also emulate off-line Turing machines on a normal Turing machine.

Is the converse also true? Can any Turing machine be emulated on an off-line one? Obviously yes, we just need to copy the input onto the read-write tape, and then proceed by ignoring the read-only tape.

Non-deterministic Turing machines: The program is no longer a function but a relation. Thus, for a state and tape symbol pair, the machine may have multiple transtitions going to different states, write different symbols on the tape, moving in different directions.

In this chapter we will talk exclusively about deterministic Turing machines. The next chapter is dedicated to comparing the complexity of deterministic and non-deterministic Turing machines.

A definition is in order here. If, on a given input, a particular Turing machine can reach both q_Y and q_N due to its non-determinism, what will the result be? As in the case of non-determinism in finite state automata and pushdown automata, we take an *angelic* view of non-determinism. The machine can be seen to always opt for the path which leads to q_Y if one exists. If this sounds too much like magic, an alternative view is that the machine tries all possible paths and chooses the one leading to q_Y if it finds one. A non-deterministic Turing machine with input x can thus either (i) accept the input if there is a path leading from the initial state to q_Y or (ii) reject the input if all paths from the initial state lead to q_N or (iii) loop forever. We assume the program to be total to avoid talking about machines which break (we reach a state and tape symbol combination for which no program instruction applies).

Formally, a language L is decided by a non-deterministic Turing machine M if (i) for every string $x \in L$, $C_0^x \vdash^* q_Y$; and (ii) for every $x \notin L$, we can never reach q_Y: $C_0^x \not\vdash^* q_Y$ but we always terminate $\neg\infty_{C_0^x}$.

Semi-decidability is defined just as before: $x \in L \Leftrightarrow C_0^x \vdash^* q_Y$.

We will show non-determinism does not increase the computing power of Turing machines in the next chapter.

5.2.2 Restrictions

The principle of Occam's Razor says that given the choice between two different explanations of a phenomenon, one should choose the simpler one. What simple means is a matter of interpretation, but in science one usually strives to build a model which requires as few basic truths as possible. In mathematics simplicity is a mixture of the count of underlying axioms and rules of inference, and a subjective measure of elegance of a theory. Surprisingly, this principle is very effective when describing a new phenomenon. If we apply Occam's Razor to Turing's model of computation we have used, we can try to shave off unnecessary parts of the model but which do not change the power of the machine.

Forced head movement: The S option to keep the tape head at the same position may be done away with.

Clearly, our model can simulate such a machine. For such a machine to simulate a Turing machine which can keep the head in the same position it suffices to replace every transition of the form $P(q, a) = (q', a', S)$ which a pair of transitions going through a new state σ'_q: $P(q, a) = (\sigma'_q, a', L)$. From a new state σ_q, we move right leaving the tape unchanged by adding a new rule for every $a \in \Gamma$: $P(\sigma_q, a) = (q, a, R)$.

Thus, by doubling the number of states and adding at most $|K| * |\Gamma|$ transitions, we can emulate a Turing machine with the definition we used.

Semi-infinite tape: This variant has a tape which is only infinite in the right direction. The input is placed on the extreme left of the tape, where the head starts off from.

Emulating a two-way infinite tape on a one-way infinite tape is not difficult. We replace Γ with Γ^2. The lower part of the tape can be seen as the left portion of the two-way infinite tape folded underneath the right part. Furthermore, the set of states K is replaced by $K_U \cup K_D$ where the subscript tells us whether the head is on the lower or the upper part of the tape.

Initially, the machine replaces an input $a_0 a_1 \ldots a_n$ with $\nabla(a_0, \Box)(a_1, \Box) \ldots (a_n, \Box)$ and goes to state $q_0 U$. The ∇ is used to signify that the tape edge has been reached.

A transition $P(q, a) = (q', a', S)$ in the original machine is now replaced by transitions $P'(q_U, (a, b)) = (q'_U, (a', b), S)$ and $P'(q_D, (b, a)) = (q'_D, (b, a'), S)$.

Left and right transitions are slightly more complex. A transition $P(q, a) = (q', a', L)$ in the original machine is replaced by transitions $P'(q_U, (a, b)) = (q'_U, (a', b), L)$ and $P'(q_D, (b, a)) = (q'_D, (b, a'), R)$. Similarly for right movement.

Finally, if we reach the end of the tape, we need to 'bounce' back: $P'(q_U, \nabla) = (q_D, \nabla, R)$ and $P'(q_D, \nabla) = (q_U, \nabla, R)$.

Binary Turing machine: The tape alphabet is restricted to 0 or 1.

The construction is a simple encoding of the original tape alphabet to a binary string. Therefore, if the original tape alphabet was $\{a, b, c\}$, we can encode these simply as 00, 01 and 10. Choosing a transition now requires us to scan two symbols and decoding (internally) the string to the original symbol.

5.3 Limitations of Turing machines

To close this brief overview of Turing machines, we will examine in more detail what the limitations of Turing machines are.

5.3.1 Universal Turing Machines

The Church-Turing thesis states that any function computable by a computer program can also be computed by a Turing machine. The main gap people usually fail to connect between Turing machines and computers is that whereas computers can run different programs, a Turing machine has one set of instructions (P) which it can execute. To see that there is, in fact, no paradox resulting from the two execution styles, we may take either of two alternative viewpoints.

A computer also has a small number of instructions hardwired into it which it executes forever without changing. The underlying circuit which fetches an instruction from memory and executes it is the underlying automaton program and the actual machine code instructions are simply the input. This is no different from a Turing machine.

On the other hand, recall that a Turing machine can be encoded as a single number, which gives the program instructions and initial and final states. The encoding/decoding process can be clearly done using a Turing machine. Thus, we can construct a Turing machine which takes a Turing machine's instructions and its input ($E(M) * x$) as input and emulates it. This is called a Universal Turing machine.

The existence of such a machine is proof of the computing power possible from Turing machines. It also raises a number of interesting questions.

Consider a universal Turing machine U with input $E(M) * x$. Clearly, it terminates and fails (by terminating on q_N) if M terminates and fails with input x, terminates and succeeds (on state q_Y) if M terminates and succeeds when given input x, and loops forever if M fails to terminate when given input x. Can we go one step further, and build a super-universal Turing machine which also terminates with an appropriate output when given input $E(M) * x$, where M does not terminate when given input x?

In fact, we cannot construct such a machine. A simpler problem — the Halting Problem — can be shown to be unsolvable, from which we can then deduce that super-universal Turing machines cannot be built.

5.3.2 The Halting Problem

Definition: The halting problem is a function defined as follows:

$$f(E(M) * x) = \begin{cases} 1, & \text{if } M \text{ terminates with input } x \\ 0, & \text{otherwise} \end{cases}$$

This function allows us to define the language of halting machines: $H = \{E(M) * x \mid f(E(M) * x) = 1\}$.

Note that, if we can build a super-universal Turing machine which decides language H, then the halting problem is solvable.

Theorem: The halting problem is not decidable. Equivalently, $H \notin \mathcal{L}_R$.

Proof: Assume it is decidable by a Turing machine M, which given input $E(T) * x$ terminates in q_Y^M if T terminates with input x, in q_N^M otherwise.

We can clearly modify this Turing machine, to start by copying an input x leaving $x * x$ on the tape, then performs M, and loops forever if it reaches M's accepting state q_Y^M, but terminates if it reaches q_N^M. Let us call this new machine P.

What does P do when given its own encoding $E(P)$ as input? After the first part, we end up with $E(P) * E(P)$. We now have two cases to consider:

- P terminates with input $E(P)$: Hence, M terminates in q_Y^M and loops forever. Hence it does not terminate.
- P does not terminate with input $E(P)$: This time round, the M part of the machine ends up in state q_N^M and then P terminates.

Therefore, the construction of M is impossible.

□

5.3.3 Reduction Techniques

This is but one of the non-Turing decidable languages. Other problems may not be as easy to prove, so we try to use our knowledge about the halting problem to show that certain other functions are non-computable. The basic idea is that if we can show that a Turing machine to solve a problem Π can be used to solve the halting problem, then so such Turing machine is possible, and Π is not Turing computable.

A problem (language) Π is said to be reducible to problem (language) Π' ($\Pi \preceq \Pi'$) if there is a Turing machine which, when given an encoding $I \in \Pi$, produces an encoding $I' \in \Pi'$ such that $I \in \Pi \Leftrightarrow I' \in \Pi'$.

Theorem: Let $\Pi \preceq \Pi'$. Then Π' is decidable implies that Π is decidable, or equivalently, Π is undecidable implies that Π' is undecidable.

Example: Consider the decision problem: Does a given Turing machine halt when given input ε?

If we refer to the halting problem *HP* and the empty string halting problem as ε*HP*, we show that *HP* $\preceq \varepsilon$*HP*. Consider an instance of *HP*. We are given a Turing machine M and input x and we have to decide whether M halts with x.

If there is a Turing machine solving ε*HP* then we can give it as input a Turing machine M_x, which initially writes x on the tape and then starts behaving like M. Thus, given an instance of *HP* we are constructing an instance of εHP. Clearly M_x terminates on empty input if M terminates with input x. Similarly, if M terminates with input x, then M_x terminates when given empty input. Hence $M \in$ *HP* if and only if $M_x \in \varepsilon$*HP*.

Therefore $HP \preceq \varepsilon HP$, and hence εHP is unsolvable.

5.4 Back to Language Classes

To conclude the classification of language classes that we have defined in this course, we need one further theorem.

Theorem: Not all recursively enumerable languages are recursive.

Proof: Consider language H — the halting problem. Clearly $H \in \mathcal{L}_E$ — we just need to emulate the given machine with the given input. If it terminates, we terminate on q_Y. Note that if an input is in H, we will eventually terminate on q_Y. If the input is not in H, then we will always loops forever (because the simulation never terminates). Therefore $H \in \mathcal{L}_E$. But we have shown that $H \notin \mathcal{L}_R$.

□

5.5 Back to Grammars

We have proved that the class of recursive languages is larger that the class of context-free languages, but smaller than the class of recursively enumerable languages. But how is $\mathcal{L}_E$ related to the class of languages accepted by general phrase structure grammars (for convenience, we will refer to this class of languages $\mathcal{L}_{psg}$)?

Theorem: Languages generated by phrase structure grammars are recursively enumerable.

$$\mathcal{L}_{psg} \subseteq \mathcal{L}_E$$

Proof: Consider $L \in \mathcal{L}_{psg}$. If we can find an algorithmic way of enumerating the strings in L (which, if you recall, can be enumerated) as $\langle w_1,\ ,w_2\ \ldots\rangle$, we can build a Turing machine which produces these strings in sequence, and compares each produced string with the input. If it matches, it goes to state q_Y and terminates, but otherwise it continues producing strings.

If the input $x \in L$, it must be equal to w_i, for some i. Hence, the Turing machine will eventually terminate successfully. However, if $x \notin L$, it matches no w_i and will thus never terminate.

But how do we algorithmically enumerate the strings in L? The easiest way is to list them in shortest derivation first order. Define the sequence of sets of strings over $(\Sigma \cup N)^*$ (N are the non-terminals)s:

$$\begin{aligned} N_0 &\stackrel{\text{df}}{=} \{S\} \\ N_{i+1} &\stackrel{\text{df}}{=} \{\beta \mid \exists \alpha \in N_i \cdot \alpha \Rrightarrow \beta\} \end{aligned}$$

Each of these sets is finite (since N_0 is a finite set of finite strings, and we have only a finite number of production rules) and, if $x \in \mathcal{L}(G)$ then $x \in N_n$ for some value of n. Clearly, there is a simple algorithm to generate the sets N_i. Also, if we sort the terminal strings in N_i (say alphabetically), we get an enumeration for every string in $\mathcal{L}(G)$.

$\square$

Theorem: For every recursively enumerable language, there is a grammar which generates it.

$$\mathcal{L}_E \subseteq \mathcal{L}_{psg}$$

Proof: Let M be a semi-infinite tape Turing machine which accepts L. We can modify M such that it never writes a blank symbol on the tape, by adding a new tape symbol $\diamond$, replacing all program instructions $P(q, a) = (q', \square, \delta)$ by $P(q, a) = (q', \diamond, \delta)$, and for every program instruction $P(q, \square) = (q', a, \delta)$ we add $P(q, \diamond) = (q', a, \delta)$. In plain language, we make sure that the machine writes a new symbol rather than blank and upon reading the new blank character, it will behave just as if it has read a normal blank character. Call this modified Turing machine M'. Clearly, the transitions of M' are just like those of M except that the output may be different because of the new blank symbols. Thus M terminates exactly when M' terminates. But M terminates if and only if its input is in L. Hence M' terminates if and only if the input it receives is in L.

Let $M' = \langle K, \Sigma, q_0, q_Y, q_N, P\rangle$. We now construct a grammar G which emulates the behaviour of M', but in reverse. Thus $S \Rrightarrow^* x$ if and only if M' reaches a final state when receiving input x.

We start off by generating all possible final states of the Turing machine. This will be represented as a string of the form $\ll \alpha q_Y \beta \gg$. $\ll$ and $\gg$ are used to mark the end of the used tape, q_Y indicates that the machine is in the accepting state, and the head ends up on the first symbol of string β.

The production rules to generate such strings can be something like the following:

$\{S \to \ll A\}$
$\{A \to aA \mid a \in \Sigma \cup \{\square\}\}$
$\{A \to q_Y B\}$
$\{B \to aB \mid a \in \Sigma \cup \{\square\}\}$
$\{B \to \gg\}$

We would now like to be able to simulate the Turing machine's transitions in reverse:

$\{q'a' \to qa \mid P(q, a) = (q', a', S)\}$
$\{a'q' \to qa \mid P(q, a) = (q', a', R)\}$
$\{q'ba' \to bqa \mid b \in \Sigma,\ P(q, a) = (q', a', L)\}$

Finally, when the initial state is reached, we can do away with it, and remove padding blanks:

$\{q_0 \to \varepsilon,\ \ll \square \to \ll,\ \square \gg \to \gg\}$
$\{\ll a \to a \mid a \in \Sigma\}$
$\{a \gg \to a \mid a \in \Sigma\}$

The result is that if, in the grammar $\alpha \Rrightarrow^* \beta$, then the Turing machine would be able to make a number of steps to go from the configuration encoded as β to the one encoded as α. Similarly, the converse holds.

□

Corollary: $\mathcal{L}_{psg} = \mathcal{L}_E$

Summary: If we call the class of context-free languages $\mathcal{L}_{cfl}$ and the class of regular languages $\mathcal{L}_{rl}$, we have shown that:

$$\mathcal{L}_{rl} \subset \mathcal{L}_{cfl} \subset \mathcal{L}_R \subset \mathcal{L}_E = \mathcal{L}_{psg} \subset 2^{\Sigma^*}$$

5.6 Exercises

1. *(Easy)* Construct a Turing machine which given an input 1^n leaves 1^{3n+1} on the input tape.

2. *(Easy)* One of the Turing machines given as an example in this chapter recognised $\{a^i b^i c^i \mid i \in \mathbb{N}\}$. What are the implications of this regarding pushdown automata?

3. *(Easy)* Design a Turing machine which recognises the language generated by the following regular grammar:

$$\begin{array}{rcl} G & \stackrel{\text{df}}{=} & \langle \{a, b, c\},\ \{S, A, B\},\ P,\ S\rangle \\ P & = & \{ \quad S \to aA \mid bB, \\ & & \quad\ \ A \to aA \mid cB \mid b \\ & & \quad\ \ B \to bB \mid cA \mid a \quad \} \end{array}$$

4. *(Moderate)* Design a Turing machine which recognises the language generated by the following regular grammar:

$$\begin{array}{rcl} G & \stackrel{\text{df}}{=} & \langle \{a, b\},\ \{S, A, B\},\ P,\ S\rangle \\ P & = & \{ \quad S \to bA \mid aB, \\ & & \quad\ \ A \to aB \mid a, \\ & & \quad\ \ B \to bA \mid b \qquad \} \end{array}$$

5. *(Moderate)* Design a Turing machine which decides the language generated by the following regular grammar:

$$\begin{array}{rcl} G & \stackrel{\text{df}}{=} & \langle \{a, b, c\},\ \{S, A, B\},\ P,\ S\rangle \\ P & = & \{ \quad S \to cA \mid cB, \\ & & \quad\ \ A \to aB \mid c, \\ & & \quad\ \ B \to bA \mid c \qquad \} \end{array}$$

6. *(Moderate)* Generalize your solutions to work with general regular grammars.

7. *(Easy)* How would you show that $\mathcal{L}_R$ is closed under language complement?

8. *(Moderate)* Show that $\mathcal{L}_R$ is closed under language union.

9. *(Difficult)* Show that $\mathcal{L}_R$ is closed under language catenation.

10. *(Difficult)* Discuss how you would show that $\mathcal{L}_E$ is closed under set union.

11. *(Difficult)* Show that $\mathcal{L}_E$ is not closed under set complement.

    Hint: If recursively enumerable languages were closed under set complement, then semi-decidable languages would all be decidable.

12. *Moderate* Formalise the definition of a k-tape Turing machine (with an independent head on each tape). Define (formally) the configuration of such a machine.

13. *Moderate/Difficult* A two-stack pushdown automaton is similar to a pushdown automaton, but uses two stacks for secondary memory. As in the case of a pushdown automaton, the machine chooses a transition based on its current state, the first symbol on the input and the value on the top of the stacks. It may then change state and write a string on both of the stacks. An input is accepted if, starting from the initial state, the machine can reach a final state with that input.

    (a) Formalise the definition of a two-stack pushdown automaton (2PDA).

    (b) Define the configuration of a 2PDA.

    (c) Formally define the language accepted by a 2PDA.

    (d) A 2PDA has the same computation power as a Turing machine. The two stacks can be seen as the tape to the left of the head and the the tape to the right. Discuss how, given a Turing machine, we can construct a 2PDA which behaves identically. Make sure that the 2PDA never reads from an empty stack.

Chapter 6

Algorithmic Complexity

In the previous chapters, we have encountered the concept of uncomputable languages. This provides a useful classification of problems: solvable and unsolvable.

However, this classification can be too weak. Consider the class of decidable problems. Within this equivalence class, some of the problems are more equal than others. Some problems may lie in the class but are not solvable in practice. Recall that a Turing machine has unlimited storage space and unlimited time in which to run, even though we live in a universe of limited resources. An algorithm is as good as useless if it need 10 times the current age of the universe to execute, or needs more storage cells than there are basic particles in the universe. Still, if we were to draw an arbitrary line dividing useful from non-useful algorithms based on the physical size or length of time it requires, it would have to be a subjective matter[1].

There is also another problem. As computers get faster, we would have to admit more and more algorithms as useful. We thus want to find a better way of choosing the practically solvable problems.

We can analyze the inherent complexity of algorithms. You have already encountered complexity in previous courses. You may remember that whereas adding a new element to the head of a linked list takes only a constant amount of time (irrespective of the size of the list), sorting a list of length n requires on the order of $n \log(n)$ time units.

Consider a problem with complexity of the order n, where n is the size of the input to the problem. By doubling the computer speed, we can solve a problem of twice the size, in the same amount of time. What about a problem of size n^2? By doubling the speed, we can solve problems of up to 1.4 times the

[1]One may classify algorithms taking longer than a century to execute, but this didn't stop an alien race from building a computer which took 5 million years to execute its program to find the answer to life, the universe and everything in Douglas Adams' *Hitchhiker's Guide To The Galaxy*.

original size. But how about a problem of size 2^n? By doubling speed, we are only increasing the maximum size of the problem solvable by 1. This shows an inherent difference between problems solvable in polynomial time to those solvable in exponential time: whereas by multiplying the speed of the underlying machine, the size of solvable polynomial problems increases geometrically, this only grows arithmetically for exponential problems.

There still is a question of what model of computing we are allowed to use. Consider the following problem:

Travelling Salesman (TS)

INSTANCE: Given a set of n cities C, a partial cost function of travelling from one city to another *cost* (such that $cost(c_1, c_2)$ is defined if there is a direct means of travelling from city c_1 to city c_2, and if defined is the cost of the travelling) and an amount of money m.

QUESTION: Is there a way of travelling through all the cities exactly once, returning back to the initial city, and in the process not spending more than the amount of money m?

Consider a non-deterministic Turing machine, which will simply fan-out trying all possible paths. Clearly, in at most n steps, each path will either succeed or fail. Hence, we will get a result in at most n steps. Hence, it seems that this problem is one of polynomial complexity. But non-determinism, involves, if you recall, an oracle function which tells us which path (if any) will yield results, or a machine with the ability to replicate itself for any number of times. This is not a very realistic means of computation. So what if we try to solve the problem on a deterministic Turing machine? If we enlist all the paths and try them one by one, in the worst case, we will need to examine all possible paths. This goes beyond the realms of polynomial complexity. You may try to find an algorithm to solve the problem on a deterministic time, but it is very improbable that you will manage to find one.

Hence, the rudimentaries of a hierarchy start to become apparent. We will define the class of problems solvable in polynomial time on a deterministic Turing machine to be P. The class of problems solvable in polynomial time on a non-deterministic Turing machine will be called *NP* (non-deterministically polynomial). Beyond this, lie the problems which need exponential time even on a non-deterministic Turing machine.

Note that a number of problems have been shown to require exponential time on a deterministic Turing machine. We refer to such problems as *intractable* because of the difficulties we encounter as soon as we try to apply these algorithms to modest sized instance of the problem.

Before concluding this brief introduction, a couple of questions arise from the above text. First of all, does there exist a solution of the travelling salesman problem which uses only polynomial time on a deterministic machine? And secondly, why do we include this material together with language hierarchies in a single unit?

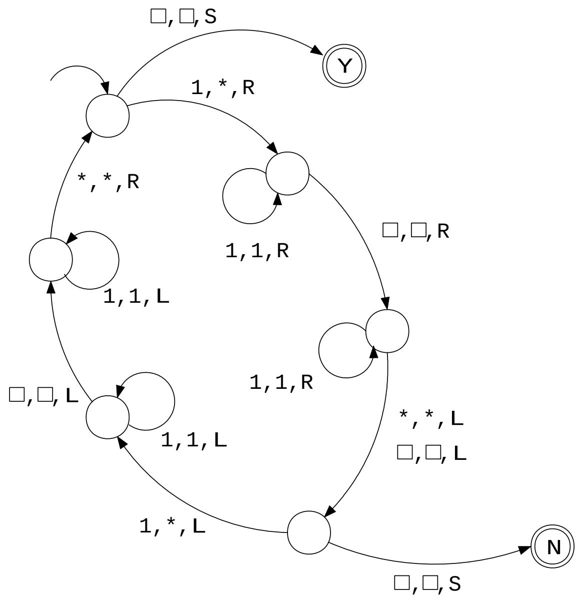
Let us start with the initial question. Before, we consider the travelling salesman, consider the following problem:

Less than or equal to (LTE)

INSTANCE: Given two numbers n_1 and n_2.

QUESTION: Is $n_1 \leq n_2$?

Consider the deterministic Turing machine which performs this operation. It will terminate and succeed if the answer to the question is positive, but terminate and fail otherwise. One such machine overwrites the numbers (one at a time) with a new symbol. If, the second runs out before the first, fail but terminate successfully otherwise:



Each right-left trip will take about $2(n_1 + 2)$ steps. This will have to be done for n_1 times. Hence, the machine terminates after at most $\alpha n_1^2 + \beta n_1 + \gamma$ steps. Since the machine is a deterministic one, and it takes only a polynomial amount of time (relative to the size of the input), this problem is in P. Clearly, every deterministic Turing machine can be seen as a non-deterministic one and thus, it can also be solved by a non-deterministic Turing machine in polynomial time and is hence also in *NP*.

How about the travelling salesman's problem? We have already given a rough idea of how to construct a non-deterministic Turing machine which computes the answer in polynomial time. Hence it is in *NP*. But is it also in P? This is probably one of the most sought after answers in mathematics today. Nobody has come up with an algorithm showing that it is in P but, on the other hand, nobody has come up with a proof that this is impossible. The mystery is even more intriguing than this. There is no problem known to be in *NP* but not P.

In 1971, an important paper identified a problem, the satisfiability problem, and showed that it is the hardest problem in *NP*. In other words, if it is shown to be in P, so must all the other problems in *NP* and P =*NP*. Other work done over these past 30 years has identified a number of other such hardest problems in *NP*. Amongst these is the travelling salesman problem.

The general consensus, is that $NP \neq P$ and hence the travelling salesman problem is intractable and not in P, but this still has to be proved.

As for the inclusion of this material together with language hierarchies, we note that we are basically using the Turing machines as language recognizers. We have a decision problem Π which, divides the set of possible inputs into two: the instances answered positively and those answered negatively, which are complements of each other. Hence, our questions are all dealing with the complexity of the language in question. Note that we are only dealing with recursive languages, since we assume that the Turing machine terminates with a yes or no answer.

6.1 Background

We will now start formally defining what we mean by the complexity of a problem. We will only be treating problems with a yes/no answer and for which there exists a terminating algorithm which gives the answer. The following definitions regarding complexity are based on languages. This may seem strange, but the concept of a yes/no decision problem is intimately related to languages.

Consider a yes/no problem Π. One way of stating such a problem is to define the language of all input strings whose answer to the Π question is yes. Clearly a Turing machine which computes the answer to this problem is just deciding whether the given input string is in the language . By defining the complexity of deciding membership in a particular language we are defining the complexity of general decision problems.

The format in which we will present decision problems is as follows:

Primality test (PRT)

Instance: Given a number n.

Question: Is n a prime number?

The first line states the name of a decision problem (in this case, primality test) and a short version of the name which will be used as an abbreviation (PRT). We then specify what an *instance* of the problem consists of. In this case it is a number n. This can be seen as the formal parametrisation of the input of the problem. Finally, we ask a yes/no question regarding the instance.

Our main task is to decide how much work has to go into showing that a particular instantiation of the input is in the language of positive answers or not. Obviously, certain inputs inherently require more work than others. For example, consider the problem of checking whether a given word is in a given dictionary. The problem becomes more 'complex' as the size of the dictionary increases. The complexity of the problem is thus to be given as a function of the size of the dictionary. This is not very convenient, since for every decision problem we must first isolate the inputs on which the problem complexity depends. However, there is a simpler solution which we will adopt. Given a reasonable encoding of the inputs, we will calculate the complexity of the problem as a function which, given the length of an input, gives the complexity of calculating the result. Consider the dictionary problem once more. We are now giving the complexity of computing the result in terms of, not just the dictionary size, but also the size of the word we are searching for. This makes sense, since every time we check whether a particular dictionary entry matches the searched word, we need to look through the symbols of the word to check for equality, and hence, the longer the word, the more work needs to be done.

Now consider the complexity of a particular algorithm to solve the dictionary problem. As you should already be aware, there is a variety of complexity (execution length) measures we can take. The ones we are usually interested in are the average and worst case length of execution. In this course we will be exclusively interested in worst case analysis.

To classify a problem we use the *big-O* notation. Since it is difficult to calculate the exact number of execution steps of an algorithm (and it is dependent on the amount of effort put into making it more efficient), the *big-O* notation allows us to approximate an upperbound to a function. The formal definition is the following:

Definition: We say that a function $f(n)$ ($f : \mathbb{N} \to \mathbb{R}^+$) is $O(g(n))$, sometimes written as $f(n) = O(g(n))$ if there is a constant c, such that, beyond a particular value of n, $c\, g(n) \geq f(n)$:

$$\exists c : \mathbb{R}^+,\ n_0 : \mathbb{N} \cdot \forall n : \mathbb{N} \cdot (n \geq n_0 \Rightarrow c\, g(n) \geq f(n))$$

For example, you may recall that $7n^2 + n - 4$ is $O(n^2)$, since for non-zero n ($n_0 = 0$), $8n^2 \geq 7n^2 + n - 4$ ($c = 8$).

Similarly, $2^n + n^{100}$ is $O(2^n)$, since for any $n \geq 1000$, $2 \cdot 2^n \geq 2^n + n^{100}$.

Finally, there is still an ambiguity in the description we have given. We have stated that the input is reasonably encoded. What exactly do we mean by this?

Some encodings are obviously unreasonable. Consider a Turing machine which checks whether two given strings are equal. There is a simple algorithm which goes back and forth to check the symbols for equality. Assume that the first string is m symbols long. Clearly, it needs, at most, $m + 1$ runs to check each of the m symbols, and in each run it moves the head $m + 1$ spaces to the left and then m symbols to the right. Hence, we have a machine taking about $m(m + 1 + m)$. Since the total length of the input $n \geq m$ symbols long, the execution length is not larger than $n(2n+1)$, which is $O(n^2)$. But now consider the encoding of the input w_1 and w_2 as $(w_1 \Box w_2 \Box)^{\sharp(w_1 w_2)}$. Using precisely the same algorithm but ignoring the extra symbols which we will not use, the analysis now yields a worst case taking $O(n)$. This is because we are using an unreasonable encoding.

Reverting back to Turing machines as language acceptors bypasses the problem very neatly. It is now clear that the two machines are accepting considerably different languages. The first machine accepted exactly those strings in $\{w\Box w \mid w \in \Sigma^*\}$ and rejected all those in $\{w\Box w' \mid w, w' \in \Sigma^*,\ w \neq w'\}$. On the other hand, the second machine is accepting strings in:

$$\{(w\Box w\Box)^{2\sharp(w)} \mid w \in \Sigma^*\}$$

but rejected strings of the form:

$$\{(w\Box w'\Box)^{\sharp(ww')} \mid w, w' \in \Sigma^*,\ w \neq w'\}$$

Note that we *assume* that the input lies in the domain of the language.

But we eventually want to apply this knowledge to actual problems. There is no hard and fast way of defining what a reasonable encoding is. It is usually used to denote a mixture of conciseness (without unnecessary padding as was used in the example) and decodability (the ability to decode the the input into a usable format using an algorithm of $O(n^\alpha)$, where α is a constant).

6.2 Machine Complexity

We have not yet analyzed the length of Turing machine executions. This is what we set out to do in this section. The basic machine we will be looking at is a single (two-way) infinite tape machine. Furthermore, we will be looking at terminating Turing machines.

Definition: A terminating deterministic Turing machine is said to have complexity t (where t is a function from natural numbers to natural numbers) if when given input x, the length of the derivation is not more than $t(\sharp(x))$.

Note that we are looking at the worst case. Consider the list of all single character inputs x_1, x_2, $\ldots x_n$. Assume that they respectively take l_1, l_2,$\ldots l_n$ transitions to terminate. Then, the definition states that $t(1) \geq \max\{l_1,\ l_2, \ldots l_n\}$.

Here, we are talking about single tape Turing machines. Usually it is much more convenient to be able to use multiple tape machines in our constructions. However, the extra tapes may allow us to perform certain problems faster than on a single tape machines, and our results for the construction we give would not be comparable to the real complexity as defined above. To get around this problem, we prove a theorem which relates the complexity of a k-tape Turing machine with that of a single tape machine.

Theorem: Provided that $t(n) \geq n$, a k-tape Turing machine of complexity $t(x)$ can be simulated on a single tape Turing machine with complexity $O(t^2(x))$.

Proof: We give the construction of a single tape Turing machine which can simulate a multi-tape Turing machine, and then we analyze the complexity of the resultant machine.

The emulator will separate the contents of the k tapes by a special symbol $\star$ and will have extra symbols, such that the contents of the location of the heads will be marked by the symbol with a bar over it. Hence, the tape contents

$\star a\bar{b}aa\star\overline{\Box}\star\bar{a}\star$ represents a 3 tape machine, where the first tape has the contents $abaa$ and the head lies on the b, the second is still unused and the third contains a single symbol over which the head resides.

The emulator, upon input $w = av$, will follow this program:

1. Write on the tape the initial configuration: $\star\bar{a}v\star(\overline{\Box}\star)^{k-1}$.
2. To simulate a single move, the machine scans the input from the first $\star$ to the $(k+1)$th to note what is under the heads. Then a second pass is made to update the tapes accordingly.
3. If, at any point the machine tries to move a virtual head onto a $\star$, it will replace the position by a blank and shift the symbols in the required direction.

Clearly, the initial stage takes $O(n)$ steps. The second and third stage requires two scans and up to k shifts. Each scan and shift requires a number of steps of the order of the length of the strings stored on the tape. Clearly, these cannot be any longer than the length of the complete execution of the k-tape machine. Hence, every time the second and third stages are performed, they take $O(t(n))$. However, these will be performed up to $t(n)$ times, and hence the total complexity is: $O(n) + t(n)O(t(n))$

Since we assumed that $t(n) \geq n$, this reduces to $O(t^2(n))$.

$\Box$

We could do the same with other extensions of Turing machines, but k-tape Turing machines are generally sufficient to allow clear descriptions of algorithms.

We now turn our attention to non-deterministic Turing machines and we will start with a definition of what we mean by the complexity of a non-deterministic Turing machine (the complexity of a multi-tape Turing machine followed naturally from that of a single tape machine, which is why we did not give it).

Definition: A terminating non-deterministic Turing machine is said to have complexity t (where t is a function from natural numbers to natural numbers) if, when given input x, the length of the longest branch of execution (whether it terminates successfully or not) is $t(\sharp(x))$.

Since deterministic Turing machines are all-powerful, they can also simulate non-deterministic Turing machines. This obviously comes at a price. The following theorem relates the complexity of a non-deterministic Turing machine with that of a deterministic one emulating it.

Theorem: Provided that $t(n) \geq n$, a non-deterministic single tape Turing machine of complexity $t(n)$ can be simulated on a single tape deterministic Turing machine with complexity $2^{O(t(n))}$.

Proof: As before, we construct an emulator and then analyze its complexity. To emulate a non-deterministic Turing machine, we use a 4-tape Turing machine. The deterministic machine proceeds to enumerate all possible computations and executes then in sequence. The four tapes are used as follows:

1. The first tape stores the input string which it does not modify in any way;
2. The second tape is used as a working tape when emulating a particular instance of the non-deterministic machine;
3. The third tape is used to store the possible computations which we have already tried;
4. The last tape is used to hold a flag to remember whether there was a computation of the length currently being investigated which has not terminated.

The main problem is that of enumerating all possible computation paths. Let α denote the maximum number of possibilities we have at any point in the computation. We can describe any computation with input w as a sequence of numbers ranging over $\{1,\ 2,\ \ldots \alpha\}$. At every stage there is a non-deterministic choice, we use the next number in the list to determine which path to follow. Obviously, not all strings over this alphabet are valid computations.

In particular, if a string is rejected there is a length beyond which all computation sequence strings over $\{1, \ldots \alpha\}$ will either reject the input (not necessarily using up all the computation sequence string), or give an invalid computation (in that a choice of a non-existent path is made).

Now consider the following algorithm:

1. Copy the contents of tape 1 to tape 2.
2. Simulate the non-deterministic machine using the string on tape 3 to take decisions as to which path to follow. A number of possibilities can occur:

   If a decision is to be made about which path to follow but no more symbols remain on tape 3 jump to step 3.

   If an invalid computation occurs (the next symbol on tape 3 is larger than the set of next possibilities) or a rejecting configuration is encountered jump to step 4.

   If, however, an accepting configuration is encountered, accept the input and terminate.
3. Write a symbol $\star$ on tape 4.
4. Calculate the next string over $\{1, \ldots \alpha\}$ to be written on tape 3. These are to be produced in order of the length of the string (and alphabetically for strings of the same length).

   If it is longer than the previous string then check tape 4. If there is a blank symbol (no $\star$ written on the tape), then all the paths were either invalid, or produced rejecting computations. Terminate and reject.

   If there is a $\star$, then there must have been some, as yet unterminated computations and we need to check computations of a longer length. Clear tape 4 and jump to instruction 1.

It should be clear (although not necessarily easily provable) that this emulator terminates and accepts exactly when the original machine did so, and terminates and rejects when the non-deterministic machine did so.

For an input of length n, the maximum depth of the computation tree is $t(n)$. At every stage, there are at most α siblings of a particular node. The total number of nodes, thus cannot exceed $\alpha^{t(n)}$ and is thus $O(\alpha^{t(n)})$. The time taken to travel down the tree path to a node does not take more than $O(t(n))$ steps. Hence, the running time of the resultant machine is $O(t(n))O(\alpha^{t(n)})$. This is $2^{O(t(n))}$. But the machine we used had 4 tapes. By the previous lemma, this can be simulated on a single tape machine with complexity $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$.

$\Box$

Note that this theorem does not say that it is impossible for a deterministic Turing machine to emulate a non-deterministic one in less than exponential time, but that we know of an algorithm which needs this amount of time. Hence, we have established an upperbound for the complexity of this simulation.

6.3 Complexity Classes

6.3.1 Class P

Clearly, we can classify languages by the complexity of a Turing machine needed to decide the language. As already indicated before, we will classify together all languages which need a polynomial complexity single tape deterministic Turing machine to decide them. This is the class of languages P.

Definition: P is the class of languages decidable in polynomial time on a deterministic single tape Turing machine:

$$P \overset{\text{df}}{=} \{L \mid L \text{ is decided by a deterministic } T \text{ such that complexity of } T \text{ is } O(n^k) \text{ for some } k \in \mathbb{N} \}$$

We have already shown, for example, that checking whether a given word is in a given dictionary is a problem in P. We will give another example before going on to further classes.

Example: Consider the following problem:

Directed path (PATH)

INSTANCE: Given a directed graph G and two nodes n_1 and n_2.

QUESTION: Is there a path from n_1 to n_2?

We will show that $PATH \in P$, by constructing a polynomial time algorithm working on a deterministic Turing machine which solves the problem.

1. Mark node n_1.

2. For every edge (a, b) such that a is marked, mark also b.
3. If some new nodes have been marked, go back to 2.
4. Accept if n_2 is marked, reject otherwise.

The first and last steps are executed just once (and take only at most $O(n)$ steps to locate n_1 and n_2). Step 2 scans every edge, checks the first node and may mark the second. This can be easily done in polynomial time on a deterministic machine (scanning every edge takes $O(n)$ and in every case checking the first node and possibly marking the second will only take $O(n)$, hence taking $O(n^2)$). Finally step 2 is repeated for a number of times. Consider the count of marked nodes which starts at 1 and increases every time step 2 is executed. Clearly, this cannot go on more than the number of nodes we have. Hence, the maximum number of repetitions is $O(n)$. Hence, the total complexity is:

$$O(n) + O(n^2)O(n) + O(n) = O(n^3)$$

Hence, $PATH \in P$.

□

Theorem: The class of languages recognizable by k-tape Turing machines in polynomial time is P.

Proof: Let M be a k-tape Turing machine such that it is decidable in $O(n^\alpha)$ ($\alpha \in \mathbb{N}$) steps. By the theorem relating multi-tape Turing machines with single tape ones, we have that it is decidable in $O(n^{2\alpha})$ on a single tape machine. Hence, it is in P. Note that in the case of $\alpha = 0$, if $f(n) = O(1)$ then $f(n) = O(n)$. Hence we can still apply the emulation theorem.

Conversely, if $L \in P$ then there is a single tape deterministic Turing machine which decides L. But we can run M on a multi-tape machine which ignores all but the first tape. This emulation takes exactly as many steps as the original machine. Hence, L is decidable in polynomial time on a k-tape machine.

□

6.3.2 Class *NP*

The theorem which constructed a non-deterministic Turing machine emulator running on a deterministic Turing machine, stated that we can perform this emulation approximately in the exponent of the original time taken by a non-deterministic machine. It does not state that there is not a more efficient emulation. However, if there is one, we do not yet know about it. This means that the class of languages decidable on a non-deterministic Turing machine in polynomial time is not necessarily the same as P. With this in mind, we define the class NP:

Definition: NP is the class of languages decidable in polynomial time on a non-deterministic single tape Turing machine:

$$NP \overset{\text{df}}{=} \{L \mid L \text{ is decided by a non-deterministic } T \text{ such that complexity of } T \text{ is } O(n^k) \text{ for some } k \in \mathbb{N}\ \}$$

NP stands for *non-deterministically polynomial.*

The first thing we note is that class P is contained in class *NP*:

Theorem: $P \subseteq NP$

Proof: The proof is trivial. Every deterministic Turing machine can be seen as a non-deterministic Turing machine with no choices. Hence, the time complexity of a deterministic Turing machine is the same as when interpreted as a non-deterministic Turing machine. Thus, all polynomial time deterministic machines run in polynomial time on a non-deterministic machine and therefore $P \subseteq NP$.

□

Example: Let us look at a problem encountered on a multiprocessor system:

Scheduler (SCH)

INSTANCE: Given a set T of tasks, a time cost function c associating a positive cost with every task, a number of processors p and a deadline D

QUESTION: Can the tasks be scheduled in the given deadline? Or more precisely, is there a partition of T, $T_1 \ldots T_n$ such that:

$$D \geq \max\{\textstyle\sum_{t \in T_i} c(t) \mid 1 \leq i \leq n\}$$

We prove that $SCH \in NP$. Consider the non-deterministic Turing machine which goes through all the tasks assigning each one to a processor non-deterministically. At the end, the total time taken is calculated and checked against D. If it larger than D, the machine terminates and fails, otherwise it terminates and succeeds. Clearly, the algorithm always terminates. If there is any assignment which will work it will be tried and the machine will succeed and accept the input, but if there are none, the machine fails and rejects the input. The assignment of tasks to processors takes $O(n)$ steps. Adding two numbers can be done in polynomial time and thus so can the adding of $O(n)$ numbers. Finding the maximum of a set of p elements is also $O(n)$ as is comparison of numbers. Hence, the total time taken by any path is $O(n^\alpha)$ for some particular value of α. Hence, we have constructed a polynomial time non-deterministic Turing machine which decides *SCH*. Hence $SCH \in NP$.

□

6.4 Reducibility

A psychologist analyzing the mathematical mind asked the following question to a mathematician:

You are walking down a street where you see a burning house, a fire hydrant, a box of matches and a hose pipe. What do you do?

After careful thought, the mathematician answered:

I connect the pipe to the hydrant, turn the water on and extinguish the fire.

Satisfied by the answer, the psychologist asked another question:

You are now walking down a street where you see a house, a fire hydrant, a box of matches and a hose pipe. What do you do?

Without even pausing for thought, the mathematician answered:

I light a match, set the house on fire, and I have now reduced the problem to a previously solved one.

In mathematics, the concept of reducing a problem into a special case of an already known one is a very important technique. The area of algorithm complexity is no exception and in this section we will define what it means for a problem to be reducible to another and how this technique can be used effectively.

Consider the following problem:

Transitive acquaintance (TA)

INSTANCE: Given set of people, a list of who knows whom and two particular persons p_1 and p_2.

QUESTION: Is p_1 a transitive acquaintance of p_2?

We say that p_1 is a transitive acquaintance of p_2 if, either p_1 is a acquaintance of p_2 or there is an intermediate person p such that p_1 is a acquaintance of p and p is a transitive acquaintance of p_2. Is this problem in P?

With a little bit of insight we realize that this problem is simply a rewording of the *PATH* problem. If we label the nodes by people's names, and make two edges (a, b) and (b, a) for every pair of people a and b who know each other (assume that acquaintance is mutual), we can run the *PATH* algorithm and solve the problem in polynomial time.

Since we can build the graph in polynomial time, and $PATH \in P$, we immediately know that $TA \in P$. How can we formalize this concept of a problem being reducible to another?

Definition: A language L_1 is said to be polynomially reducible to another language L_2, written as $L_1 \preceq_P L_2$, if there is a total function $f : L_1 \to L_2$ which is computable in polynomial time on a deterministic Turing machine, such that

$$w \in L_1 \Leftrightarrow f(w) \in L_2$$

To show that $TA \preceq_P PATH$ we need to define function f, which given an instance of *TA* gives an instance of *PATH*. This function, is the one already defined: $f(\langle P, A\rangle) = \langle P, A \cup A^{-1}\rangle$ (where $\langle P, A\rangle$ is an instance of the acquaintance problem — P is the set of people and A the set of acquaintances, and $\langle N, E\rangle$ is an instance of *PATH* if N is the set of nodes and E the set of edges). This function can clearly be calculated in polynomial time since it requires only copying.

Since *PATH* $\in P$ and *TA* $\preceq_P$ *PATH* we expect to be able to conclude that *TA* $\in P$. This is proved in the following theorem:

Theorem: If $L_1 \preceq_P L_2$ and $L_2 \in P$ then $L_1 \in P$.

Proof: Simply compute function f, taking $O(n^\alpha)$ steps, then execute the algorithm deciding L_2, taking $O(n^\beta)$. Now $w \in L_1$ if and only if $f(w) \in L_2$. Hence this algorithm decides L_1. Its complexity is $O(n^\alpha) + (n^\beta) = O(n^{\max(\alpha,\beta)})$ and L_1 is hence in P.

$\square$

Note that the function need not be as simple as the one for *TA*. Consider the following extension to the *TA* problem:

Country acquaintance (CA)

Instance: Given set of people and their nationality, a list of who knows whom and two distinct countries c_1 and c_2.

Question: Is there a person from c_1 who is a transitive acquaintance of a person from c_2?

We will show that *CA*$\preceq_P$*TA*, by giving a function which maps any instance of *CA* into an instance of *TA*.

We start by building a graph as we did in *TA*. Now we add a person p_1 such that p_1 knows all people from c_1 and nobody else. Similarly, we add p_2 for c_2. Clearly, this means $O(n)$ more edges. We now ask the *TA* question on p_1 and p_2. This function is computable in polynomial time. Now, if some person q_1 in c_1 transitively knows some person q_2 in c_2, then p_1 knows q_1 who transitively knows q_2 who knows p_2. Hence p_1 knows p_2. Conversely, if p_1 knows p_2, then p_1 knows somebody q_1 (who must be from c_1) who transitively knows p_2. But p_2 knows only people of c_2 (which is different from c_1) and hence there must be a person q_2 from c_2 who knows p_2 and such that q_1 transitively knows q_2. Hence there exist q_1 from c_1 and q_2 from c_2 who are transitive acquaintances. Hence, $w \in CA \Leftrightarrow f(w) \in TA$ and therefore $CA \preceq_P TA$. But $TA \in P$ and hence $CA \in P$.

$\square$

Before we conclude this section we give a few properties of the reducible relation.

Proposition: The reducible relation ($\preceq_P$) is reflexive and transitive. Also $P \times NP \subseteq \preceq_P$.

6.5 *NP*-completeness

The main use of reducibility, is however not in *P*, but in *NP*. The following theorem can easily be proved:

Theorem: If $L_1 \preceq_P L_2$ and $L_2 \in NP$ then $L_1 \in NP$.

Reducibility is basically a measure of how hard a problem is. When we say that $L_1 \preceq_P L_2$, we are basically saying that (ignoring any 'insignificant' polynomial delays) deciding L_2 is at least as hard as deciding L_1. Is there a problem which is the hardest in *NP*?

Assume there is, and let us call the problem X. Thus, for any language $L \in NP$, $L \preceq_P X$. Now assume that somebody manages to show that $X \in P$. By the theorem we have proved earlier, $L \preceq_P X$ and $X \in P$ implies that $L \in P$. Hence, the whole of *NP* would collapse to P. Therefore, if such a problem exists, researchers can concentrate exclusively on whether $X \in P$. There turns out to be not just one, but a whole family of such problems. We call this family of functions *NP*-complete:

Definition: A language L is said to be *NP*-complete if:

- $L \in NP$
- For every language $L' \in NP$, $L' \preceq_P L$.

The property that if we find an *NP*-complete problem to lie in P then the whole of *NP* would collapse to P, can now be proved.

Theorem: If *NP*-complete $\cap\, P \neq \emptyset$ then $P = NP$.

Proof: Recall that if $A \preceq_P B$ and $B \in P$ then $A \in P$.

Assume *NP*-complete $\cap\, P \neq \emptyset$ and let X be one element from this set. Consider any $Y \in NP$. By the definition of *NP*-completeness, $Y \preceq_P X$. But $X \in P$. Hence $Y \in P$. Therefore $NP \subseteq P$. We also proved that $P \subseteq NP$ and therefore $P = NP$.

□

Assume we know at least one *NP*-complete problem X. If we find an *NP* hard problem Y such that X is reducible to Y, then Y is at least as hard as X and Y should therefore also classify as 'the hardest problem in *NP*'. The following theorem proves this result.

Theorem: If L is *NP*-complete and $L \preceq_P L'$ (where $L' \in NP$) then L' is also *NP*-complete.

Proof: To prove that L' is *NP*-complete, we need to prove two properties: that it is in *NP* and that all *NP* hard problems are reducible to L'. The first is given in the hypothesis. Now consider any problem in *NP* L''. Since L is *NP*-complete, we know that $L'' \preceq_P L$. But we also know that $L \preceq_P L'$. Hence, by transitivity of $\preceq_P$, we get $L'' \preceq_P L'$.

□

We will now end this chapter by the grand proof of this part of the course. Originally given in a pioneering paper by Cook in 1971, this theorem gives us a first *NP*-complete problem which we can then use to prove that other problems are *NP*-complete.

6.6 Cook's Theorem

Before we state the theorem and proof, we need some background information about Boolean logic, which is the domain of the problem given in Cook's theorem.

Given a set of Boolean variables V, we call a total function $s : V \to \{T, F\}$ a *truth assignment*. We say that variable v is true under s if $s(v) = T$. Otherwise we say that v is false under s.

Literals can be either a variable, or a variable with a bar over it $\overline{v}$ (corresponding to *not* v). We can extend s to give the truth value of literals in the following manner: given a variable v, $s(\overline{v}) = T$ if $s(v) = F$, otherwise it is equal to F.

A set of literals is called a *clause* (corresponding to disjunction). The interpretation of s can be extended once more to get:

$$s(C) \quad = \quad \begin{cases} T & \text{if } \exists c \in C \cdot s(c) = T \\ F & \text{otherwise} \end{cases}$$

Finally, a set of clauses is said to be satisfiable under interpretation s if for every clause c, $s(c) = T$.

We can now state Cook's theorem.

Theorem: $SAT \in NP$-complete, where SAT is defined as:

Satisfiability (SAT)

INSTANCE: Given a set of variables V and a set of clauses C.

QUESTION: Is there a truth assignment s satisfying C?

Comment: If you recall Boolean logic, a collection of clauses is effectively an expression in disjunctive normal form. Also, all boolean expressions are expressible in disjunctive normal form. Hence, we are effectively showing that checking whether a general boolean expression can be satisfied is NP-complete.

Proof: That $SAT \in NP$ is not hard to prove. Consider the problem of deciding whether a given interpretation satisfies C. For every variable $v \in V$ we pass over C to replace literals v and $\overline{v}$ by T or F. Another pass over C replaces clauses by T or F. Finally a last pass is needed to check whether each clause has been satisfied or not. Clearly, this takes polynomial time. We can now construct a non-deterministic machine which starts off by branching for every variable, choosing whether it is true or not. This guarantees that $SAT \in NP$.

We now need to prove that all problems in NP can be reduced to SAT. Consider $L \in NP$. We need to prove that $L \preceq_P SAT$. What do we know about L? The property defining NP guarantees that there is non-deterministic Turing machine which decides L in polynomial time. What we thus try to do, is transform a non-deterministic Turing machine into an instance of SAT. If this can be done in polynomial time, we can transform L into an instance of SAT via the non-deterministic Turing machine which recognizes L, whatever L is.

Let M be a terminating non-deterministic Turing machine such that $M = \langle K, \Sigma, \Gamma, q_0, q_Y, q_N, P\rangle$ and with polynomial time complexity $p(n)$. We now set up a set of Boolean clauses such that they can be satisfiable if and only if an input string is accepted by M. We then show that this construction is possible in polynomial time.

What boolean variables do we need? We need to remember the following information:

- In what state is M at time t? If we enumerate the states starting from q_0 up to $q_{\sharp(K)}$, we can have variables ${}_tQ_i$ which is true if and only if M is in state q_i at time step t;
- Which tape square is the head scanning at time t? We will use ${}_tH_i$ which is set to true if and only if the head is scanning square i at time t;
- What are the contents of the tape? If we enumerate the tape symbols as σ_1 to $\sigma_{\sharp(\Gamma)}$, we can use variables ${}_tS[i]_j$ which is true if and only if square i has symbol j at time t.

But we have to place bounds on the subscripts of the above variables. Recall that M has polynomial complexity $p(n)$. In this time it cannot use any tape square not in $S[-p(n)] \ldots S[p(n)]$. Similarly, the time variables cannot exceed $p(n)$. The result is that each subscript has $O(p(n))$ possible values. Hence, we have $O(p^2(n))+O(p^2(n))+O(p^3(n))$ variables which is $O(p^3(n))$ — a polynomial in n.

Now, we will give a set of clauses C which can be satisfied if and only if M accepts the input in at most $p(n)$ steps. Hence:

$$\begin{aligned} & w \in L \\ \Leftrightarrow\ & M \text{ accepts } x \\ \Leftrightarrow\ & M \text{ accepts } x \text{ in at most } p(n) \text{ steps} \\ \Leftrightarrow\ & C \text{ can be satisfied} \end{aligned}$$

We now set out to define the clauses. We will have four sets of clauses. Each will serve a specific function:

1. Machine invariants, split into three subclasses:
    (a) M can only be in one state at a time;
    (b) The head can only be at one position at a time;
    (c) Each square can only contain one piece of information at a time.
2. Initialization of the machine.

3. Application of program P to calculate the next state, position and tape contents.

4. Termination of the machine.

We now define the clauses.

1. Machine invariants:

   (a) One state at a time: The machine has to be in at least one state at any time t. For t ranging over: $0 \leq t \leq p(n) \ldots$
   $$\{_tQ_0,\ _tQ_2 \ldots _t Q_{|K|}\}$$
   But there may be no more than one state which is true. This is expressible as $\neg(_tQ_i \wedge_t Q_j)$ where $i \neq j$. In disjunctive form, this appears as:
   $$\{_t\overline{Q}_i,\ _t\overline{Q}_j\}$$
   where $0 \leq t \leq p(n)$ and $0 \leq i < j < |K|$.
   Note that we have $O(|K|^2)$ of these clauses. Hence, the encoding of the clauses of this type is polynomial in size.

   (b) The head is at one place at a time: This is almost identical to the previous case:
   $$\{_tH_{-p(n)}, \ldots _t H_{p(n)}\}$$
   where $0 \leq t \leq p(n)$ and ...
   $$\{_t\overline{H}_i,\ _t\overline{H}_j\}$$
   where $0 \leq t \leq p(n)$ and $-p(n) \leq i < j \leq p(n)$.
   As before, the encoding of these clauses only takes $O(p^2(n))$, which is a polynomial.

   (c) Tape squares can contain only one piece of information at a time:
   $$\{_tS[i]_0, \ldots _t S[i]_{|\Gamma|}\}$$
   where $0 \leq t \leq p(n)$ and $-p(n) \leq i \leq p(n)$ and ...
   $$\{_t\overline{S}[i]_r,\ _t\overline{Q}[i]_s\}$$
   where $0 \leq t \leq p(n)$ and $0 \leq r < s < |\Gamma|$ and and $-p(n) \leq i \leq p(n)$.
   As before, the encoding of these clauses only takes polynomial time.

2. Initialization: Initially, the head is on the 0th position, M is in the initial state q_0 and the tape contains the input string:

$$\begin{array}{rl} \{_0H_0\} & \\ \{_0Q_0\} & \\ \{_0S[-p(n)]_0\} & \qquad (\sigma_0 = \Box) \\ \vdots & \\ \{_0S[-1]_0\} & \end{array}$$

$$\begin{array}{rl} \{{}_0S[0]_i\} & \text{where the first symbol of the input is } \sigma_i \\ \vdots & \\ \{{}_0S[n-1]_j\} & \text{where the last symbol of the input is } \sigma_j \\ \{{}_0S[n]_0\} & \\ \vdots & \\ \{{}_0S[p(n)]_0\} & \end{array}$$

Again, notice that we have $O(p(n))$ clauses.

3. Calculation of next state: Consider a transition instruction $(q_j, \sigma_m, S) \in P(q_i, \sigma_l)$. In boolean logic terms we want to say that: for any time t and position s, if the state is q_i and the information at the current tape position s is σ_l, then we want to leave the head where it is, but change the information at the current position to σ_m and change the state to q_j.

   $$\begin{array}{c} ({}_tH_s \wedge_t S[s]_l \wedge_t Q_i) \Rightarrow \\ ({}_{t+1}H_s \wedge \ {}_{t+1}Q_j \wedge \ {}_{t+1}S[s]_m) \end{array}$$

   With some simple manipulation, we can get them into the right format:

   $$\begin{array}{r} \{{}_t\overline{H}_s,\ {}_t\overline{S}[s]_l,\ {}_t\overline{Q}_i,\ {}_{t+1}H_s\} \\ \{{}_t\overline{H}_s,\ {}_t\overline{S}[s]_l,\ {}_t\overline{Q}_i,\ {}_{t+1}Q_j\} \\ \{{}_t\overline{H}_s,\ {}_t\overline{S}[s]_l,\ {}_t\overline{Q}_i,\ {}_{t+1}S[s]_m\} \end{array}$$

   where $0 \leq t \leq p(n)$, $-p(n) \leq s \leq p(n)$.

   The rest of the tape contents must not change.

   $$({}_t\overline{H}_s \wedge_t S[s]_i) \Rightarrow_{t+1} S[s]_i$$

   This can be converted to:

   $$\{{}_t\overline{S}[s]_i,\ {}_tH_s,\ {}_{t+1}S[s]_i\}$$

   where $0 \leq t \leq p(n)$, $-p(n) \leq s \leq p(n)$ and $1 \leq i \leq \sharp(\Sigma)$.

   If we count the number of clauses, we still have a number proportional to a polynomial of the input size.

   We can construct similar triplets for instructions which move to the left or to the right.

4. Termination: Once, a program terminates (before $p(n)$ steps), we must keep the same state and leave the current position unchanged. If y and n are the positive and negative state indices, we add:

$$\begin{array}{rcl} ({}_tQ_y \wedge_t H_s \wedge_t S[s]_j) & \Rightarrow & ({}_{t+1}H_s \wedge_{t+1} S[s]_j \wedge_{t+1} Q_y) \\ ({}_tQ_n \wedge_t H_s \wedge_t S[s]_j) & \Rightarrow & ({}_{t+1}H_s \wedge_{t+1} S[s]_j \wedge_{t+1} Q_n) \end{array}$$

   This can be written as:

$$\begin{array}{c} \{{}_t\overline{Q}_y,\ {}_t\overline{H}_s,\ {}_t\overline{S}[s]_j,\ {}_{t+1}H_s\} \\ \{{}_t\overline{Q}_y,\ {}_t\overline{H}_s,\ {}_t\overline{S}[s]_j,\ {}_{t+1}S[s]_j\} \\ \{{}_t\overline{Q}_y,\ {}_t\overline{H}_s,\ {}_t\overline{S}[s]_j,\ {}_{t+1}Q_y\} \end{array}$$

   (and similarly for index n) where $0 \leq t \leq p(n)$, $-p(n) \leq s \leq p(n)$.

   Finally we want the set of clauses to be satisfiable only if we end up in the accepting state:

$$\{{}_{p(n)}Q_y\}$$

   The size of these clauses again does not exceed a polynomial function of the input size.

We now have constructed a collection of clauses which, upon input w, can be satisfied if and only if M accepts w. Hence, for any language $L \in NP$ we have defined a function f_L which transforms the language (or rather the non-deterministic machine accepting the language in polynomial time) into an instance of *SAT*. Also $w \in L \Leftrightarrow f(w) \in SAT$. Hence, if we can show that f_L can be computed in polynomial time, we have shown that for any $L \in NP$, $L \preceq_P SAT$ and thus *SAT* is *NP*-complete.

Note that once we know L (and hence M and $p(n)$), constructing f_L consists of nothing but replacing values into the long formulae given above. It is therefore polynomial as long as we do not have to produce output which is more than polynomial in length. But we have seen that each subdivision of clauses in at most a polynomial of n in length. Hence, so is their sum. Therefore f_L has polynomial complexity.

$\square$

6.7 More *NP*-Complete Problems

If we were to use a technique similar to the one used in Cook's theorem for the satisfiability problem, there probably would not be far too many results showing problems to be *NP*-complete. However, we can also use the polynomial reduction technique shown earlier. Since, at this point we only know of SAT to be *NP*-complete, we can only reduce SAT to our new problem but as we gather more and more *NP*-complete problems we have a wider repertoire to use to show *NP*-completeness.

6.7.1 3-Satisfiability

A restricted version of the satisfiability problem is the case when all the clauses are exactly 3 literals long. We will show that this problem is also *NP*-complete by reducing SAT into it.

3-Satisfiability (3SAT)

Instance: Given a set of variables V and a set of clauses C each of which is 3 literals long.

Question: Is there an assignment of V which satisfies C?

Theorem: *3SAT* is *NP*-complete.

Proof: We show that $3SAT \in NP$ and that $SAT \preceq_P 3SAT$ to conclude the desired result.

- $3SAT \in NP$: We have already shown that SAT is in *NP*. Hence, there is a non-deterministic polynomial Turing machine which decides SAT. But *3SAT* is just a restricted version of SAT and we can thus still use the same machine. Since such a machine exists, we conclude that $3SAT \in NP$.

- $SAT \preceq_P 3SAT$: We have to show that every instance of SAT can be reduced into an instance of *3SAT*.

  Consider a clause which is only two literals long: $\{\alpha, \beta\}$ (corresponding to $\alpha \vee \beta$). Clearly, this can be satisfied exactly when $\{\alpha, \beta, z\}$ can be satisfied for all values of z. How can we express this?

  $$\{\ \{\alpha, \beta, z\},\ \{\alpha, \beta, \overline{z}\}\ \}$$

  Using the definition of satisfaction, we can easily prove the equivalence of the two sets of clauses.

  In the case of 1 literal-long clauses, we use a similar trick to transform $\{\alpha\}$ into

  $$\{\ \{\alpha, z, z'\},\ \{\alpha, \overline{z}, z'\},\ \{\alpha, z, \overline{z}'\},\ \{\alpha, \overline{z}, \overline{z}'\}\ \}$$

  What about clauses with more than three literals? Let us first consider the case with four literals. We note that the clause $a \vee b \vee c \vee d$ can be satisfied exactly when $(a \vee b \vee z) \wedge (\overline{z} \vee c \vee d)$ can be satisfied.

  Hence, we can transform $\{a, b, c, d\}$ into $\{\ \{a, b, z\},\ \{\overline{z}, c, d\}\ \}$.

  But what about longer clauses? The answer is to apply the rule for 4 literals repeatedly. Thus, in the case of 5 literals we have: $a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5$ which can be satisfied exactly when $a_1 \vee a_2 \vee z_1) \wedge (\overline{z} \vee a_3 \vee a_4 \vee a_5)$

can be satisfied. We can now apply the 4-literal clause rule again to the second clause and get that the original 5-literal clause is equivalent to:

$$\begin{array}{rl} & (a_1 \vee a_2 \vee z_1) \\ \wedge & (\overline{z}_1 \vee a_3 \vee z_2) \\ \wedge & (\overline{z}_2 \vee a_4 \vee a_5) \end{array}$$

In general we get $\{a_1, \ldots a_n\}$ transformed into:

$$\{\ \{a_1, a_2, z_1\},\ \{\overline{z}_{n-3}, a_{n-1}, a_n\}\ \} \quad \cup \quad \{\{\overline{z}_i, a_{i+2}, z_{i+1}\} \mid 1 \leq i \leq n-4\}$$

We have thus defined a function f, which, given an instance in *SAT*, produces an equivalent instance of *3SAT*. The informal reasoning above should convince you that an instance x of *SAT* can be satisfied if and only if $f(x)$ can be satisfied in *3SAT*, or $x \in SAT \Leftrightarrow f(x) \in 3SAT$.

But is the transformation f polynomial? As in the case of *SAT* we notice that an algorithm that performs the replacements is trivial and can be designed to work in polynomial time, provided it does not have to produce an exponential amount of output. In the cases of single or double literal clauses, we are only adding a constant number of variables and new clauses per such clause. Hence, the increase in size is only $O(1)$ per clause. What about longer clauses? For every such clause, we are adding $O(n)$ new variables, and $O(n)$ new clauses (all of fixed length) — an increase of $O(n)$ per clause. It should thus be clear that the total output is $O(n^2)$, making f polynomial and thus $SAT \preceq_P 3SAT$.

□

6.7.2 Clique

Clique (CLIQUE)

INSTANCE: Given a directed graph G and a positive integer k.

QUESTION: Does G contain a clique of size k?

Note that a *clique* is a set of nodes which are all interconnected.

Theorem: *CLIQUE* is *NP*-complete.

Proof: We reduce *3SAT* to *CLIQUE* to show that it is *NP*-complete.

- First, we must show that *CLIQUE* is in *NP*.

  A polynomial time non-deterministic algorithm to decide *CLIQUE* is the following:

1. Choose, non-deterministically, a subset of k nodes from G.
2. Check whether the chosen subset is a clique.
3. Accept the input if it is, reject otherwise.

Note that to check whether a given subset is a clique we only need $O(n^2)$ operations to check every element with every other element. Hence $\mathit{CLIQUE} \in$ NP.

- We now want to show that $\mathit{3SAT} \preceq_P \mathit{CLIQUE}$.

  Given a set of 3-literal clauses, we want to construct a graph and a constant k such that the transformation f takes polynomial time and a set of clauses C can be satisfied ($C \in \mathit{3SAT}$) if and only if the graph has a clique of size k ($f(C) \in \mathit{CLIQUE}$).

  Consider a set of clauses C:

  $$\{\ \{\alpha_1, \beta_1, \gamma_1\},\ \dots \{\alpha_m, \beta_m, \gamma_m\},\ \}$$

  Now consider the graph which has $3m$ nodes — one for every α_i, β_i and γ_i. As for edges, two nodes will be connected if and only if:

  – they come from different clauses; and
  – they are not contradictory (they are not x and $\overline{x}$, where x is a variable).

  Thus, for example, the set of clauses $\{\{x, y, \overline{y}\}, \{\overline{x}, x, y\}\}$ has 6 nodes n_1 to n_6 corresponding to the 6 literals in the set of clauses respectively (we have renamed them to avoid having to deal with two nodes named x). The set of edges is:

  $$\{(n_1, n_5), (n_1, n_6), (n_2, n_4), (n_2, n_5), (n_2, n_6), (n_3, n_4), (n_3, n_5)\}$$

  The constant k is set to the number of clauses m.

  Now we need to show that $C \in \mathit{3SAT} \Leftrightarrow f(C) \in \mathit{CLIQUE}$. Assume that C is in *3SAT* (and can thus be satisfied). Then there is at least one member of every clause which is true, such that none of these are contradictory. The construction guarantees that these form a clique of size k.

  On the other hand, assume that the graph has a clique of size k. No two of these can be nodes from the same clause (otherwise they would not be connected). Also there are no contradictory nodes in this clique (again, by the definition of the construction). Hence, there is at least one assignment of variables (as appear in the node labels) which makes sure that each one of the clauses is true (since each clause has a node in the clique). Hence, the original set of 3-literal clauses is satisfiable and thus in *3SAT*.

  But can the construction be done in polynomial time? As usual, this construction is simple to construct and we only need to check that it is of polynomial size. Note that, given an instance of *3SAT,* we construct a graph with $O(n)$ nodes, and $O(n^2)$ edges. Hence, the size of $f(C)$ is polynomially related to the size of C.

This ensures that *3SAT*$\preceq_P$*CLIQUE*.

Using the reduction theorem proved earlier, and the earlier result that *3SAT* is *NP*-complete, we can conclude that *CLIQUE* is also *NP*-complete.

□

6.8 Exercises

1. *(Easy)* Show that the following problem is in P:

   Triangle (TRI)

   INSTANCE: Given three numbers.

   QUESTION: Can these three numbers be the lengths of the three sides of a triangle?

2. *(Moderate)* Show that the travelling salesman's problem but with distinct start and terminal cities is also in *NP*. We have mentioned that the travelling salesman problem is *NP*-complete. Assuming it is so, can you outline, how to show that the modified problem given here is also, *NP*-complete?

3. *(Moderate)* Show that the following problem is in P:

   Shortest path (SP)

   INSTANCE: Given a directed graph G, cost function c associating with each edge a positive whole number, two nodes n_1 and n_2 and a bound $B \in \mathbb{N}$.

   QUESTION: Is there a path in G from n_1 to n_2 costing less than B?

   Use the answer to prove that *CkA*, as defined below, is also in P.

   Given an acquaintance relation A and a number k, we say that p is a k-acquaintance of q if $(p, q) \in \bigcup_{i=1}^{k} A^k$. In other words, there is a list of up to $k - 1$ persons r_1 to r_i such that p knows r_1 who knows r_2 ... who knows r_i who knows q.

   Country k-acquaintance (CkA)

   INSTANCE: Given set of people and their nationality, a list of who knows whom, two distinct countries c_1 and c_2 and a bound k.

   QUESTION: Is there a person from c_1 who is a k-acquaintance of a person from c_2?

   As an aside, it is interesting to note that the minimum k such that everybody on Earth is k-acquainted to everybody else is very low — possibly as low as 10. This is because of the hierarchical way our society is organized.

Everybody in the world knows somebody important in their village who knows somebody important in their region who know somebody important in the country who knows the official head of state. Since the heads of states have met quite a few other heads of states (and are hence acquaintances), you can then follow down any other person on earth through their head of state!

4. *(Difficult)* Consider the following single player game: For a particular value of $\alpha \in \mathbb{N}$, the game is played on a square board made up of $n \times n$ squares (where $n^2 > \alpha$). Initially α pieces are placed on the board. The player makes moves, each of which consists of a piece jumping over an adjacent piece vertically or horizontally onto an empty square. The piece jumped over is removed. The aim of the game is to end up with only one piece.

   α-Solitaire (αSOL)

   Instance: Given a value n such that $n > \alpha$ and an initial position I.

   Question: Is the game winnable from position I?

   Prove that, for a constant α, $\alpha SOL \in P$.

   Hint: You can show that $\alpha SOL \preceq_P PATH$. Start by informally showing that checking whether B' is a 'valid next position' of B can be calculated in polynomial time.

Bibliography

[1] Daniel E. Cohen. *Computability and Logic.* Mathematics and its Applications. Ellis Horwood Ltd, 1987.

[2] Daniel I.A. Cohen. *Introduction to Computer Theory*, volume 1997. John Wiley & Sons Inc, second edition, 2003.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Bell Telephone Laboratories, 1979.

[4] David Harel. *Algorithmics: The Spirit of Computing.* Addison-Wesley Publishing Company Inc, second edition, 1993.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, second edition, 2001.

[6] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, 1979.

[7] Dean Kelley. *Automata and Formal Languages: An Introduction.* Prentice-Hall, 1995.

[8] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, second edition, 1998.

[9] John C. Martin. *Introduction to Languages and the Theory of Computation.* McGraw-Hill Higher Education, third edition, 2003.

[10] Nicholas Pippenger. *Theories of Computability.* Cambridge University Press, 1997.

[11] V.J. Rayward-Smith. *A First Course in Computability.* Computer Science Texts. McGraw-Hill, 1995.

[12] V.J. Rayward-Smith. *A First Course in Formal Language Theory.* McGraw-Hill, second edition, 1995.

[13] György E. Révész. *Introduction to Formal Languages*. Dover Publications, 1991.

[14] Henry M. Walker. *The Limits of Computing*. Jones and Bartlett Publishers, 1994.