List of Lectures

1 Course overview and mathematical foundations 5
   1.1 Course overview ................................................. 5
   1.2 Sets and countability ............................................. 6
   1.3 Alphabets, strings, and languages ............................. 11

2 Countability for languages; deterministic finite automata 15
   2.1 Countability and languages ..................................... 15
   2.2 Deterministic finite automata ................................. 18

3 Nondeterministic finite automata 25
   3.1 Nondeterministic finite automata basics .................... 25
   3.2 Equivalence of NFAs and DFAs ................................. 29

4 Regular operations and regular expressions 37
   4.1 Regular operations ................................................. 37
   4.2 Other closure properties of regular languages .............. 43
   4.3 Regular expressions .............................................. 44

5 Proving languages to be nonregular 51
   5.1 The pumping lemma (for regular languages) ............... 51
   5.2 Using the pumping lemma to prove nonregularity .......... 54
   5.3 Nonregularity from closure properties ...................... 59

6 Further discussion of regular languages 61
   6.1 Other operations on languages ................................. 61
   6.2 Example problems concerning regular languages ........... 66

7 Context-free grammars and languages 73
   7.1 Definitions of CFGs and CFLs ................................. 73
   7.2 Basic examples .................................................. 76
   7.3 A tougher example ............................................... 80
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Start Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Parse trees, ambiguity, and Chomsky normal form</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>8.1 Left-most derivations and parse trees</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>8.2 Ambiguity</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>8.3 Chomsky normal form</td>
<td>87</td>
</tr>
<tr>
<td>9</td>
<td>Closure properties for context-free languages</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>9.1 Closure under the regular operations</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>9.2 Every regular language is context-free</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>9.3 Intersections of regular and context-free languages</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>9.4 Prefixes, suffixes, and substrings</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>Proving languages to be non-context-free</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>10.1 The pumping lemma (for context-free languages)</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>10.2 Using the context-free pumping lemma</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>10.3 Non-context-free languages and closure properties</td>
<td>110</td>
</tr>
<tr>
<td>11</td>
<td>Pushdown automata</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>11.1 Pushdown automata</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>11.2 Further examples</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>11.3 Equivalence of PDAs and CFGs</td>
<td>122</td>
</tr>
<tr>
<td>12</td>
<td>Stack machines</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>12.1 Nondeterministic stack machines</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>12.2 Deterministic stack machines</td>
<td>129</td>
</tr>
<tr>
<td>13</td>
<td>Stack machine computations, languages, and functions</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>13.1 Stack machine configurations and computations</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>13.2 Languages and functions from DSMs</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>13.3 New computable functions from old</td>
<td>143</td>
</tr>
<tr>
<td>14</td>
<td>Turing machines and their equivalence to stack machines</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>14.1 Turing machine definitions</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>14.2 Equivalence of DTMs and DSMs</td>
<td>156</td>
</tr>
<tr>
<td>15</td>
<td>Encodings; examples of decidable languages</td>
<td>161</td>
</tr>
<tr>
<td></td>
<td>15.1 Encodings of interesting mathematical objects</td>
<td>161</td>
</tr>
<tr>
<td></td>
<td>15.2 Decidability of formal language problems</td>
<td>168</td>
</tr>
<tr>
<td>16</td>
<td>Universal stack machines and a non-semidecidable language</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>16.1 An encoding scheme for DSMs</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>16.2 A universal stack machine</td>
<td>178</td>
</tr>
</tbody>
</table>
16.3 A non-semidecidable language ................................................. 182

17 Undecidable languages ......................................................... 185
  17.1 Undecidability proofs through contradiction ....................... 185
  17.2 Proving undecidability through reductions ....................... 189

18 Further discussion of computability ....................................... 197
  18.1 Closure properties of decidable and semidecidable languages .... 197
  18.2 The range of a computable function ................................. 202

19 Time-bounded computations ................................................ 207
  19.1 DTIME and time-constructible functions ............................ 208
  19.2 The time-hierarchy theorem ........................................... 210
  19.3 Polynomial and exponential time .................................... 212

20 NP, polynomial-time mapping reductions, and NP-completeness .... 215
  20.1 The complexity class NP ............................................. 215
  20.2 Polynomial-time reductions and NP-completeness ............... 219

21 Ladner’s theorem ................................................................ 223
  21.1 Some useful concepts and ideas ....................................... 223
  21.2 Proof of Ladner’s theorem ............................................. 229
Lecture 1

Course overview and mathematical foundations

1.1 Course overview

This course is about the theory of computation, which deals with mathematical properties of abstract models of computation and the problems they solve. An important idea to keep in mind as we begin the course is this:

*Computational problems, devices, and processes can themselves be viewed as mathematical objects.*

We can, for example, think about each program written in a particular programming language as a single element in the set of all programs written in that language, and we can investigate not only those programs that might be interesting to us, but also properties that must hold for all programs. We can also consider problems that some computational models can solve and that others cannot.

The notion of a computation is very general. Examples of things that can be viewed or described as computations include the following:

- Computers running programs (of course).
- Networks of computers running protocols.
- People performing calculations with a pencil and paper.
- Proofs of theorems (in a sense to be discussed from time to time throughout this course).
- Certain biological processes.
One could debate the definition of a computation (which is not something we will do), but a reasonable starting point for a definition is that a computation is a manipulation of symbols according to a fixed set of rules.

One interesting connection between computation and mathematics, which is particularly important from the viewpoint of this course, is that mathematical proofs and computations performed by the models we will discuss throughout this course have a low-level similarity: they both involve symbolic manipulations according to fixed sets of rules. Indeed, fundamental questions about proofs and mathematical logic have played a critical role in the development of theoretical computer science.

We will begin the course working with very simple models of computation (finite automata, regular expressions, context-free grammars, and related models), and later on we will discuss more powerful computational models (e.g., the stack machine and Turing machine models of computation). Before we get to any of these models, however, it is appropriate that we discuss some of the mathematical foundations and definitions upon which our discussions will be based.

1.2 Sets and countability

It is assumed throughout these notes that the reader is familiar with naive set theory and basic propositional logic.

Naive set theory treats the concept of a set to be self-evident. This will not be problematic for the purposes of this course, but it does lead to problems and paradoxes—such as Russell’s paradox—when it is pushed to its limits. Here is one formulation of Russell’s paradox, in case you are interested:

Russell’s paradox. Let $S$ be the set of all sets that are not elements of themselves:

$$S = \{ T : T \notin T \}.$$

Is it the case that $S$ is an element of itself?

If $S \in S$, then by the condition that a set must satisfy to be included in $S$, it must be that $S \notin S$. On the other hand, if $S \notin S$, then the definition of $S$ says that $S$ is to be included in $S$. It therefore holds that $S \in S$ if and only if $S \notin S$, which is a contradiction.

If you want to avoid this sort of paradox, you need to replace naive set theory with axiomatic set theory, which is quite a bit more formal and disallows objects such as the set of all sets (which is what opens the door to let in Russell’s paradox). Set theory is the foundation on which mathematics is built, so axiomatic set theory is the better choice for making this foundation sturdy. Moreover, if you really wanted
to reduce mathematical proofs to a symbolic form that a computer can handle, something along the lines of axiomatic set theory would be needed.

On the other hand, axiomatic set theory is quite a bit more complicated than naive set theory, and it is also outside of the scope of this course. Fortunately, there will be no specific situations that arise in this course for which the advantages of axiomatic set theory over naive set theory explicitly appear, and for this reason we are safe in thinking about set theory from the naive point of view—and meanwhile we can trust that everything would work out the same way if axiomatic set theory had been used instead.

The size of a finite set is the number of elements it contains. If $A$ is a finite set, then we write $|A|$ to denote this number. For example, the empty set is denoted $\emptyset$ and has no elements, so $|\emptyset| = 0$. A couple of simple examples are

$$|\{a, b, c\}| = 3 \quad \text{and} \quad |\{1, \ldots, n\}| = n. \quad \text{(1.1)}$$

In the second example, we are assuming $n$ is a positive integer, and $\{1, \ldots, n\}$ is the set containing the positive integers from 1 to $n$.

Sets can also be infinite. For example, the set of natural numbers\footnote{Some people choose not to include 0 in the set of natural numbers, but for this course we will include 0 as a natural number. It is not right or wrong to make such a choice, it is only a definition, and what is most important is that we make clear the precise meaning of the terms we use.} $\mathbb{N} = \{0, 1, 2, \ldots\}$

(1.2)
is infinite, as are the sets of integers

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}, \quad \text{(1.3)}$$

and rational numbers

$$\mathbb{Q} = \left\{ \frac{n}{m} : n, m \in \mathbb{Z}, m \neq 0 \right\}. \quad \text{(1.4)}$$

The sets of real and complex numbers are also infinite, but we won’t define these sets here because they won’t play a major role in this course and the definitions are a bit more complicated than one might initially expect.

While it is sometimes sufficient to say that a set is infinite, we will require a more refined notion, which is that of a set being countable or uncountable.

**Definition 1.1.** A set $A$ is countable if either (i) $A$ is empty, or (ii) there exists an onto (or surjective) function of the form $f : \mathbb{N} \to A$. If a set is not countable, then we say that it is uncountable.
These three statements are equivalent for any choice of a set $A$:

1. $A$ is countable.
2. There exists a one-to-one (or injective) function of the form $g : A \rightarrow \mathbb{N}$.
3. Either $A$ is finite or there exists a one-to-one and onto (or bijective) function of the form $h : \mathbb{N} \rightarrow A$.

It is not obvious that these three statements are actually equivalent, but it can be proved. We will, however, not discuss the proof.

**Example 1.2.** The set of natural numbers $\mathbb{N}$ is countable. Of course this is not surprising, but it is sometimes nice to start out with a simple example. The fact that $\mathbb{N}$ is countable follows from the fact that we may take $f : \mathbb{N} \rightarrow \mathbb{N}$ to be the identity function, meaning $f(n) = n$ for all $n \in \mathbb{N}$, in Definition 1.1. Notice that substituting $f$ for the function $g$ in statement 2 makes that statement true, and likewise for statement 3 when $f$ is substituted for the function $h$.

The function $f(n) = n$ is not the only function that works to establish that $\mathbb{N}$ is countable. For example, the function

$$f(n) = \begin{cases} n + 1 & \text{if } n \text{ is even} \\ n - 1 & \text{if } n \text{ is odd} \end{cases}$$

also works. The first few values of this function are

$$f(0) = 1, \quad f(1) = 0, \quad f(2) = 3, \quad f(3) = 2,$$

and it is not too hard to see that this function is both one-to-one and onto. There are (infinitely) many other choices of functions that work equally well to establish that $\mathbb{N}$ is countable.

**Example 1.3.** The set $\mathbb{Z}$ of integers is countable. To prove that this is so, it suffices to show that there exists an onto function of the form

$$f : \mathbb{N} \rightarrow \mathbb{Z}.$$  \hspace{1cm} (1.7)

As in the previous example, there are many possible choices of $f$ that work, one of which is this function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \frac{n+1}{2} & \text{if } n \text{ is odd} \\ -\frac{n}{2} & \text{if } n \text{ is even}. \end{cases}$$  \hspace{1cm} (1.8)
Thus, we have

$$ f(0) = 0, \quad f(1) = 1, \quad f(2) = -1, \quad f(3) = 2, \quad f(4) = -2, \quad (1.9) $$

and so on. This is a well-defined function\(^2\) of the correct form \(f : \mathbb{N} \to \mathbb{Z}\), and it is onto; for every integer \(m\), there is a natural number \(n \in \mathbb{N}\) so that \(f(n) = m\), as is quite evident from the pattern in (1.9).

**Example 1.4.** The set \(\mathbb{Q}\) of rational numbers is countable, which we can prove by defining an onto function taking the form \(f : \mathbb{N} \to \mathbb{Q}\). Once again, there are many choices of functions that would work, and we’ll pick just one.

First, imagine that we create a sequence of ordered lists of numbers, starting like this:

- **List 0:** 0
- **List 1:** \(-1, 1\)
- **List 2:** \(-2, -\frac{1}{2}, \frac{1}{2}, 2\)
- **List 3:** \(-3, -\frac{3}{2}, -\frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{3}{2}, 3\)
- **List 4:** \(-4, -\frac{4}{3}, -\frac{2}{3}, -\frac{1}{4}, \frac{1}{4}, \frac{3}{4}, \frac{4}{3}, 4\)
- **List 5:** \(-5, -\frac{5}{2}, -\frac{5}{3}, -\frac{5}{4}, -\frac{5}{5}, -\frac{2}{5}, -\frac{1}{5}, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, \frac{5}{3}, \frac{5}{4}, \frac{5}{5}, 5\)

and so on. In general, for \(n \geq 1\) we let the \(n\)-th list be the sorted list of all numbers that can be written as

$$ \frac{k}{m}, \quad (1.10) $$

where \(k, m \in \{-n, \ldots, n\}, \ m \neq 0\), and the value of the number \(k/m\) does not already appear in one of the previous lists. The lists get longer and longer, but for every natural number \(n\) it is surely the case that the corresponding list is finite.

Now consider the single list obtained by concatenating all of the lists together, starting with List 0, then List 1, and so on. Because the lists are finite, we have no problem defining the concatenation of all of them, and every number that appears in any one of the lists above will also appear in the single concatenated list. For instance, this single list begins as follows:

$$ 0, -1, 1, -2, -\frac{1}{2}, \frac{1}{2}, 2, -3, -\frac{3}{2}, -\frac{3}{3}, -\frac{1}{3}, 1, \frac{1}{3}, \frac{2}{3}, 3, 4, -\frac{4}{3}, -\frac{3}{4}, \ldots \quad (1.11) $$

and naturally the complete list is infinitely long. Finally, let \(f : \mathbb{N} \to \mathbb{Q}\) be the function we obtain by setting \(f(n)\) to be the number in position \(n\) in the infinite list

---

\(^2\) We can think of well-defined as meaning that there are no “undefined” values, and moreover that every reasonable person that understands the definition would agree on the values the function takes, irrespective of when and where they lived.
we just defined, starting with position 0. For example, \( f(0) = 0, f(1) = -1, \) and \( f(8) = -3/2. \)

Even though we didn’t write down an explicit formula for the function \( f \), it is a well-defined function of the proper form \( f : \mathbb{N} \to \mathbb{Q} \). Moreover, it is an onto function: for any rational number you choose, you will eventually find that rational number in the list constructed above. It is therefore the case that \( \mathbb{Q} \) is countable. The function \( f \) also happens to be one-to-one, although we don’t need to know this to conclude that \( \mathbb{Q} \) is countable.

It is natural at this point to ask a question: Is every set countable? The answer is “no,” and we will soon see an example of an uncountable set. First, however, we will need the following definition.

**Definition 1.5.** For any set \( A \), the **power set** of \( A \) is the set \( \mathcal{P}(A) \) containing all subsets of \( A \):

\[
\mathcal{P}(A) = \{ B : B \subseteq A \}. \tag{1.12}
\]

For example, the power set of \( \{1, 2, 3\} \) is

\[
\mathcal{P}({1, 2, 3}) = \{ \emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \}. \tag{1.13}
\]

Notice, in particular, that the empty set \( \emptyset \) and the set \( \{1, 2, 3\} \) itself are contained in the power set \( \mathcal{P}({1, 2, 3}) \). For any finite set \( A \), the power set \( \mathcal{P}(A) \) always contains \( 2^{|A|} \) elements, which is why it is called the power set.

Also notice that there is nothing that prevents us from taking the power set of an infinite set. For instance, \( \mathcal{P}(\mathbb{N}) \), the power set of the natural numbers, is the set containing all subsets of \( \mathbb{N} \). This set, in fact, is our first example of an uncountable set.

**Theorem 1.6 (Cantor).** The power set of the natural numbers, \( \mathcal{P}(\mathbb{N}) \), is uncountable.

**Proof.** Assume toward contradiction that \( \mathcal{P}(\mathbb{N}) \) is countable, which implies that there exists an onto function of the form \( f : \mathbb{N} \to \mathcal{P}(\mathbb{N}) \). From this function we may define\(^3\) a subset of natural numbers as follows:

\[
S = \{ n \in \mathbb{N} : n \notin f(n) \}. \tag{1.14}
\]

Because \( S \) is a subset of \( \mathbb{N} \), it is the case that \( S \in \mathcal{P}(\mathbb{N}) \). We have assumed that \( f \) is onto, so there must therefore exist a natural number \( m \in \mathbb{N} \) such that \( f(m) = S \). Fix such a choice of \( m \) for the remainder of the proof.

---

\(^3\) This definition makes sense because, for each \( n \in \mathbb{N} \), \( f(n) \) is an element of \( \mathcal{P}(\mathbb{N}) \), which means it is a subset of \( \mathbb{N} \). It is therefore true that either \( n \in f(n) \) or \( n \notin f(n) \), so if you knew what the function \( f \) was, you could determine whether or not a given number \( n \) is contained in \( S \) or not.
Lecture 1

Now we may ask ourselves a question: Is \( m \) contained in \( S \)? We have

\[ [m \in S] \Leftrightarrow [m \in f(m)] \]  \hspace{1cm} (1.15)

because \( S = f(m) \). On the other hand, by the definition of the set \( S \) we have

\[ [m \in S] \Leftrightarrow [m \notin f(m)]. \]  \hspace{1cm} (1.16)

It is therefore the case that

\[ [m \in f(m)] \Leftrightarrow [m \notin f(m)], \]  \hspace{1cm} (1.17)

or, equivalently,

\[ [m \in S] \Leftrightarrow [m \notin S], \]  \hspace{1cm} (1.18)

which is a contradiction.

Having obtained a contradiction, we conclude that our assumption that \( \mathcal{P}(\mathbb{N}) \) is countable was wrong, so the theorem is proved.

The method used in the proof above is called \textit{diagonalization}, for reasons we will discuss later in the course. This is a fundamentally important proof technique in the theory of computation. Using this technique, one can prove that the sets \( \mathbb{R} \) and \( \mathbb{C} \) of real and complex numbers are uncountable—the central idea of the proof is the same as the proof above, but the fact that some real numbers have multiple decimal representations (or, for any other choice of a base \( b \), that some real numbers have multiple base \( b \) representations) makes the proof a bit more complicated.

### 1.3 Alphabets, strings, and languages

The last thing we will do for this lecture is to introduce some terminology that you may already be familiar with from other courses.

First let us define what we mean by an \textit{alphabet}. Intuitively speaking, when we refer to an alphabet, we mean a collection of symbols that could be used for writing, encoding information, or performing calculations. Mathematically speaking, there is not much to say—there is nothing to be gained by defining what is meant by the words \textit{symbol}, \textit{writing}, \textit{encoding information}, or \textit{calculation} in this context, so instead we keep things as simple as possible and stick to the mathematical essence of the concept.

\textbf{Definition 1.7.} An \textit{alphabet} is a finite and nonempty set.
Typical names used for alphabets in this course are capital Greek letters such as $\Sigma$, $\Gamma$, and $\Delta$. We refer to elements of alphabets as *symbols*, and we will often use lower-case letters appearing at the beginning of the Roman alphabet, such as $a$, $b$, $c$, and $d$, as variable names when referring to symbols.

Our favorite alphabet in this course will be the *binary alphabet* $\Sigma = \{0, 1\}$. Sometimes we will refer to the *unary alphabet* $\Sigma = \{0\}$ that has just one symbol. Although it is not a very efficient choice for encoding information, the unary alphabet is a valid alphabet—it’s an excellent choice for an alphabet if you’re stuck in a prison cell and want to count the days. We can also imagine an abstract sort of alphabet $\Sigma = \{0, 1, \ldots, n - 1\}$, where $n$ is a large positive integer, like $n = 1,000,000$. Of course we do not need to actually think up one million different symbols to contemplate such an alphabet in a mathematical sense. Alphabets could consist of other symbols, such as $\Sigma = \{A, B, C, \ldots, Z\}$, $\Sigma = \{\heartsuit, \diamondsuit, \spadesuit, \clubsuit\}$, or $\Sigma = \{\text{a}, \text{b}, \text{c}, \text{d}\}$ but the specific symbols that actually appear in the alphabets we consider won’t actually matter all that much. From a mathematical point of view, there’s really nothing special about the alphabets $\Sigma = \{\heartsuit, \diamondsuit, \spadesuit, \clubsuit\}$ and $\Sigma = \{\text{a}, \text{b}, \text{c}, \text{d}\}$ in comparison to the alphabet $\Sigma = \{0, 1, 2, 3\}$, for instance. For this reason, when it is convenient to do so, we may assume without loss of generality that a given alphabet we’re working with takes the form $\Sigma = \{0, \ldots, n - 1\}$ for some positive integer $n$.

Next we have *strings*, which are defined with respect to a particular alphabet as follows.

**Definition 1.8.** Let $\Sigma$ be an alphabet. A *string* over the alphabet $\Sigma$ is a finite, ordered sequence of symbols from $\Sigma$. The *length* of a string is the total number of symbols in the sequence.

For example, 11010 is a string of length 5 over the binary alphabet $\Sigma = \{0, 1\}$. It is also a string over the alphabet $\Gamma = \{0, 1, 2\}$ that doesn’t happen to include the symbol 2. On the other hand,

$$01010101 \cdots \quad (\text{repeating forever})$$

(1.19)

is not a string because it is not finite. There are situations where it is interesting or useful to consider infinitely long sequences of symbols like this, but we just won’t refer to them as *strings*.

There is a special string, called the *empty string* and denoted $\varepsilon$, that has no symbols in it (and therefore it has length 0). It is a string over every alphabet.

We will typically use lower-case letters appearing near the end of the Roman alphabet, such as $u$, $v$, $w$, $x$, $y$, and $z$, as names that refer to strings. Saying that these are *names that refer to strings* is just meant to clarify that we’re not thinking
Lecture 1

about \(u, v, w, x, y,\) and \(z\) as being single symbols from the Roman alphabet in this context. Because we’re essentially using symbols and strings to communicate ideas about symbols and strings, there is hypothetically a chance for confusion, but once we establish some simple conventions this will not be an issue. If \(w\) is a string, we denote the length of \(w\) as \(|w|\).

Finally, the term language refers to any collection of strings over some alphabet.

**Definition 1.9.** Let \(\Sigma\) be an alphabet. A language over \(\Sigma\) is a set of strings, with each string being a string over the alphabet \(\Sigma\).

Notice that there has to be an alphabet associated with a language. We would not, for instance, consider a set of strings that includes infinitely many different symbols appearing among all of the strings to be a language.

A simple but nevertheless important example of a language over a given alphabet \(\Sigma\) is the set of all strings over \(\Sigma\). We denote this language as \(\Sigma^*\). Another simple and important example of a language is the empty language, which is the set containing no strings at all. The empty language is denoted \(\emptyset\) because it is the same thing as the empty set; there is no point in introducing any new notation here because we already have a notation for the empty set. The empty language is a language over an arbitrary choice of an alphabet.

In this course we will typically use capital letters near the beginning of the Roman alphabet, such as \(A, B, C,\) and \(D,\) to refer to languages. Sometimes we will also give special languages special names, such as PAL and DIAG, as you will see later.

We will see many other examples of languages throughout the course. Here are a few examples involving the binary alphabet \(\Sigma = \{0, 1\}:\)

\[
A = \{0010, 110110, 011000010110, 11111000110100010110\}. \quad (1.20)
\]
\[
B = \{x \in \Sigma^* : x \text{ starts with } 0 \text{ and ends with } 1\}. \quad (1.21)
\]
\[
C = \{x \in \Sigma^* : x \text{ is a binary representation of a prime number}\}. \quad (1.22)
\]
\[
D = \{x \in \Sigma^* : |x| \text{ and } |x| + 2 \text{ are prime numbers}\}. \quad (1.23)
\]

The language \(A\) is finite, \(B\) and \(C\) are not finite (they both have infinitely many strings), and at this point in time nobody knows if \(D\) is finite or infinite (because the so-called twin primes conjecture remains unproved).
Lecture 2

Countability for languages; deterministic finite automata

The main goal of this lecture is to introduce the finite automata model, but first we will finish our introductory discussion of alphabets, strings, and languages by connecting them with the notion of countability.

2.1 Countability and languages

We discussed a few examples of languages last time, and considered whether or not those languages were finite or infinite. Now let us think about the notion of countability in the context of languages.

Languages are countable

We will begin with the following proposition.¹

Proposition 2.1. For every alphabet $\Sigma$, the language $\Sigma^*$ is countable.

Let us focus on how this proposition may be proved just for the binary alphabet $\Sigma = \{0, 1\}$ for simplicity; the argument is easily generalized to any other alphabet. To prove that $\Sigma^*$ is countable, it suffices to define an onto function

$$f : \mathbb{N} \to \Sigma^*. \quad (2.1)$$

¹ In mathematics, names including proposition, theorem, corollary, and lemma refer to facts, and which name you use depends on the nature of the fact. Informally speaking, theorems are important facts that we’re proud of and propositions are also important facts, but we’re embarrassed to call them theorems because they’re so easy to prove. Corollaries are facts that follow easily from theorems, and lemmas (or lemmata for Latin purists) are boring technical facts that nobody cares about except for the fact that they are useful for proving certain theorems.
In fact, we can easily obtain a one-to-one and onto function $f$ of this form by considering the lexicographic ordering of strings. This is what you get by ordering strings by their length, and using the “dictionary” ordering among strings of equal length. The lexicographic ordering of $\Sigma^*$ begins like this:

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$$

(2.2)

From this ordering we can define a function $f$ of the form (2.1) simply by setting $f(n)$ to be the $n$-th string in the lexicographic ordering of $\Sigma^*$, starting from 0. Thus, we have

$$f(0) = \varepsilon, \quad f(1) = 0, \quad f(2) = 1, \quad f(3) = 00, \quad f(4) = 01,$$

(2.3)

and so on. An explicit method for calculating $f(n)$ is to write $n + 1$ in binary notation and then throw away the leading 1.

It is not hard to see that the function $f$ we’ve just defined is an onto function; every binary string appears as an output value of the function $f$. It therefore follows that $\Sigma^*$ is countable. It is also the case that $f$ is a one-to-one function.

It is easy to generalize this argument to any other alphabet. The first thing we need to do is to decide on an ordering of the alphabet symbols themselves. For the binary alphabet we order the symbols in the way we were trained: first 0, then 1. If we started with a different alphabet, such as $\Gamma = \{\epsilon, \alpha, \beta, \gamma\}$, it might not be clear how to order the symbols, but it doesn’t matter as long as we pick a single ordering and remain consistent with it. Once we’ve ordered the symbols in a given alphabet $\Gamma$, the lexicographic ordering of the language $\Gamma^*$ is defined in a similar way to what we did above, using the ordering of the alphabet symbols to determine what is meant by “dictionary” ordering. From the resulting lexicographic ordering we obtain a one-to-one and onto function $f : \mathbb{N} \to \Gamma^*$.

**Remark 2.2.** A brief remark is in order concerning the term lexicographic order. Some use this term to mean something different: dictionary ordering without first ordering strings according to length. They then use the term quasi-lexicographic order to refer to what we have called lexicographic order. There is no point in worrying too much about such discrepancies; there are many cases in science and mathematics where people disagree on terminology. What is important is that everyone is clear about what the terminology means when it is being used. With that in mind, in this course lexicographic order means strings are ordered first by length, and by “dictionary” ordering among strings of the same length.

It follows from the fact that the language $\Sigma^*$ is countable, for any choice of an alphabet $\Sigma$, that every language $A \subseteq \Sigma^*$ is countable. This is because every subset of a countable set is also countable. (I will leave it to you to prove this yourself, both for practice in writing proofs and to gain familiarity with the concept of countability.)
The set of all languages over any alphabet is uncountable

Next we will consider the set of all languages over a given alphabet. If $\Sigma$ is an alphabet, then saying that $A$ is a language over $\Sigma$ is equivalent to saying that $A$ is a subset of $\Sigma^*$, and being a subset of $\Sigma^*$ is the same thing as being an element of the power set of $\Sigma^*$. The following three statements are therefore equivalent, for any choice of an alphabet $\Sigma$:

1. $A$ is a language over the alphabet $\Sigma$.
2. $A \subseteq \Sigma^*$.
3. $A \in \mathcal{P}(\Sigma^*)$.

We have observed, for any alphabet $\Sigma$, that every language $A \subseteq \Sigma^*$ is countable, and it is natural to ask next if the set of all languages over $\Sigma$ is countable. It is not.

**Proposition 2.3.** Let $\Sigma$ be an alphabet. The set $\mathcal{P}(\Sigma^*)$ is uncountable.

To prove this proposition, we don’t need to repeat the same sort of diagonalization argument used to prove that $\mathcal{P}(\mathbb{N})$ is uncountable. Instead, we can simply combine that theorem with the fact that there exists a one-to-one and onto function from $\mathbb{N}$ to $\Sigma^*$.

In greater detail, let

$$f : \mathbb{N} \to \Sigma^*$$

be a one-to-one and onto function, such as the function we obtained earlier from the lexicographic ordering of $\Sigma^*$. We can extend this function, so to speak, to the power sets of $\mathbb{N}$ and $\Sigma^*$ as follows. Let

$$g : \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\Sigma^*)$$

be the function defined as

$$g(A) = \{ f(n) : n \in A \}$$

for all $A \subseteq \mathbb{N}$. In words, the function $g$ simply applies $f$ to each of the elements in a given subset of $\mathbb{N}$. It is not hard to see that $g$ is one-to-one and onto; we can express the inverse of $g$ directly, in terms of the inverse of $f$, as follows:

$$g^{-1}(B) = \{ f^{-1}(w) : w \in B \}$$

for every $B \subseteq \Sigma^*$.
Now, because there exists a one-to-one and onto function of the form (2.5), we conclude that \( \mathcal{P}(\mathbb{N}) \) and \( \mathcal{P}(\Sigma^*) \) have the “same size.” That is, because \( \mathcal{P}(\mathbb{N}) \) is uncountable, the same must be true of \( \mathcal{P}(\Sigma^*) \). To be more formal about this statement, one may assume toward contradiction that \( \mathcal{P}(\Sigma^*) \) is countable, which implies that there exists an onto function of the form

\[
h : \mathbb{N} \to \mathcal{P}(\Sigma^*).
\]

(2.8)

By composing this function with the inverse of the function \( g \) specified above, we obtain an onto function

\[
g^{-1} \circ h : \mathbb{N} \to \mathcal{P}(\mathbb{N}),
\]

(2.9)

which contradicts what we already know, which is that \( \mathcal{P}(\mathbb{N}) \) is uncountable.

### 2.2 Deterministic finite automata

The first model of computation we will discuss in this course is a simple one, called the deterministic finite automata model. You should have already learned something about finite automata (also called finite state machines) in CS 241, so we aren’t necessarily starting from the very beginning—but we do of course need a formal definition to proceed mathematically.

Please keep in mind the following two points as you consider the definition of the deterministic finite automata model:

1. The definition is based on sets (and functions, which can be formally described in terms of sets, as you may have learned in a discrete mathematics course). This is not surprising: set theory provides a foundation for much of mathematics, and it is only natural that we look to sets as we formulate definitions.

2. Although deterministic finite automata are not very powerful in computational terms, the model is important nevertheless, and it is just the start. Do not be bothered if it seems like a weak and useless model; we’re not trying to model general purpose computers at this stage, and the concept of finite automata is far from useless.

**Definition 2.4.** A deterministic finite automaton (or DFA, for short) is a 5-tuple

\[
M = (Q, \Sigma, \delta, q_0, F),
\]

(2.10)

where \( Q \) is a finite and nonempty set (whose elements we will call states), \( \Sigma \) is an alphabet, \( \delta \) is a function (called the transition function) having the form

\[
\delta : Q \times \Sigma \to Q,
\]

(2.11)

\( q_0 \in Q \) is a state (called the start state), and \( F \subseteq Q \) is a subset of states (whose elements we will call accept states).
State diagrams

It is common that DFAs are expressed using *state diagrams*, such as this one that appears in Figure 2.1. State diagrams express all 5 parts of the formal definition of DFAs:

1. States are denoted by circles.
2. Alphabet symbols label the arrows.
3. The transition function is determined by the arrows and the circles they connect.
4. The start state is determined by the arrow coming in from nowhere.
5. The accept states are those with double circles.

For the state diagram in Figure 2.1, for example, the state set is

\[ Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}, \]  

the alphabet is

\[ \Sigma = \{0, 1\}, \] 

the start state is \(q_0\), the set of accepts states is

\[ F = \{q_0, q_2, q_5\}. \]
and the transition function $\delta : Q \times \Sigma \to Q$ is as follows:

\[
\begin{align*}
\delta(q_0, 0) &= q_0, & \delta(q_1, 0) &= q_3, & \delta(q_2, 0) &= q_5, \\
\delta(q_0, 1) &= q_1, & \delta(q_1, 1) &= q_2, & \delta(q_2, 1) &= q_5, \\
\delta(q_3, 0) &= q_3, & \delta(q_4, 0) &= q_4, & \delta(q_5, 0) &= q_4, \\
\delta(q_3, 1) &= q_3, & \delta(q_4, 1) &= q_1, & \delta(q_5, 1) &= q_2.
\end{align*}
\]

In order for a state diagram to correspond to a DFA, and more specifically for it to determine a valid transition function, it must be that for every state and every symbol, there is exactly one arrow exiting from that state labeled by that symbol. Note that when a single arrow is labeled by multiple symbols, such as in the case of the arrows labeled “0,1” in Figure 2.1, it should be interpreted that there are actually multiple arrows, each labeled by a single symbol; we’re just making our diagrams a bit less cluttered by reusing the same arrow to express multiple transitions.

You can also go the other way and draw a state diagram from a formal description of a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. It is a routine exercise to do this.

**DFA computations**

You may already know what it means for a DFA $M = (Q, \Sigma, \delta, q_0, F)$ to accept or reject a given input string $w \in \Sigma^*$, either based on what you learned in CS 241 or from a natural guess after a moment’s thought about the definition. It is easy enough to say it in words, particularly when we think in terms of state diagrams: we start on the start state, follow transitions from one state to another according to the symbols of $w$ (reading one at a time, left to right), and we accept if and only if we end up on an accept state (and otherwise we reject).

This all makes sense, but it is useful nevertheless to think about how it is expressed formally. That is, how do we define in precise, mathematical terms what it means for a DFA to accept or reject a given string? In particular, phrases like “follow transitions” and “end up on an accept state” can be replaced by more precise mathematical notions.

Here is one way to define acceptance and rejection more formally. Notice again that the definition focuses on sets and functions.

**Definition 2.5.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w \in \Sigma^*$ be a string. The DFA $M$ accepts the string $w$ if one of the following statements holds:

1. $w = \varepsilon$ and $q_0 \in F$.
2. $w = a_1 \cdots a_n$ for a positive integer $n$ and symbols $a_1, \ldots, a_n \in \Sigma$, and there exist states $r_0, \ldots, r_n \in Q$ such that $r_0 = q_0$, $r_n \in F$, and $r_{k+1} = \delta(r_k, a_{k+1})$ for all $k \in \{0, \ldots, n-1\}$.
If $M$ does not accept $w$, then $M$ rejects $w$.

In words, the formal definition of acceptance is that there exists a sequence of states $r_0, \ldots, r_n$ such that the first state is the start state, the last state is an accept state, and each state in the sequence is determined from the previous state and the corresponding symbol read from the input as the transition function describes: if we are in the state $q$ and read the symbol $a$, the new state becomes $p = \delta(q, a)$. The first statement in the definition is simply a special case that handles the empty string.

It is natural to consider why we would prefer a formal definition like this to what is perhaps a more human-readable definition. Of course, the human-readable version beginning with “Start on the start state, follow transitions . . . ” is effective for explaining the concept of a DFA, but the formal definition has the benefit that it reduces the notion of acceptance to elementary mathematical statements about sets and functions. It is also quite succinct and precise, and leaves no ambiguities about what it means for a DFA to accept or reject.

It is sometimes useful to define a new function

$$\delta^* : Q \times \Sigma^* \rightarrow Q,$$

based on a given transition function $\delta : Q \times \Sigma \rightarrow Q$, in the following recursive way:

1. $\delta^*(q, \epsilon) = q$ for every $q \in Q$, and
2. $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ for all $q \in Q, a \in \Sigma$, and $w \in \Sigma^*$.

Intuitively speaking, $\delta^*(q, w)$ is the state you end up on if you start at state $q$ and follow the transitions specified by the string $w$.

It is the case that a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string $w \in \Sigma^*$ if and only if $\delta^*(q_0, w) \in F$. A natural way to argue this formally, which we will not do in detail, is to prove by induction on the length of $w$ that $\delta^*(q, w) = p$ if and only if one of these two statements is true:

1. $w = \epsilon$ and $p = q$.
2. $w = a_1 \cdots a_n$ for a positive integer $n$ and symbols $a_1, \ldots, a_n \in \Sigma$, and there exist states $r_0, \ldots, r_n \in Q$ such that $r_0 = q, r_n = p$, and $r_{k+1} = \delta(r_k, a_{k+1})$ for all $k \in \{0, \ldots, n - 1\}$.

Once that equivalence is proved, the statement $\delta^*(q_0, w) \in F$ can be equated to $M$ accepting $w$.

**Remark 2.6.** By now it is evident that we will not formally prove every statement we make in this course. If we did, we wouldn’t get far, and even then we might
look back and feel as if we could probably have been even more formal. If we insisting on proving everything with more and more formality, we could in principle reduce every mathematical claim we make to axiomatic set theory—but then we would have covered little material about computation in a one-term course. Moreover, our proofs would most likely be incomprehensible, and would quite possibly contain as many errors as you would expect to find in a complicated and untested program written in assembly language.

Naturally we won’t take this path, but from time to time we will discuss the nature of proofs, how we would prove something if we took the time to do it, and how certain high-level statements and arguments could be reduced to more basic and concrete steps pointing in the general direction of completely formal proofs that could be verified by a computer. If you are unsure at this point what actually constitutes a proof, or how much detail and formality you should aim for in your own proofs, don’t worry—sorting this out is one of the principal aims of this course.

Languages recognized by DFAs and regular languages

Suppose \( M = (Q, \Sigma, \delta, q_0, F) \) is a DFA. We may then consider the set of all strings that are accepted by \( M \). This language is denoted \( L(M) \), so that

\[
L(M) = \{ w \in \Sigma^* : M \text{ accepts } w \}. \tag{2.17}
\]

We refer to this as the language recognized by \( M \).\(^2\) It is important to understand that this is a single, well-defined language consisting precisely of those strings accepted by \( M \) and not containing any strings rejected by \( M \).

For example, here is a simple DFA over the binary alphabet \( \Sigma = \{0, 1\} \):

\[ q_0 \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quantity{2}{\text{Alternative, we might also refer to } L(M) \text{ as the language accepted by } M. \text{ Unfortunately this terminology sometimes leads to confusion because it overloads the term accepted.}}
then of course it is true that \( M \) accepts every string in \( A \). However, \( M \) also accepts some strings that are not in \( A \), so \( A \) is not the language recognized by \( M \).

We have one more definition for this lecture, which introduces some important terminology.

**Definition 2.7.** Let \( \Sigma \) be an alphabet and let \( A \subseteq \Sigma^* \) be a language over \( \Sigma \). The language \( A \) is regular if there exists a DFA \( M \) such that \( A = L(M) \).

We have not seen many DFAs thus far, so we don’t have many examples of regular languages to mention at this point, but we will see plenty of them throughout the course.

Let us finish off the lecture with a question.

**Question 1.** For a given alphabet \( \Sigma \), is the number of regular languages over the alphabet \( \Sigma \) countable or uncountable?

The answer is “countable.” The reason is that there are countably many DFAs over any alphabet \( \Sigma \), and we can combine this fact with the observation that the function that maps each DFA to the regular language it recognizes is, by definition, an onto function.

When we say that there are countably many DFAs, we really should be a bit more precise. In particular, we are not considering two DFAs to be different if they are exactly the same except for the names we have chosen to give the states. This is reasonable because the names we give to different states of a DFA has no influence on the language recognized by that DFA—we may as well assume that the state set of a DFA is \( Q = \{q_0, \ldots, q_{m-1}\} \) for some choice of a positive integer \( m \). In fact, people often don’t even bother assigning names to states when drawing state diagrams of DFAs, because the state names are irrelevant to the way DFAs operates.

To see that there are countably many DFAs over a given alphabet \( \Sigma \), we can use a similar strategy to what we did when proving that the set rational numbers \( Q \) is countable. First imagine that there is just one state: \( Q = \{q_0\} \). There are only finitely many DFAs with just one state over a given alphabet \( \Sigma \). (In fact there are just two, one where \( q_0 \) is an accept state and one where \( q_0 \) is a reject state.) Now consider the set of all DFAs with two states: \( Q = \{q_0, q_1\} \). Again, there are only finitely many. Continuing on like this, for any choice of a positive integer \( m \), there will be only finitely many DFAs with \( m \) states for a given alphabet \( \Sigma \). The number of DFAs with \( m \) states happens to grow exponentially with \( m \), but this is not important right now—we just need to known that the number is finite. If you chose some arbitrary way of sorting each of these finite lists of DFAs, and then you concatenated the lists together starting with the 1 state DFAs, then the 2 state DFAs,
and so on, you would end up with a single list containing every DFA. From such a list you can obtain an onto function from $\mathbb{N}$ to the set of all DFAs over $\Sigma$ in a similar way to what we did for the rational numbers.

Because there are uncountably many languages $A \subseteq \Sigma^*$, and only countably many regular languages $A \subseteq \Sigma^*$, we can immediately conclude that some languages are not regular. This is just an existence proof, and doesn’t give us a specific language that is not regular—it just tells us that there is one. We’ll see methods later that allow us to conclude that certain specific languages are not regular.
Lecture 3

Nondeterministic finite automata

This lecture is focused on the nondeterministic finite automata (NFA) model and its relationship to the DFA model.

Nondeterminism is a critically important concept in the theory of computing. It refers to the possibility of having multiple choices for what can happen at various points in a computation. We then consider the possible outcomes that these choices can have, usually focusing on whether or not there exists a sequence of choices that leads to a particular outcome (such as acceptance for a finite automaton).

This may sound like a fantasy mode of computation not likely to be relevant from a practical viewpoint, because real computers don’t make nondeterministic choices: each step a real computer makes is uniquely determined by its configuration at any given moment. Our great interest in nondeterminism is, however, not meant to suggest otherwise. We will see that nondeterminism is a powerful analytic tool (in the sense that it helps us to design things and prove facts), and its close connection with proofs and verification has fundamental importance.

3.1 Nondeterministic finite automata basics

Let us begin our discussion of the NFA model with its definition. The definition is similar to the definition of the DFA model, but with a key difference.

Definition 3.1. A nondeterministic finite automaton (or NFA, for short) is a 5-tuple

\[ N = (Q, \Sigma, \delta, q_0, F), \]

where \( Q \) is a finite and nonempty set of states, \( \Sigma \) is an alphabet, \( \delta \) is a transition function having the form

\[ \delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q), \]

\( q_0 \in Q \) is a start state, and \( F \subseteq Q \) is a subset of accept states.
The key difference between this definition and the analogous definition for DFAs is that the transition function has a different form. For a DFA we had that \( \delta(q, a) \) was a state, for any choice of a state \( q \in Q \) and a symbol \( a \in \Sigma \), representing the next state that the DFA would move to if it was in the state \( q \) and read the symbol \( a \). For an NFA, each \( \delta(q, a) \) is not a state, but rather a subset of states, which is equivalent to \( \delta(q, a) \) being an element of the power set \( \mathcal{P}(Q) \). This subset represents all of the possible states that the NFA could move to when in state \( q \) and reading symbol \( a \). There could be just a single state in this subset, or there could be multiple states, or there might even be no states at all—it is possible to have \( \delta(q, a) = \emptyset \).

We also have that the transition function of an NFA is not only defined for every pair \((q, a) \in Q \times \Sigma\), but also for every pair \((q, \varepsilon)\). Here, as always in this course, \( \varepsilon \) denotes the empty string. By defining \( \delta \) for such pairs we are allowing for so-called \( \varepsilon \)-transitions, where an NFA may move from one state to another without reading a symbol from the input.

**State diagrams**

Similar to DFAs, we sometimes represent NFAs with state diagrams. This time, for each state \( q \) and each symbol \( a \), there may be multiple arrows leading out of the circle representing the state \( q \) labeled by \( a \), which tells us which states are contained in \( \delta(q, a) \), or there may be no arrows like this when \( \delta(q, a) = \emptyset \). We may also label arrows by \( \varepsilon \), which indicates where the \( \varepsilon \)-transitions lead.

Figure 3.1 gives an example of a state diagram for an NFA. In this figure, we see that \( Q = \{q_0, q_1, q_2, q_3\} \), \( q_0 \) is the start state, and \( F = \{q_1\} \), just like we would have if this diagram represented a DFA. It is reasonable to guess from the diagram that the alphabet for the NFA it describes is \( \Sigma = \{0, 1\} \), although all we can be sure of is that \( \Sigma \) includes the symbols 0 and 1; it could be, for instance, that \( \Sigma = \{0, 1, 2\} \), but it so happens that \( \delta(q, 2) = \emptyset \) for every \( q \in Q \). Let us agree, however, that unless we explicitly indicate otherwise, the alphabet for an NFA described by a state diagram includes precisely those symbols (not including \( \varepsilon \) of course) that label transitions in the diagram, so that \( \Sigma = \{0, 1\} \) for this particular example. The transition function, which must take the form

\[
\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q),
\]

is given by

\[
\begin{align*}
\delta(q_0, 0) &= \{q_1\}, & \delta(q_0, 1) &= \{q_0\}, & \delta(q_0, \varepsilon) &= \emptyset, \\
\delta(q_1, 0) &= \{q_1\}, & \delta(q_1, 1) &= \{q_3\}, & \delta(q_1, \varepsilon) &= \{q_2\}, \\
\delta(q_2, 0) &= \{q_1, q_2\}, & \delta(q_2, 1) &= \emptyset, & \delta(q_2, \varepsilon) &= \{q_3\}, \\
\delta(q_3, 0) &= \{q_0, q_3\}, & \delta(q_3, 1) &= \emptyset, & \delta(q_3, \varepsilon) &= \emptyset.
\end{align*}
\]

(3.4)
Lecture 3

![NFA State Diagram]

Figure 3.1: The state diagram of an NFA.

NFA computations

Next let us consider the definition of acceptance and rejection for NFAs. This time we will start with the formal definition and then try to understand what it says.

Definition 3.2. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let $w \in \Sigma^*$ be a string. The NFA $N$ accepts $w$ if there exists a natural number $m \in \mathbb{N}$, a sequence of states $r_0, \ldots, r_m$, and a sequence of either symbols or empty strings $a_1, \ldots, a_m \in \Sigma \cup \{\varepsilon\}$ such that the following statements all hold:

1. $r_0 = q_0$.
2. $r_m \in F$.
3. $w = a_1 \cdots a_m$.
4. $r_{k+1} \in \delta(r_k, a_{k+1})$ for every $k \in \{0, \ldots, m - 1\}$.

If $N$ does not accept $w$, then we say that $N$ rejects $w$.

As you may already know, we can think of the computation of an NFA $N$ on an input string $w$ as being like a single-player game, where the goal is to start on the start state, make moves from one state to another, and end up on an accept state. If you want to move from a state $q$ to a state $p$, there are two possible ways to do this: you can move from $q$ to $p$ by reading a symbol $a$ from the input, provided that $p \in \delta(q, a)$; or you can move from $q$ to $p$ without reading a symbol, provided that $p \in \delta(q, \varepsilon)$ (i.e., there is an $\varepsilon$-transition from $q$ to $p$). To win the game, you
must not only end on an accept state, but you must also have read every symbol from the input string \( w \). To say that \( N \) accepts \( w \) means that it is possible to win the corresponding game.

Definition 3.2 essentially formalizes the notion of winning the game we just discussed: the natural number \( m \) represents the number of moves you make and \( r_0, \ldots, r_m \) represent the states that are visited. In order to win the game you have to start on state \( q_0 \) and end on an accept state, which is why the definition requires \( r_0 = q_0 \) and \( r_m \in F \), and it must also be that every symbol of the input is read by the end of the game, which is why the definition requires \( w = a_1 \cdots a_m \). The condition \( r_{k+1} \in \delta(r_k, a_{k+1}) \) for every \( k \in \{0, \ldots, m - 1\} \) corresponds to every move being a legal move in which a valid transition is followed.

We should take a moment to note how the definition works when \( m = 0 \). The natural numbers (as we have defined them) include 0, so there is nothing that prevents us from considering \( m = 0 \) as one way that a string might potentially be accepted. If we begin with the choice \( m = 0 \), then we must consider the existence of a sequence of states \( r_0, \ldots, r_0 \) and a sequence of symbols or empty strings \( a_1, \ldots, a_0 \in \Sigma \cup \{ \varepsilon \} \), and whether or not these sequences satisfy the four requirements listed in the definition. There is nothing wrong with a sequence of states having the form \( r_0, \ldots, r_0 \), by which we really just mean the sequence \( r_0 \) having a single element. The sequence \( a_1, \ldots, a_0 \in \Sigma \cup \{ \varepsilon \} \), on the other hand, looks like it does not make any sense—but it actually does make sense if you interpret it as an empty sequence having no elements in it. The condition \( w = a_1 \cdots a_0 \) in this case, which refers to a concatenation of an empty sequence of symbols or empty strings, is that it means \( w = \varepsilon \).\(^1\) Asking that the condition \( r_{k+1} \in \delta(r_k, a_{k+1}) \) should hold for every \( k \in \{0, \ldots, m - 1\} \) is a vacuous statement, and is therefore trivially true, because there are no values of \( k \) to worry about.

Thus, if it is the case that the initial state \( q_0 \) of the NFA we are considering happens to be an accept state, and our input is the empty string, then the NFA accepts—for we can take \( m = 0 \) and \( r_0 = q_0 \), and the definition is satisfied. Note that we could have done something similar in our definition for when a DFA accepts: if we allowed \( n = 0 \) in the second statement of that definition, it would be equivalent to the first statement, and so we really didn’t need to take the two possibilities separately. (Alternatively, we could have added a special case to Definition 3.2, but it would make the definition longer, and the convention described above is good to know about anyway.)

Along similar lines to what we did for DFAs, we can define an extended version

---

\(^1\) Note that it is a convention, and not something you can deduce, that the concatenation of an empty sequence of symbols gives you the empty string. It is similar to the convention that the sum of an empty sequence of numbers is 0 and the product of an empty sequence of numbers is 1.
of the transition function of an NFA. In particular, if
\[
    \delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)
\]
(3.5)
is a transition of an NFA, we define a new function
\[
    \delta^* : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)
\]
(3.6)
as follows. First, we define the \( \varepsilon \)-closure of any set \( R \subseteq Q \) as
\[
    \varepsilon(R) = \left\{ q \in Q : q \text{ is reachable from some } r \in R \text{ by following zero or more } \varepsilon \text{-transitions} \right\}.
\]
(3.7)
Another way of defining \( \varepsilon(R) \) is to say that it is the intersection of all subsets \( T \subseteq Q \) satisfying these conditions:

1. \( R \subseteq T \).
2. \( \delta(q, \varepsilon) \subseteq T \) for every \( q \in T \).

We can interpret this alternative definition as saying that \( \varepsilon(R) \) is the \textit{smallest} subset of \( Q \) that contains \( R \) and is such that you can never get out of this set by following an \( \varepsilon \)-transition.

With the notion of the \( \varepsilon \)-closure in hand, we define \( \delta^* \) recursively as follows:

1. \( \delta^*(q, \varepsilon) = \varepsilon(\{q\}) \) for every \( q \in Q \), and
2. \( \delta^*(q, wa) = \varepsilon(\bigcup_{r \in \delta^*(q, w)} \delta(r, a)) \) for every \( q \in Q, a \in \Sigma, \) and \( w \in \Sigma^* \).

Intuitively speaking, \( \delta^*(q, w) \) is the set of all states that you could potentially reach by starting on the state \( q \), reading \( w \), and making as many \( \varepsilon \)-transitions along the way as you like. To say that an NFA \( N = (Q, \Sigma, \delta, q_0, F) \) accepts a string \( w \in \Sigma^* \) is equivalent to the condition that \( \delta^*(q_0, w) \cap F \neq \emptyset \).

Also similar to DFAs, the notation \( L(N) \) denotes the language \textit{recognized} by an NFA \( N \):
\[
    L(N) = \{ w \in \Sigma^* : N \text{ accepts } w \}.
\]
(3.8)

3.2 Equivalence of NFAs and DFAs

It seems like NFAs might potentially be more powerful than DFAs because NFAs have the option to use nondeterminism. Perhaps you already know from a previous course, however, that this is not the case, as the following theorem states.
Theorem 3.3. Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language $A$ is regular (i.e., recognized by a DFA) if and only if $A = L(N)$ for some NFA $N$.

Let us begin by breaking this theorem down, to see what needs to be shown in order to prove it. First, it is an “if and only if” statement, so there are two things to prove:

1. If $A$ is regular, then $A = L(N)$ for some NFA $N$.
2. If $A = L(N)$ for some NFA $N$, then $A$ is regular.

If you were in a hurry and had to choose one of these two statements to prove, you would be wise to choose the first: it’s the easier of the two by far. In particular, suppose $A$ is regular, so by definition there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes $A$. The goal is to define an NFA $N$ that also recognizes $A$. This is simple, as we can just take $N$ to be the NFA whose state diagram is the same as the state diagram for $M$. At a formal level, $N$ isn’t exactly the same as $M$; because $N$ is an NFA, its transition function will have a different form from a DFA transition function, but in this case the difference is only cosmetic. More formally speaking, we can define $N = (Q, \Sigma, \mu, q_0, F)$ where the transition function $\mu : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is defined as

$$\mu(q, a) = \{\delta(q, a)\} \quad \text{and} \quad \mu(q, \epsilon) = \emptyset \quad (3.9)$$

for all $q \in Q$ and $a \in \Sigma$. It is the case that $L(N) = L(M) = A$, and so we’re done.

Now let us consider the second statement listed above. We assume $A = L(N)$ for some NFA $N = (Q, \Sigma, \delta, q_0, F)$, and our goal is to show that $A$ is regular. That is, we must prove that there exists a DFA $M$ such that $L(M) = A$. The most direct way to do this is to argue that, by using the description of $N$, we are able to come up with an equivalent DFA $M$. That is, if we can show how an arbitrary NFA $N$ can be used to define a DFA $M$ such that $L(M) = L(N)$, then the proof will be complete.

We will use the description of an NFA $N$ to define an equivalent DFA $M$ using a simple idea: each state of $M$ will keep track of a subset of states of $N$. After reading any part of its input string, there will always be some subset of states that $N$ could possibly be in, and we will design $M$ so that after reading the same part of its input string it will be in the state corresponding to this subset of states of $N$.

A simple example

Let us see how this works for a simple example before we describe it in general. Consider the NFA $N$ described in Figure 3.2. If we describe this NFA formally,
according to the definition of NFAs, it is given by

\[ N = (Q, \Sigma, \delta, q_0, F) \]  \hspace{1cm} (3.10)

where \( Q = \{q_0, q_1\} \), \( \Sigma = \{0, 1\} \), \( F = \{q_1\} \), and \( \delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q) \) is defined as follows:

\[
\begin{align*}
\delta(q_0, 0) &= \{q_0, q_1\}, & \delta(q_0, 1) &= \{q_1\}, & \delta(q_0, \varepsilon) &= \emptyset, \\
\delta(q_1, 0) &= \emptyset, & \delta(q_1, 1) &= \{q_0\}, & \delta(q_1, \varepsilon) &= \emptyset.
\end{align*}
\]  \hspace{1cm} (3.11)

We are going to define an DFA \( M \) having one state for every subset of states of \( N \). We can name the states of \( M \) however we like, so we may as well name them directly with the subsets of \( Q \). In other words, the state set of \( M \) will be the power set \( \mathcal{P}(Q) \).

Have a look at the state diagram in Figure 3.3 and think about if it makes sense as a good choice for \( M \). Formally speaking, this DFA is given by

\[ M = (\mathcal{P}(Q), \Sigma, \mu, \{q_0\}, \{\{q_1\}, \{q_0, q_1\}\}), \]  \hspace{1cm} (3.12)
where the transition function \( \mu : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q) \) is defined as

\[
\begin{align*}
\mu(\{q_0\}, 0) &= \{q_0, q_1\}, & \mu(\{q_0\}, 1) &= \{q_1\}, \\
\mu(\{q_1\}, 0) &= \emptyset, & \mu(\{q_1\}, 1) &= \{q_0\}, \\
\mu(\{q_0, q_1\}, 0) &= \{q_0, q_1\}, & \mu(\{q_0, q_1\}, 1) &= \{q_0, q_1\}, \\
\mu(\emptyset, 0) &= \emptyset, & \mu(\emptyset, 1) &= \emptyset.
\end{align*}
\]  
(3.13)

One can verify that this DFA description indeed makes sense, one transition at a time.

For instance, suppose at some point in time \( N \) is in the state \( q_0 \). If a 0 is read, it is possible to either follow the self-loop and remain on state \( q_0 \) or follow the other transition and end on \( q_1 \). This is why there is a transition labeled 0 from the state \( \{q_0\} \) to the state \( \{q_0, q_1\} \) in \( M \); the state \( \{q_0, q_1\} \) in \( M \) is representing the fact that \( N \) could be either in the state \( q_0 \) or the state \( q_1 \). On the other hand, if \( N \) is in the state \( q_1 \) and a 0 is read, there are no possible transitions to follow, and this is why \( M \) has a transition labeled 0 from the state \( \{q_1\} \) to the state \( \emptyset \). The state \( \emptyset \) in \( M \) is representing the fact that there aren’t any states that \( N \) could possibly be in (which is sensible because \( N \) is an NFA). The self-loop on the state \( \emptyset \) in \( M \) labeled by 0 and 1 represents the fact that if \( N \) cannot be in any states at a given moment, and a symbol is read, there still aren’t any states it could be in. You can go through the other transitions and verify that they work in a similar way.

There is also the issue of which state is chosen as the start state of \( M \) and which states are accept states. This part is simple: we let the start state of \( M \) correspond to the states of \( N \) we could possibly be in without reading any symbols at all, which is \( \{q_0\} \) in our example, and we let the accept states of \( M \) be those states corresponding to any subset of states of \( N \) that includes at least one element of \( F \).

**The construction in general**

Now let us think about the idea suggested above in greater generality. That is, we will specify a DFA \( M \) satisfying \( L(M) = L(N) \) for an arbitrary NFA

\[ N = (Q, \Sigma, \delta, q_0, F). \]  
(3.14)

One thing to keep in mind as we do this is that \( N \) could have \( \epsilon \)-transitions, whereas our simple example did not. It will, however, be easy to deal with \( \epsilon \)-transitions by referring to the notion of the \( \epsilon \)-closure that we discussed earlier. Another thing to keep in mind is that \( N \) really is arbitrary—maybe it has 1,000,000 states or more. It is therefore hopeless for us to describe what’s going on using state diagrams, so we’ll do everything abstractly.
First, we know what the state set of $M$ should be based on the discussion above: the power set $\mathcal{P}(Q)$ of $Q$. Of course the alphabet is $\Sigma$ because it has to be the same as the alphabet of $N$. The transition function of $M$ should therefore take the form

$$\mu : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q)$$

in order to be consistent with these choices. In order to define the transition function $\mu$ precisely, we must therefore specify the output subset

$$\mu(R, a) \subseteq Q$$

for every subset $R \subseteq Q$ and every symbol $a \in \Sigma$. One way to do this is as follows:

$$\mu(R, a) = \bigcup_{q \in R} \varepsilon(\delta(q, a)).$$

In words, the right-hand side of (3.17) represents every state in $N$ that you can get to by (i) starting at any state in $R$, then (ii) following a transition labeled $a$, and finally (iii) following any number of $\varepsilon$-transitions.

The last thing we need to do is to define the initial state and the accept states of $M$. The initial state is $\varepsilon(\{q_0\})$, which is every state you can reach from $q_0$ by just following $\varepsilon$-transitions, while the accept states are those subsets of $Q$ containing at least one accept state of $N$. If we write $G \subseteq \mathcal{P}(Q)$ to denote the set of accept states of $M$, then we may define this set as

$$G = \{ R \in \mathcal{P}(Q) : R \cap F \neq \emptyset \}.$$  

The DFA $M$ can now be specified formally as

$$M = (\mathcal{P}(Q), \Sigma, \mu, \varepsilon(\{q_0\}), G).$$

Now, if we are being honest with ourselves, we cannot say that we have proved that for every NFA $N$ there is an equivalent DFA $M$ satisfying $L(M) = L(N)$. All we’ve done is to define a DFA $M$ from a given NFA $N$ that seems like it should satisfy this equality. It is, in fact, true that $L(M) = L(N)$, but we won’t go through a formal proof that this really is the case. It is worthwhile, however, to think about how we would do this if we had to.

First, if we are to prove that the two languages $L(M)$ and $L(N)$ are equal, the natural way to do it is to split it into two separate statements:

1. $L(M) \subseteq L(N)$.
2. $L(N) \subseteq L(M)$. 

33
This is often the way to prove the equality of two sets. Nothing tells us that the two statements need to be proved in the same way, and by doing them separately we give ourselves more options about how to approach the proof. Let’s start with the subset relation \( L(N) \subseteq L(M) \), which is equivalent to saying that if \( w \in L(N) \), then \( w \in L(M) \). We can now fall back on the definition of what it means for \( N \) to accept a string \( w \), and try to conclude that \( M \) must also accept \( w \). It’s a bit tedious to write everything down carefully, but it is possible and maybe you can convince yourself that this is so. The other relation \( L(M) \subseteq L(N) \) is equivalent to saying that if \( w \in L(M) \), then \( w \in L(N) \). The basic idea here is similar in spirit, although the specifics are a bit different. This time we start with the definition of acceptance for a DFA, applied to \( M \), and then try to reason that \( N \) must accept \( w \).

A different way to prove that the construction works correctly is to make use of the functions \( \delta^* \) and \( \mu^* \), which are defined from \( \delta \) and \( \mu \) as we discussed in the previous lecture and earlier in this lecture. In particular, using induction on the length of \( w \), it can be proved that

\[
\mu^*(\epsilon(R), w) = \bigcup_{q \in R} \delta^*(q, w)
\]

for every string \( w \in \Sigma^* \) and every subset \( R \subseteq Q \). Once we have this, we see that \( \mu^*(\epsilon(\{q_0\}), w) \) is contained in \( G \) if and only if \( \delta^*(q_0, w) \cap F \neq \emptyset \), which is equivalent to \( w \in L(M) \) if and only if \( w \in L(N) \).

In any case, you are not being asked to formalize and verify the proofs just suggested at this stage, but only to think about how it would be done.

**On the process of converting NFAs to DFAs**

It is a typical type of exercise in courses such as CS 360 that students are presented with an NFA and asked to come up with an equivalent DFA using the construction described above. This is a mechanical exercise that does not require creativity, and it will be important later in the course to observe that the construction itself can be performed by a computer. This claim may, however, become more clear once you’ve gone through a few examples by hand.

If you do find yourself performing this construction by hand, it is worth pointing out that you really don’t need to write down every subset of states of \( N \) and then draw the arrows. There will be exponentially many more states in \( M \) than in \( N \), and it will sometimes be that many of these states are completely useless—being unreachable from the start state of \( M \). A better option is to first write down the start state of \( M \), which corresponds to the \( \epsilon \)-closure of the set containing just the start state of \( N \), and then to only draw new states of \( M \) as you need them.
In the worst case, however, you might actually need all of those states. There are examples known of languages that have an NFA with $n$ states, while the smallest DFA for the same language has $2^n$ states, for every choice of a positive integer $n$. So, while NFAs and DFAs are equivalent in computational power, there is sometimes a significant cost to be paid in converting an NFA into a DFA, which is that this might require the DFA to have a huge number of states in comparison to the number of states of the original NFA.
Lecture 4

Regular operations and regular expressions

In this lecture we will discuss the regular operations, as well as regular expressions and their relationship to regular languages.

4.1 Regular operations

The regular operations are three operations on languages, as the following definition makes clear.

Definition 4.1. Let $\Sigma$ be an alphabet and let $A, B \subseteq \Sigma^*$ be languages. The regular operations are as follows:

1. **Union.** The language $A \cup B \subseteq \Sigma^*$ is defined as

   $$A \cup B = \{ w : w \in A \text{ or } w \in B \}. \quad (4.1)$$

   In words, this is just the ordinary union of two sets that happen to be languages.

2. **Concatenation.** The language $AB \subseteq \Sigma^*$ is defined as

   $$AB = \{ wx : w \in A \text{ and } x \in B \}. \quad (4.2)$$

   In words, this is the language of all strings obtained by concatenating together a string from $A$ and a string from $B$, with the string from $A$ on the left and the string from $B$ on the right.

   Note that there is nothing about a string of the form $wx$ that indicates where $w$ stops and $x$ starts; it is just the sequence of symbols you get by putting $w$ and $x$ together.
3. **Kleene star** (or just *star*, for short). The language $A^*$ is defined as

$$A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \cdots \quad (4.3)$$

In words, $A^*$ is the language obtained by selecting any finite number of strings from $A$ and concatenating them together. (This includes the possibility to select no strings at all from $A$, where we follow the convention that concatenating together no strings at all gives the empty string.)

Note that the name *regular operations* is just a name that has been chosen for these three operations. They are special operations and they do indeed have a close connection to the regular languages, but naming them *the regular operations* is a choice we’ve made and not something mandated in a mathematical sense.

**Closure of regular languages under regular operations**

Next let us prove a theorem connecting the regular operations with the regular languages.

**Theorem 4.2.** The regular languages are closed with respect to the regular operations: if $A, B \subseteq \Sigma^*$ are regular languages, then the languages $A \cup B$, $AB$, and $A^*$ are also regular.

**Proof.** First let us observe that, because the languages $A$ and $B$ are regular, there must exist DFAs

$$M_A = (P, \Sigma, \delta, p_0, F) \quad \text{and} \quad M_B = (Q, \Sigma, \mu, q_0, G) \quad (4.4)$$

such that $L(M_A) = A$ and $L(M_B) = B$. We will make use of these DFAs as we prove that the languages $A \cup B$, $AB$, and $A^*$ are regular. Because we are free to give whatever names we like to the states of a DFA without influencing the language it recognizes, there is no generality lost in assuming that $P$ and $Q$ are disjoint sets (meaning that $P \cap Q = \emptyset$).

Let us begin with $A \cup B$. From last lecture we know that if there exists an NFA $N$ such that $L(N) = A \cup B$, then $A \cup B$ is regular. With that fact in mind, our goal will be to define such an NFA. We will define this NFA $N$ so that its states include all elements of both $P$ and $Q$, as well as an additional state $r_0$ that is in neither $P$ nor $Q$. This new state $r_0$ will be the start state of $N$. The transition function of $N$ is to be defined so that all of the transitions among the states $P$ defined by $\delta$ and all of the transitions among the states $Q$ defined by $\mu$ are present, as well as two $\varepsilon$-transitions, one from $r_0$ to $p_0$ and one from $r_0$ to $q_0$.

Figure 4.1 illustrates what the NFA $N$ looks like in terms of a state diagram. You should imagine that the shaded rectangles labeled $M_A$ and $M_B$ are the state
diagrams of $M_A$ and $M_B$. (The illustrations in the figure are only meant to suggest hypothetical state diagrams for these two DFAs. The actual state diagrams for $M_A$ and $M_B$ could, of course, be arbitrary.)

We can specify $N$ more formally as follows:

\[ N = (R, \Sigma, \eta, r_0, F \cup G) \]  \hspace{1cm} (4.5)

where $R = P \cup Q \cup \{r_0\}$ (and we assume $P$, $Q$, and $\{r_0\}$ are disjoint sets as suggested above) and the transition function

\[ \eta : R \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(R) \]  \hspace{1cm} (4.6)

is defined as follows:

\[
\begin{align*}
\eta(p, a) &= \{\delta(p, a)\} \quad \text{(for all } p \in P \text{ and } a \in \Sigma), \\
\eta(p, \varepsilon) &= \emptyset \quad \text{(for all } p \in P), \\
\eta(q, a) &= \{\mu(q, a)\} \quad \text{(for all } q \in Q \text{ and } a \in \Sigma), \\
\eta(q, \varepsilon) &= \emptyset \quad \text{(for all } q \in Q), \\
\eta(r_0, a) &= \emptyset \quad \text{(for all } a \in \Sigma), \\
\eta(r_0, \varepsilon) &= \{p_0, q_0\}.
\end{align*}
\]
The accept states of $N$ are $F \cup G$.

Every string that is accepted by $M_A$ is also accepted by $N$ because we can simply take the $\varepsilon$-transition from $r_0$ to $p_0$ and then follow the same transitions that would be followed in $M_A$ to an accept state. By similar reasoning, every string accepted by $M_B$ is also accepted by $N$. Finally, every string that is accepted by $N$ must be accepted by either $M_A$ or $M_B$ (or both), because every accepting computation of $N$ begins with one of the two $\varepsilon$-transitions and then necessarily mimics an accepting computation of either $M_A$ or $M_B$ depending on which $\varepsilon$-transition was taken. It therefore follows that

$$L(N) = L(M_A) \cup L(M_B) = A \cup B,$$

and so we conclude that $A \cup B$ is regular.

Next we will prove that $AB$ is regular. The idea is similar to the proof that $A \cup B$ is regular: we will use the DFAs $M_A$ and $M_B$ to define an NFA $N$ for the language $AB$. This time we will take the state set of $N$ to be the union $P \cup Q$, and the start state $p_0$ of $M_A$ will be the start state of $N$. All of the transitions defined by $M_A$ and $M_B$ will be included in $N$, and in addition we will add an $\varepsilon$-transition from each accept state of $M_A$ to the start state of $M_B$. Finally, the accept states of $N$ will be just the accept states $G$ of $M_B$ (and not the accept states of $M_A$). Figure 4.2 illustrates the construction of $N$ based on $M_A$ and $M_B$.

In formal terms, $N$ is the NFA defined as

$$N = (P \cup Q, \Sigma, \eta, p_0, G)$$

where the transition function

$$\eta : (P \cup Q) \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(P \cup Q)$$
Lecture 4

is given by

\[
\begin{align*}
\eta(p, a) &= \{\delta(p, a)\} \quad \text{(for all } p \in P \text{ and } a \in \Sigma), \\
\eta(q, a) &= \{\mu(q, a)\} \quad \text{(for all } q \in Q \text{ and } a \in \Sigma), \\
\eta(p, \varepsilon) &= \{q_0\} \quad \text{(for all } p \in F), \\
\eta(p, \varepsilon) &= \emptyset \quad \text{(for all } p \in P \setminus F), \\
\eta(q, \varepsilon) &= \emptyset \quad \text{(for all } q \in Q).
\end{align*}
\]

Along similar lines to what was done in the proof that \(A \cup B\) is regular, one can argue that \(N\) recognizes the language \(AB\), from which it follows that \(AB\) is regular.

Finally we will prove that \(A^*\) is regular, and once again the proof proceeds along similar lines. This time we will just consider \(M_A\) and not \(M_B\) because the language \(B\) is not involved. Let us start with the formal specification of \(N\) this time; define

\[
N = (R, \Sigma, \eta, r_0, \{r_0\})
\]

where \(R = P \cup \{r_0\}\) and the transition function

\[
\eta : R \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(R)
\]

is defined as

\[
\begin{align*}
\eta(r_0, a) &= \emptyset \quad \text{(for all } a \in \Sigma), \\
\eta(r_0, \varepsilon) &= p_0, \\
\eta(p, a) &= \{\delta(p, a)\} \quad \text{(for every } p \in P \text{ and } a \in \Sigma), \\
\eta(p, \varepsilon) &= \{r_0\} \quad \text{(for every } p \in F), \\
\eta(p, \varepsilon) &= \emptyset \quad \text{(for every } p \in P \setminus F). \\
\end{align*}
\]

In words, we take \(N\) to be the NFA whose states are the states of \(M_A\) along with an additional state \(r_0\), which is both the start state of \(N\) and its only accept state. The transitions of \(N\) include all of the transitions of \(M_A\), along with an \(\varepsilon\)-transition from \(r_0\) to the start state \(p_0\) of \(M_A\), and \(\varepsilon\)-transitions from all of the accept states of \(M_A\) back to \(r_0\). Figure 4.3 provides an illustration of how \(N\) relates to \(M_A\).

It is evident that \(N\) recognizes the language \(A^*\). This is because the strings it accepts are precisely those strings that cause \(N\) to start at \(r_0\) and loop back to \(r_0\) zero or more times, with each loop corresponding to some string that is accepted by \(M_A\). As \(L(N) = A^*\), it follows that \(A^*\) is regular, and so the proof is complete. \(\square\)

It is natural to ask why we could not easily conclude, for a regular language \(A\), that \(A^*\) is regular using the facts that the regular languages are closed under union and concatenation. In more detail, we have that

\[
A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \cdots
\]
It is easy to see that the language \( \{ \varepsilon \} \) is regular—here is the state diagram for an
NFA that recognizes the language \( \{ \varepsilon \} \) (for any choice of an alphabet):

The language \( \{ \varepsilon \} \cup A \) is therefore regular because the union of two regular lan-
guages is also regular. We also have that \( AA \) is regular because the concatenation
of two regular languages is regular, and therefore \( \{ \varepsilon \} \cup A \cup AA \) is regular because
it is the union of the two regular languages \( \{ \varepsilon \} \cup A \) and \( AA \). Continuing on like
this we find that the language

\[
\{ \varepsilon \} \cup A \cup AA \cup AAA
\]

is regular, the language

\[
\{ \varepsilon \} \cup A \cup AA \cup AAA \cup AAAAA
\]

is regular, and so on. Does this imply that \( A^* \) is regular?

The answer is “no.” Although it is true that \( A^* \) is regular whenever \( A \) is regular,
as we proved earlier, the argument just suggested based on combining unions and
concatenations alone does not establish it. This is because we can never conclude
from this argument that the infinite union (4.12) is regular, but only that finite
unions such as (4.14) are regular.
If you are still skeptical or uncertain, consider this statement:

If $A$ is a finite language, then $A^*$ is also a finite language.

This statement is false in general. For example, $A = \{0\}$ is finite, but

$$A^* = \{\varepsilon, 0, 00, 000, \ldots\}$$

is infinite. On the other hand, it is true that the union of two finite languages is finite, and the concatenation of two finite languages is finite, so something must go wrong when you try to combine these facts in order to conclude that $A^*$ is finite. The situation is similar when the property of being finite is replaced by the property of being regular.

### 4.2 Other closure properties of regular languages

There are many other operations on languages aside from the regular operations under which the regular languages are closed. For example, the complement of a regular language is also regular. Just to be sure the terminology is clear, here is the definition of the complement of a language.

**Definition 4.3.** Let $A \subseteq \Sigma^*$ be a language over the alphabet $\Sigma$. The complement of $A$, which is denoted $\overline{A}$, is the language consisting of all strings over $\Sigma$ that are not contained in $A$:

$$\overline{A} = \Sigma^* \setminus A.$$  \hspace{1cm} (4.16)

(We have already used the notation $S \setminus T$ a few times, but for the sake of clarity we will use a backslash to denote set differences in this course: $S \setminus T$ is the set of all elements in $S$ that are not in $T$.)

**Proposition 4.4.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language over the alphabet $\Sigma$. The language $\overline{A}$ is also regular.

This proposition is very easy to prove: because $A$ is regular, there must exist a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L(M) = A$. We obtain a DFA for the language $\overline{A}$ simply by swapping the accept and reject states of $M$. That is, the DFA $K = (Q, \Sigma, \delta, q_0, Q \setminus F)$ recognizes $\overline{A}$.

While it is easy to obtain a DFA for the complement of a language if you have a DFA for the original language simply by swapping the accept and reject states, this does not work for NFAs. You might, for instance, swap the accept and reject states of an NFA and end up with an NFA that recognizes something very different from
the complement of the language you started with. This is due to the asymmetric nature of accepting and rejecting for nondeterministic models.

Within the next few lectures we will see more examples of operations under which the regular languages are closed. Here is one more for this lecture.

**Proposition 4.5.** Let \( \Sigma \) be an alphabet and let \( A \) and \( B \) be regular languages over the alphabet \( \Sigma \). The intersection \( A \cap B \) is also regular.

This time we can just combine closure properties we already know to obtain this one. This is because De Morgan’s laws imply that

\[
A \cap B = \overline{A \cup \overline{B}}.
\]

If \( A \) and \( B \) are regular, then it follows that \( \overline{A} \) and \( \overline{B} \) are regular, and therefore \( \overline{A \cup \overline{B}} \) is regular, and because the complement of this regular language is \( A \cap B \) we have that \( A \cap B \) is regular.

There is another way to conclude that \( A \cap B \) is regular, which is arguably more direct. Because the languages \( A \) and \( B \) are regular, there must exist DFAs

\[
M_A = (P, \Sigma, \delta, p_0, F) \quad \text{and} \quad M_B = (Q, \Sigma, \mu, q_0, G)
\]

such that \( L(M_A) = A \) and \( L(M_B) = B \). We can obtain a DFA \( M \) recognizing \( A \cap B \) using a **Cartesian product** construction:

\[
M = (P \times Q, \Sigma, \eta, (p_0, q_0), F \times G)
\]

where

\[
\eta((p, q), a) = (\delta(p, a), \mu(q, a))
\]

for every \( p \in P \), \( q \in Q \), and \( a \in \Sigma \). In essence, the DFA \( M \) is what you get if you build a DFA that runs \( M_A \) and \( M_B \) in parallel, and accepts if and only if both \( M_A \) and \( M_B \) accept. You could also get a DFA for \( A \cup B \) using a similar idea (but accepting if and only if \( M_A \) accepts or \( M_B \) accepts).

### 4.3 Regular expressions

I expect that you already have some experience with regular expressions; they are very commonly used in programming languages and other applications to specify patterns for searching and string matching. When we use regular expressions in practice, we typically endow them with a rich set of operations, but in this class we take a minimal definition of regular expressions allowing only the three regular
operations (and not other operations like negation or special symbols marking the first or last characters of an input).

Here is the formal definition of regular expressions we will adopt. The definition is an example of an inductive definition, and some comments on inductive definitions will follow.

**Definition 4.6.** Let Σ be an alphabet. It is said that \( R \) is a *regular expression* over the alphabet \( \Sigma \) if any of these properties holds:

1. \( R = \emptyset \).
2. \( R = \varepsilon \).
3. \( R = a \) for some choice of \( a \in \Sigma \).
4. \( R = (R_1 \cup R_2) \) for regular expressions \( R_1 \) and \( R_2 \).
5. \( R = (R_1 R_2) \) for regular expressions \( R_1 \) and \( R_2 \).
6. \( R = (R_1^* \) for a regular expression \( R_1 \).

When you see an inductive definition such as this one, you should interpret it in the most sensible way, as opposed to thinking of it as something circular or paradoxical. For instance, when it is said that \( R = (R_1^* \) for a regular expression \( R_1 \), it is to be understood that \( R_1 \) is already well-defined as a regular expression. We cannot, for instance, take \( R_1 \) to be the regular expression \( R \) that we are defining—for then we would have \( R = (R)^* \), which might be interpreted as a strange, fractal-like expression that looks like this:

\[
R = (((\cdots (\cdots)^{*} \cdots)^{*})^*)^*.
\] (4.21)

Such a thing makes no sense as a regular expression, and is not valid according to a sensible interpretation of the definition. Here are some valid examples of regular expressions over the binary alphabet \( \Sigma = \{0, 1\} \):

\[
\emptyset \\
\varepsilon \\
0 \\
1 \\
(0 \cup 1) \\
((0 \cup 1)^*) \\
(((0 \cup \varepsilon)^*)1)
\]
When we are talking about regular expressions over an alphabet $\Sigma$, you should think of them as being strings over the alphabet

$$\Sigma \cup \{ (, ), *, \cup, \varepsilon, \emptyset \} \quad (4.22)$$

(assuming of course that $\Sigma$ and $\{ (, ), *, \cup, \varepsilon, \emptyset \}$ are disjoint). Some authors will use a different font for regular expressions so that this is more obvious, but I will not do this (partly because my handwriting on the board only has one font, but also because I don’t think it is really necessary).

Next we will define the language recognized (or matched) by a given regular expression. Again it is an inductive definition, and it directly parallels the regular expression definition itself. If it looks to you like it is stating something obvious, then your impression is correct; we require a formal definition, but it essentially says that we should define the language matched by a regular expression in the most straightforward and natural way.

**Definition 4.7.** Let $R$ be a regular expression over the alphabet $\Sigma$. The language recognized by $R$, which is denoted $L(R)$, is defined as follows:

1. If $R = \emptyset$, then $L(R) = \emptyset$.
2. If $R = \varepsilon$, then $L(R) = \{ \varepsilon \}$.
3. If $R = a$ for $a \in \Sigma$, then $L(R) = \{ a \}$.
4. If $R = (R_1 \cup R_2)$ for regular expressions $R_1$ and $R_2$, then $L(R) = L(R_1) \cup L(R_2)$.
5. If $R = (R_1 R_2)$ for regular expressions $R_1$ and $R_2$, then $L(R) = L(R_1) L(R_2)$.
6. If $R = (R_1^*)$ for a regular expression $R_1$, then $L(R) = L(R_1)^*$.

**Order of precedence for regular operations**

It might appear that regular expressions arising from Definition 4.6 have a lot of parentheses. For instance, the regular expression $(((0 \cup \varepsilon)^*)1)$ has more parentheses than it has non-parenthesis symbols. The parentheses ensure that every regular expression has an unambiguous meaning.

We can, however, reduce the need for so many parentheses by introducing an order of precedence for the regular operations. The order is as follows:

1. star (highest precedence)
2. concatenation
3. union (lowest precedence).
To be more precise, we’re not changing the formal definition of regular expressions, we’re just introducing a convention that allows some parentheses to be implicit, which makes for simpler-looking regular expressions. For example, we write

\[ 10^* \cup 1 \]  

rather than

\[ ((1(0^*)) \cup 1). \]

Having agreed upon the order of precedence above, the simpler-looking expression is understood to mean the second expression.

A simple way to remember the order of precedence is to view the regular operations as being analogous to algebraic operations that you are already familiar with: star looks like exponentiation, concatenation looks like multiplication, and unions are similar to additions. So, just as the expression \( xy^2 + z \) has the same meaning as \( ((x(y^2)) + z) \), the expression \( 10^* \cup 1 \) has the same meaning as \( ((1(0^*)) \cup 1) \).

**Regular expressions characterize the regular languages**

At this point it is natural to ask which languages have regular expressions. The answer is that the class of languages having regular expressions is precisely the class of regular languages. (If it were otherwise, you would have to wonder why the regular languages were named in this way!)

There are two implications needed to establish that the regular languages coincides with the class of languages having regular expressions. Let us start with the first implication, which is the content of the following proposition.

**Proposition 4.8.** Let \( \Sigma \) be an alphabet and let \( R \) be a regular expression over the alphabet \( \Sigma \). The language \( L(R) \) is regular.

The idea behind a proof of this proposition is simple enough: we can easily build DFAs for the languages \( \emptyset \), \( \{\epsilon\} \), and \( \{a\} \) (for each symbol \( a \in \Sigma \)), and by repeatedly using the constructions described in the proof of Theorem 4.2, one can combine together such DFAs to build an NFA recognizing the same language as any given regular expression.

The other implication is the content of the following theorem, which is more difficult to prove than the proposition above. I’ve included a proof in case you are interested, but you can also feel free to skip it if you’re not interested; there won’t be questions on any exams or homework assignments in this class that require you to have studied the proof.

**Theorem 4.9.** Let \( \Sigma \) be an alphabet and let \( A \subseteq \Sigma^* \) be a regular language. There exists a regular expression over the alphabet \( \Sigma \) such that \( L(R) = A \).
Proof. Because $A$ is regular, there must exist a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L(M) = A$. We are free to use whatever names we like for the states of a DFA, so no generality is lost in assuming $Q = \{1, \ldots, n\}$ for some positive integer $n$.

We are now going to define a language $B^k_{p,q} \subseteq \Sigma^*$, for every choice of states $p, q \in \{1, \ldots, n\}$ and an integer $k \in \{0, \ldots, n\}$. The language $B^k_{p,q}$ is the set of all strings $w$ causing $M$ to operate as follows:

If we start $M$ in the state $p$, then by reading $w$ the DFA $M$ moves to the state $q$. Moreover, aside from the beginning state $p$ and the ending state $q$, the DFA $M$ only touches states contained in the set $\{1, \ldots, k\}$ when reading $w$ in this way.

For example, the language $B^n_{p,q}$ is simply the set of all strings causing $M$ to move from $p$ to $q$ (because restricting the intermediate states that $M$ touches to those contained in the set $\{1, \ldots, n\}$ is no restriction whatsoever). At the other extreme, the set $B^0_{p,q}$ must be a finite set; it could be the empty set if there are no direct transitions from $p$ to $q$, it includes the empty string in the case $p = q$, and in general it includes a length-one string corresponding to each symbol that causes $M$ to transition from $p$ to $q$.

Now, we will prove by induction on $k$ that there exists a regular expression $R^k_{p,q}$ satisfying $L(R^k_{p,q}) = B^k_{p,q}$, for every choice of $p, q \in \{1, \ldots, n\}$ and $k \in \{0, \ldots, n\}$. The base case is $k = 0$. The language $B^0_{p,q}$ is finite for every $p, q \in \{1, \ldots, n\}$, consisting entirely of strings of length 0 or 1, so it is straightforward to define a corresponding regular expression $R^0_{p,q}$ that matches $B^0_{p,q}$.

For the induction step, we assume $k \geq 1$, and that there exists a regular expression $R^{k-1}_{p,q}$ satisfying $L(R^{k-1}_{p,q}) = B^{k-1}_{p,q}$ for every $p, q \in \{1, \ldots, n\}$. It is the case that

$$B^k_{p,q} = B^{k-1}_{p,q} \cup B^{k-1}_{p,k} (R^{k-1}_{k,k})^* B^{k-1}_{k,q}. \quad (4.25)$$

This equality reflects the fact that the strings that cause $M$ to move from $p$ to $q$ through the intermediate states $\{1, \ldots, k\}$ are precisely those strings that either (i) cause $M$ to move from $p$ to $q$ through the intermediate states $\{1, \ldots, k\}$, but in fact without actually visiting $k$ as an intermediate state, or (ii) cause $M$ to move from $p$ to $q$ through the intermediate states $\{1, \ldots, k\}$, and visiting state $k$ as an intermediate state one or more times. We may therefore define a regular expression $R^k_{p,q}$ satisfying $L(R^k_{p,q}) = B^k_{p,q}$ for every $p, q \in \{1, \ldots, n\}$ as

$$R^k_{p,q} = R^{k-1}_{p,q} \cup R^{k-1}_{p,k} (R^{k-1}_{k,k})^* R^{k-1}_{k,q}. \quad (4.26)$$

Finally, we obtain a regular expression $R$ satisfying $L(R) = A$ by defining

$$R = \bigcup_{q \in F} R^0_{q_0,q}. \quad (4.27)$$
Lecture 4

(In words, $R$ is the regular expression we obtain by forming the union over all regular expressions $R^m_{q_0,q}$ where $q$ is an accept state.) This completes the proof. □

There is a procedure that can be used to convert a given DFA into an equivalent regular expression. The idea behind this conversion process is similar to the idea of the proof above. It tends to get messy, producing rather large and complicated-looking regular expressions from relatively simple DFAs, but it works—and just like the conversion of an NFA to an equivalent DFA, it can be implemented by a computer.
Lecture 5

Proving languages to be nonregular

We already know, for any alphabet $\Sigma$, that there exist languages $A \subseteq \Sigma^*$ that are nonregular. This is because there are uncountably many languages over $\Sigma$ but only countably many regular languages over $\Sigma$. However, this observation does not give us a method to prove that specific nonregular languages are indeed nonregular. In this lecture we will discuss a method that can be used to prove that a fairly wide selection of languages are nonregular.

5.1 The pumping lemma (for regular languages)

We will begin by proving a simple fact—known as the pumping lemma—that must hold for all regular languages. A bit later in the lecture, in the section following this one, we will then use this fact to conclude that certain languages are nonregular.

**Lemma 5.1** (Pumping lemma for regular languages). Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language. There exists a positive integer $n$ (called a pumping length of $A$) that possesses the following property. For every string $w \in A$ with $|w| \geq n$, it is possible to write $w = xyz$ for some choice of strings $x, y, z \in \Sigma^*$ such that

1. $y \neq \epsilon$,
2. $|xy| \leq n$, and
3. $xy^iz \in A$ for all $i \in \mathbb{N}$.

The pumping lemma is essentially a precise, technical way of expressing one simple consequence of the following fact:

If a DFA with $n$ or fewer states reads $n$ or more symbols from an input string, at least one of its states must have been visited more than once.
This means that if a DFA with \( n \) states reads a particular string having length at least \( n \), then there must be a substring of that input string that causes a loop, meaning that the DFA starts and ends on the same state. If the DFA accepts the original string, then by repeating that substring that caused a loop multiple times, or alternatively removing it altogether, we obtain a different string that is also accepted by the DFA. It may be helpful to try to match this intuition to the proof that follows.

**Proof of Lemma 5.1.** Let \( M = (Q, \Sigma, \delta, q_0, F) \) be a DFA that recognizes \( A \) and let \( n = |Q| \) be the number of states of \( M \). We will prove that the property stated in the pumping lemma is satisfied for this choice of \( n \).

Let us note first that if there is no string contained in \( A \) that has length \( n \) or larger, then there is nothing more we need to do: the property stated in the lemma is trivially satisfied in this case. We may therefore move on to the case in which \( A \) does contain at least one string having length at least \( n \). In particular, suppose that \( w \in A \) is a string such that \( |w| \geq n \). We may write

\[
w = a_1 \cdots a_m\]  

for \( m = |w| \) and \( a_1, \ldots, a_m \in \Sigma \). Because \( w \in A \) it must be the case that \( M \) accepts \( w \), and therefore there exist states

\[
r_0, r_1, \ldots, r_m \in Q\]  

such that \( r_0 = q_0, r_m \in F \), and

\[
r_{k+1} = \delta(r_k, a_{k+1})\]  

for every \( k \in \{0, \ldots, m - 1\} \).

Now, the sequence \( r_0, r_1, \ldots, r_n \) has \( n + 1 \) members, but there are only \( n \) different elements in \( Q \), so at least one of the states of \( Q \) must appear more than once in this sequence. (This is an example of the so-called pigeon hole principle: if \( n + 1 \) pigeons fly into \( n \) holes, then at least one of the holes must contain two or more pigeons.) Thus, there must exist indices \( s, t \in \{0, \ldots, n\} \) satisfying \( s < t \) such that \( r_s = r_t \).

Next, define strings \( x, y, z \in \Sigma^* \) as follows:

\[
x = a_1 \cdots a_s, \quad y = a_{s+1} \cdots a_t, \quad z = a_{t+1} \cdots a_m.\]  

It is the case that \( w = xyz \) for this choice of strings, so to complete the proof, we just need to demonstrate that these strings fulfill the three conditions that are listed in the lemma. The first two conditions are immediate: we see that \( y \) has length \( t - s \), which is at least 1 because \( s < t \), and therefore \( y \neq \varepsilon \); and we see that \( xy \) has length \( t \), which is at most \( n \) because \( t \) was chosen from the set \( \{0, \ldots, n\} \). It remains
Lecture 5

to verify that $xy^iz \in A$, which is equivalent to $M$ accepting $xy^iz$, for every $i \in \mathbb{N}$. That fact that $xy^iz$ is accepted by $M$ follows from the verification that the sequence of states

$$r_0, \ldots, r_s, r_{s+1}, \ldots, r_t, r_{t+1}, \ldots, r_m$$

repeated $i$ times

satisfies the definition of acceptance of the string $xy^iz$ by the DFA $M$. □

If the proof of the pumping lemma, or the idea behind it, is not clear, it may be helpful to see it in action for an actual DFA and a long enough string accepted by that DFA. For instance, let us take $M$ to be the DFA having the state diagram illustrated in Figure 5.1.

![State Diagram](image)

Figure 5.1: The state diagram of a DFA $M$, to be used to provide an example to explain the pumping lemma.

Now consider any string $w$ having length at least 6 (which is the number of states of $M$) that is accepted by $M$. For instance, let us take $w = 0110111$. This causes $M$ to move through this sequence of states:

$$q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_5 \xrightarrow{0} q_4 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_5$$

(5.6)

(The arrows represent the transitions and the symbols above the arrows indicate which input symbol has caused this transition.) Sure enough, there is at least one state that appears multiple times in the sequence, and in this particular case there are actually three such states: $q_1$, $q_2$, and $q_5$, each of which appear twice. Let us focus on the two appearances of the state $q_1$, just because this state happens to be
the one that gets revisited first. It is the substring 1101 that causes $M$ to move in a
loop starting and ending on the state $q_1$. In the statement of the pumping lemma
this corresponds to taking

$$x = 0, \quad y = 1101, \quad \text{and} \quad z = 11.$$ (5.7)

Because the substring $y$ causes $M$ to move from the state $q_1$ back to $q_1$, it is as if
reading $y$ when $M$ is in the state $q_1$ has no effect. So, given that $x$ causes $M$ to move
from the initial state $q_0$ to the state $q_1$, and $z$ causes $M$ to move from $q_1$ to an accept
state, we see that $M$ must not only accept $w = xyz$, but it must also accept $xz, xyyz,$
$xyyyz$, and so on.

There is nothing special about the example just described; something similar
always happens. Pick any DFA whatsoever, and then pick any string accepted by
that DFA that has length at least the number of states of the DFA, and you will
be able to find a loop like we did above. By repeating input symbols in the most
natural way so that the loop is followed multiple times (or no times) you will ob-
tain different strings accepted by the DFA. This is essentially all that the pumping
lemma is saying.

### 5.2 Using the pumping lemma to prove nonregularity

It is helpful to keep in mind that the pumping lemma is a statement about regular
languages: it establishes a property that must always hold for every chosen regular
language.

Although the pumping lemma is a statement about regular languages, we can
use it to prove that certain languages are not regular using the technique of proof
by contradiction. In particular, we take the following steps:

1. For $A$ being the language we hope to prove is nonregular, we make the assump-
tion that $A$ is regular. Operating under the assumption that the language $A$ is
regular, we apply the pumping lemma to it.
2. Using the property that the pumping lemma establishes for $A$, we derive a
contradiction. The contradiction will almost always be that we conclude that
some particular string is contained in $A$ that we know is actually not contained
in $A$.
3. Having derived a contradiction, we conclude that it was our assumption that $A$
is regular that led to the contradiction, and so we deduce that $A$ is nonregular.

Let us see this method in action for a few examples. These examples will be
stated as propositions, with the proofs showing you how the argument works.
Lecture 5

**Proposition 5.2.** Let $\Sigma = \{0, 1\}$ be the binary alphabet and define a language over $\Sigma$ as follows:

$$ A = \{0^m1^m : m \in \mathbb{N}\}. \quad (5.8) $$

The language $A$ is not regular.

*Proof.* Assume toward contradiction that $A$ is regular. By the pumping lemma for regular languages, there must exist a pumping length $n \geq 1$ for $A$ for which the property stated by that lemma holds. We will fix such a pumping length $n$ for the remainder of the proof.

Define $w = 0^n1^n$ (where $n$ is the pumping length we just fixed). It is the case that $w \in A$ and $|w| = 2n \geq n$, so the pumping lemma tells us that there exist strings $x, y, z \in \Sigma^*$ so that $w = xyz$ and the following conditions hold:

1. $y \neq \varepsilon$,
2. $|xy| \leq n$, and
3. $xy^iz \in A$ for all $i \in \mathbb{N}$.

Now, because $xyz = 0^n1^n$ and $|xy| \leq n$, the substring $y$ cannot have any 1s in it, as the substring $xy$ is not long enough to reach the 1s in $xyz$. This means that $y = 0^k$ for some choice of $k \in \mathbb{N}$, and because $y \neq \varepsilon$, we conclude moreover that $k \geq 1$. We may also conclude that

$$ xy^2z = xyyz = 0^{n+k}1^n. \quad (5.9) $$

This is because $xyyz$ is the string obtained by inserting $y = 0^k$ somewhere in the initial portion of the string $xyz = 0^n1^n$, before any 1s have appeared. More generally it holds that

$$ xy^iz = 0^{n+(i-1)k}1^n \quad (5.10) $$

for each $i \in \mathbb{N}$. (We don’t actually need this more general formula for the sake of the current proof, but in other similar cases a formula like this can be helpful.)

However, because $k \geq 1$, we see that the string $xy^2z = 0^{n+k}1^n$ is *not* contained in $A$. This contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^iz \in A$ for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that our assumption that $A$ is regular was wrong. The language $A$ is therefore nonregular, as required. \qed

**Proposition 5.3.** Let $\Sigma = \{0, 1\}$ be the binary alphabet and define a language over $\Sigma$ as follows:

$$ B = \{0^m1^r : m, r \in \mathbb{N}, m > r\}. \quad (5.11) $$

The language $B$ is not regular.
Proof. Assume toward contradiction that $B$ is regular. By the pumping lemma for regular languages, there must exist a pumping length $n \geq 1$ for $B$ for which the property stated by that lemma holds. We will fix such a pumping length $n$ for the remainder of the proof.

Define $w = 0^{n+1}1^n$. We see that $w \in B$ and $|w| = 2n + 1 \geq n$, so the pumping lemma tells us that there exist strings $x, y, z \in \Sigma^*$ so that $w = xyz$ and the following conditions are satisfied:

1. $y \neq \varepsilon$,
2. $|xy| \leq n$, and
3. $xy^iz \in B$ for all $i \in \mathbb{N}$.

Now, because $xyz = 0^{n+1}1^n$ and $|xy| \leq n$, it must be that $y = 0^k$ for some choice of $k \geq 1$. (The reasoning here is just like in the previous proposition.) This time we have

$$xy^iz = 0^{n+1+(i-1)k}1^n$$

for each $i \in \mathbb{N}$. In particular, if we choose $i = 0$, then we have

$$xy^0z = xz = 0^{n+1-k}1^n.$$ (5.13)

However, because $k \geq 1$, and therefore $n + 1 - k \leq n$, we see that the string $xy^0z$ is not contained in $B$. This contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^iz \in B$ for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that our assumption that $B$ is regular was wrong. The language $B$ is therefore nonregular, as required. \qed

Remark 5.4. In the previous proof, it was important that we could choose $i = 0$ to get a contradiction—no other choice of $i$ would have worked.

Proposition 5.5. Let $\Sigma = \{0\}$ and define a language over $\Sigma$ as follows:

$$C = \{0^m : m \text{ is a perfect square (i.e., } m = j^2 \text{ for some } j \in \mathbb{N}\}.$$ (5.14)

The language $C$ is not regular.

Proof. Assume toward contradiction that $C$ is regular. By the pumping lemma for regular languages, there must exist a pumping length $n \geq 1$ for $C$ for which the property stated by that lemma holds. We will fix such a pumping length $n$ for the remainder of the proof.
Lecture 5

Define \( w = 0^{n^2} \). We observe that \( w \in C \) and \( |w| = n^2 \geq n \), so the pumping lemma tells us that there exist strings \( x, y, z \in \Sigma^* \) so that \( w = xyz \) and the following conditions are satisfied:

1. \( y \neq \epsilon \),
2. \( |xy| \leq n \), and
3. \( xy^iz \in C \) for all \( i \in \mathbb{N} \).

There is only one symbol in the alphabet \( \Sigma \), so this time it is immediate that \( y = 0^k \) for some choice of \( k \in \mathbb{N} \). Because \( y \neq \epsilon \) and \( |y| \leq |xy| \leq n \), it must be the case that \( 1 \leq k \leq n \), and therefore

\[
xy^iz = 0^{n^2+(i-1)k}
\]

for each \( i \in \mathbb{N} \). In particular, if we choose \( i = 2 \), then we have

\[
xy^2z = xyyz = 0^{n^2+k}.
\]

However, because \( 1 \leq k \leq n \), it cannot be that \( n^2 + k \) is a perfect square; the number \( n^2 + k \) is larger than \( n^2 \), but the next perfect square after \( n^2 \) is

\[
(n+1)^2 = n^2 + 2n + 1,
\]

which is strictly larger than \( n^2 + k \) because \( k \leq n \). The string \( xy^2z \) is therefore not contained in \( C \), which contradicts the third condition stated by the pumping lemma, which guarantees us that \( xy^iz \in C \) for all \( i \in \mathbb{N} \).

Having obtained a contradiction, we conclude that our assumption that \( C \) is regular was wrong. The language \( C \) is therefore nonregular, as required. \( \square \)

For the next example we will use some notation that will appear from time to time throughout the course. For a given string \( w \), the string \( w^R \) denotes the reverse of the string \( w \). Formally speaking, we may define the string reversal operation inductively as follows:

1. \( \epsilon^R = \epsilon \), and
2. \( (aw)^R = w^Ra \) for every \( w \in \Sigma^* \) and \( a \in \Sigma \).

**Proposition 5.6.** Let \( \Sigma = \{0, 1\} \) and define a language over \( \Sigma \) as follows:

\[
D = \{ w \in \Sigma^* : w = w^R \}.
\]

The language \( D \) is not regular.
Proof. Assume toward contradiction that \( D \) is regular. By the pumping lemma for regular languages, there must exist a pumping length \( n \geq 1 \) for \( D \) for which the property stated by that lemma holds. We will fix such a pumping length \( n \) for the remainder of the proof.

Define \( w = 0^n10^n \). We observe that \( w \in D \) and \( |w| = 2n + 1 \geq n \), so the pumping lemma tells us that there exist strings \( x, y, z \in \Sigma^* \) so that \( w = xyz \) and the following conditions are satisfied:

1. \( y \neq \varepsilon \),
2. \( |xy| \leq n \), and
3. \( xy^iz \in D \) for all \( i \in \mathbb{N} \).

Once again, we may conclude that \( y = 0^k \) for \( k \geq 1 \). This time it is the case that

\[
xy^iz = 0^{n+(i-1)k}10^n
\]  
(5.19)

for each \( i \in \mathbb{N} \). In particular, if we choose \( i = 2 \), then we have

\[
xy^2z = xyyz = 0^{n+k}10^n.
\]  
(5.20)

Because \( k \geq 1 \), this string is not equal to its own reverse, and therefore \( xy^2z \) is therefore not contained in \( D \). This contradicts the third condition stated by the pumping lemma, which guarantees us that \( xy^iz \in D \) for all \( i \in \mathbb{N} \).

Having obtained a contradiction, we conclude that our assumption that \( D \) is regular was wrong. The language \( D \) is therefore nonregular, as required. \( \square \)

The four propositions above should give you an idea of how the pumping lemma can be used to prove languages are nonregular. The set-up is always the same: we assume toward contradiction that a particular language is regular, and observe that the pumping lemma gives us a pumping length \( n \). At that point it is time to choose the string \( w \), try to use some reasoning, and derive a contradiction.

It may not always be clear what string \( w \) to choose or how exactly to get a contradiction; these steps will depend on the language you’re working with, there may be multiple good choices for \( w \), and there may be some creativity and/or insight involved in getting it all to work. One thing you can always try is to make a reasonable guess for what string might work, try to get a contradiction, and if you don’t succeed then make another choice for \( w \) based on whatever insight you’ve gained by failing to get a contradiction from your first choice. Of course you should always be convinced by your own arguments and actively look for ways they might be going wrong. If you don’t truly believe your own proof, it’s not likely anyone else will believe it either.
5.3 Nonregularity from closure properties

Sometimes you can prove that a particular language is nonregular by combining together closure properties for regular languages with your knowledge of other languages being nonregular. Here are two examples, again stated as propositions. (In the proofs of these propositions, the languages $B$ and $D$ refer to the languages proved to be nonregular in the previous section.)

**Proposition 5.7.** Let $\Sigma = \{0, 1\}$ and define a language over $\Sigma$ as follows:

$$E = \{ w \in \Sigma^* : w \neq w^R \}. \quad (5.21)$$

The language $E$ is not regular.

**Proof.** Assume toward contradiction that $E$ is regular. The regular languages are closed under complementation, and therefore $\overline{E}$ is regular. However, $\overline{E} = D$, which we already proved is nonregular. This is a contradiction, and therefore our assumption that $E$ is regular was wrong. We conclude that $E$ is nonregular, as claimed.  

**Proposition 5.8.** Let $\Sigma = \{0, 1\}$ and define a language over $\Sigma$ as follows:

$$F = \{ w \in \Sigma^* : w \text{ has more 0s than 1s} \}. \quad (5.22)$$

The language $F$ is not regular.

**Proof.** Assume toward contradiction that $F$ is regular. We know that the language $L(0^*1^*)$ is regular because it is the language matched by a regular expression. The regular languages are closed under intersection, so $F \cap L(0^*1^*)$ is regular. However, we have that

$$F \cap L(0^*1^*) = B, \quad (5.23)$$

which we already proved is nonregular. This is a contradiction, and therefore our assumption that $F$ is regular was wrong. We conclude that $F$ is nonregular, as claimed.  

It is important to remember, when using this method, that it is the regular languages that are closed under operations such as complementation, intersection, union, and so on, not the nonregular languages. For instance, it is not generally the case that the intersection of two nonregular languages is nonregular—so a proof would not be valid if it were to rely on such a claim.
Lecture 6

Further discussion of regular languages

This is the last lecture of the course to be devoted to regular languages, but we will refer back to regular languages frequently and relate them to various computational models and classes of languages as the course progresses. For the most part we will use this lecture to relate some of the different concepts we have already discussed, introduce a few new concepts along the way, and go over some examples of problems concerning regular languages.

6.1 Other operations on languages

We’ve discussed some basic operations on languages, including the regular operations (union, concatenation, and star) and a few others (such as complementation and intersection). There are many other operations that one can consider—you could probably sit around all day thinking of increasingly obscure examples if you wanted to—but for now we’ll take a look at just a few more.

Reverse

Suppose $\Sigma$ is an alphabet and $w \in \Sigma^*$ is a string. The reverse of the string $w$, which we denote by $w^R$, is the string obtained by rearranging the symbols of $w$ so that they appear in the opposite order. As we observed in the previous lecture, the reverse of a string may be defined inductively as follows:

1. If $w = \epsilon$, then $w^R = \epsilon$.
2. If $w = ax$ for $a \in \Sigma$ and $x \in \Sigma^*$, then $w^R = x^Ra$.
Now suppose that $A \subseteq \Sigma^*$ is a language. We define the *reverse* of $A$, which we denote by $A^R$, to be the language obtained by taking the reverse of each element of $A$. That is, we define

$$A^R = \{ w^R : w \in A \}.$$  

(6.1)

You can check that the following identities hold that relate the reverse operation to the regular operations:

$$(A \cup B)^R = A^R \cup B^R, \quad (AB)^R = B^R A^R, \quad \text{and} \quad (A^*)^R = (A^R)^*.$$  

(6.2)

A natural question concerning the reverse of a languages is this one:

If a language $A$ is regular, must its reverse $A^R$ also be regular?

The answer to this question is “yes.” Let us state this fact as a proposition and then consider two ways to prove it.

**Proposition 6.1.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language. The language $A^R$ is regular.

**First proof.** There is a natural way of defining the reverse of a regular expression that mirrors the identities (6.2) above. In particular, if $S$ is a regular expression, then its reverse regular expression can be defined inductively as follows:

1. If $S = \emptyset$ then $S^R = \emptyset$.
2. If $S = \varepsilon$ then $S^R = \varepsilon$.
3. If $S = a$ for some choice of $a \in \Sigma$, then $S^R = a$.
4. If $S = (S_1 \cup S_2)$ for regular expressions $S_1$ and $S_2$, then $S^R = (S_1^R \cup S_2^R)$.
5. If $S = (S_1 S_2)$ for regular expressions $S_1$ and $S_2$, then $S^R = (S_2^R S_1^R)$.
6. If $S = (S_1^*)$ for a regular expression $S_1$, then $S^R = ((S_1^R)^*)$.

It is evident that $L(S^R) = L(S)^R$; for any regular expression $S$, the reverse regular expression $S^R$ matches the reverse of the language matched by $S$.

Now, under the assumption that $A$ is regular, there must exist a regular expression $S$ such that $L(S) = A$, because every regular language is matched by some regular expression. The reverse of the regular expression $S$ is $S^R$, which is also a valid regular expression. The language matched by any regular expression is regular, and therefore $L(S^R)$ is regular. Because $L(S^R) = L(S)^R = A^R$, we have that $A^R$ is regular, as required.
Lecture 6

*Second proof (sketch).* (We’ll consider this as a proof “sketch” because it just summarizes the main idea without covering the details of why it works.) Under the assumption that $A$ is regular, there must exist a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L(M) = A$. We can design an NFA $N$ such that $L(N) = A^R$, thereby implying that $A^R$ is regular, by effectively running $M$ backwards in time (using the power of nondeterminism to do this because deterministic computations are generally not reversible).

Here is the natural way to define an NFA $N$ that does what we want:

$$N = (Q \cup \{r_0\}, \Sigma, \mu, r_0, \{q_0\}),$$

where it is assumed that $r_0$ is not contained in $Q$ (i.e., we are letting $N$ have the same states as $M$ along with a new start state $r_0$), and we take the transition function $\mu$ to be defined as follows:

$$\mu(r_0, \varepsilon) = F, \quad \mu(r_0, a) = \emptyset,$$
$$\mu(q, \varepsilon) = \emptyset, \quad \mu(q, a) = \{p \in Q : \delta(p, a) = q\},$$

for all $q \in Q$ and $a \in \Sigma$.

The way $N$ works is to first nondeterministically guess an accepting state of $M$, then it reads symbols from the input and nondeterministically chooses to move to a state for which $M$ would allow a move in the opposite direction on the same input symbol, and finally it accepts if it ends on the start state of $M$.

The most natural way to formally prove that $L(N) = L(M)^R$ is to refer to the definitions of acceptance for $N$ and $M$, and to check that a sequence of states satisfies the definition for $M$ accepting a string $w$ if and only if the reverse of that sequence of states satisfies the definition of acceptance for $N$ accepting $w^R$. \hfill \Box

**Symmetric difference**

Given two sets $A$ and $B$, we define the *symmetric difference* of $A$ and $B$ as

$$A \triangle B = (A \setminus B) \cup (B \setminus A).$$

In words, the elements of the symmetric difference $A \triangle B$ are those objects that are contained in either $A$ or $B$, but not both. Figure 6.1 illustrates the symmetric difference in the form of a Venn diagram.

It is not hard to conclude that if $\Sigma$ is an alphabet and $A, B \subseteq \Sigma^*$ are regular languages, then the symmetric difference $A \triangle B$ of these two languages is also regular. This is because the regular languages are closed under the operations union, intersection, and complementation, and the symmetric difference can be described in
Figure 6.1: The shaded region denotes the symmetric difference $A \triangle B$ of two sets $A$ and $B$.

terms of these operations. More specifically, if we assume that $A$ and $B$ are regular, then their complements $\overline{A}$ and $\overline{B}$ are also regular; which implies that the intersections $A \cap \overline{B}$ and $\overline{A} \cap B$ are also regular; and therefore the union $(A \cap \overline{B}) \cup (\overline{A} \cap B)$ of these two intersections is regular as well. Observing that we have

$$A \triangle B = (A \cap \overline{B}) \cup (\overline{A} \cap B),$$

we see that the symmetric difference of $A$ and $B$ is regular.

Prefix, suffix, and substring

Let $\Sigma$ be an alphabet and let $w \in \Sigma^*$ be a string. A prefix of $w$ is any string you can obtain from $w$ by removing zero or more symbols from the right-hand side of $w$; a suffix of $w$ is any string you can obtain by removing zero or more symbols from the left-hand side of $w$; and a substring of $w$ is any string you can obtain by removing zero or more symbols from either or both the left-hand side and right-hand side of $w$. We can state these definitions more formally as follows: (i) a string $x \in \Sigma^*$ is a prefix of $w \in \Sigma^*$ if there exists $v \in \Sigma^*$ such that $w = xv$, (ii) a string $x \in \Sigma^*$ is a suffix of $w \in \Sigma^*$ if there exists $u \in \Sigma^*$ such that $w = ux$, and (iii) a string $x \in \Sigma^*$ is a substring of $w \in \Sigma^*$ if there exist $u, v \in \Sigma^*$ such that $w = uv$.

For any language $A \subseteq \Sigma^*$, we will write Prefix$(A)$, Suffix$(A)$, and Substring$(A)$ to denote the languages containing all prefixes, suffixes, and substrings (respectively) that can be obtained from any choice of a string $w \in A$. That is, we define

$$\text{Prefix}(A) = \{ x \in \Sigma^* : \text{there exists } v \in \Sigma^* \text{ such that } xv \in A \},$$

$$\text{Suffix}(A) = \{ x \in \Sigma^* : \text{there exists } u \in \Sigma^* \text{ such that } ux \in A \},$$

$$\text{Substring}(A) = \{ x \in \Sigma^* : \text{there exist } u, v \in \Sigma^* \text{ such that } u xv \in A \}.$$
Lecture 6

Again we have a natural question concerning these concepts:

If a language $A$ is regular, must the languages $\text{Prefix}(A)$, $\text{Suffix}(A)$, and $\text{Substring}(A)$ also be regular?

The answer is “yes,” as the following proposition establishes.

**Proposition 6.2.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language over the alphabet $\Sigma$. The languages $\text{Prefix}(A)$, $\text{Suffix}(A)$, and $\text{Substring}(A)$ are regular.

**Proof.** Because $A$ is regular, there must exist a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L(M) = A$. Some of the states in $Q$ are reachable from the start state $q_0$, by following zero or more transitions specified by the transition function $\delta$. We may call this set $R$, so that

$$R = \{ q \in Q : \text{there exists } w \in \Sigma^* \text{ such that } \delta^*(q_0, w) = q \}. \quad (6.10)$$

Also, from some of the states in $Q$, it is possible to reach an accept state of $M$, by following zero or more transitions specified by the transition function $\delta$. We may call this set $P$, so that

$$P = \{ q \in Q : \text{there exist } w \in \Sigma^* \text{ such that } \delta^*(q, w) \in F \}. \quad (6.11)$$

(See Figure 6.2 for a simple example illustrating the definitions of these sets.)

First, define a DFA $K = (Q, \Sigma, \delta, q_0, P)$. In words, $K$ is the same as $M$ except that its accept states are all of the states in $M$ from which it is possible to reach an accept state of $M$. We see that $L(K) = \text{Prefix}(A)$, and therefore $\text{Prefix}(A)$ is regular.

Next, define an NFA $N = (Q \cup \{ r_0 \}, \Sigma, \eta, r_0, F)$, where the transition function $\eta$ is defined as

$$\eta(r_0, \varepsilon) = R,$$

$$\eta(q, a) = \{ \delta(q, a) \} \quad (\text{for each } q \in Q \text{ and } a \in \Sigma),$$

and $\eta$ takes the value $\emptyset$ in all other cases. In words, we define $N$ from $M$ by adding a new start state $r_0$, along with $\varepsilon$-transitions from $r_0$ to every reachable state in $M$. It holds that $L(N) = \text{Suffix}(A)$, and therefore $\text{Suffix}(A)$ is regular.

Finally, the fact that $\text{Substring}(A)$ is regular follows from the observation that $\text{Substring}(A) = \text{Suffix}(\text{Prefix}(A))$ (or that $\text{Substring}(A) = \text{Prefix}(\text{Suffix}(A))$).

---

1 If you were defining a DFA for some purpose, there would be no point in having states that are not reachable from the start state—but there is nothing in the definition of DFAs that forces all states to be reachable.
6.2 Example problems concerning regular languages

We will conclude with a few other examples of problems concerning regular languages along with their solutions.

**Problem 6.1.** Let $\Sigma = \{0, 1\}$ and let $A \subseteq \Sigma^*$ be a regular language. Prove that the language

$$B = \{uv : u, v \in \Sigma^* \text{ and } uav \in A \text{ for some choice of } a \in \Sigma\}$$

(6.12)

is regular.

The language $B$ can be described in intuitive terms as follows: it is the language of all strings that can be obtained by choosing a nonempty string $w$ from $A$ and deleting exactly one symbol of $w$.

**Solution.** A natural way to solve this problem is to describe an NFA for $B$, based on a DFA for $A$, which must exist by the assumption that $A$ is regular. This will imply that $B$ is regular, as every language recognized by an NFA is necessarily regular.

Along these lines, let us suppose that

$$M = (Q, \Sigma, \delta, q_0, F)$$

(6.13)

is a DFA for which $A = L(M)$. Define an NFA

$$N = (P, \Sigma, \eta, p_0, G)$$

(6.14)
as follows. First, we will define
\[ P = \{0, 1\} \times Q, \] (6.15)
and we will take the start state of \( N \) to be
\[ p_0 = (0, q_0). \] (6.16)
The accept states of \( N \) will be
\[ G = \{(1, q) : q \in F\}. \] (6.17)

It remains to describe the transition function \( \eta \) of \( N \), which will be as follows:

1. \( \eta((0, q), a) = \{(0, \delta(q, a))\} \) for every \( q \in Q \) and \( a \in \Sigma \).
2. \( \eta((0, q), \varepsilon) = \{(1, \delta(q, 0)), (1, \delta(q, 1))\} \) for every \( q \in Q \).
3. \( \eta((1, q), a) = \{(1, \delta(q, a))\} \) for every \( q \in Q \) and \( a \in \Sigma \).
4. \( \eta((1, q), \varepsilon) = \emptyset \) for every \( q \in Q \).

The idea behind the way that \( N \) operates is as follows. The NFA \( N \) starts in the state \((0, q_0)\) and simulates \( M \) for some number of steps. This is the effect of the transitions listed as 1 above. At some point, which is nondeterministically chosen, \( N \) follows an \( \varepsilon \)-transition from a state of the form \((0, q)\) to either the state \((1, \delta(q, 0))\) or the state \((1, \delta(q, 1))\). Intuitively speaking, \( N \) is reading nothing from its input while “hypothesizing” that \( M \) has read some symbol \( a \) (which is either 0 or 1). This is the effect of the transitions listed as 2. Then \( N \) simply continues simulating \( M \) on the remainder of the input string, which is the effect of the transitions listed as 3. There are no \( \varepsilon \)-transitions leading out of the states of the form \((1, q)\), which is why we have the values for \( \eta \) listed as 4.

If you think about the NFA \( N \) for a moment or two, it should become evident that it recognizes \( B \).

**Alternative Solution.** Here is a somewhat different solution that may appeal to some of you. Part of its appeal is that it illustrates a method that may be useful in other cases. In this case we will also discuss a somewhat more detailed proof of correctness (partly because it happens to be a bit easier for this solution).

Again, let
\[ M = (Q, \Sigma, \delta, q_0, F) \] (6.18)
be a DFA for which \( L(M) = A \). For each choice of \( p, q \in Q \), define a new DFA
\[ M_{p,q} = (Q, \Sigma, \delta, p, \{q\}), \] (6.19)
and let $A_{p,q} = L(M_{p,q})$. In words, $A_{p,q}$ is the regular language consisting of all strings that cause $M$ to transition to the state $q$ when started in the state $p$. For any choice of $p$, $q$, and $r$, we must surely have $A_{p,r}A_{r,q} \subseteq A_{p,q}$. Indeed, $A_{p,r}A_{r,q}$ represents all of the strings that cause $M$ to transition from $p$ to $q$, touching $r$ somewhere along the way.

Now consider the language
\[
\bigcup_{(p,a,r) \in Q \times \Sigma \times F} A_{q_0,p}A_{\delta(p,a),r}. \tag{6.20}
\]
This is a regular language because each $A_{p,q}$ is regular and the regular languages are closed under finite unions and concatenations. To complete the solution, let us observe that the language above is none other than $B$:
\[
B = \bigcup_{(p,a,r) \in Q \times \Sigma \times F} A_{q_0,p}A_{\delta(p,a),r}. \tag{6.21}
\]

To prove this equality, we do the natural thing, which is to separate it into two separate set inclusions. First let us prove that
\[
B \subseteq \bigcup_{(p,a,r) \in Q \times \Sigma \times F} A_{q_0,p}A_{\delta(p,a),r}. \tag{6.22}
\]
Every string in $B$ takes the form $uv$, for some choice of $u, v \in \Sigma^*$ and $a \in \Sigma$ for which $uav \in A$. Let $p \in Q$ be the unique state for which $u \in A_{q_0,p}$, which we could alternatively describe as the state of $M$ reached from the start state on input $u$, and let $r \in F$ be the unique state (which is necessarily an accepting state) for which $uav \in A_{q_0,r}$. As $a$ causes $M$ to transition from $p$ to $\delta(p,a)$, it follows that $v$ must cause $M$ to transition from $\delta(p,a)$ to $r$, i.e., $v \in A_{\delta(p,a),r}$. It therefore holds that $uv \in A_{q_0,p}A_{\delta(p,a),r}$, which implies the required inclusion.

Next we will prove that
\[
\bigcup_{(p,a,r) \in Q \times \Sigma \times F} A_{q_0,p}A_{\delta(p,a),r} \subseteq B. \tag{6.23}
\]
The argument is quite similar to the other inclusion just considered. Pick any choice of $p \in Q$, $a \in \Sigma$, and $r \in F$. An element of $A_{q_0,p}A_{\delta(p,a),r}$ must take the form $uv$ for $u \in A_{q_0,p}$ and $v \in A_{\delta(p,a),r}$. One finds that $uav \in A_{p_0,r} \subseteq A$, and therefore $uv \in B$, as required.

**Problem 6.2.** Let $\Sigma = \{0,1\}$ and let $A \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the language
\[
C = \{vu : u, v \in \Sigma^* \text{ and } uv \in A\} \tag{6.24}
\]
is regular.
Solution. Again, a natural way to solve this problem is to give an NFA for $C$. Let us assume

$$M = (Q, \Sigma, \delta, q_0, F)$$

(6.25)

is a DFA for which $L(M) = A$, like we did above. This time our NFA will be slightly more complicated. In particular, let us define

$$N = (P, \Sigma, \eta, p_0, G)$$

(6.26)

as follows. First, we will define

$$P = (\{0, 1\} \times Q \times Q) \cup \{p_0\},$$

(6.27)

for $p_0$ being a special start state of $N$ that is not contained in $\{0, 1\} \times Q \times Q$. The accept states of $N$ will be

$$G = \{(1, q, q) : q \in Q\}.$$  

(6.28)

It remains to describe the transition function $\eta$ of $N$, which will be as follows:

1. $\eta(p_0, \epsilon) = \{(0, q, q) : q \in Q\}$.
2. $\eta((0, r, q), a) = \{(0, \delta(r, a), q)\}$ for all $q, r \in Q$ and $a \in \Sigma$.
3. $\eta((0, r, q), \epsilon) = \{(1, q_0, q)\}$ for every $r \in F$ and $q \in Q$.
4. $\eta((1, r, q), a) = \{(1, \delta(r, a), q)\}$ for all $q, r \in Q$ and $a \in \Sigma$.

All other values of $\eta$ that have not been listed are to be understood as $\emptyset$.

Let us consider this definition in greater detail to understand how it works. $N$ starts out in the start state $p_0$, and the only thing it can do is to make a guess for some state of the form $(0, q, q)$ to jump to. The idea is that the 0 indicates that $N$ is entering the first phase of its computation, in which it will read a portion of its input string corresponding to $v$ in the definition of $C$. It jumps to any state $q$ of $M$, but it also remembers which state it jumped to. Every state $N$ ever moves to from this point on will have the form $(a, r, q)$ for some $a \in \{0, 1\}$ and $r \in Q$, but for the same $q$ that it first jumped to; the third coordinate $q$ represents the memory of where it first jumped, and it will never forget or change this part of its state. Intuitively speaking, the state $q$ is a guess made by $N$ for the state that $M$ would be on after reading $u$ (which $N$ hasn’t seen yet, so it’s just a guess).

Then, $N$ starts reading symbols and essentially mimicking $M$ on those input symbols—this is the point of the transitions listed in item 2. At some point, non-deterministically chosen, $N$ decides that it’s time to move to the second phase of its computation, reading the second part of its input, which corresponds to the string $u$ in the definition of $C$. It can only make this nondeterministic move, from a state
of the form \((0, r, q)\) to \((1, q_0, q)\), when \(r\) is an accepting state of \(M\). The reason is that \(N\) only wants to accept \(vu\) when \(M\) accepts \(uv\), so \(M\) should be in the initial state at the start of \(u\) and in an accepting state at the end of \(v\). This is the point of the transitions listed in item 3. Finally, in the second phase of its computation, \(N\) simulates \(M\) on the second part of its input, which corresponds to the string \(u\). It accepts only for states of the form \((1, q, q)\), because those are the states that indicate that \(N\) made the correct guess on its first step for the state that \(M\) would be in after reading \(u\).

This is just an intuitive description, not a formal proof. It is the case, however, that \(L(N) = C\), as a low-level, formal proof would reveal, which implies that \(C\) is regular.

**Alternative Solution.** Again, there is another solution along the same lines as the alternative solution to the previous problem. This time it’s actually a much easier solution. Let \(M\) be a DFA for \(A\), precisely as above, and define \(A_{p,r}q_0,p\) for each \(p, q \in Q\) as in the alternative solution to the previous problem. The language

\[
\bigcup_{(p,r) \in Q \times F} A_{p,r}A_{q_0,p}
\]  

(6.29)

is regular, again by the closure of the regular languages under finite unions and concatenations. It therefore suffices to prove

\[
C = \bigcup_{(p,r) \in Q \times F} A_{p,r}A_{q_0,p}.
\]  

(6.30)

By definition, every element of \(C\) may be expressed as \(vu\) for \(u, v \in \Sigma^*\) satisfying \(uv \in A\). Let \(p \in Q\) and \(r \in F\) be the unique states for which \(u \in A_{q_0,p}\) and \(uv \in A_{q_0,r}\). It follows that \(v \in A_{p,r}\), and therefore \(vu \in A_{p,r}A_{q_0,p}\), implying

\[
C \subseteq \bigcup_{(p,r) \in Q \times F} A_{p,r}A_{q_0,p}.
\]  

(6.31)

Along similar lines, for any choice of \(p \in Q\), \(r \in F\), \(u \in A_{q_0,p}\), and \(v \in A_{p,r}\) it holds that \(uv \in A_{q_0,p}A_{p,r} \subseteq A\), and therefore \(vu \in C\), from which the inclusion

\[
\bigcup_{(p,r) \in Q \times F} A_{p,r}A_{q_0,p} \subseteq C
\]  

(6.32)

follows.

The final problem demonstrates that closure properties holding for all regular languages may fail for nonregular languages. In particular, the nonregular languages are not closed under the regular operations.
Problem 6.3. For each of the following statements, give specific examples of languages over some alphabet $\Sigma$ for which the statements are satisfied.

(a) There exist nonregular languages $A, B \subseteq \Sigma^*$ such that $A \cup B$ is regular.
(b) There exist nonregular languages $A, B \subseteq \Sigma^*$ such that $AB$ is regular.
(c) There exists a nonregular language $A \subseteq \Sigma^*$ such that $A^*$ is regular.

Solution. For statement (a), let us let $\Sigma = \{0\}$, let $A \subseteq \Sigma^*$ be any nonregular language whatsoever, such as $A = \{0^n : n \text{ is a perfect square}\}$, and let $B = \overline{A}$. We know that $B$ is also nonregular (because if it were regular, then its complement would also be regular, but its complement is $A$ which we know is nonregular). On the other hand, $A \cup B = \Sigma^*$, which is regular.

For statement (b), let us let $\Sigma = \{0\}$, and let us start by taking $C \subseteq \Sigma^*$ to be any nonregular language (such as $C = \{0^n : n \text{ is a perfect square}\}$). Then let us take

$$A = C \cup \{\varepsilon\} \quad \text{and} \quad B = \overline{C} \cup \{\varepsilon\}. \quad (6.33)$$

The languages $A$ and $B$ are nonregular, by virtue of the fact that $C$ is nonregular (and therefore $\overline{C}$ is nonregular as well). On the other hand, $AB = \Sigma^*$, which is regular.

Finally, for statement (c), let us again take $\Sigma = \{0\}$, and let $A \subseteq \Sigma^*$ be any nonregular language that contains the single-symbol string 0. (Again, the language $A = \{0^n : n \text{ is a perfect square}\}$ will work.) We have that $A$ is nonregular, but $A^* = \Sigma^*$, which is regular.
Lecture 7

Context-free grammars and languages

The next class of languages we will study in this course is the class of context-free languages. They are defined by the notion of a context-free grammar, or a CFG for short, which you will have encountered previously in your studies (such as in CS 241).

7.1 Definitions of CFGs and CFLs

We will start with the following definition for context-free grammars.

**Definition 7.1.** A context-free grammar (or CFG for short) is a 4-tuple

\[ G = (V, \Sigma, R, S), \]  

where \( V \) is a finite and non-empty set (whose elements we will call *variables*), \( \Sigma \) is an alphabet (disjoint from \( V \)), \( R \) is a finite and nonempty set of *rules*, each of which takes the form

\[ A \rightarrow w \]  

for some choice of \( A \in V \) and \( w \in (V \cup \Sigma)^* \), and \( S \in V \) is a variable called the *start variable*.

**Example 7.2.** For our first example of a CFG, we may consider \( G = (V, \Sigma, R, S) \), where \( V = \{S\} \) (so that there is just one variable in this grammar), \( \Sigma = \{0, 1\} \), \( S \) is the start variable, and \( R \) contains these two rules:

\[ S \rightarrow 0S1 \]

\[ S \rightarrow \varepsilon. \]  

It is often convenient to describe a CFG just by listing the rules, like in (7.3). When we do this, it is to be understood that the set of variables \( V \) and the alphabet
Σ are determined implicitly: the variables are the capital letters appearing on the left-hand side of the rules and the alphabet contains the symbols on the right-hand side of the rules that are left over. Moreover, the start variable is understood to be the variable appearing on the left-hand side of the first rule that is listed.

Note that these are just conventions that allow us to save time, and you could simply list each of the elements \(V, \Sigma, R,\) and \(S\) if it was likely that the conventions would cause confusion (like in a situation in which, for some reason, you want to have a variable or alphabet symbol that does not appear in any rule).

Every context-free grammar \(G = (V, \Sigma, R, S)\) generates a language \(L(G) \subseteq \Sigma^*\). Informally speaking, this is the language consisting of all strings that can be obtained by the following process:

1. Write down the start variable \(S\).
2. Repeat the following steps any number of times:
   1. Choose any rule \(A \rightarrow w\) from \(R\).
   2. Within the string of variables and alphabet symbols you currently have written down, replace any instance of the variable \(A\) with the string \(w\).
3. If you are eventually left with a string of the form \(x \in \Sigma^*\), so that no variables remain, then stop. The string \(x\) has been obtained by the process, and is therefore among the strings generated by \(G\).

**Example 7.3.** The CFG \(G\) described in Example 7.2 generates the language

\[
L(G) = \{0^n1^n : n \in \mathbb{N}\}.
\]  

This is because we begin by writing down the start variable \(S\), then we choose one of the two rules and perform the replacement in the only way possible: there will always be a single variable \(S\) in the middle of the string, and we replace it either by \(0S1\) or by \(\varepsilon\). The process ends precisely when we choose the rule \(S \rightarrow \varepsilon\), and depending on how many times we chose the rule \(S \rightarrow 0S1\) we obtain one of the strings

\[
\varepsilon, 01, 0011, 000111, \ldots
\]

and so on. The set of all strings that can possibly be obtained is therefore given by (7.4).

The description of the process through which the language generated by a CFG is determined provides an intuitive, human-readable way to explain this concept, but it is not very satisfying from a mathematical viewpoint. We would prefer a definition based on sets, functions, and so on (rather than one that refers to “writing down” variables, for instance). One way to define this notion mathematically
begins with the specification of the yields relation of a grammar that captures the notion of performing a substitution.

**Definition 7.4.** Let $G = (V, \Sigma, R, S)$ be a context-free grammar. The yields relation defined by $G$ is a relation defined for pairs of strings over the alphabet $V \cup \Sigma$ as follows:

$$uAv \Rightarrow_G uwv$$

for every choice of strings $u, v, w \in (V \cup \Sigma)^*$ and a variable $A \in V$, provided that the rule $A \rightarrow w$ is included in $R$.\(^1\)

The interpretation of this relation is that $x \Rightarrow_G y$, for $x, y \in (V \cup \Sigma)^*$, when it is possible to replace one of the variables appearing in $x$ according to one of the rules of $G$ in order to obtain $y$.

It will also be convenient to consider the reflexive transitive closure of this relation, which is defined as follows.

**Definition 7.5.** Let $G = (V, \Sigma, R, S)$ be a context-free grammar. For any two strings $x, y \in (V \cup \Sigma)^*$ one has that

$$x \Rightarrow^*_G y$$

if there exists a positive integer $m$ and strings $z_1, \ldots, z_m \in (V \cup \Sigma)^*$ such that

1. $x = z_1$,
2. $y = z_m$, and
3. $z_k \Rightarrow_G z_{k+1}$ for every $k \in \{1, \ldots, m - 1\}$.

In this case, the interpretation of this relation is that $x \Rightarrow^*_G y$ holds when it is possible to transform $x$ into $y$ by performing zero or more substitutions according to the rules of $G$.

When a CFG $G$ is fixed or can be safely taken as implicit, we will sometimes write $\Rightarrow$ rather than $\Rightarrow_G$, and likewise for the starred version.

We can now use the relation just defined to formally define the language generated by a given context-free grammar.

**Definition 7.6.** Let $G = (V, \Sigma, R, S)$ be a context-free grammar. The language generated by $G$ is

$$L(G) = \{ x \in \Sigma^* : S \Rightarrow^*_G x \}. \quad (7.8)$$

---

1 Recall that a relation is a subset of a Cartesian product of two sets. In this case, the relation is the subset $\{(uAv, uwv) : u, v, w \in (V \cup \Sigma)^*, A \in V, \text{ and } A \rightarrow w \text{ is a rule in } R\}$. The notation $uAv \Rightarrow_G uwv$ is a more readable way of indicating that the pair $(uAv, uwv)$ is an element of the relation.
If \( x \in L(G) \) for a CFG \( G = (V, \Sigma, R, S) \), and \( z_1, \ldots, z_m \in (V \cup \Sigma)^* \) is a sequence of strings for which \( z_1 = S \), \( z_m = x \), and \( z_k \Rightarrow_G z_{k+1} \) for all \( k \in \{1, \ldots, m - 1\} \), then the sequence \( z_1, \ldots, z_m \) is said to be a derivation of \( x \). If you unravel the definitions above, it becomes clear that there must of course exist at least one derivation for every string \( x \in L(G) \), but in general there might be more than one derivation of a given string \( x \in L(G) \).

Finally, we define the class of context-free languages to be those languages that are generated by context-free grammars.

**Definition 7.7.** Let \( \Sigma \) be an alphabet and let \( A \subseteq \Sigma^* \) be a language. The language \( A \) is context-free if there exists a context-free grammar \( G \) such that \( L(G) = A \).

**Example 7.8.** The language \( \{0^n1^n : n \in \mathbb{N}\} \) is a context-free language, as has been established in Example 7.3.

### 7.2 Basic examples

We’ve seen one example of a context-free language so far: \( \{0^n1^n : n \in \mathbb{N}\} \). Let us now consider a few more examples.

**Example 7.9.** The language

\[
\text{PAL} = \{ w \in \Sigma^* : w = w^R \}
\]

over the alphabet \( \Sigma = \{0, 1\} \) is context-free. (In fact this is true for any choice of an alphabet \( \Sigma \), but let us stick with the binary alphabet for now for simplicity). To verify that this language is context-free, it suffices to exhibit a context-free grammar that generates it. Here is one that works:

\[
\begin{align*}
S &\rightarrow 0 \ S \ 0 \\
S &\rightarrow 1 \ S \ 1 \\
S &\rightarrow 0 \\
S &\rightarrow 1 \\
S &\rightarrow \varepsilon
\end{align*}
\]

We have named the language in the above example PAL because it is short for palindrome, which is something that reads the same forwards and backwards. The example is concerned with binary string palindromes, but you might be familiar with English language examples, such as “never odd or even” or “yo, banana boy” (which of course require you to ignore punctuation and spaces).
Lecture 7

We often use a short-hand notation for describing grammars in which the same variable appears on the left-hand side of multiple rules, as is the case for the grammar described in the previous example. The short-hand notation is to write the variable on the left-hand side and the arrow just once, and to draw a vertical bar (which can be read as “or”) among the possible alternatives for the right-hand side like this:

\[ S \rightarrow 0S0 | 1S1 | 0 | 1 | \varepsilon \]  
(7.11)

When you use this short-hand notation when you’re writing by hand, such as on an exam, be sure to make your bars tall enough so that they are easily distinguished from 1s.

Sometimes it is easy to see that a particular CFG generates a given language—for instance, I would consider this to be obvious in the case of the previous example. In other cases it can be more challenging, or even impossibly difficult, to verify that a particular grammar generates a particular language. The next example illustrates a case in which such a verification is nontrivial.

**Example 7.10.** Let \( \Sigma = \{0, 1\} \) be the binary alphabet, and define a language \( A \subseteq \Sigma^* \) as follows:

\[ A = \{ w \in \Sigma^*: |w|_0 = |w|_1 \} \]  
(7.12)

Here we are using a convenient notation: \( |w|_0 \) denotes the number of times the symbol 0 appears in \( w \), and similarly \( |w|_1 \) denotes the number of times the symbol 1 appears in \( w \). The language \( A \) therefore contains all binary strings having the same number of 0s and 1s. This is a context-free language, as it is generated by this context-free grammar:

\[ S \rightarrow 0S1S | 1S0S | \varepsilon. \]  
(7.13)

Now, it is pretty clear that every string generated by the grammar (which we will call \( G \)) described in the above example is contained in \( A \); we begin a derivation with just the variable \( S \), so there are an equal number of 0s and 1s written down at the start (zero of each, to be precise), and every rule maintains this property as an invariant.

On the other hand, it is not immediately obvious that every element of \( A \) can be generated by \( G \). Let us prove that this is indeed the case.

**Claim 7.11.** \( A \subseteq L(G) \).

**Proof.** Let \( w \in A \) be a string contained in \( A \) and let \( n = |w| \). We will prove that \( w \in L(G) \) by (strong) induction on \( n \).

The base case is \( n = 0 \), which means that \( w = \varepsilon \). We have that \( S \Rightarrow_G \varepsilon \) represents a derivation of \( \varepsilon \), and therefore \( w \in L(G) \).
For the induction step, we assume that \( n \geq 1 \), and the hypothesis of induction is that \( x \in L(G) \) for every string \( x \in A \) with \( |x| < n \). Our goal is to prove that \( G \) generates \( w \). Let us write

\[
w = a_1 \cdots a_n
\]  

(7.14)

for \( a_1, \ldots, a_n \in \Sigma \). We have assumed that \( w \in A \), and therefore

\[
|a_1 \cdots a_n|_0 = |a_1 \cdots a_n|_1.
\]  

(7.15)

Next, let \( m \in \{1, \ldots, n\} \) be the minimum value for which

\[
|a_1 \cdots a_m|_0 = |a_1 \cdots a_m|_1; \tag{7.16}
\]

we know that this equation is satisfied when \( m = n \), and there might be a smaller value of \( m \) that works, but in any case we know that \( m \) is a well-defined number. We will prove that \( a_1 \neq a_m \).

We can reason that \( a_1 \neq a_m \) through a proof by contradiction. Toward this goal, assume \( a_1 = a_m \), and define

\[
N_k = |a_1 \cdots a_k|_1 - |a_1 \cdots a_k|_0 \tag{7.17}
\]

for every \( k \in \{1, \ldots, m\} \). We know that \( N_m = 0 \) because equation (7.16) is satisfied. Moreover, using the assumption \( a_1 = a_m \), we observe the equations

\[
\begin{align*}
|a_1 \cdots a_m|_1 &= |a_1 \cdots a_{m-1}|_1 + |a_m|_1 = |a_1 \cdots a_{m-1}|_1 + |a_1|_1, \\
|a_1 \cdots a_m|_0 &= |a_1 \cdots a_{m-1}|_0 + |a_m|_0 = |a_1 \cdots a_{m-1}|_0 + |a_1|_0,
\end{align*}
\]  

(7.18)

and conclude that

\[
N_m = N_{m-1} + N_1 \tag{7.19}
\]

by subtracting the second equation from the first. Therefore, because \( N_m = 0 \) and \( N_1 \) is nonzero, it must be that \( N_{m-1} \) is also nonzero, and more importantly \( N_1 \) and \( N_{m-1} \) must have opposite sign. However, because consecutive values of \( N_k \) must always differ by 1 and can only take integer values, we conclude that there must exist a choice of \( k \) in the range \( \{2, \ldots, m-2\} \) for which \( N_k = 0 \), for otherwise it would not be possible for \( N_1 \) and \( N_{m-1} \) to have opposite sign. This, however, is in contradiction with \( m \) being the minimum value for which (7.16) holds. We have therefore concluded that \( a_1 \neq a_m \).

At this point it is possible to describe a derivation for \( w \). We have \( w = a_1 \cdots a_n \), and we have that

\[
|a_1 \cdots a_m|_0 = |a_1 \cdots a_m|_1 \quad \text{and} \quad a_1 \neq a_m \tag{7.20}
\]
Lecture 7

for some choice of \( m \in \{1, \ldots, n\} \). We conclude that

\[
|a_2 \cdots a_{m-1}|_0 = |a_2 \cdots a_{m-1}|_1 \quad \text{and} \quad |a_{m+1} \cdots a_n|_0 = |a_{m+1} \cdots a_n|_1.
\]  

(7.21)

By the hypothesis of induction it follows that

\[
S \xrightarrow{*} G a_2 \cdots a_{m-1} \quad \text{and} \quad S \xrightarrow{*} G a_{m+1} \cdots a_n.
\]  

(7.22)

Therefore the string \( w \) satisfies

\[
S \Rightarrow_G 0 S 1 S \xrightarrow{*} G 0 a_2 \cdots a_{m-1} 1 a_{m+1} \cdots a_n = w
\]  

(7.23)

(in case \( a_1 = 0 \) and \( a_m = 1 \)) or

\[
S \Rightarrow_G 1 S 0 S \xrightarrow{*} G 1 a_2 \cdots a_{m-1} 0 a_{m+1} \cdots a_n = w
\]  

(7.24)

(in case \( a_1 = 1 \) and \( a_m = 0 \)). We have proved that \( w \in L(G) \) as required.

Here is another example that is related to the previous one. It is an important example and we’ll refer to it from time to time throughout the course.

**Example 7.12.** Consider the alphabet \( \Sigma = \{ (, ) \} \). That is, we have two symbols in this alphabet: left-parenthesis and right-parenthesis.

To say that a string \( w \) over the alphabet \( \Sigma \) is *properly balanced* means that by repeatedly removing the substring \(( )\), you can eventually reach \( \epsilon \). More intuitively speaking, a string over \( \Sigma \) is properly balanced if it would make sense to use this pattern of parentheses in an ordinary arithmetic expression (ignoring everything besides the parentheses). These are examples of properly balanced strings:

\[
( ( ) ( ) ) ( ), \quad ( ( ( ) ) ) ), \quad ( ( ) ), \quad \text{and} \quad \epsilon.
\]  

(7.25)

These are examples of strings that are not properly balanced:

\[
( ( ( ) ) ) , \quad \text{and} \quad ( ) ).
\]  

(7.26)

Now define a language

\[
\text{BAL} = \{ w \in \Sigma^* : w \text{ is properly balanced} \}.
\]  

(7.27)

The language BAL is context-free; here is a simple CFG that generates it:

\[
S \rightarrow ( S ) S \mid \epsilon.
\]  

(7.28)

See if you can convince yourself that this CFG indeed generates BAL!
7.3 A tougher example

Sometimes it is more challenging to come up with a context-free grammar for a given language. The following example concerns one such language.

Example 7.13. Let \( \Sigma = \{0, 1\} \) and let \( \Gamma = \{0, 1, \#\} \), and define

\[ A = \{u\#v : u, v \in \Sigma^* \text{ and } u \neq v\}. \]  

Here is a context-free grammar for \( A \):

\[
S \rightarrow W_01Y \mid W_10Y \mid Z \\
W_0 \rightarrow XW_0X \mid 0Y\# \\
W_1 \rightarrow XW_1X \mid 1Y\# \\
Z \rightarrow XZX \mid XY\# \mid #XY \\
X \rightarrow 0 \mid 1 \\
Y \rightarrow XY \mid \epsilon. 
\]

(7.30)

The idea behind this CFG is as follows. First, the variable \( Z \) generates strings of the form \( u\#v \) where \( u \) and \( v \) have different lengths. The variable \( W_0 \) generates strings that look like this:

\[
\begin{array}{ccc}
\square \ldots \square & 0 & \square \ldots \square \# \square \ldots \square \\
n \text{bits} & m \text{bits} & n \text{bits}
\end{array}
\]

(7.31)

(where \( \square \) denotes either 0 or 1), so that \( W_01Y \) generates strings that look like this:

\[
\begin{array}{cccc}
\square \ldots \square 0 \square \ldots \square \# \square \ldots \square 1 \square \ldots \square \\
n \text{bits} & m \text{bits} & n \text{bits} & k \text{bits}
\end{array}
\]

(7.32)

(for any choice of \( n, m, k \in \mathbb{N} \)). Similarly, \( W_10Y \) generates strings that look like this:

\[
\begin{array}{cccc}
\square \ldots \square 1 \square \ldots \square \# \square \ldots \square 0 \square \ldots \square \\
n \text{bits} & m \text{bits} & n \text{bits} & k \text{bits}
\end{array}
\]

(7.33)

Taken together, these two possibilities generate \( u\#v \) for all binary strings \( u \) and \( v \) that differ in at least one position (and that may or may not have the same length). The three options together cover all possible \( u\#v \) for which \( u \) and \( v \) are non-equal binary strings.
Parse trees, ambiguity, and Chomsky normal form

In this lecture we will discuss a few important notions connected with context-free grammars, including *parse trees*, *ambiguity*, and a special form for context-free grammars known as the *Chomsky normal form*.

8.1 Left-most derivations and parse trees

In the previous lecture we covered the definition of context-free grammars as well as *derivations* of strings by context-free grammars. Let us consider one of the context-free grammars from the previous lecture:

\[
S \rightarrow 0S1S \mid 1S0S \mid \varepsilon.
\]  

Again we’ll call this CFG \( G \), and as we proved last time we have

\[
L(G) = \{ w \in \Sigma^* : |w|_0 = |w|_1 \},
\]

where \( \Sigma = \{0, 1\} \) is the binary alphabet and \( |w|_0 \) and \( |w|_1 \) denote the number of times the symbols 0 and 1 appear in \( w \), respectively.

**Left-most derivations**

Here is an example of a derivation of the string 0101:

\[
S \Rightarrow 0S1S \Rightarrow 01S0S1S \Rightarrow 010S1S \Rightarrow 0101S \Rightarrow 0101.
\]

This is an example of a *left-most derivation*, which means that it is always the left-most variable that gets replaced at each step. For the first step there is only one
variable that can possibly be replaced; this is true both in this example and in general. For the second step, however, one could choose to replace either of the occurrences of the variable $S$, and in the derivation above it is the left-most occurrence that gets replaced. That is, if we underline the variable that gets replaced and the symbols and variables that replace it, we see that this step replaces the left-most occurrence of the variable $S$:

$$0S1S \Rightarrow 01S0S1S.$$ (8.4)

The same is true for every other step: always we choose the left-most variable occurrence to replace, and that is why we call this a left-most derivation. The same terminology is used in general, for any context-free grammar.

If you think about it for a moment, you will quickly realize that every string that can be generated by a particular context-free grammar can also be generated by that same grammar using a left-most derivation. This is because there is no “interaction” among multiple variables and/or symbols in any context-free grammar derivation; if we know which rule is used to substitute each variable, then it doesn’t matter what order the variable occurrences are substituted, so you might as well always take care of the left-most variable during each step.

We could also define the notion of a right-most derivation, in which the right-most variable occurrence is always evaluated first, but there isn’t really anything important about right-most derivations that isn’t already represented by the notion of a left-most derivation, at least from the viewpoint of this course. For this reason, we won’t have any reason to discuss right-most derivations further.

**Parse trees**

With any derivation of a string by a context-free grammar we may associate a tree, called a *parse tree*, according to the following rules:

1. We have one node of the tree for each new occurrence of either a variable, a symbol, or an $\varepsilon$ in the derivation, with the root node of the tree corresponding to the start variable. We only have nodes labelled $\varepsilon$ when rules of the form $V \rightarrow \varepsilon$ are applied.

2. Each node corresponding to a symbol or an $\varepsilon$ is a leaf node (having no children), while each node corresponding to a variable has one child for each symbol, variable, or $\varepsilon$ with which it is replaced. The children of each variable node are ordered in the same way as the symbols and variables in the rule used to replace that variable.

For example, the derivation (8.3) yields the parse tree illustrated in Figure 8.1.
8.2 Ambiguity

Sometimes a context-free grammar will allow multiple parse trees (or, equivalently, multiple left-most derivations) for some strings in the language that it generates. For example, a left-most derivation of the string 0101 by the CFG (8.1) that is different from the one given by the derivation (8.3) is

\[
S \Rightarrow 0S1S \Rightarrow 01S \Rightarrow 010S1S \Rightarrow 0101S \Rightarrow 0101.
\] (8.5)

The parse tree corresponding to this derivation is illustrated in Figure 8.2.

When it is the case, for a given context-free grammar $G$, that there exists at least one string $w \in L(G)$ having at least two different parse trees, the CFG $G$ is said to
be ambiguous. Note that this is so even if there is just a single string having multiple parse trees; in order to be unambiguous, a CFG must have just a single, unique parse tree for every string it generates.

Being unambiguous is generally considered to be a positive attribute of a CFG, and indeed it is a requirement for some applications of context-free grammars.

**Designing unambiguous CFGs**

In some cases it is possible to come up with an unambiguous context-free grammar that generates the same language as an ambiguous context-free grammar. For example, we can come up with a different context-free grammar for the language

\[ \{ w \in \{0, 1\}^* : |w|_0 = |w|_1 \} \quad (8.6) \]

that, unlike the CFG (8.1), is unambiguous. Here is such a CFG:

\[
\begin{align*}
S & \rightarrow 0X1S \mid 1Y0S \mid \epsilon \\
X & \rightarrow 0X1X \mid \epsilon \\
Y & \rightarrow 1Y0Y \mid \epsilon 
\end{align*}
\]

(8.7)

We won’t take the time to go through a proof that this CFG is unambiguous, but if you think about it for a few moments you should be able to convince yourself that it is unambiguous. The variable \( X \) generates strings having the same number of 0s and 1s, where the number of 1s never exceeds the number of 0s when you read from left to right, and the variable \( Y \) is similar except the role of the 0s and 1s is reversed. If you try to generate a particular string by a left-most derivation with this CFG, you’ll never have more than one option as to which rule to apply.

Here is another example of how an ambiguous CFG can be modified to make it unambiguous. Let us define an alphabet

\[ \Sigma = \{ a, b, +, \ast, (, ) \} \quad (8.8) \]

along with a CFG

\[
S \rightarrow S + S \mid S * S \mid (S) \mid a \mid b 
\]

(8.9)

This grammar generates strings that look like arithmetic expressions in variables \( a \) and \( b \), where we allow the operations \( \ast \) and \( + \), along with parentheses. For instance, the string

\[ (a + b) \ast a + b \quad (8.10) \]

is such an expression, and we can generate it (for instance) as follows:

\[
\begin{align*}
S \Rightarrow S \ast S \Rightarrow (S) \ast S \Rightarrow (S + S) \ast S \Rightarrow (a + S) \ast S \Rightarrow (a + b) \ast S \\
\Rightarrow (a + b) \ast S + S \Rightarrow (a + b) \ast a + S \Rightarrow (a + b) \ast a + b. 
\end{align*}
\]

(8.11)
This happens to be a left-most derivation, as it is always the left-most variable that is substituted. The parse tree corresponding to this derivation is shown in Figure 8.3. You can of course imagine a more complex version of this grammar allowing for other arithmetic operations, variables, and so on, but we will stick to the grammar in (8.9) for the sake of simplicity.

Now, the CFG (8.9) is certainly ambiguous. For instance, a different (left-most) derivation for the same string \((a + b) \ast a + b\) as before is

\[
S \Rightarrow S + S \Rightarrow S \ast S + S \Rightarrow (S) \ast S + S \Rightarrow (S + S) \ast S + S \\
(\Rightarrow (a + S) \ast S + S \Rightarrow (a + b) \ast S + S \Rightarrow (a + b) \ast a + S) \Rightarrow (a + b) \ast a + b,
\]

and the parse tree for this derivation is shown in Figure 8.4. Notice that there is something appealing about the parse tree illustrated in Figure 8.4, which is that it actually carries the meaning of the expression \((a + b) \ast a + b\), in the sense that the tree structure properly captures the order in which the operations should be applied. In contrast, the first parse tree seems to represent what the expression \((a + b) \ast a + b\) would evaluate to if we lived in a society where addition was given higher precedence than multiplication.

The ambiguity of the grammar (8.9), along with the fact that parse trees may not represent the meaning of an arithmetic expression in the sense just described, is a problem in some settings. For example, if we were designing a compiler and wanted a part of it to represent arithmetic expressions (presumably allowing much more complicated ones than our grammar from above allows), a CFG along the lines of (8.9) would be completely inadequate.
We can, however, come up with a new CFG for the same language that is much better, in the sense that it is unambiguous and properly captures the meaning of arithmetic expressions (given that we give multiplication higher precedence than addition). Here it is:

\[
\begin{align*}
S & \rightarrow T \mid S + T \\
T & \rightarrow F \mid T * F \\
F & \rightarrow I \mid (S) \\
I & \rightarrow a \mid b
\end{align*}
\] (8.13)

For example, the unique parse tree corresponding to the string \((a + b) * a + b\) is as shown in Figure 8.5.

In order to better understand the CFG (8.13), it may help to associate meanings with the different variables. In this CFG, the variable \(T\) generates terms, the variable \(F\) generates factors, and the variable \(I\) generates identifiers. An expression is either a term or a sum of terms, a term is either a factor or a product of factors, and a factor is either an identifier or an entire expression inside of parentheses.

**Inherently ambiguous languages**

While we have seen that it is sometime possible to come up with an unambiguous CFG that generates the same language as an ambiguous CFG, it is not always possible. There are some context-free languages that can only be generated by ambiguous CFGs. Such languages are called *inherently ambiguous* context-free languages.
Figure 8.5: Unique parse tree for \((a + b) * a + b\) for the CFG \((8.13)\).

An example of an inherently ambiguous context-free language is this one:

\[
\left\{0^n1^m2^k : n = m \text{ or } m = k \right\}.
\quad (8.14)
\]

We will not prove that this language is inherently ambiguous, but the intuition is that no matter what CFG you come up with for this language, the string \(0^n1^m2^n\) will always have multiple parse trees for some sufficiently large natural number \(n\).

### 8.3 Chomsky normal form

Some context-free grammars are strange. For example, the CFG

\[
S \rightarrow S S S S \mid \varepsilon
\]  
\quad (8.15)
simply generates the language \{\epsilon\}; but it is obviously ambiguous, and even worse it has infinitely many parse trees (which of course can be arbitrarily large) for the only string \epsilon it generates. While we know we cannot always eliminate ambiguity from CFGs, as some context-free languages are inherently ambiguous, we can at least eliminate the possibility to have infinitely many parse trees for a given string. Perhaps more importantly, for any given CFG \( G \), we can always come up with a new CFG \( H \) for which \( L(H) = L(G) \), and for which we are guaranteed that every parse tree for a given string \( w \in L(H) \) has the same size and a very simple, binary-tree-like structure.

To be more precise about the specific sort of CFGs and parse trees we’re talking about, it is appropriate at this point to define what is called the Chomsky normal form for context-free grammars.

**Definition 8.1.** A context-free grammar \( G \) is in Chomsky normal form if every rule of \( G \) has one of the following three forms:

1. \( X \to YZ \), for variables \( X \), \( Y \), and \( Z \), and where neither \( Y \) nor \( Z \) is the start variable,
2. \( X \to a \), for a variable \( X \) and a symbol \( a \), or
3. \( S \to \epsilon \), for \( S \) the start variable.

Now, the reason why a CFG in Chomsky normal form is nice is that every parse tree for such a grammar has a simple form: the variable nodes form a binary tree, and for each variable node that doesn’t have any variable node children, a single symbol node hangs off. A hypothetical example meant to illustrate the structure we are talking about is given in Figure 8.6. Notice that the start variable always appears exactly once at the root of the tree because it is never allowed on the right-hand side of any rule.

If the rule \( S \to \epsilon \) is present in a CFG in Chomsky normal form, then we have a special case that doesn’t fit exactly into the structure described above. In this case we can have the very simple parse tree shown in Figure 8.7 for \( \epsilon \), and this is the only possible parse tree for this string.

Because of the special form that a parse tree must take for a CFG \( G \) in Chomsky normal form, we have that *every* parse tree for a given string \( w \in L(G) \) must have exactly \( 2|w| - 1 \) variable nodes and \( |w| \) leaf nodes (except for the special case \( w = \epsilon \), in which we have one variable node and 1 leaf node). An equivalent statement is that every derivation of a (nonempty) string \( w \) by a CFG in Chomsky normal form requires exactly \( 2|w| - 1 \) substitutions.

The following theorem establishes that every context-free language is generated by a CFG in Chomsky normal form.
Figure 8.6: A hypothetical example of a parse tree for a CFG in Chomsky normal form.

Figure 8.7: The unique parse tree for $\varepsilon$ for a CFG in Chomsky normal form, assuming it includes the rule $S \rightarrow \varepsilon$.

**Theorem 8.2.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a context-free language. There exists a CFG $G$ in Chomsky normal form such that $A = L(G)$.

The usual way to prove this theorem is through a construction that converts an arbitrary CFG $G$ into a CFG $H$ in Chomsky normal form for which $L(H) = L(G)$. The conversion is, in fact, fairly straightforward—a summary of the steps one may perform to do this conversion for an arbitrary CFG $G = (V, \Sigma, R, S)$ appear below. To illustrate how the rules work, let us start with the following CFG, which generates the balanced parentheses language BAL from the previous lecture:

$$S \rightarrow (S)S \mid \varepsilon$$  \hspace{1cm} (8.16)

1. Add a new start variable $S_0$ along with the rule $S_0 \rightarrow S$.

   Doing this will ensure that the start variable $S_0$ never appears on the right-hand side of any rule.
Applying this step to the CFG (8.16) yields
\[
S_0 \rightarrow S \\
S \rightarrow (S)S \mid \varepsilon
\] (8.17)

2. Introduce a new variable \( X_a \) for each symbol \( a \in \Sigma \).

First include the new rule \( X_a \rightarrow a \). Then, for every other rule in which \( a \) appears on the right-hand side, except for the cases when \( a \) appears all by itself on the right-hand side, replace each \( a \) with \( X_a \).

Continuing with our example, the CFG (8.17) is transformed into this CFG (where we’ll use the names \( L \) and \( R \) rather than the weird-looking variables \( X_\_ \) and \( X_\_) \) in the interest of style):
\[
S_0 \rightarrow S \\
S \rightarrow LSRS \mid \varepsilon \\
L \rightarrow ( \\
R \rightarrow )
\] (8.18)

3. Split up rules of the form \( X \rightarrow Y_1 \cdots Y_m \), whenever \( m \geq 3 \), using auxiliary variables in a straightforward way.

In particular, \( X \rightarrow Y_1 \cdots Y_m \) can be broken up as
\[
X \rightarrow Y_1 Z_2 \\
Z_2 \rightarrow Y_2 Z_3 \\
\vdots \\
Z_{m-2} \rightarrow Y_{m-2} Z_{m-1} \\
Z_{m-1} \rightarrow Y_{m-1} Y_m
\]
(8.19)

We need to be sure to use separate auxiliary variables for each rule so that there is no “cross talk” between separate rules, so do not reuse the same auxiliary variables to break up multiple rules.

Transforming the CFG (8.18) in this way results in the following CFG:
\[
S_0 \rightarrow S \\
S \rightarrow LZ_2 \mid \varepsilon \\
Z_2 \rightarrow SZ_3 \\
Z_3 \rightarrow RS \\
L \rightarrow ( \\
R \rightarrow )
\] (8.20)
4. Eliminate $\varepsilon$-rules of the form $X \rightarrow \varepsilon$ and “repair the damage.”

Aside from the special case $S_0 \rightarrow \varepsilon$, there is never any need for rules of the form $X \rightarrow \varepsilon$; you can get the same effect by simply duplicating rules in which $X$ appears on the right-hand side, and directly replacing or not replacing $X$ with $\varepsilon$ in each possible combination. You might introduce new $\varepsilon$-rules in this way, but they can be handled recursively—and any time a new $\varepsilon$-rule is generated that was already eliminated, it is not added back in.

Transforming the CFG (8.20) in this way results in the following CFG:

$$
\begin{align*}
S_0 & \rightarrow S \mid \varepsilon \\
S & \rightarrow LZ_2 \\
Z_2 & \rightarrow SZ_3 \mid Z_3 \\
Z_3 & \rightarrow RS \mid R \\
L & \rightarrow ( \\
R & \rightarrow )
\end{align*}
$$

(8.21)

Note that we do end up with the $\varepsilon$-rule $S_0 \rightarrow \varepsilon$, but we do not eliminate this one because $S_0 \rightarrow \varepsilon$ is the special case that we allow as an $\varepsilon$-rule.

5. Eliminate unit rules, which are rules of the form $X \rightarrow Y$.

Rules like this are never necessary, and they can be eliminated provided that we also include the rule $X \rightarrow w$ in the CFG whenever $Y \rightarrow w$ appears as a rule. If you obtain a new unit rule that was already eliminated (or is the unit rule currently being eliminated), it is not added back in.

Transforming the CFG (8.21) in this way results in the following CFG:

$$
\begin{align*}
S_0 & \rightarrow LZ_2 \mid \varepsilon \\
S & \rightarrow LZ_2 \\
Z_2 & \rightarrow SZ_3 \mid RS \mid ) \\
Z_3 & \rightarrow RS \mid ) \\
L & \rightarrow ( \\
R & \rightarrow )
\end{align*}
$$

(8.22)

At this point we are finished; this context-free grammar is in Chomsky normal form.

The description above is only meant to give you the basic idea of how the construction works and does not constitute a formal proof of Theorem 8.2. It is possible,
however, to be more formal and precise in describing this construction in order to obtain a proper proof of Theorem 8.2.

We will make use of the theorem from time to time; when we are proving things about context-free languages it is sometimes extremely helpful to know that we can always assume that a given context-free language is generated by a CFG in Chomsky normal form.

Finally, it must be stressed that the Chomsky normal form says nothing about ambiguity in general. A CFG in Chomsky normal form may or may not be ambiguous, just like we have for arbitrary CFGs.
Lecture 9

Closure properties for context-free languages

In this lecture we will examine various properties of the context-free languages, including the fact that they are closed under the regular operations, that every regular language is context-free, and that the intersection of a context-free language and a regular language is always context-free.

9.1 Closure under the regular operations

We will begin by proving that the context-free languages are closed under each of the regular operations.

**Theorem 9.1.** Let \( \Sigma \) be an alphabet and let \( A, B \subseteq \Sigma^* \) be context-free languages. The languages \( A \cup B, AB, \) and \( A^* \) are context-free.

**Proof.** Because \( A \) and \( B \) are context-free languages, there must exist CFGs

\[
G_A = (V_A, \Sigma, R_A, S_A) \quad \text{and} \quad G_B = (V_B, \Sigma, R_B, S_B)
\]

such that \( L(G_A) = A \) and \( L(G_B) = B \). Because the specific names we choose for the variables in a context-free grammar have no effect on the language it generates, there is no loss of generality in assuming \( V_A \) and \( V_B \) are disjoint sets.

First let us construct a CFG \( G \) for the language \( A \cup B \). This CFG will include all of the variables and rules of \( G_A \) and \( G_B \) together, along with a new variable \( S \) (which we assume is not already contained in \( V_A \) or \( V_B \), and which we will take to be the start variable of \( G \)) and two new rules:

\[
S \rightarrow S_A \mid S_B.
\]
Formally speaking we may write

$$G = (V, \Sigma, R, S)$$  \hfill (9.3)

where $V = V_A \cup V_B \cup \{S\}$ and $R = R_A \cup R_B \cup \{S \to S_A, S \to S_B\}$. In the typical style in which we write CFGs, the grammar $G$ looks like this:

$$S \to S_A \mid S_B$$

\begin{itemize}
  \item all rules of $G_A$
  \item all rules of $G_B$
\end{itemize}

It is evident that $L(G) = A \cup B$; each derivation may begin with $S \Rightarrow S_A$ or $S \Rightarrow S_B$, after which either $S_A$ generates any string in $A$ or $S_B$ generates any string in $B$. As the language $A \cup B$ is generated by the CFG $G$, we have that it is context-free.

Next we will construct a CFG $H$ for the language $AB$. The construction of $H$ is very similar to the construction of $G$ above. The CFG $H$ will include all of the variables and rules of $G_A$ and $G_B$, along with a new start variable $S$ and one new rule:

$$S \to S_AS_B.$$  \hfill (9.4)

Formally speaking we may write

$$H = (V, \Sigma, R, S)$$  \hfill (9.5)

where $V = V_A \cup V_B \cup \{S\}$ and $R = R_A \cup R_B \cup \{S \to S_AS_B\}$. In the typical style in which we write CFGs, the grammar $G$ looks like this:

$$S \to S_AS_B$$

\begin{itemize}
  \item all rules of $G_A$
  \item all rules of $G_B$
\end{itemize}
Lecture 9

It is evident that \( L(G) = AB \); each derivation must begin with \( S \Rightarrow S_AS_B \), and then \( S_A \) generates any string in \( A \) and \( S_B \) generates any string in \( B \). As the language \( AB \) is generated by the CFG \( H \), we have that it is context-free.

Finally we will construct a CFG \( K \) for \( A^* \). This time the CFG \( K \) will include just the rules and variables of \( G_A \), along with a new start variable \( S \) and two new rules:

\[
S \rightarrow S \mid S_A | \epsilon. \tag{9.6}
\]

Formally speaking we may write

\[
K = (V, \Sigma, R, S) \tag{9.7}
\]

where \( V = V_A \cup \{S\} \) and \( R = R_A \cup \{S \rightarrow S_SS_A, S \rightarrow \epsilon\} \). In the typical style in which we write CFGs, the grammar \( K \) looks like this:

\[
S \rightarrow S \mid S_A | \epsilon
\]

all rules of \( G_A \)

Every possible left-most derivation of a string by \( K \) must begin with zero or more applications of the rule \( S \rightarrow S_SS_A \) followed by the rule \( S \rightarrow \epsilon \). This means that every left-most derivation begins with a sequence of rule applications that is consistent with one of the following relationships:

\[
S \Rightarrow \epsilon \\
S \Rightarrow S_A \\
S \Rightarrow S_A S_A \\
S \Rightarrow S_A S_A S_A \\
\vdots
\]

and so on. After this, each occurrence of \( S_A \) generates any string in \( A \). It is therefore the case that \( L(K) = A^* \), so that \( A^* \) is context-free.

\[
\square
\]

9.2 Every regular language is context-free

Next we will prove that every regular language is also context-free. In fact, we will see two different ways to prove this fact.
Theorem 9.2. Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language. The language $A$ is context-free.

**First proof.** With every regular expression $R$ over the alphabet $\Sigma$, one may associate a CFG $G$ by recursively applying these simple constructions:

1. If $R = \emptyset$, then $G$ is the CFG
   \[ S \rightarrow S, \]  
   (9.9)
   which generates the empty language $\emptyset$.
2. If $R = \epsilon$, then $G$ is the CFG
   \[ S \rightarrow \epsilon, \]  
   (9.10)
   which generates the language $\{\epsilon\}$.
3. If $R = a$ for $a \in \Sigma$, then $G$ is the CFG
   \[ S \rightarrow a, \]  
   (9.11)
   which generates the language $\{a\}$.
4. If $R = (R_1 \cup R_2)$, then $G$ is the CFG generating the language $L(G_1) \cup L(G_2)$, as described in the proof of Theorem 9.1, where $G_1$ and $G_2$ are CFGs associated with the regular expressions $R_1$ and $R_2$, respectively.
5. If $R = (R_1 R_2)$, then $G$ is the CFG generating the language $L(G_1) L(G_2)$, as described in the proof of Theorem 9.1, where $G_1$ and $G_2$ are CFGs associated with the regular expressions $R_1$ and $R_2$, respectively.
6. If $R = (R_1^*)$, then $G$ is the CFG generating the language $L(G_1)^*$, as described in the proof of Theorem 9.1, where $G_1$ is the CFG associated with the regular expression $R_1$.

In each case, we observe that $L(G) = L(R)$.

Now, by the assumption that $A$ is regular, there must exist a regular expression $R$ such that $L(R) = A$. For the CFG $G$ obtained from $R$ as described above, we find that $L(G) = A$, and therefore $A$ is context-free.

**Second proof.** Because $A$ is regular, there must exist a DFA

\[ M = (Q, \Sigma, \delta, q_0, F) \]  
(9.12)

such that $L(M) = A$. Because it does not matter what names we assign to the states of a DFA, there is no loss of generality in assuming $Q = \{q_0, \ldots, q_{n-1}\}$ for some choice of a positive integer $n$. 

96
We will define a CFG $G$ that effectively simulates $M$, generating exactly those strings that are accepted by $M$. In particular, we will define

$$G = (V, \Sigma, R, X_0) \quad (9.13)$$

where the variables are $V = \{X_0, \ldots, X_{n-1}\}$ (i.e., one variable for each state of $M$) and the following rules are to be included in $R$:

1. For each choice of $k, m \in \{0, \ldots, n-1\}$ and $a \in \Sigma$ satisfying $\delta(q_k, a) = q_m$, the rule
   $$X_k \rightarrow aX_m \quad (9.14)$$
   is included in $R$.
2. For each $m \in \{0, \ldots, n-1\}$ satisfying $q_m \in F$, the rule
   $$X_m \rightarrow \varepsilon \quad (9.15)$$
   is included in $R$.

Now, by examining the rules suggested above, we see that every derivation of a string by $G$ starts with $X_0$ (of course), involves zero or more applications of rules of the first type listed above, and then ends when a rule of the second type is applied. There will always be a single variable appearing after each step of the derivation, until the very last step in which this variable is eliminated. It is important that this final step is only possible when the variable $X_k$ corresponds to an accept state $q_k \in F$. By considering the rules of the first type, it is evident that

$$[X_0 \Rightarrow^* wX_m] \iff [\delta^*(q_0, w) = q_m]. \quad (9.16)$$

We therefore have $X_0 \Rightarrow^* w$ if and only if there exists a choice of $m \in \{0, \ldots, n-1\}$ for which $\delta^*(q_0, w) = q_m$ and $q_m \in F$. This is equivalent to the statement that $L(G) = L(M)$, which completes the proof. \qed

### 9.3 Intersections of regular and context-free languages

The context-free languages are not closed under some operations for which the regular languages are closed. For example, the complement of a context-free language may fail to be context-free, and the intersection of two context-free languages may fail to be context-free. (We will observe both of these facts in the next lecture.) It is the case, however, that the intersection of a context-free language and a regular language is always context-free.

97
Theorem 9.3. Let $\Sigma$ be an alphabet, let $A, B \subseteq \Sigma^*$ be languages, and assume $A$ is context-free and $B$ is regular. The language $A \cap B$ is context-free.

Remark 9.4. Before discussing the proof of this theorem, let us note that it implies Theorem 9.2; one is free to choose $A = \Sigma^*$ (which is context-free) and $B$ to be any regular language, and the implication is that $\Sigma^* \cap B = B$ is context-free. Because the proof is quite a bit more involved than the two proofs of Theorem 9.2 that we already discussed, however, it made sense that we considered them first.

Proof. The language $A$ is context-free, so there exists a CFG that generates it. As discussed in the previous lecture, we may in fact assume that there exists a CFG in Chomsky normal form that generates $A$. Having this CFG be in Chomsky normal form will greatly simplify the proof. Hereafter we will assume

$$G = (V, \Sigma, R, S)$$ (9.17)

is a CFG in Chomsky normal form such that $L(G) = A$. Because the language $B$ is regular, there must also exist a DFA

$$M = (Q, \Sigma, \delta, q_0, F)$$ (9.18)

such that $L(M) = B$.

The main idea of the proof is to define a new CFG $H$ such that $L(H) = A \cap B$. The CFG $H$ will have $|Q|^2$ variables for each variable of $G$, which may be a lot but that’s not a problem; it is a finite number, and that is all we require of a set of variables of a context-free grammar. In particular, for each variable $X \in V$, we will include a variable $X_{p,q}$ in $H$ for every choice of $p, q \in Q$. In addition, we will add a new start variable $S_0$ to $H$.

The intended meaning of each variable $X_{p,q}$ is that it should generate all strings that (i) are generated by $X$ with respect to the grammar $G$, and (ii) cause $M$ to move from state $p$ to state $q$. We will accomplish this by adding a collection of rules to $H$ for each rule of $G$. Because the grammar $G$ is assumed to be in Chomsky normal form, there are just three possible forms for its rules, and they can be handled one at a time as follows:

1. For each rule of the form $X \rightarrow a$ in $G$, include the rule

$$X_{p,q} \rightarrow a$$ (9.19)

in $H$ for every pair of states $p, q \in Q$ for which $\delta(p, a) = q$.

2. For each rule of the form $X \rightarrow YZ$ in $G$, include the rule

$$X_{p,q} \rightarrow Y_{p,r}Z_{r,q}$$ (9.20)

in $H$ for every choice of states $p, q, r \in Q$. 
Lecture 9

3. If the rule $S \rightarrow \varepsilon$ is included in $G$ and $q_0 \in F$ (i.e., $\varepsilon \in A \cap B$), then include the rule

$$S_0 \rightarrow \varepsilon$$

(9.21)

in $H$, where $S_0$ is the new start variable for $H$ mentioned above.

Once we have added all of these rules in $H$, we also include the rule

$$S_0 \rightarrow S_{q_0,p}$$

(9.22)

in $H$ for every accept state $p \in F$.

The intended meaning of each variable $X_{p,q}$ in $H$ has been suggested above. More formally speaking, we wish to prove that the following equivalence holds for every nonempty string $w \in \Sigma^*$, every variable $X \in V$, and every choice of states $p,q \in Q$:

$$[X_{p,q} \Rightarrow_H w] \iff [(X \Rightarrow_G w) \land (\delta^*(p,w) = q)].$$

(9.23)

The two implications can naturally be handled separately, and one of the two implications naturally splits into two parts.

First, it is almost immediate that the implication

$$[X_{p,q} \Rightarrow_H w] \Rightarrow [X \Rightarrow_G w]$$

(9.24)

holds, as a derivation of $w$ starting from $X_{p,q}$ in $H$ gives a derivation of $w$ starting from $X$ in $G$ if we simply remove all of the subscripts on all of the variables.

Next, we can prove the implication

$$[X_{p,q} \Rightarrow_H w] \Rightarrow [\delta^*(p,w) = q]$$

(9.25)

by induction on the length of $w$. The base case is $|w| = 1$ (because we are assuming $w \neq \varepsilon$), and in this case we must have $X_{p,q} \Rightarrow_H a$ for some $a \in \Sigma$. The only rules that allow such a derivation are of the first type above, which require $\delta(p,a) = q$. In the general case in which $|w| \geq 2$, it must be that

$$X_{p,q} \Rightarrow Y_{p,r} Z_{r,q}$$

(9.26)

for variables $Y_{p,r}$ and $Z_{r,q}$ satisfying

$$Y_{p,r} \Rightarrow_H y \quad \text{and} \quad Z_{r,q} \Rightarrow_H z$$

(9.27)

for strings $y,z \in \Sigma^*$ for which $w = yz$. By the hypothesis of induction we conclude that $\delta^*(p,y) = r$ and $\delta^*(r,z) = q$, so that $\delta^*(p,w) = q$. 99
Finally, we can prove
\[ [(X \Rightarrow_G^* w) \land (\delta^*(p, w) = q)] \Rightarrow [X_{p,q} \Rightarrow_H^* w], \]  
(9.28)
again by induction on the length of \( w \). The base case is \( |w| = 1 \), which is straightforward: if \( X \Rightarrow_G^* a \) and \( \delta(p, a) = q \), then \( X_{p,q} \Rightarrow_H^* a \) because the rule that allows for this derivation has been included among the rules of \( H \). In the general case in which \( |w| \geq 2 \), the relation \( X \Rightarrow_G^* w \) implies that \( X \Rightarrow_G YZ \) for variables \( Y, Z \in V \) such that \( Y \Rightarrow_G^* y \) and \( Z \Rightarrow_G^* z \), for strings \( y, z \in \Sigma^* \) satisfying \( w = yz \). Choosing \( r \in Q \) so that \( \delta^*(p, y) = r \) (and therefore \( \delta^*(r, z) = q \)), we have that \( Y_{p,r} \Rightarrow_H^* y \) and \( Z_{r,q} \Rightarrow_H^* z \) by the hypothesis of induction, and therefore \( X_{p,q} \Rightarrow_H Y_{p,r}Z_{r,q} \Rightarrow_H^* yz = w \).

Because every derivation of a nonempty string by \( H \) must begin with
\[ S_0 \Rightarrow_H S_{q_0,p} \]  
(9.29)
for some \( p \in F \), we find that the nonempty strings \( w \) generated by \( H \) are precisely those strings that are generated by \( G \) and satisfy \( \delta^*(q_0, w) = p \) for some \( p \in F \). Equivalently, for \( w \neq \varepsilon \) it is the case that \( w \in L(H) \iff w \in A \cap B \). The empty string has been handled as a special case, so it follows that \( L(H) = A \cap B \). The language \( A \cap B \) is therefore context-free. \( \square \)

### 9.4 Prefixes, suffixes, and substrings

Let us finish off the lecture with just a few quick examples. Recall from Lecture 6 that for any language \( A \subseteq \Sigma^* \) we define
\[
\text{Prefix}(A) = \{ x \in \Sigma^* : \text{there exists } v \in \Sigma^* \text{ such that } xv \in A \}, \tag{9.30}
\]
\[
\text{Suffix}(A) = \{ x \in \Sigma^* : \text{there exists } u \in \Sigma^* \text{ such that } ux \in A \}, \tag{9.31}
\]
\[
\text{Substring}(A) = \{ x \in \Sigma^* : \text{there exist } u, v \in \Sigma^* \text{ such that } uxv \in A \}. \tag{9.32}
\]

Let us prove that if \( A \) is context-free, then each of these languages is also context-free. In the interest of time, we will just explain how to come up with context-free grammars for these languages and not go into details regarding the proofs that these CFGs are correct. In all three cases, we will assume that \( G = (V, \Sigma, R, S) \) is a CFG in Chomsky normal form such that \( L(G) = A \).

We will need to make one additional assumption on the grammar \( G \), which is that none of the variables in \( G \) generates the empty language. A variable that generates the empty language is called a **useless variable**, and it should not be hard
to convince yourself that useless variables are indeed useless (with one exception). That is, if you have any CFG $G$ in Chomsky normal form that generates a nonempty language, you can easily come up with a new CFG in Chomsky normal form for the same language that does not contain any useless variables simply by removing the useless variables and every rule in which a useless variable appears.

The one exception is the empty language itself, which by definition requires that the start variable is useless (and you will need at least one additional useless variable to ensure that the grammar has a nonempty set of rules and obeys the conditions of a CFG in Chomsky normal form). However, we don’t need to worry about this case because Prefix($\emptyset$), Suffix($\emptyset$), and Substring($\emptyset$) are all equal to the empty language, and are therefore context-free.

For the language Prefix($A$), we will design a CFG $H$ as follows. First, for every variable $X \in V$ used by $G$ we will include this variable in $H$, and in addition we will also include a variable $X_0$. The idea is that $X$ will generate exactly the same strings in $H$ that it does in $G$, while $X_0$ will generate all the prefixes of the strings generated by $X$ in $G$. We include rules in $H$ as follows:

1. For every rule of the form $X \to YZ$ in $G$, include these rules in $H$:

   \[ X \to YZ \]
   \[ X_0 \to YZ_0 \mid Y_0 \]  \hspace{1cm} (9.33)

2. For every rule of the form $X \to a$ in $G$, include these rules in $H$:

   \[ X \to a \]
   \[ X_0 \to a \mid \epsilon \]  \hspace{1cm} (9.34)

Finally, we take $S_0$ to be the start variable of $H$.

The idea is similar for the language Suffix($A$), for which we will construct a CFG $K$. This time, for every variable $X \in V$ used by $G$ we will include this variable in $K$, and in addition we will also include a variable $X_1$. The idea is that $X$ will generate exactly the same strings in $K$ that it does in $G$, while $X_1$ will generate all the suffixes of the strings generated by $X$ in $G$. We include rules in $K$ as follows:

1. For every rule of the form $X \to YZ$ in $G$, include these rules in $K$:

   \[ X \to YZ \]
   \[ X_1 \to Y_1Z \mid Z_1 \]  \hspace{1cm} (9.35)

2. For every rule of the form $X \to a$ in $G$, include these rules in $K$:

   \[ X \to a \]
   \[ X_1 \to a \mid \epsilon \]  \hspace{1cm} (9.36)
Finally, we take $S_1$ to be the start variable of $K$.

To obtain a CFG $J$ for $\text{Substring}(A)$, we can simply combine the two constructions above (i.e., apply either one to $G$, then apply the other to the resulting CFG). Equivalently, we can include variables $X$, $X_0$, $X_1$, and $X_2$ in $J$ for every $X \in V$ and include rules as follows:

1. For every rule of the form $X \rightarrow YZ$ in $G$, include these rules in $J$:
   
   $\begin{align*}
   X & \rightarrow YZ \\
   X_0 & \rightarrow YZ_0 | Y_0 \\
   X_1 & \rightarrow Y_1 Z | Z_1 \\
   X_2 & \rightarrow Y_1 Z_0 | Y_2 | Z_2
   \end{align*}$

   \hspace{1cm} (9.37)

2. For every rule of the form $X \rightarrow a$ in $G$, include these rules in $J$:

   $\begin{align*}
   X & \rightarrow a \\
   X_0 & \rightarrow a | \epsilon \\
   X_1 & \rightarrow a | \epsilon \\
   X_2 & \rightarrow a | \epsilon.
   \end{align*}$

   \hspace{1cm} (9.38)

Finally, we take $S_2$ to be the start variable of $J$. The meaning of the variables $X$, $X_0$, $X_1$, and $X_2$ in $J$ is that they generate precisely the strings generated by $X$ in $G$, the prefixes, the suffixes, and the substrings of these strings, respectively.
Lecture 10

Proving languages to be non-context-free

In this lecture we will study a method through which certain languages can be proved to be non-context-free. The method will appear to be quite familiar, because it closely resembles the one we discussed in Lecture 5 for proving certain languages to be nonregular.

10.1 The pumping lemma (for context-free languages)

Along the same lines as the method we discussed in Lecture 5 for proving some languages to be nonregular, we will start with a variant of the pumping lemma that holds for context-free languages.

The proof of this lemma is, naturally, different from the proof of the pumping lemma for regular languages, but there are similar underlying ideas. The main idea is that if you have a parse tree for the derivation of a particular string by some context-free grammar, and the parse tree is sufficiently deep, then there must be a variable that appears multiple times on some path from the root to a leaf—and by modifying the parse tree in certain ways, one obtains a similar type of pumping effect that we had in the case of the pumping lemma for regular languages.

Lemma 10.1 (Pumping lemma for context-free languages). Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a context-free language. There exists a positive integer $n$ (called a pumping length of $A$) that possesses the following property. For every string $w \in A$ with $|w| \geq n$, it is possible to write $w = uvxyz$ for some choice of strings $u, v, x, y, z \in \Sigma^*$ such that

1. $vy \neq \varepsilon$,
2. $|vxy| \leq n$, and
3. $uv^i xy^i z \in A$ for all $i \in \mathbb{N}$.
Figure 10.1: At least one path from the root to a leaf in a CNF parse tree for a string of length $2^m$ or more must have $m + 1$ or more variable nodes. If this were not so, the total number of variable nodes (represented by the shaded region) would be at most $2^m - 1$, contradicting the fact that there must be at least $2^m$ variable nodes.

Proof. Given that $A$ is context-free, we know that there must exist a CFG $G$ in Chomsky normal form such that $A = L(G)$. Let $m$ be the number of variables in $G$. We will prove that the property stated in the lemma holds for $n = 2^m$.

Suppose that a string $w \in A$ satisfies $|w| \geq n = 2^m$. As $G$ is in Chomsky normal form, every parse tree for $w$ has exactly $2|w| - 1$ variable nodes and $|w|$ leaf nodes. Hereafter let us fix any one of these parse trees, and let us call this tree $T$. For the sake of this proof, what is important about the size of $T$ is that the number of variable nodes is at least $2^m$. (This is true because $2|w| - 1 \geq 2 \cdot 2^m - 1 \geq 2^m$. In fact, the last inequality must be strict because $m \geq 1$, but this makes no difference to the proof.) Because the number of variable nodes in $T$ is at least $2^m$, there must exist at least one path in $T$ from the root to a leaf along which there are at least $m + 1$ variable nodes—for if all such paths had $m$ or fewer variable nodes, there could be at most $2^m - 1$ variable nodes in the entire tree.

Next, choose any path in $T$ from the root to a leaf having the maximum possible length. (There may be multiple choices, but any one of them is fine.) We know that at least $m + 1$ variable nodes must appear in this path, as argued above—and because there are only $m$ different variables in total, there must be at least one variable that appears multiple times along this path. In fact, we know that some variable (let us call it $X$) must appear at least twice within the $m + 1$ variable nodes.
closest to the leaf on the path we have selected. Let $T_1$ and $T_2$ be the subtrees of $T$ rooted at these two bottom-most occurrences of this variable $X$. By the way we have chosen these subtrees, we know that $T_2$ is a proper subtree of $T_1$, and $T_1$ is not very large: every path from the root of the subtree $T_1$ to one of its leaves can have at most $m + 1$ variable nodes, and therefore $T_1$ has no more than $2^m = n$ leaf nodes.

Now, let $x$ be the string for which $T_2$ is a parse tree (starting from the variable $X$) and let $v$ and $y$ be the strings formed by the leaves of $T_1$ to the left and right, respectively, of the subtree $T_2$, so that $vxy$ is the string for which $T_1$ is a parse tree (also starting from the variable $X$). Finally, let $u$ and $z$ be the strings represented by the leaves of $T$ to the left and right, respectively, of the subtree $T_1$, so that $w = uvxyz$. Figure 10.2 provides an illustration of the strings $u$, $v$, $x$, $y$, and $z$ and how they related to the trees $T$, $T_1$, and $T_2$.

It remains to prove that $u$, $v$, $x$, $y$, and $z$ have the properties required by the statement of the lemma. Let us first prove that $uv^i xy^j z \in A$ for all $i \in \mathbb{N}$. To see that $uxz = uv^0 xy^0 z \in A$, we observe that we can obtain a valid parse tree for $uxz$ by replacing the subtree $T_1$ with the subtree $T_2$, as illustrated in Figure 10.3. This replacement is possible because both $T_1$ and $T_2$ have root nodes corresponding to the variable $X$. Along similar lines, we have that $uv^2 xy^2 z \in A$ because we can obtain a valid parse tree for this string by replacing the subtree $T_2$ with a copy of $T_1$, as suggested by Figure 10.4. By repeatedly replacing $T_2$ with a copy of $T_1$, a valid parse tree for any string of the form $uv^i xy^j z$ is obtained.

Next, the fact that $vy \neq \varepsilon$ follows from the fact that every parse tree for a string
Figure 10.3: By replacing the subtree $T_1$ by the subtree $T_2$ in $T$, a parse tree for the string $uxz = uv^0 xy^0 z$ is obtained.

Figure 10.4: By replacing the subtree $T_2$ by the subtree $T_1$ in $T$, a parse tree for the string $uv^2 xy^2 z$ is obtained. By repeatedly replacing $T_2$ with $T_1$ in this way, a parse tree for the string $uv^i xy^i z$ is obtained for any positive integer $i \geq 2$. 
corresponding to a CFG in Chomsky normal form has the same size. It therefore cannot be that the parse tree suggested by Figure 10.3 generates the same string as the one suggested by Figure 10.2, as the two trees have differing numbers of variable nodes. This implies that $uvxyz \neq uxz$, so $vy \neq \varepsilon$.

Finally, we have $|vxy| \leq n$ because the subtree $T_1$ has at most $2^m = n$ leaf nodes, as was already argued above. □

### 10.2 Using the context-free pumping lemma

Now that we have the pumping lemma for context-free languages in hand, we can prove that certain languages are not context-free. The methodology is very similar to what we used in Lecture 5 to prove some languages to be nonregular. Some examples, stated as propositions, follow.

**Proposition 10.2.** Let $\Sigma = \{0, 1, 2\}$ and let $A$ be a language defined as follows:

$$A = \{0^m1^m2^m : m \in \mathbb{N}\}. \quad (10.1)$$

The language $A$ is not context-free.

**Proof.** Assume toward contradiction that $A$ is context-free. By the pumping lemma for context-free languages, there must exist a pumping length $n \geq 1$ for $A$.

Let $w = 0^n1^n2^n$. We have that $w \in A$ and $|w| = 3n \geq n$, so the pumping lemma guarantees that there must exist strings $u, v, x, y, z \in \Sigma^*$ so that $w = uvxyz$ and the three properties in the statement of that lemma hold: (i) $vy \neq \varepsilon$, (ii) $|vxy| \leq n$, and (iii) $uv^ixyz \in A$ for all $i \in \mathbb{N}$.

Now, given that $|vxy| \leq n$, it cannot be that the symbols 0 and 2 both appear in the string $vy$; the 0s and 2s are too far apart for this to happen. On the other hand, at least one of the symbols of $\Sigma$ must appear within $vy$, because this string is nonempty. This implies that the string

$$uv^0xy^0z = uxz \quad (10.2)$$

must have strictly fewer occurrences of either 1 or 2 than 0, or strictly fewer occurrences of either 0 or 1 than 2. That is, if the symbol 0 does not appear in $vy$, then it must be that either

$$|uxz|_1 < |uxz|_0 \quad \text{or} \quad |uxz|_2 < |uxz|_0, \quad (10.3)$$

and if the symbol 2 does not appear in $vy$, then it must be that either

$$|uxz|_0 < |uxz|_2 \quad \text{or} \quad |uxz|_1 < |uxz|_2. \quad (10.4)$$
This, however, is in contradiction with the fact that $uv^0xy^0z = uxz$ is guaranteed to be in $A$ by the third property.

Having obtained a contradiction, we conclude that $A$ is not context-free, as claimed.

In some cases, such as the following one, a language can be proved to be non-context-free in almost exactly the same way that it can be proved to be nonregular.

**Proposition 10.3.** Let $\Sigma = \{0\}$ and define a language over $\Sigma$ as follows:

$$B = \{0^m : m \text{ is a perfect square}\}.$$  \hspace{1cm} (10.5)

The language $B$ is not context-free.

**Proof.** Assume toward contradiction that $B$ is context-free. By the pumping lemma for context-free languages, there must exist a pumping length $n \geq 1$ for $B$ for which the property stated by that lemma holds. We will fix such a pumping length $n$ for the remainder of the proof.

Define $w = 0^{n^2}$. We see that $w \in B$ and $|w| = n^2 \geq n$, so the pumping lemma tells us that there exist strings $u, v, x, y, z \in \Sigma^*$ so that $w = uvxyz$ and the following conditions hold:

1. $vy \neq \varepsilon$,
2. $|vx| \leq n$, and
3. $uv^i xy^i z \in B$ for all $i \in \mathbb{N}$.

There is only one symbol in the alphabet $\Sigma$, so it is immediate that $vy = 0^k$ for some choice of $k \in \mathbb{N}$. Because $vy \neq \varepsilon$ and $|vy| \leq |vx| \leq n$ it must be the case that $1 \leq k \leq n$. Observe that

$$uv^i xy^i z = 0^{n^2+(i-1)k}$$  \hspace{1cm} (10.6)

for each $i \in \mathbb{N}$. In particular, if we choose $i = 2$, then we have

$$uv^2 xy^2 z = 0^{n^2+k}.$$  \hspace{1cm} (10.7)

However, because $1 \leq k \leq n$, it cannot be that $n^2 + k$ is a perfect square. This is because $n^2 + k$ is larger than $n^2$, but the next perfect square after $n^2$ is

$$(n+1)^2 = n^2 + 2n + 1,$$  \hspace{1cm} (10.8)

which is strictly larger than $n^2 + k$ because $k \leq n$. The string $uv^2 xy^2 z$ is therefore not contained in $B$, which contradicts the third condition stated by the pumping lemma, which guarantees us that $uv^i xy^i z \in B$ for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that $B$ is not context-free, as claimed. \qed
Remark 10.4. We will not discuss the proof, but it turns out that every context-free language over a single-symbol alphabet must be regular. By combining this fact with the fact that $B$ is nonregular, we obtain a different proof that $B$ is not context-free.

Here is one more example of a proof that a particular language is not context-free using the pumping lemma for context-free languages. For this one things get a bit messy because there are multiple cases to worry about as we try to get a contradiction, which turns out to be fairly common when using this method. Of course, one has to be sure to get a contradiction in all of the cases in order to have a valid proof by contradiction, so be sure to keep this in mind.

Proposition 10.5. Let $\Sigma = \{0, 1, \#\}$ and define a language $C$ over $\Sigma$ as follows:

$$C = \{ r\#s : r, s \in \{0, 1\}^*, r \text{ is a substring of } s \}.$$  \hspace{1cm} (10.9)

The language $C$ is not context-free.

Proof. Assume toward contradiction that $C$ is context-free. By the pumping lemma for context-free languages, there exists a pumping length $n \geq 1$ for $C$.

Let

$$w = 0^n1^n\#0^n1^n.$$  \hspace{1cm} (10.10)

It is the case that $w \in C$ (because $0^n1^n$ is a substring of itself) and $|w| = 4n + 1 \geq n$. The pumping lemma therefore guarantees that there exist strings $u, v, x, y, z \in \Sigma^*$ so that $w = uvxyz$ and the three properties in the statement of that lemma hold: (i) $vy \neq \varepsilon$, (ii) $|vxy| \leq n$, and (iii) $uv^ixy^iz \in C$ for all $i \in \mathbb{N}$.

There is just one occurrence of the symbol $\#$ in $w$, so it must appear in one of the strings $u, v, x, y, \text{ or } z$. We will consider each case separately:

Case 1: the $\#$ lies within $u$. In this case we have that all of the symbols in $v$ and $y$ appear to the right of the symbol $\#$ in $w$. It follows that

$$uv^0xy^0z = 0^n1^n\#0^{n-j}1^{n-k}$$  \hspace{1cm} (10.11)

for some choice of integers $j$ and $k$ with $j + k \geq 1$, because by removing $v$ and $y$ from $w$ we must have removed at least one symbol to the right of the symbol $\#$ (and none from the left of that symbol). The string (10.11) is not contained in $C$, even though the third property guarantees it is, and so we have a contradiction in this case.

Case 2: the $\#$ lies within $v$. This is an easy case: because the $\#$ symbol lies in $v$, the string $uv^0xy^0z = uxz$ does not contain the symbol $\#$ at all, so it cannot be in $C$. This is again in contradiction with the third property, which guarantees that $uv^0xy^0z \in C$, and so we have a contradiction in this case.
Case 3: the # lies within x. In this case, we know that \( vxy = 1^j#0^k \) for some choice of integers \( j \) and \( k \) for which \( j + k \geq 1 \). The reason why \( vxy \) must take this form is that \( |vxy| \leq n \), so this substring cannot both contain the symbol # and reach either the first block of 0s or the last block of 1s, and the reason why \( j + k \geq 1 \) is that \( vy \neq \varepsilon \).

If it happens that \( j \geq 1 \), then we may choose \( i = 2 \) to obtain a contradiction, as

\[
vuv^2xy^2z = 0^n1^n^j#0^n^k1^n,\tag{10.12}
\]

which is not in \( C \) because the string to the left of the # symbol has more 1s than the string to the right of the # symbol. If it happens that \( k \geq 1 \), then we may choose \( i = 0 \) to obtain a contradiction: we have

\[
vuv^0xy^0z = 0^n1^n^j#0^n^k1^n \tag{10.13}
\]

in this case, which is not contained in \( C \) because the string to the left of the # symbol has more 0s than the string to the right of the # symbol.

Case 4: the # lies within y. This case is identical to case 2—the string \( uv^0xy^0z \) cannot be in \( C \) because it does not contain the symbol #.

Case 5: the # lies within z. In this case we have that all of the symbols in \( v \) and \( y \) appear to the left of the symbol # in \( w \). Because \( vy \neq \varepsilon \), it follows that

\[
vuv^2xy^2z = r#0^n1^n \tag{10.14}
\]

for some string \( r \) that has length strictly larger than \( 2n \). The string (10.14) is not contained in \( C \), even though the third property guarantees it is, and so we have a contradiction in this case.

Having obtained a contradiction in all of the cases, we conclude that there must really be a contradiction—so \( C \) is not context-free, as claimed.

### 10.3 Non-context-free languages and closure properties

In the previous lecture it was stated that the context-free languages are not closed under either intersection or complementation. That is, there exist context-free languages \( A \) and \( B \) such that neither \( A \cap B \) nor \( \overline{A} \) are context-free. We can now verify these claims.

First, let us consider the case of intersection. Suppose we define languages \( A \) and \( B \) as follows:

\[
A = \{0^n1^n2^m : n, m \in \mathbb{N}\},
\]

\[
B = \{0^n1^m2^m : n, m \in \mathbb{N}\}.\tag{10.15}
\]
These are certainly context-free languages—a CFG generating $A$ is given by

\[
S \rightarrow XY \\
X \rightarrow 0X1 \mid \varepsilon \tag{10.16} \\
Y \rightarrow 2Y \mid \varepsilon
\]

and a CFG generating $B$ is given by

\[
S \rightarrow XY \\
X \rightarrow 0X \mid \varepsilon \tag{10.17} \\
Y \rightarrow 1Y2 \mid \varepsilon
\]

On the other hand, the intersection $A \cap B$ is not context-free, as our first proposition from the previous section established.

Having proved that the context-free languages are not closed under intersection, it follows immediately that the context-free languages are not closed under complementation. This is because we already know that the context-free languages are closed under union, and if they were also closed under complementation we would conclude that they must also be closed under intersection by De Morgan’s laws.

Finally, let us observe that one can sometimes use closure properties to prove that certain languages are not context-free. For example, consider the language

\[
D = \{w \in \{0, 1, 2\}^* : |w|_0 = |w|_1 = |w|_2\}. \tag{10.18}
\]

It would be possible to prove that $D$ is not context-free using the pumping lemma in a similar way to the first proposition from the previous section. A simpler way to conclude this fact is as follows. We assume toward contradiction that $D$ is context-free. Because the intersection of a context-free language and a regular language must always be context-free, it follows that $D \cap L(0^*1^*2^*)$ is context-free (because $L(0^*1^*2^*)$ is the language matched by a regular expression and is therefore regular). However,

\[
D \cap L(0^*1^*2^*) = \{0^m1^m2^m : m \in \mathbb{N}\}, \tag{10.19}
\]

which we already know is not context-free. Having obtained a contradiction, we conclude that $D$ is not context-free, as required.
Lecture 11

Pushdown automata

This is the last lecture of the course that is devoted to context-free languages. As for regular languages, however, we will refer to context-free languages from time to time throughout the remainder of the course.

The first part of the lecture will focus on the pushdown automata model of computation, which gives an alternative characterization of context-free languages to the definition based on CFGs. The second part of the lecture will be devoted to some further properties of context-free languages that we have not discussed thus far, and that happen to be useful for understanding pushdown automata.

11.1 Pushdown automata

The pushdown automaton (or PDA) model of computation is essentially what you get if you equip NFAs with a stack. As we shall see, the class of languages recognized by PDAs is precisely the class of context-free languages, which provides a useful tool for reasoning about these languages.

A few simple examples

Let us begin with an example of a PDA, expressed in the form of a state diagram in Figure 11.1. The state diagram naturally looks a bit different from the state diagram of an NFA or DFA, because it includes instructions for operating with the stack, but the basic idea is the same. A transition labeled by an input symbol or \( \epsilon \) means that we read a symbol or take an \( \epsilon \)-transition, just like an NFA; a transition labeled \((\downarrow, a)\) means that we push the symbol \( a \) onto the stack; and a transition labeled \((\uparrow, a)\) means that we pop the symbol \( a \) off of the stack.

Thus, the way the PDA \( P \) illustrated in Figure 11.1 works is that it first pushes the stack symbol \( \Diamond \) onto the stack (which we assume is initially empty) and enters
Figure 11.1: The state diagram of a PDA $P$

diagram text

state $q_1$ (without reading anything from the input). From state $q_1$ it is possible to either read the left-parenthesis symbol “(” and move to $r_0$ or read the right-parenthesis symbol “)” and move to $r_1$. To get back to $q_1$ we must either push the symbol $\ast$ onto the stack (in the case that we just read a left-parenthesis) or pop the symbol $\ast$ off of the stack (in the case that we just read a right-parenthesis). Finally, to get to the accept state $q_2$ from $q_1$, we must pop the symbol $\diamond$ off of the stack. Note that a transition requiring a pop operation can only be followed if that symbol is actually there on the top of the stack to be popped. It is not too hard to see that the language recognized by this PDA is the language BAL of balanced parentheses; these are precisely the input strings for which it will be possible to perform the required pops to land on the accept state $q_2$ after the entire input string is read.

A second example is given in Figure 11.2. In this case the PDA accepts every string in the language

$$\{0^n1^n : n \in \mathbb{N}\}.$$  (11.1)

In this case the stack is essentially used as a counter: we push a star for every 0, pop a star for every 1, and by using the “bottom of the stack marker” $\diamond$ we check that an equal number of the two symbols have been read.

**Definition of pushdown automata**

The formal definition of the pushdown automata model is similar to that of nonde­terministic finite automata, except that one must also specify the alphabet of stack
symbols and alter the form of the transition function so that it specifies how the stack operates.

**Definition 11.1.** A pushdown automaton (or PDA for short) is a 6-tuple

\[ P = (Q, \Sigma, \Gamma, \delta, q_0, F) \]  

where \( Q \) is a finite and nonempty set of states, \( \Sigma \) is an alphabet (called the input alphabet), \( \Gamma \) is an alphabet (called the stack alphabet), \( \delta \) is a function of the form

\[ \delta : Q \times (\Sigma \cup \text{Stack}(\Gamma) \cup \{ \varepsilon \}) \rightarrow \mathcal{P}(Q), \]

where \( \text{Stack}(\Gamma) = \{ \downarrow, \uparrow \} \times \Gamma \), \( q_0 \in Q \) is the start state, and \( F \subseteq Q \) is a set of accept states. It is required that \( \Sigma \cap \text{Stack}(\Gamma) = \emptyset \).

The way to interpret a transition function having the above form is that the set of possible labels on transitions is

\[ \Sigma \cup \text{Stack}(\Gamma) \cup \{ \varepsilon \}; \]

we can either read a symbol \( a \), push a symbol from \( \Gamma \) onto the stack, pop a symbol from \( \Gamma \) off of the stack, or take an \( \varepsilon \)-transition.

**Strings of valid stack operations**

Before we discuss the formal definition of acceptance for PDAs, it will be helpful to think about stacks and valid sequences of stack operations. Consider an alphabet \( \Gamma \) that we will think of as representing a stack alphabet, and define

\[ \text{Stack}(\Gamma) = \{ \downarrow, \uparrow \} \times \Gamma \]  

Figure 11.2: A PDA recognizing the language \( \{0^n1^n : n \in \mathbb{N} \} \).
as we have done in Definition 11.1. The alphabet \( \text{Stack}(\Gamma) \) represents the possible stack operations for a stack that uses the alphabet \( \Gamma \); for each \( a \in \Gamma \) we imagine that the symbol \((\downarrow, a)\) represents pushing \( a \) onto the stack, and that the symbol \((\uparrow, a)\) represents popping \( a \) off of the stack.

Now, we can view a string \( v \in \text{Stack}(\Gamma)^* \) as either representing or failing to represent a valid sequence of stack operations, assuming we read it from left to right and imagine starting with an empty stack. If a string does represent a valid sequence of stack operations, we will say that it is a valid stack string; and if a string fails to represent a valid sequence of stack operations, we will say that it is an invalid stack string.

For example, if \( \Gamma = \{0, 1\} \), then these strings are valid stack strings:
\[
(\downarrow, 0)(\downarrow, 1)(\uparrow, 1)(\downarrow, 0)(\uparrow, 0),
(\downarrow, 0)(\downarrow, 1)(\uparrow, 1)(\downarrow, 0)(\uparrow, 0).
\]

In the first case the stack is transformed like this (where the left-most symbol represents the top of the stack):
\[
\varepsilon \rightarrow 0 \rightarrow 10 \rightarrow 0 \rightarrow 00 \rightarrow 0 \rightarrow \varepsilon.
\]

The second case is similar, except that we don’t leave the stack empty at the end:
\[
\varepsilon \rightarrow 0 \rightarrow 10 \rightarrow 0 \rightarrow 00 \rightarrow 0.
\]

On the other hand, these strings are invalid stack strings:
\[
(\downarrow, 0)(\downarrow, 1)(\uparrow, 0)(\downarrow, 0)(\uparrow, 1)(\uparrow, 0),
(\downarrow, 0)(\downarrow, 1)(\uparrow, 1)(\downarrow, 0)(\uparrow, 0)(\uparrow, 1).
\]

For the first case we start by pushing 0 and then 1, which is fine, but then we try to pop 0 even though 1 is on the top of the stack. In the second case the very last symbol is the problem: we try to pop 1 even though the stack is empty.

It is the case that the language over the alphabet \( \text{Stack}(\Gamma) \) consisting of all valid stack strings is a context-free language. To see that this is so, let us first consider the language of all valid stack strings that also leave the stack empty after the last operation. For instance, the first sequence in (11.6) has this property while the second does not. We can obtain a CFG for this language by mimicking the CFG for the balanced parentheses language, but imagining a different parenthesis type for each symbol. To be more precise, let us define a CFG \( G \) so that it includes the rule
\[
S \rightarrow (\downarrow, a) S (\uparrow, a) S
\]
Lecture 11

for every symbol $a \in \Gamma$, as well as the rule $S \rightarrow \epsilon$. This CFG generates the language of valid stack strings for the stack alphabet $\Gamma$ that leave the stack empty at the end.

If we drop the requirement that the stack be left empty after the last operation, then we still have a context-free language. This is because this is the language of all prefixes of the language generated by the CFG in the previous paragraph, and the context-free languages are closed under taking prefixes.

**Definition of acceptance for PDAs**

Next let us consider a formal definition of what it means for a PDA $P$ to accept or reject a string $w$.

**Definition 11.2.** Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA and let $w \in \Sigma^*$ be a string. The PDA $P$ accepts the string $w$ if there exists a natural number $m \in \mathbb{N}$, a sequence of states $r_0, \ldots, r_m$, and a sequence $a_1, \ldots, a_m \in \Sigma \cup \operatorname{Stack}(\Gamma) \cup \{\epsilon\}$ for which these properties hold:

1. $r_0 = q_0$ and $r_m \in F$.
2. $r_{k+1} \in \delta(r_k, a_{k+1})$ for every $k \in \{0, \ldots, m-1\}$.
3. By removing every symbol from the alphabet $\operatorname{Stack}(\Gamma)$ from $a_1 \cdots a_m$, the input string $w$ is obtained.
4. By removing every symbol from the alphabet $\Sigma$ from $a_1 \cdots a_m$, a valid stack string is obtained.

If $P$ does not accept $w$, then $P$ rejects $w$.

For the most part the definition is straightforward. In order for $P$ to accept $w$, there must exist a sequence of states, along with moves between these states, that agree with the input string and the transition function. In addition, the usage of the stack must be consistent with our understanding of what a stack is, and this is represented by the fourth property.

As you would expect, for a given PDA $P$, we let $L(P)$ denote the language recognized by $P$, which is the language of all strings accepted by $P$.

**Some useful shorthand notation for PDA state diagrams**

There is a shorthand notation for PDA state diagrams that is sometimes useful, which is essentially to represent a sequence of transitions as if it were a single
transition. In particular, if a transition is labeled

$$a \uparrow b \downarrow c,$$

(11.12)

the meaning is that the symbol $a$ is read, $b$ is popped off of the stack, and then $c$ is pushed onto the stack. Figure 11.3 illustrates how this shorthand is to be interpreted. It is to be understood that the “implicit” states in a PDA represented by this shorthand are unique to each edge. For instance, the states $r_1$ and $r_2$ in Figure 11.3 are only used to implement this one transition from $p$ to $q$, and are not reachable from any other states or used to implement other transitions.

This sort of shorthand notation can also be used in case multiple symbols are to be pushed or popped. For instance, an edge labeled

$$a \uparrow b_1b_2b_3 \downarrow c_1c_2c_3c_4$$

(11.13)

means that $a$ is read from the input, $b_1b_2b_3$ is popped off the top of the stack, and then $c_1c_2c_3c_4$ is pushed onto the stack. We will always follow the convention that the top of the stack corresponds to the left-hand side of any string of stack symbols, so such a transition requires $b_1$ on the top of the stack, $b_2$ next on the stack, and $b_3$ third on the stack—and when the entire operation is done, $c_1$ is on top of the stack, $c_2$ is next, and so on. One can follow a similar pattern to what is shown in Figure 11.3 to implement such a transition using the ordinary types of transitions from the definition of PDAs, along with intermediate states to perform the operations in the right order. Finally, we can simply omit parts of a transition of the above form if those parts are not used. For instance, the transition label

$$a \uparrow b$$

(11.14)

means “read $a$ from the input, pop $a$ off of the stack, and push nothing,” the transition label

$$\uparrow b \downarrow c_1c_2$$

(11.15)

means “read nothing from the input, pop $b$ off of the stack, and push $c_1c_2$,” and so on. Figure 11.4 illustrates the same PDA as in Figure 11.2 using this shorthand.
A remark on deterministic pushdown automata

It must be stressed that pushdown automata are, by default, considered to be non-deterministic. It is possible to define a deterministic version of the PDA model, but if we do this we end up with a strictly weaker computational model. That is, every deterministic PDA will recognize a context-free language, but some context-free languages cannot be recognized by a deterministic PDA. One example is the language PAL of palindromes over the alphabet $\Sigma = \{0, 1\}$; this language is recognized by the PDA in Figure 11.5, but no deterministic PDA can recognize it.\footnote{We will not prove this fact, and indeed we have not even discussed a formal definition for deterministic PDAs, but the intuition is clear enough. Deterministic PDAs cannot detect when they have reached the middle of a string, and for this reason the use of a stack is not enough to recognize palindromes; no matter how you do it, the machine will never know when to stop pushing and start popping. A nondeterministic machine, on the other hand, can simply guess when to do this.}

11.2 Further examples

Next we will consider a few additional operations under which the context-free languages are closed. These include string reversals, symmetric differences with finite languages, and a couple of operations that involve inserting and deleting certain alphabet symbols from strings.
Reverse

We already discussed string reversals in Lecture 6, where we observed that the reverse of a regular language is always regular. The same thing is true of context-free languages, as the following proposition establishes.

**Proposition 11.3.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a context-free language. The language $A^R$ is context-free.

**Proof.** Because $A$ is context-free, there must exist a CFG $G$ such that $A = L(G)$. Define a new CFG $H$ as follows: $H$ contains exactly the same variables as $G$, and for each rule $X \rightarrow w$ of $G$ we include the rule $X \rightarrow w^R$ in $H$. In words, $H$ is the CFG obtained by reversing the right-hand side of every rule in $G$. It is evident that $L(H) = L(G)^R = A^R$, and therefore $A^R$ is context-free. \qed

Symmetric difference with a finite language

Next we will consider symmetric differences, which were also defined in Lecture 6. It is certainly not the case that the symmetric difference between two context-free languages is always context-free, or even that the symmetric difference between a context-free language and a regular language is context-free.

For example, if $A \subseteq \Sigma^*$ is context-free but $\overline{A}$ is not, then the symmetric difference between $A$ and the regular language $\Sigma^*$ is not context-free, because

$$A \triangle \Sigma^* = \overline{A}. \quad (11.16)$$

On the other hand, the symmetric difference between a context-free language and a finite language must always be context-free, as the following proposition shows. This is interesting because the symmetric difference between a given language and a finite language carries an intuitive meaning: it means we modify that language on a finite number of strings, by either including or excluding these strings. The proposition therefore shows that the property of being context-free does not change when a language is modified on a finite number of strings.

**Proposition 11.4.** Let $\Sigma$ be an alphabet, let $A \subseteq \Sigma^*$ be a context-free language, and let $B \subseteq \Sigma^*$ be a finite language. The language $A \triangle B$ is context-free.

**Proof.** First, given that $B$ is finite, we have that $B$ is regular, and therefore $\overline{B}$ is regular as well, because the regular languages are closed under complementation. This implies that $A \cap \overline{B}$ is context-free, because the intersection of a context-free language and a regular language is context-free.

Next, we observe that $\overline{A} \cap B$ is contained in $B$, and is therefore finite. Every finite language is context-free, and therefore $\overline{A} \cap B$ context-free.
Finally, given that we have proved that both $A \cap B$ and $\overline{A} \cap B$ are context-free, it follows that $A \Delta B = (A \cap B) \cup (\overline{A} \cap B)$ is context-free because the union of two context-free languages is necessarily context-free.

**Closure under string projections**

Suppose that $\Sigma$ and $\Gamma$ are disjoint alphabets, and we have a string $w \in (\Sigma \cup \Gamma)^*$ that may contain symbols from either or both of these alphabets. We can imagine deleting all of the symbols in $w$ that are contained in the alphabet $\Gamma$, which leaves us with a string over $\Sigma$. We call this operation the *projection* of a string over the alphabet $\Sigma \cup \Gamma$ onto the alphabet $\Sigma$.

We will prove two simple closure properties of the context-free languages that concern this notion. The first one says that if you have a context-free language over the alphabet $\Sigma \cup \Gamma$, and you project all of the strings in $A$ onto the alphabet $\Sigma$, you’re left with a context-free language.

**Proposition 11.5.** Let $\Sigma$ and $\Gamma$ be disjoint alphabets, let $A \subseteq (\Sigma \cup \Gamma)^*$ be a context-free language, and define

$$B = \left\{ w \in \Sigma^* : \text{there exists a string } x \in A \text{ such that } w \text{ is obtained from } x \text{ by deleting all symbols in } \Gamma \right\}.$$  \hfill (11.17)

The language $B$ is context-free.

**Proof.** Because $A$ is context-free, there exists a CFG $G$ in Chomsky normal form such that $L(G) = A$. We will create a new CFG $H$ as follows:

1. For every rule of the form $X \rightarrow YZ$ appearing in $G$, include the same rule in $H$. Also, if the rule $S \rightarrow \epsilon$ appears in $G$, include this rule in $H$ as well.
2. For every rule of the form $X \rightarrow a$ in $G$, where $a \in \Sigma$, include the same rule $X \rightarrow a$ in $H$.
3. For every rule of the form $X \rightarrow b$ in $G$, where $b \in \Gamma$, include the rule $X \rightarrow \epsilon$ in $H$.

It is apparent that $L(H) = B$, and therefore $B$ is context-free.

We can also go the other way, so to speak: if $A$ is a context-free language over the alphabet $\Sigma$, and we consider the language consisting of all strings over the alphabet $\Sigma \cup \Gamma$ that result in a string in $A$ when they are projected onto the alphabet $\Sigma$, then this new language over $\Sigma \cup \Gamma$ will also be context-free. In essence, this is the language you get by picking any string in $A$, and then inserting any number of symbols from $\Gamma$ anywhere into the string.
Proposition 11.6. Let \( \Sigma \) and \( \Gamma \) be disjoint alphabets, let \( A \subseteq \Sigma^* \) be a context-free language, and define

\[
B = \left\{ x \in (\Sigma \cup \Gamma)^* : \text{the string } w \text{ obtained from } x \text{ by deleting all symbols in } \Gamma \text{ satisfies } w \in A \right\}.
\]  

(11.18)

The language \( B \) is context-free.

Proof. Because \( A \) is context-free, there exists a CFG \( G \) in Chomsky normal form such that \( L(G) = A \). Define a new CFG \( H \) as follows:

1. Include the rule

\[
W \rightarrow bW
\]

(11.19)

in \( H \) for each \( b \in \Gamma \), as well as the rule \( W \rightarrow \varepsilon \), for a new variable \( W \) not already used in \( G \). The variable \( W \) generates any string of symbols from \( \Gamma \), including the empty string.

2. For each rule of the form \( X \rightarrow YZ \) in \( G \), include the same rule in \( H \) without modifying it.

3. For each rule of the form \( X \rightarrow a \) in \( G \), include this rule in \( H \):

\[
X \rightarrow WaW
\]

(11.20)

4. If the rule \( S \rightarrow \varepsilon \) is contained in \( G \), then include this rule in \( H \):

\[
S \rightarrow W
\]

(11.21)

Intuitively speaking, \( H \) operates in much the same way as \( G \), except that any time \( G \) generates a symbol or the empty string, \( H \) is free to generate the same string with any number of symbols from \( \Gamma \) inserted. We have that \( L(H) = B \), and therefore \( B \) is context-free. \( \square \)

11.3 Equivalence of PDAs and CFGs

As suggested earlier in the lecture, it is the case that a language is context-free if and only if it is recognized by a PDA. This section gives a high-level description of one way to prove this equivalence.
Every context-free language is recognized by a PDA

To prove that every context-free language is recognized by some PDA, we can define a PDA that corresponds directly to a given CFG. That is, if \( G = (V, \Sigma, R, S) \) is a CFG, then we can obtain a PDA \( P \) such that \( L(P) = L(G) \) in the manner suggested by Figure 11.6. The stack symbols of \( P \) are taken to be \( V \cup \Sigma \), along with a special bottom of the stack marker \( \diamond \) (which we assume is not contained in \( V \cup \Sigma \)), and during the computation the stack will provide a way to store the symbols and variables needed to carry out a derivation with respect to the grammar \( G \).

If you consider how derivations of strings by a grammar \( G \) and the operation of the corresponding PDA \( P \) work, it will be evident that \( P \) accepts precisely those strings that can be generated by \( G \). We start with just the start variable on the stack (in addition to the bottom of the stack marker). In general, if a variable appears on the top of the stack, we can pop it off and replace it with any string of symbols and variables appearing on the right-hand side of a rule for the variable that was popped; and if a symbol appears on the top of the stack we essentially just match it up with an input symbol—so long as the input symbol matches the symbol on the top of the stack we can pop it off, move to the next input symbol, and process whatever is left on the stack. We can move to the accept state whenever the stack is empty (meaning that just the bottom of the stack marker is present), and if all of the input symbols have been read we accept. This situation is representative of the input string having been derived by the grammar.

Every language recognized by a PDA is context-free

We will now argue that every language recognized by a PDA is context-free. There is a method through which a given PDA can actually be converted into an equivalent CFG, but it is messy and the intuition tends to get lost in the details. Here we will summarize a different way to prove that every language recognized by a PDA is context-free that is pretty simple, given the tools that we’ve already collected in
our study of context-free languages. If you wanted to, you could turn this proof into an explicit construction of a CFG for a given PDA, and it wouldn’t be all that different from the method just mentioned, but we’ll focus just on the proof and not on turning it into an explicit construction.

Suppose we have a PDA \( P = (Q, \Sigma, \Gamma, \delta, q_0, F) \). The transition function \( \delta \) takes the form

\[
\delta : Q \times (\Sigma \cup \text{Stack}(\Gamma) \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q),
\]

so if we wanted to, we could think of \( P \) as being an NFA for some language over the alphabet \( \Sigma \cup \text{Stack}(\Gamma) \). Slightly more formally, let \( N \) be the NFA defined as

\[
N = (Q, \Sigma \cup \text{Stack}(\Gamma), \delta, q_0, F);
\]

we don’t even need to change the transition function because it already has the right form of a transition function for an NFA over the alphabet \( \Sigma \cup \text{Stack}(\Gamma) \). Also define \( B = L(N) \subseteq (\Sigma \cup \text{Stack}(\Gamma))^* \) to be the language recognized by \( N \). In general, the strings in \( B \) include symbols in both \( \Sigma \) and \( \text{Stack}(\Gamma) \). Even though symbols in \( \text{Stack}(\Gamma) \) may be present in the strings accepted by \( N \), there is no requirement on these strings to actually represent a valid use of a stack, because \( N \) doesn’t have a stack with which to check this condition.

Now let us consider a second language \( C \subseteq (\Sigma \cup \text{Stack}(\Gamma))^* \). This will be the language consisting of all strings over the alphabet \( \Sigma \cup \text{Stack}(\Gamma) \) having the property that by deleting every symbol in \( \Sigma \), a valid stack string is obtained. We already discussed the fact that the language consisting of all valid stack strings is context-free, and so it follows from Proposition 11.6 that the language \( C \) is also context-free.

Next, we consider the intersection \( D = B \cap C \). Because \( D \) is the intersection of a regular language and a context-free language, it is context-free. The strings in \( D \) actually correspond to valid computations of the PDA \( P \) that lead to an accept state; but in addition to the input symbols in \( \Sigma \) that are read by \( P \), these strings also include symbols in \( \text{Stack}(\Gamma) \) that represent transitions of \( P \) that involve stack operations.

The language \( D \) is therefore not the same as the language \( A \), but it is closely related; \( A \) is the language that is obtained from \( D \) by deleting all of the symbols in \( \text{Stack}(\Gamma) \) and leaving the symbols in \( \Sigma \) alone. Because we know that \( D \) is context-free, it therefore follows that \( A \) is context-free by Proposition 11.5, which is what we wanted to prove.
Lecture 12

Stack machines

Our focus in this course will now shift to more powerful computational models. We will begin with a new model of computation called the stack machine model. This model resembles the pushdown automata model, but unlike that model the stack machine model permits the use of multiple stacks. As it turns out, this makes a huge difference; whereas pushdown automata are highly limited computational models, stack machines are computationally universal. Informally speaking, this means that stack machines have the same language recognition power as a modern day programming language, or any other “reasonable” model of a general computer.

As you gain familiarity with the stack machine model, you will likely notice that it has a different character from the more limited models discussed previously in the course. In particular, through the use of multiple stacks one can mimic complex data structures, and transitions between states can mimic the sorts of control flow that is familiar in standard programming languages. Generally speaking, one has a great deal of freedom when defining a stack machine to make it operate in a desired way, just like an ordinary computer program.

12.1 Nondeterministic stack machines

We will mostly be concerned with a deterministic variant of the stack machine model in this course, but it is nevertheless convenient to begin by defining a nondeterministic variant of this model. There are two principal reasons for beginning with the nondeterministic version:

1. We have already studied the pushdown automata model, for which the nondeterministic variant is considered the default (and indeed we did not even cover the deterministic variant of this model), and the nondeterministic stack machine model is a conceptually simple variation on the pushdown automata model.
2. We will arrive at a formal definition for the deterministic stack machine model by placing restrictions on the nondeterministic version of the model. These restrictions will be natural, and they will also have a side-effect of making the model easier to work with, but they are not quite as simple as the analogous situation for finite automata, for instance.

In short, a nondeterministic stack machine (NSM) is just like a pushdown automaton, except that there are multiple stacks. The number of stacks is fixed for a particular machine, and when we wish to make explicit that the number of stacks is some positive integer \( r \), we will refer to that machine as an \( r \)-NSM.

Here are a couple of additional points, which highlight differences between stack machines and pushdown automata that should be kept in mind as you consider the formal definition:

1. We will assume that a given NSM begins with its input string stored on its first stack (to be indexed by 0), and there will be a special bottom-of-the-stack symbol \( \Diamond \) at the bottom of this input stack underneath the input. This is just a simple way of allowing the machine access to its input without necessitating a special “read from the input” action that differs from an ordinary stack operation.

2. We will assume that every stack has the same stack alphabet, which must include the symbols in the input alphabet (because the input is assumed to be loaded into the first stack) along with the bottom-of-the-stack symbol \( \Diamond \), which may not be included in the input alphabet. We are free to include additional symbols in the stack alphabet if we choose. The computational power of the model would not actually change if we allowed each stack to have its own alphabet, but placing this restriction on the model keeps the definition and associated notation simpler.

**Definition 12.1.** An \( r \)-stack nondeterministic stack machine (\( r \)-NSM, for short) is an 8-tuple

\[
M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \tag{12.1}
\]

where

1. \( Q \) is a finite and nonempty set of states,
2. \( \Sigma \) is an input alphabet, which may not include the bottom-of-the-stack symbol \( \Diamond \),
3. \( \Delta \) is a stack alphabet, which must satisfy \( \Sigma \cup \{\Diamond\} \subseteq \Delta \),
4. \( \delta \) is a transition function of the form

\[
\delta : (Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}) \times \{\uparrow_0, \downarrow_0, \ldots, \uparrow_{r-1}, \downarrow_{r-1}\} \times \Delta \to \mathcal{P}(Q), \tag{12.2}
\]

and
Lecture 12

5. \( q_0, q_{\text{acc}}, q_{\text{rej}} \in Q \) are the initial state, accept state, and reject state, respectively, which must satisfy \( q_{\text{acc}} \neq q_{\text{rej}} \).

There are two operations a nondeterministic stack machine may perform: push operations and pop operations. The machine may also transition between states as it performs these operations. An interpretation of the transition function \( \delta \), which specifies how these operations and transitions may be performed, is as follows:

1. If it is the case that \( q \in \delta(p, \downarrow_k, a) \), then when the machine is in the state \( p \), it may push the symbol \( a \) onto stack \( k \) and transition to state \( q \).
2. If it is the case that \( q \in \delta(p, \uparrow_k, a) \), then when the machine is in the state \( p \) and the symbol \( a \) is on the top of stack \( k \), it may pop the symbol \( a \) off of this stack and transition to state \( q \).

We can describe NSMs using state diagrams in a similar way to finite automata and pushdown automata. As usual, states are represented by circles, and the inclusions \( q \in \delta(p, \downarrow_k, a) \) and \( q \in \delta(p, \uparrow_k, a) \) are indicated by arrows like this:

\[
\begin{align*}
p & \xrightarrow{(\downarrow_k, a)} q \\
p & \xrightarrow{(\uparrow_k, a)} q
\end{align*}
\]

Stack machine computations begin as follows:

1. The input string \( x \in \Sigma^+ \) is assumed to be stored in stack 0. Specifically, the top symbol of stack 0 stores the first symbol of \( x \), the second to top symbol of stack 0 contains the second symbol of \( x \), and so on. At the bottom of stack 0, underneath all of the input symbols, is the bottom-of-the-stack symbol \( \diamond \).
2. All of the other stacks initially contain just the bottom-of-the-stack symbol \( \diamond \).
3. The computation starts in the initial state \( q_0 \).

The computation of a stack machine continues (nondeterministically, in general) so long as the machine is in one of the states other than \( q_{\text{acc}} \) and \( q_{\text{rej}} \) and there are valid transitions that the machine may follow. Whenever one of the states \( q_{\text{acc}} \) or \( q_{\text{rej}} \) is reached, the computation stops (or halts, according to traditional parlance); the corresponding computation path is deemed as accepting or rejecting, accordingly. Computation paths that effectively terminate because they result in a situation in which there are no possible transitions that can be followed are also considered to be rejecting computation paths. It is important to note that there is also a possibility for computations to carry on indefinitely, and in such a situation we will refer to the machine running forever.
Based on the discussion above, it would be routine to formulate a definition for when an NSM $M$ accepts a given string $x \in \Sigma^*$, but we’ll postpone the formal definition of acceptance for now. In the next lecture we will discuss computations of NSMs somewhat more generally, and it will be most efficient for us to include the formal definition of acceptance of NSMs in this more general discussion.

**Example 12.2.** An example of a 3-NSM $M$, described by a state diagram, is illustrated in Figure 12.1. The operation of this machine is as follows. In its first phase (represented by states $p, p_0, p_1$), $M$ pops 0 and 1 symbols off of the input stack (stack 0) and pushes the same symbol it pops onto stack 1. Assuming it encounters a # symbol in the input string, it transitions to $r$. The states $r, r_0, r_1$ represent the second phase of the computation, in which 0 and 1 symbols are popped off of the input stack and onto stack 2 rather than stack 1. There will at some point be no legal transitions for the machine to make if the input contains two or more occurrences of the symbol #, or if it reaches the end of the input string without encountering a # symbol. If the input contains exactly one # symbol, then the computation will transition to $s$ after the entire input string has been popped off of the input stack. The final phase of the computation compares stacks 1 and 2, and allows the accept state $q_{\text{acc}}$ to be reached if and only if these two stacks contain the same string. The strings accepted by $M$ are therefore represented by the language

$$ \{w\#w : w \in \{0, 1\}^*\}. $$
The state $q_{rej}$ happens to be unreachable, and is not really needed at all—but it has been included in the diagram because the definition requires that this state exists.\(^\text{1}\)

The language (12.3) is not context-free, so with our very first example we have established that nondeterministic stack machines are computationally more powerful than pushdown automata. (We did not actually prove that nondeterministic stack machines are at least as powerful as pushdown automata, but this is nearly obvious: a 2-NSM can easily simulate a PDA.)

### 12.2 Deterministic stack machines

We will now define a deterministic variant of the stack machine model. Before doing this, it should be noted that this is not quite as simple as demanding that there should always exist exactly one possible transition that can be followed from every state at every instant during a computation of a given NSM. Of course we want this property to hold, but if we simply demand this property and nothing more, we run into a problem: it turns out that it is a undecidable problem to determine whether or not this property holds for a given NSM.

With that in mind, our definition for deterministic stack machines will be somewhat more restricted. What we will do is to insist that every non-halting state is either a push state or a pop state—so that only push transitions or pop transitions, but not both, may be followed from each state—and moreover we will always associate exactly one of the stacks with each state, so that all of the push transitions or pop transitions from that state are limited to the one stack associated with that state. Finally, if the state is a push state, there must be exactly one push transition leading from that state, and if the state is a pop state, then there must be exactly one pop transition leading from that state for every possible stack symbol.

**Definition 12.3.** An $r$-NSM

\[
M = (Q, \Sigma, \Delta, \delta, q_0, q_{acc}, q_{rej})
\]

is an $r$-stack deterministic stack machine ($r$-DSM, for short) if, for every state $q \in Q \setminus \{q_{acc}, q_{rej}\}$, there exists an index $k \in \{0, \ldots, r - 1\}$ such that exactly one of the following two properties is satisfied:

1. (The state $q$ is a push state.) There exists a symbol $a \in \Delta$ such that

\[
|\delta(q, \downarrow_k, a)| = 1.
\]

\(^1\)There is no need for a reject state in general for nondeterministic stack machines, but this state is important in the deterministic case. It is included in the definition of NSMs so that deterministic stack machines can be more easily described as a special case of nondeterministic stack machines.
Moreover,

$$\delta(q, \downarrow j, b) = \emptyset$$ \hspace{1cm} \text{(12.6)}

for all \( j \in \{0, \ldots, r - 1\} \) and \( b \in \Delta \) satisfying \((j, b) \neq (k, a)\), as well as

$$\delta(q, \uparrow j, b) = \emptyset$$ \hspace{1cm} \text{(12.7)}

for all \( j \in \{0, \ldots, r - 1\} \) and \( b \in \Delta \).

2. (The state \( q \) is a pop state.) For every symbol \( a \in \Delta \) it is the case that

$$|\delta(q, \uparrow k, a)| = 1,$$ \hspace{1cm} \text{(12.8)}

as well as

$$\delta(q, \uparrow j, a) = \emptyset$$ \hspace{1cm} \text{(12.9)}

for all \( j \in \{0, \ldots, r - 1\} \setminus \{k\} \). Moreover,

$$\delta(q, \downarrow j, b) = \emptyset$$ \hspace{1cm} \text{(12.10)}

for all \( j \in \{0, \ldots, r - 1\} \) and \( b \in \Delta \).

**State diagrams for DSMs**

Deterministic stack machines may be represented by state diagrams in a way that is somewhat different from general nondeterministic stack machines. As usual, states will be represented by nodes in a directed graph, directed edges (with labels) will represent transitions, and the accept and reject states are labeled as such. You will be able to immediately recognize that a state diagram represents a DSM in these notes from the fact that the nodes are square-shaped (with slightly rounded corners) rather than circle or oval shaped.

What is different from the sort of state diagram we discussed earlier in the lecture for NSMs is that the nodes themselves, rather than the transitions, will indicate the operation (push or pop) that are to be performed, as well as the stack associated with that operation. Each push state must have a single transition leading from it, with the label indicating which symbol is pushed and with the transition pointing to the next state. Each pop state must have one directed edge leading from it for each possible stack symbol, indicating to which state the computation is to transition (depending on the symbol popped). Finally, we will commonly assign names like \( X, Y, \) and \( Z \) to different stacks, rather than calling them \( \text{stack 0, stack 1,} \) and so on, as this makes for more natural, algorithmically focused descriptions of stack machines, where we view the stacks as being akin to variables in a computer program.

Figure 12.2 gives an example of a state diagram of a 3-DSM. Two additional comments on state diagrams for DSMs follow.
1. Aside from the accept and reject states, we tend not to include the names of individual states in state diagrams. This is because the names we choose for the states are irrelevant to the functioning of a given machine, and omitting them makes for less cluttered diagrams. In rare cases in which it is important to include the name of a state in a state diagram, we will just write the state name above or beside its corresponding node.

2. Although every deterministic stack machine is assumed to have a reject state, we often do not bother to include it in state diagrams. Whenever there is a state with which a pop operation is associated, and one or more of the possible stack symbols does not appear on any transition leading out of this state, it is assumed that the “missing” transitions lead to the reject state.

For example, in Figure 12.2, there is no transition labeled \( \Diamond \) leading out of the initial state, so it is implicit that if \( \Diamond \) is popped off of X from this state, the machine enters the reject state.

Figure 12.2: A 3-DSM for the language \( \{w \# w : w \in \{0, 1\}^* \} \). The input stack is named X and the other two stacks are named Y and Z.
Subroutines

Just like we often do with ordinary programming languages, we can define subroutines for stack machines. This can sometimes offer a major simplification to the descriptions of stack machines.

For example, consider the DSM whose state diagram is shown in Figure 12.3. Before discussing this machine, let us agree that whenever we say that a particular stack stores a string $x$, we mean that the bottom-of-the-stack marker $\diamond$ appears on the bottom of the stack, and the symbols of $x$ appear above this bottom-of-the-stack marker on the stack, with the leftmost symbol of $x$ on the top of the stack. (We will only use this terminology in the situation that the symbol $\diamond$ does not appear in $x$.) Using this terminology, the behavior of the DSM illustrated in Figure 12.3 is that if its computation begins with $X$ storing an arbitrary string $x \in \{0, 1\}^*$, then the computation always results in acceptance, with $X$ storing $\epsilon$. In other words, the DSM erases the string stored by $X$ and halts.

Now, the simple process performed by this DSM might be useful as a subroutine inside of some more complicated DSM, and of course a simple modification allows us to choose any stack in place of $X$ that gets erased. Rather than replicating the description of the DSM from Figure 12.5 inside of this more complicated hypothetical DSM, we can simply use the sort of shorthand suggested by Figure 12.4.

More explicitly, the diagram on the left-hand side of Figure 12.4 suggests a small part of a hypothetical DSM, where the DSM from Figure 12.3 appears inside of the dashed box. Note that we have not included the accept state in the

Figure 12.3: An example of a state diagram describing a 1-DSM, whose sole stack is named $X$. This particular machine is not very interesting from a language-recognition viewpoint—it accepts every string—but it performs the useful task of erasing the contents of a stack. Here the stack alphabet is assumed to be $\Delta = \{0, 1, \diamond\}$, but the idea is easily extended to other stack alphabets.
Figure 12.4: The diagrams on the left and right describe equivalent portions of a larger DSM; the contents of the dotted rectangle in the left-hand diagram is viewed as a subroutine that is represented by a single rectangle labeled “\( \text{X} \leftarrow \varepsilon \)” in the right-hand diagram.

dashed box because, rather than accepting, we wish for control to be passed to the state labeled “another state” as the erasing process completes. We also do not have that the “pop \( \text{X} \)” state is the initial state any longer, because rather than starting at this state, we have that control passes to this state from the state labeled “some state.” (There could, in fact, be multiple transitions from multiple states leading to the “pop \( \text{X} \)” state in the hypothetical DSM we’re considering.) In the diagram on the right-hand side of Figure 12.4 we have replaced the dashed box with a single rectangle labeled “\( \text{X} \leftarrow \varepsilon \)” This is just a label that we’ve chosen, but of course it is a fitting label in this case. The rectangle labeled “\( \text{X} \leftarrow \varepsilon \)” looks like a state, and we can think of it as being like a state with which a more complicated operation than push or pop is associated—but the reality is that it is just a short-hand for the contents of the dashed box on the left-hand side diagram.

The same general pattern can be replicated for just about any choice of a DSM. That is, if we have a DSM that we would like to use as a subroutine, we can always do this as follows:

1. Let the original start state of the DSM be the state to which some transition points.

2. Remove the accept state, modifying transitions to this removed accept state so that they point to some other state elsewhere in the larger DSM to which control is to pass once the subroutine is complete.
Figure 12.5: An example of a state diagram describing a 3-DSM (with stacks named \( X \), \( Y \), and \( Z \)). This machine performs the task of copying the contents of one stack to another: \( X \) is copied to \( Y \). The stack \( Z \) is used as workspace to perform this operation.

Naturally, one must be careful when defining and using subroutines like this, as computations could easily become corrupted if subroutines modify stacks that are being used for other purposes elsewhere in a computation. (The same thing can, of course, be said concerning subroutines in ordinary computer programs.)

Another example of a subroutine is illustrated in Figure 12.5. This stack machine copies the contents of one stack to another, using a third stack as an auxiliary (or workspace) stack to accomplish this task. Specifically, under the assumption that a stack \( X \) stores a string \( x \in \{0, 1\}^* \) and stacks \( Y \) and \( Z \) store the empty string, the illustrated 3-DSM will always lead to acceptance—and when it does accept, the stacks \( X \) and \( Y \) will both store the string \( x \), while \( Z \) will revert to its initial configuration in which it stores the empty string. This action can be summarized as follows:

\[
\begin{align*}
\text{X stores } x & \quad \rightarrow \quad \text{X stores } x \\
\text{Y stores } \varepsilon & \quad \rightarrow \quad \text{Y stores } x \\
\text{Z stores } \varepsilon & \quad \rightarrow \quad \text{Z stores } \varepsilon
\end{align*}
\]
If we wish to use this DSM as a subroutine in a more complicated DSM, we could again represent the entire DSM (minus the accept state) by a single rectangle, just like we did in Figure 12.4. A fitting label in this case is “\( Y \leftarrow X \).”

One more example of a DSM that is useful as a subroutine is pictured in Figure 12.6. Notice that in this state diagram we have made use of the two previous subroutines to make the figure simpler. After each new subroutine is defined, we’re naturally free to use it to describe new DSMs. The DSM in the figure reverses the string stored by \( X \). It uses a workspace stack \( Y \) to accomplish this task—but in fact it also uses a workspace stack \( Z \), which is hidden inside the subroutine labeled “\( Y \leftarrow X \).” In summary, it performs this transformation:

\[
\begin{array}{c|c|c|c}
\text{X stores } x & \text{Y stores } \varepsilon & \text{Z stores } \varepsilon & \text{X stores } x^R \\
\end{array}
\]

Hereafter we will tend not to list explicitly all of the workspace stacks used by our DSMs. So long as we are careful to always revert our workspace stacks back to their initial configuration, in which they store the empty string, we can just imagine that each occurrence of a subroutine in a given state diagram has its own workspace stacks associated with it that we never use in other subroutine occurrences or elsewhere in the larger DSM under consideration. We also won’t worry about using an excessive number of stacks; there are strategies for limiting the number of stacks required to perform computations (and in fact just two stacks are always enough, as we will see later), but for the purposes of this course there will be no compelling reasons to focus on this issue.
Lecture 13

Stack machine computations, languages, and functions

In this lecture we will discuss computations of stack machines at a formal level, and then we will define a few fundamental notions that we will carry with us for the rest of the course, including the notions of semidecidable and decidable languages and computable functions. The last part of the lecture will be devoted to further developing the stack machine model of computation, with a principle aim being to recognize that the deterministic stack machine model is, in an abstract mathematical sense, representative of the computational power of an ordinary computer.¹

13.1 Stack machine configurations and computations

In the previous lecture we defined the NSM model, and then we defined the DSM model by restricting the definition of NSMs in a way that guarantees that we always have just one possible transition out of each non-halting state at each instant. It remains for us to formally define acceptance and rejection for NSMs and DSMs; and although it would be routine to do this by mimicking the analogous definitions for PDAs, we will take a different approach that will allow us to define the important concept of computable functions more or less simultaneously. The approach is reminiscent of the definition for when a CFG generates a given string.

Let us begin by defining the set of configurations of a stack machine. Intuitively speaking, a configuration of a stack machine describes its current state and the contents of all of its stacks at a particular instant.

¹You might be inclined to think that ordinary computers are actually better modeled by finite automata than by stack machines, due to the fact that there is a finite (albeit large) amount of data storage available in the world. Keep in mind, however, that there is a difference between the physical world and the mathematical abstractions through which we aim to understand the world, and recognize that modeling an ordinary computer as a finite automaton is not very informative.
Definition 13.1. Let \( M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \) be an \( r\)-NSM. A configuration of \( M \) is an element of the set \( Q \times (\Delta^*)^r \).

The configuration \( (p, x_0, \ldots, x_{r-1}) \) of a stack machine \( M \) indicates that the current state of \( M \) is \( p \), the contents of stack 0 is given by \( x_0 \) (with the left-most symbol of \( x_0 \) on the top of stack 0 and the rightmost symbol of \( x_0 \) on the bottom of stack 0), the contents of stack 1 is given by \( x_1 \), and so on.

Next, we will define a yields relation on pairs of configurations that represents the possibility that a given stack machine can transition from the first configuration to the second. Similar to the yields relation for context-free grammars, we will define two variants of this relation: the first represents the possibility to move from one configuration to another in a single step, and the second relation is the reflexive transitive closure of the first relation. In other words, the second relation is a starred version, representing the possibility to transition from one configuration to another through zero or more steps of the stack machine under consideration.

Definition 13.2. Let \( M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \) be an \( r\)-NSM. The yields relation defined by \( M \) is the relation denoted \( \vdash_M \) that consists of all of the following pairs of configurations:

1. For every pair of states \( p \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\} \) and \( q \in Q \), every stack symbol \( a \in \Delta \), and every stack index \( k \in \{0, \ldots, r-1\} \) for which \( q \in \delta(p, \downarrow_k, a) \), the yields relation for \( M \) includes the pair

\[
(p, x_0, \ldots, x_{r-1}) \vdash_M (q, x_0, \ldots, x_{r-1}, a x_k, x_{k+1}, \ldots, x_{r-1})
\]  

(13.1)

for all strings \( x_0, \ldots, x_{r-1} \in \Delta^* \).

2. For every pair of states \( p \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\} \) and \( q \in Q \), every stack symbol \( a \in \Delta \), and every stack index \( k \in \{0, \ldots, r-1\} \) for which \( q \in \delta(p, \uparrow_k, a) \), the yields relation for \( M \) includes the pair

\[
(p, x_0, \ldots, x_{r-1}, a x_k, x_{k+1}, \ldots, x_{r-1}) \vdash_M (q, x_0, \ldots, x_{r-1})
\]  

(13.2)

for all strings \( x_0, \ldots, x_{r-1} \in \Delta^* \).

3. For every state \( p \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\} \), every stack symbol \( a \in \Delta \), and every stack index \( k \in \{0, \ldots, r-1\} \) for which \( q \in \delta(p, \uparrow_k, a) \), the yields relation for \( M \) includes the pairs

\[
(p, x_0, \ldots, x_{r-1}) \vdash_M (q_{\text{rej}}, x_0, \ldots, x_{r-1})
\]  

(13.3)

for all strings \( x_0, \ldots, x_{r-1} \in \Delta^* \).
The relation $\vdash^*_M$ is the symmetric, transitive closure of $\vdash_M$. In symbols, for every pair of configurations $(p, x_0, \ldots, x_{r-1})$ and $(q, y_0, \ldots, y_{r-1})$, it is the case that

$$(p, x_0, \ldots, x_{r-1}) \vdash^*_M (q, y_0, \ldots, y_{r-1})$$

if there exist a sequence of configurations $c_0, \ldots, c_t \in Q \times (\Delta^*)^r$, for some non-negative integer $t \in \mathbb{N}$, such that $c_0 = (p, x_0, \ldots, x_{r-1}), c_t = (q, y_0, \ldots, y_{r-1})$, and $c_j \vdash_M c_{j+1}$ for every $j \in \{0, \ldots, t-1\}$.

**Remark 13.3.** The third category in the previous definition is a boundary case that we did not discuss in the previous lecture. It essentially says that it is possible to transition to the reject state by attempting to pop an empty stack. (When this happens, the empty stack remains empty.) By defining the yields relation in this way, we ensure that every non-halting configuration of a DSM has a well-defined next configuration, which is a rejecting configuration when a DSM attempts to pop an empty stack.

### 13.2 Languages and functions from DSMs

Having defined the yields relation $\vdash_M$ for a given stack machine $M$, we are now prepared to define a few important notions. Our primary focus will be on deterministic stack machines, but one can also consider analogous notions for non-deterministic stack machines.

**Acceptance, rejection, and running forever**

First we will formally define what it means for a DSM to accept or to reject a given input string, or to do neither.

**Definition 13.4.** Let $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be an $r$-DSM and let $w \in \Sigma^*$ be an input string.

1. *M accepts w* if there exist strings $x_0, \ldots, x_{r-1} \in \Delta^*$ such that

   $$(q_0, w\diamond, \diamond, \ldots, \diamond) \vdash^*_M (q_{\text{acc}}, x_0, \ldots, x_{r-1}).$$

2. *M rejects w* if there exist strings $x_0, \ldots, x_{r-1} \in \Delta^*$ such that

   $$(q_0, w\diamond, \diamond, \ldots, \diamond) \vdash^*_M (q_{\text{rej}}, x_0, \ldots, x_{r-1}).$$

3. *M runs forever on w* if $M$ neither accepts nor rejects $w$. 

139
It is important to recognize that running forever is a possibility for deterministic stack machines; but you are surely familiar with the same possibility for ordinary computer programs, so it will not likely come as a surprise that this might happen for a stack machine. For the first two possibilities in the previous definition (M accepts $w$ and $M$ rejects $w$), the strings $x_0, \ldots, x_{r-1}$ represent whatever happens to be on the stacks of $M$ when it reaches the accept or reject state. We don’t place any restrictions on what these stacks might contain, all that matters for the sake of this definition is that either the state $q_{acc}$ or the state $q_{rej}$ was reached.

At this point we can be precise about the definition of the language recognized by a DSM.

**Definition 13.5.** Let $M = (Q, \Sigma, \Delta, \delta, q_0, q_{acc}, q_{rej})$ be an $r$-DSM. The language recognized by $M$ is defined as

$$L(M) = \{w \in \Sigma^*: M \text{ accepts } w\}. \quad (13.7)$$

**Semidecidable languages**

We now define the class of *semidecidable* languages to be the class of languages for which there exists a DSM that recognizes that language.

**Definition 13.6.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language $A$ is *semidecidable* if there exists a DSM $M$ such that $A = L(M)$.

There are several alternative names that people often use in place of *semidecidable*, including *Turing recognizable, partially decidable*, and *recursively enumerable* (or r.e. for short).

The name *semidecidable* reflects the fact that if $A = L(M)$ for some DSM $M$, and $w \in A$, then running $M$ on $w$ will necessarily lead to acceptance; but if $w \notin A$, then $M$ might either reject or run forever on input $w$. That is, $M$ does not really decide whether a string $w$ is in $A$ or not, it only “semidecides”; for if $w \notin A$, you might not ever really know this as a result of running $M$ on $w$.

**Decidable languages**

Next, we define the class of *decidable* languages. As the following definition makes clear, a decidable language is one for which there exists a DSM that correctly answers whether or not a given string is in the language (and therefore never runs forever).
Definition 13.7. Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language $A$ is **decidable** if there exists a DSM $M$ with these two properties:

1. $M$ accepts every string $w \in A$.
2. $M$ rejects every string $w \in \overline{A}$.

Example 13.8. The language $\{w#w : w \in \{0, 1\}^*\}$ is decidable, as the DSM for this language defined in Lecture 12 accepts every string in this language and rejects every string not in this language.

**Analogous notions for NSMs**

Although our main focus for the remainder of the course will be on deterministic stack machines, it is appropriate to take a moment to briefly discuss nondeterministic stack machines and analogous notions to those discussed above for DSMs.

We will begin by formally defining acceptance for NSMs as well as the language recognized by a given NSM. These definitions are actually exactly the same as for DSMs, as you might have expected. We will not, however, define rejection or running forever for NSMs.

Definition 13.9. Let $M = (Q, \Sigma, \Delta, \delta, q_0, q_{acc}, q_{rej})$ be an $r$-NSM. The NSM $M$ **accepts** the string $w \in \Sigma^*$ if there exist strings $x_0, \ldots, x_{r-1} \in \Delta^*$ such that

$$ (q_0, w\diamond, \diamond, \ldots, \diamond) \vdash^*_{M} (q_{acc}, x_0, \ldots, x_{r-1}). $$

(13.8)

The language **recognized by** $M$ is defined as

$$ L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}. $$

(13.9)

You may wonder how the class of languages recognized by NSMs compares with the semidecidable languages. We saw previously in the course that NFAs and DFAs are equivalent in terms of which languages they recognize, while deterministic PDAs are strictly less powerful than (nondeterministic) PDAs, so it seems like it could go either way. It turns out that NSMs and DSMs are equivalent in terms of the languages they recognize, as the following theorem states.

**Theorem 13.10.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language $A$ is **semidecidable** if and only if there exists an NSM $M$ such that $A = L(M)$.

We won’t actually prove this theorem because, as was already stated above, our focus will mainly be on deterministic stack machines. It is not a difficult theorem to prove, although a proof would be properly placed after a bit more development of the DSM model than we have seen thus far. The fundamental idea behind
the proof is that for a given NSM \( N \), we can construct a DSM \( M \) that effectively searches through all of the possible computations of \( N \) on a given input string \( w \) using a breadth-first search of the “computation tree” of \( N \). This search might take a long time, this is not a problem because the theorem makes no claims about computational efficiency.

It is also possible to prove a variant of Theorem 13.10 that gives a characterization of decidable languages in terms of NSMs, but we’ll skip this in the interest of saving time. In short, a language \( A \) is decidable if and only if there exists an NSM \( M \) with two properties: the first is that \( A = L(M) \), and the second is that it is never possible for \( M \) to make nondeterministic transitions that cause it to run forever. The idea behind the proof of this fact is exactly the same as the idea of the proof of Theorem 13.10.

### Computable functions

Next we will define the class of computable functions. The motivation behind this concept is quite straightforward: we often want to compute functions, or consider computations that evaluate functions, as opposed to just deciding membership in languages. The concept of a computable function is also useful as a tool for studying decidability and semidecidability of languages.

It is easy to adapt the stack machine model so that it computes functions rather than just deciding membership in a language by (i) requiring that a given DSM \( M \) that computes a function always accepts, and (ii) requiring that the output of the function is stored on the first stack (and all other stacks store the empty string) when the computation eventually enters the accept state. We can generalize this definition without difficulty to allow for different input and output alphabets, and to allow for functions having multiple input and output arguments.

**Definition 13.11.** Let \( \Sigma \) and \( \Gamma \) be alphabets, neither of which includes the bottom-of-the-stack symbol \( \odot \). A function \( f : \Sigma^* \to \Gamma^* \) is computable if there exists an \( r \)-DSM \( M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \) such that, for every string \( x \in \Sigma^* \), it is the case that

\[
(q_0, x \odot, \odot, \ldots, \odot) \vdash_M^* (q_{\text{acc}}, f(x) \odot, \odot, \ldots, \odot).
\]

(13.10)

More generally, a function \( g : (\Sigma^*)^n \to (\Gamma^*)^m \), for positive integers \( n \) and \( m \), is computable if there exists an \( r \)-DSM \( M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \) such that, for every choice of strings \( x_1, \ldots, x_n \in \Sigma^* \), it is the case that

\[
(q_0, x_1 \odot, \ldots, x_n \odot, \odot, \ldots, \odot) \vdash_M^* (q_{\text{acc}}, y_1 \odot, \ldots, y_m \odot, \odot, \ldots, \odot)
\]

(13.11)

for \((y_1, \ldots, y_m) = g(x_1, \ldots, x_n)\).
You could generalize the definition even further by allowing each of the arguments to have its own alphabet, but the definition above is sufficient for our needs. Although we did not refer to them as such, we already saw a few examples of computable functions in Lecture 12:

1. The function \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \) defined as \( f(x) = \varepsilon \) for every \( x \in \{0, 1\}^* \) is computable.
2. The function \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \times \{0, 1\}^* \) defined as \( f(x) = (x, x) \) for every \( x \in \{0, 1\}^* \) is computable.
3. The function \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \) defined as \( f(x) = x^R \) for every \( x \in \{0, 1\}^* \) is computable.

All three of these examples could easily be generalized to any alphabet \( \Sigma \) in place of \( \{0, 1\} \). We will see more examples later.

Subroutines from computable functions

It is perhaps evident, but nevertheless worth mentioning, that computable functions can easily be turned into subroutines. In particular, suppose that we have a computable function

\[ f : (\Sigma^*)^n \rightarrow \Gamma^*. \]  
(13.12)

For any choice of distinct stacks \( Y_1, \ldots, Y_n, Z \), we can easily transform a DSM that computes \( f \) into a stack machine subroutine for the instruction

\[ Z \leftarrow f(Y_1, \ldots, Y_n). \]  
(13.13)

Our interpretation of such a subroutine is that \( Z \) is overwritten with the string \( f(y_1, \ldots, y_n) \), for \( y_1, \ldots, y_n \) being the strings stored by \( Y_1, \ldots, Y_n \), and that once the subroutine is finished the stacks \( Y_1, \ldots, Y_n \) will still store their original strings \( y_1, \ldots, y_n \).

Of course it is possible to generalize this sort of construction, but this one will be sufficient for our needs.

13.3 New computable functions from old

We will end this lecture by discussing a few ways in which computable functions can be combined or modified so that new computable functions are obtained. Using these methods is often an effective way to prove that certain functions are computable without explicitly constructing DSMs that compute them.

Let us begin with three propositions that state simple facts concerning computable functions that may be considered to be straightforward.
Proposition 13.12 (Tuples of computable functions are computable). Let $\Sigma$ and $\Gamma$ be alphabets, let $n$ and $m$ be positive integers, and let

$$g_1 : (\Sigma^*)^n \rightarrow \Gamma^*$$

$$\vdots$$

$$g_m : (\Sigma^*)^n \rightarrow \Gamma^*$$

be computable functions. The function $f : (\Sigma^*)^n \rightarrow (\Gamma^*)^m$ defined as

$$f(x_1, \ldots, x_n) = (g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

for all $x_1, \ldots, x_n \in \Sigma^*$ is a computable function.

Proposition 13.13 (Projection functions are computable). Let $\Sigma$ be an alphabet, let $n$ be a positive integer, and let $k \in \{1, \ldots, n\}$. The function $\pi^k_n : (\Sigma^*)^n \rightarrow (\Sigma^*)$ defined as

$$\pi^k_n(x_1, \ldots, x_n) = x_k$$

is computable.

Proposition 13.14 (Compositions of computable functions are computable). Let $\Sigma$, $\Gamma$, and $\Lambda$ be alphabets, let $n$ and $m$ be positive integers, and let $f : (\Gamma^*)^m \rightarrow \Lambda^*$ and

$$g_1 : (\Sigma^*)^n \rightarrow \Gamma^*$$

$$\vdots$$

$$g_m : (\Sigma^*)^n \rightarrow \Gamma^*$$

be computable functions. The function $h : (\Sigma^*)^n \rightarrow \Lambda^*$ defined as

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

for all $x_1, \ldots, x_n \in \Sigma^*$ is a computable function.

Finally, let us prove the following theorem, which will allow us to conclude that certain functions obtained by recursion from computable functions are also computable. Using this theorem effectively can sometimes greatly simplify the job of proving that a given function is computable.

Theorem 13.15 (Recursions on computable functions are computable). Let $\Sigma$ be an alphabet, let $g_a : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ be a computable function for each symbol $a \in \Sigma$, and let $w \in \Sigma^*$ be a string. The function $h : \Sigma^* \rightarrow \Sigma^*$ defined as

$$h(\varepsilon) = w$$

$$h(ax) = g_a(x, h(x))$$

(for every $a \in \Sigma$ and $x \in \Sigma^*$)

is computable.
Figure 13.1: This DSM computes the function $f(x) = (x, h(x))$, where $h$ is obtained from $\{g_a : a \in \Sigma\}$ and $w$ by recursion, as defined in (13.19).

Proof. Consider the DSM $M$ described in Figure 13.1. Our goal will be to prove that $M$ computes the function $f(x) = (x, h(x))$. It is to be understood that $X$ is stack 0 and $Y$ is stack 1, so that $X$ initially stores the input string while the pair $(X, Y)$ represents the output. The stacks $Z$ and $W$ are used as workspace. Let us note explicitly that we have represented the execution of multiple subroutines as single nodes to make the figure more compact and readable. All of the subroutines used have already been discussed, with the exception of the one labeled “$Y \leftarrow w$.” As $w$ is a fixed string, this subroutine is easily implemented by simply pushing the symbols of $w$ onto $Y$ one at a time.

We will prove by induction that $M$ computes $f$. The base case is that $x = \epsilon$. An inspection of the DSM $M$ reveals that it accepts with $X$ containing $\epsilon$, $Y$ containing $w$, and $Z$ and $W$ containing $\epsilon$, which is consistent with the value $f(\epsilon) = (\epsilon, h(\epsilon))$.

Now suppose that $a \in \Sigma$ is any symbol, and assume that for a given input string $x \in \Sigma^*$, the DSM $M$ correctly computes $f(x) = (x, h(x))$. For the input $ax$, we combine this assumption with an examination of $M$ to conclude that this DSM must reach the state labeled “pop $Z$” with $X$ storing $x$, $Y$ storing $h(x)$, and $Z$ storing $a$ (as opposed to $Z$ storing $\epsilon$, as it would if the input had been $x$ rather than $ax$). The loop is iterated one more time, and we find that $X$ stores $ax$, $Y$ stores $h(ax) = g_a(x, h(x))$, and $Z$ stores $\epsilon$, from which it follows that $M$ correctly computes $f(ax) = (ax, h(ax))$, as required.
Given that \( f(x) = (x, h(x)) \) is a computable function, it follows from Propositions 13.13 and 13.14 that \( h \) is computable.

**Example 13.16.** Let \( \Sigma = \{0, 1\} \) be the binary alphabet and define two functions, \( g_0, g_1 : (\Sigma^*)^2 \to \Sigma^* \) as follows:

\[
\begin{align*}
g_0(x, y) &= 1x \\
g_1(x, y) &= 0y
\end{align*}
\]  

(13.20)

for all \( x, y \in \Sigma^* \). These are easily shown to be computable functions. Now consider the function \( h : \Sigma^* \to \Sigma^* \) obtained by recursion as follows:

\[
\begin{align*}
h(\varepsilon) &= 0 \\
h(ax) &= g_a(x, h(x)) \\
&= \begin{cases} 
1x & \text{if } a = 0 \\
0h(x) & \text{if } a = 1.
\end{cases}
\end{align*}
\]  

(13.21)

The first few values of \( h \) are as follows:

\[
\begin{align*}
h(\varepsilon) &= 0, & h(0) &= 1, & h(1) &= 00, & h(00) &= 10, \\
h(01) &= 01, & h(11) &= 11, & h(000) &= 100.
\end{align*}
\]  

(13.22)

The function \( h \) is almost a function that increments with respect to lexicographic ordering, except that left and right are reversed. The function \( \text{inc} : \Sigma^* \to \Sigma^* \) defined as

\[
\text{inc}(x) = (h(x^R))^R
\]  

(13.23)

will actually increment with respect to lexicographic order—but because it is obtained from \( h \) and the reverse function by composition (two times), and both \( h \) and the reverse function are computable, it follows that \( \text{inc} \) is also computable.

It is possible to prove more general forms of Theorem 13.15. For example, if \( h : (\Sigma^*)^{n+1} \to \Sigma^* \) is defined as

\[
\begin{align*}
h(\varepsilon, y_1, \ldots, y_n) &= f(y_1, \ldots, y_n) \\
h(ax, y_1, \ldots, y_n) &= g_a(x, h(x, y_1, \ldots, y_n), y_1, \ldots, y_n)
\end{align*}
\]

(13.24)

for every \( a \in \Sigma \) and \( x, y_1, \ldots, y_n \in \Sigma^* \), for computable functions \( f : (\Sigma^*)^n \to \Sigma^* \) and \( g_a : (\Sigma^*)^{n+2} \to \Sigma^* \) (for each \( a \in \Sigma \)), then \( h \) is computable.
Lecture 14

Turing machines and their equivalence to stack machines

In this lecture we will discuss the Turing machine model of computation. This model is named after Alan Turing (1912–1954), who proposed it in 1936. It is difficult to overstate the importance of Alan Turing’s work to this course; the subject of theoretical computer science effectively started with Turing’s work, and for this reason he is sometimes referred to as the father of theoretical computer science.

The intention of the Turing machine model is to provide a simple mathematical abstraction of general computations. This is also the intention of the stack machine model, and we will soon see that the two models are in fact equivalent—so it is simply a matter of preference to base the theory of computability on one model or the other, or even on a different model that is equivalent to Turing machines and stack machines.\(^1\) It should be made clear, however, that Turing machines do, in fact, reign supreme: in most courses similar to CS 360 taught around the world, there is no mention at all of stack machines, and it is just the Turing machine that is studied in the context of computability theory.

The idea that Turing machine computations are representative of a fully general computational model is called the Church–Turing thesis. Here is one statement of this thesis (although it is the idea rather than the exact choice of words that is important):

**Church–Turing thesis:** Any function that can be computed by a mechanical process can be computed by a Turing machine.

Note that this is not a mathematical statement that can be proved or disproved. If you wanted to try to prove a statement along these lines, the first thing you

\(^1\) One well-known example is λ-calculus, which was proposed by Alonzo Church a short time before Turing proposed the Turing machine. A sketch of a proof of the equivalence of Turing machines and λ-calculus appeared in Turing’s 1936 paper.
would most likely do is to look for a mathematical definition of what it means for a function to be “computed by a mechanical process,” and this is precisely what the Turing machine model was intended to provide.

While people have actually built machines that behave like Turing machines, this is mostly a recreational activity. The Turing machine model was never intended to serve as a practical device for performing computations, but rather was intended to provide a rigorous mathematical foundation for reasoning about computation, and it has served this purpose very well since its introduction.

14.1 Turing machine definitions

We will begin with an informal description of the Turing machine model before stating the formal definition. There are three components of a Turing machine:

1. The finite state control. This component is in one of a finite number of states at each instant, and is connected to the tape head component.

2. The tape head. This component scans one of the tape squares of the tape at each instant, and is connected to the finite state control. It can read and write symbols from/to the tape, and it can move left and right along the tape.

3. The tape. This component consists of an infinite number of tape squares, each of which can store one of a finite number of tape symbols at each instant. The tape is infinite both to the left and to the right.

Figure 14.1 illustrates these three components and the way they are connected.

The idea is that the action of a Turing machine at each instant is determined by the state of the finite state control together with just the one symbol that is
stored in the tape square that the tape head is currently reading. Thus, the action is determined by a finite number of possible alternatives: one action for each state/symbol pair. Depending on the state and the symbol being scanned, the action that the machine is to perform may involve changing the state of the finite state control, changing the symbol on the tape square being scanned, and moving the tape head to the left or right. Once this action is performed, the machine will again have some state for its finite state control and will be reading some symbol on its tape, and the process continues. Just like the stack machine model, one may consider both deterministic and nondeterministic variants of the Turing machine model.

When a Turing machine begins a computation, an input string is written on its tape, and every other tape square is initialized to a special blank symbol (which may not be included in the input alphabet). Naturally, we need an actual symbol to represent the blank symbol in these notes, and we will use the symbol $\$\$ for this purpose. More generally, we will allow the possible symbols written on the tape to include other non-input symbols in addition to the blank symbol, as it is sometimes convenient to allow this possibility.

Similar to stack machines, we require that Turing machines have two special states: an accept state $q_{\text{acc}}$ and a reject state $q_{\text{rej}}$. If the machine enters one of these two states, the computation immediately stops and accepts or rejects accordingly. When we discuss language recognition, our focus is on whether or not a given Turing machine eventually reaches one of the states $q_{\text{acc}}$ or $q_{\text{rej}}$, but we can also use the Turing machine model to discuss function computations by taking into account the contents of the tape after the accept state (let us say) has been reached.

Formal definition of DTMs

With the informal description of Turing machines from above in mind, we will now proceed to the formal definition.

**Definition 14.1.** A deterministic Turing machine (or DTM, for short) is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \quad (14.1)$$

where $Q$ is a finite and nonempty set of states; $\Sigma$ is an alphabet called the input alphabet, which may not include the blank symbol $\$\$; $\Gamma$ is an alphabet called the tape alphabet, which must satisfy $\Sigma \cup \{\$\$\} \subseteq \Gamma$; $\delta$ is a transition function having the form

$$\delta : Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}; \quad (14.2)$$

$q_0 \in Q$ is the initial state; and $q_{\text{acc}}, q_{\text{rej}} \in Q$ are the accept and reject states, which satisfy $q_{\text{acc}} \neq q_{\text{rej}}$. 

149
The interpretation of the transition function is as follows. Suppose the DTM is currently in a state \( p \in Q \), the symbol stored in the tape square being scanned by the tape head is \( a \in \Gamma \), and it is the case that \( \delta(p, a) = (q, b, D) \) for \( D \in \{\leftarrow, \rightarrow\} \). The action performed by the DTM is then to (i) change state to \( q \), (ii) overwrite the contents of the tape square being scanned by the tape head with \( b \), and (iii) move the tape head in direction \( D \) (either left or right). In the case that the state is \( q_{\text{acc}} \) or \( q_{\text{rej}} \), the transition function does not specify an action, because we assume that the DTM halts once it reaches one of these two states.

**Turing machine computations**

If we have a DTM \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \), and we wish to consider its operation on some input string \( w \in \Sigma^* \), we assume that it is started with its components initialized as illustrated in Figure 14.2. That is, the input string is written on the tape, one symbol per square, with each other tape square containing the blank symbol, and with the tape head scanning the tape square immediately to the left of the first input symbol. (In the case that the input string is \( \varepsilon \), all of the tape squares start out storing blanks.)

Once the initial arrangement of the DTM is set up, the DTM begins taking steps, as determined by the transition function \( \delta \) in the manner suggested above. So long as the DTM does not enter one of the two states \( q_{\text{acc}} \) or \( q_{\text{rej}} \), the computation continues. If the DTM eventually enters the state \( q_{\text{acc}} \), it accepts the input string, and if it eventually enters the state \( q_{\text{rej}} \), it rejects the input string. Thus, there are three possible alternatives for a DTM \( M \) on a given input string \( w \):

1. \( M \) accepts \( w \).
2. \( M \) rejects \( w \).
3. \( M \) runs forever on input \( w \).

In some cases one can design a particular DTM so that the third alternative does not occur, but in general it might.
Representing configurations of DTMs

In order to speak more precisely about Turing machines and state a formal definition concerning their behavior, we will require a bit more terminology. When we speak of a configuration of a DTM, we are speaking of a description of all of the Turing machine’s components at some instant. This includes

1. the state of the finite state control,
2. the contents of the entire tape, and
3. the tape head position on the tape.

Rather than drawing pictures depicting the different parts of Turing machines, like in Figure 14.2, we use the following compact notation to represent configurations. If we have a DTM \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \), and we wish to refer to a configuration of this DTM, we express it in the form

\[ u(q, a)v \quad (14.3) \]

for some state \( q \in Q \), a tape symbol \( a \in \Gamma \), and (possibly empty) strings of tape symbols \( u \) and \( v \) such that

\[ u \in \Gamma^*\setminus\{\omega\}\Gamma^* \quad \text{and} \quad v \in \Gamma^*\setminus\{\omega\} \quad (14.4) \]

(i.e., \( u \) and \( v \) are strings of tape symbols such that \( u \) does not start with a blank and \( v \) does not end with a blank). What the expression (14.3) means is that the string \( uav \) is written on the tape in consecutive squares, with all other tape squares containing the blank symbol; the state of \( M \) is \( q \); and the tape head of \( M \) is positioned over the symbol \( a \) that occurs between \( u \) and \( v \).

For example, the configuration of the DTM in Figure 14.1 is expressed as

\[ 0\$1(q_4, 0)0\# \quad (14.5) \]

while the configuration of the DTM in Figure 14.2 is

\[ (q_0, \omega)w \quad (14.6) \]

(for \( w = a_1 \cdots a_n \)).

When working with descriptions of configurations, it is convenient to define a few functions as follows. We define \( \alpha : \Gamma^* \to \Gamma^*\setminus\{\omega\}\Gamma^* \) and \( \beta : \Gamma^* \to \Gamma^*\setminus\{\omega\} \Gamma^* \) recursively as

\[ \alpha(w) = w \quad \text{(for } w \in \Gamma^*\setminus\{\omega\}\Gamma^*) \]
\[ \alpha(\omega w) = \alpha(w) \quad \text{(for } w \in \Gamma^*) \quad (14.7) \]
and

\[ \beta(w) = w \quad \text{(for } w \in \Gamma^* \backslash \Gamma^* \{ \omega \}) \]
\[ \beta(w \omega) = \beta(w) \quad \text{(for } w \in \Gamma^*) \]

and we define

\[ \gamma : \Gamma^*(Q \times \Gamma)^* \to (\Gamma^* \backslash \{ \omega \}) (Q \times \Gamma) (\Gamma^* \backslash \{ \omega \}) \]

as

\[ \gamma(u(q,a)\nu) = \alpha(u)(q,a)\beta(\nu) \]

for all \( u, \nu \in \Gamma^* \), \( q \in Q \), and \( a \in \Gamma \). This is not as complicated as it might appear: the function \( \gamma \) just throws away all blank symbols on the left-most end of \( u \) and the right-most end of \( \nu \), so that a proper expression of a configuration remains.

**A yields relation for DTMs**

Now we will define a *yields* relation, in a similar way to what we did for context-free grammars and stack machines. This will in turn allow us to formally define acceptance and rejection for DTMs.

**Definition 14.2.** Let \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}) \) be a DTM. We define a *yields* relation \( \vdash_M \) on pairs of expressions representing configurations as follows:

1. For every choice of \( p \in Q \backslash \{ q_{acc}, q_{rej} \} \), \( q \in Q \), and \( a, b \in \Gamma \) satisfying

\[ \delta(p,a) = (q, b, \rightarrow), \]

the yields relation includes these pairs for all \( u \in \Gamma^* \backslash \{ \omega \} \), \( v \in \Gamma^* \backslash \{ \omega \} \), and \( c \in \Gamma \):

\[ u(p,a)cv \vdash_M \gamma(ub(q,c)\nu) \]
\[ u(p,a) \vdash_M \gamma(ub(q,\omega)) \]  

(14.12)

2. For every choice of \( p \in Q \backslash \{ q_{acc}, q_{rej} \} \), \( q \in Q \), and \( a, b \in \Gamma \) satisfying

\[ \delta(p,a) = (q, b, \leftarrow), \]

the yields relation includes these pairs for all \( u \in \Gamma^* \backslash \{ \omega \} \), \( v \in \Gamma^* \backslash \{ \omega \} \), and \( c \in \Gamma \):

\[ uc(p,a)\nu \vdash_M \gamma(u(q,c)b\nu) \]
\[ (p,a)\nu \vdash_M \gamma((q, \omega)b\nu) \]  

(14.14)
In addition, we let \( \vdash_\star M \) denote the reflexive, transitive closure of \( \vdash M \). That is, we have

\[
\vdash_\star M \text{ if and only if there exists an integer } m \geq 1, \text{ strings } w_1, \ldots, w_m, x_1, \ldots, x_m \in \Gamma^*, \text{ symbols } c_1, \ldots, c_m \in \Gamma, \text{ and states } r_1, \ldots, r_m \in Q \text{ such that }
\]

\[
u(p, a)\vdash_M y(q, b)z \tag{14.15}
\]

\[
w_k(r_k, c_k)x_k \vdash_M w_{k+1}(r_{k+1}, c_{k+1})x_{k+1} \tag{14.16}
\]

for all \( k \in \{1, \ldots, m - 1\} \).

A somewhat more intuitive explanation of this definition is as follows. Whenever we have

\[
u(p, a)\vdash_M y(q, b)z \tag{14.17}
\]

it means that by running \( M \) for one step we move from the configuration represented by \( u(p, a) \) to the configuration represented by \( y(q, b) \); and whenever we have

\[
u(p, a)\vdash_\star M y(q, b)z \tag{14.18}
\]

it means that by running \( M \) for some number of steps, possibly zero steps, we will move from the configuration represented by \( u(p, a) \) to the configuration represented by \( y(q, b) \).

### Acceptance and rejection for DTMs

Finally, we can write down a definition for acceptance and rejection by a DTM, using the relation \( \vdash_\star M \) we just defined.

**Definition 14.3.** Let \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \) be a DTM and let \( w \in \Sigma^* \) be a string. If there exist strings \( u, v \in \Gamma^* \) and a symbol \( a \in \Gamma \) such that

\[
(q_0, \omega)w \vdash_\star M u(q_{\text{acc}}, a)v, \tag{14.19}
\]

then \( M \) accepts \( w \). If there exist strings \( u, v \in \Gamma^* \) and a symbol \( a \in \Gamma \) such that

\[
(q_0, \omega)w \vdash_\star M u(q_{\text{rej}}, a)v, \tag{14.20}
\]

then \( M \) rejects \( w \). If neither of these conditions hold, then \( M \) runs forever on input \( w \).

In words, if a DTM is set in its initial configuration, for some input string \( w \), and starts running, it accepts \( w \) if it eventually enters its accept state, it rejects \( w \) if it eventually enters its reject state, and it run forever if neither of these possibilities holds.
Similar to what we have done for other computational models, we write $L(M)$ to denote the language of all strings that are accepted by a DTM $M$. As for stack machines, the language $L(M)$ doesn’t really tell the whole story, because a string $w \not\in L(M)$ might either be rejected or it may cause $M$ to run forever, but the notation is useful nevertheless.

**A simple example of a Turing machine**

Let us now see an example of a DTM, which we will describe using a state diagram. In the DTM case, we represent the property that the transition function satisfies $\delta(p, a) = (q, b, \rightarrow)$ with a transition of the form

$$
\begin{align*}
\text{p} & \quad a, b \rightarrow \quad \text{q}
\end{align*}
$$

and similarly we represent the property that $\delta(p, a) = (q, b, \leftarrow)$ with a transition of the form

$$
\begin{align*}
\text{p} & \quad a, b \leftarrow \quad \text{q}
\end{align*}
$$

Figure 14.3: A DTM $M$ for the language $\{0^n1^n : n \in \mathbb{N}\}$. 

The state diagram for the example is given in Figure 14.3. The DTM $M$ described by this diagram is for the language

$$A = \{0^n1^n : n \in \mathbb{N}\}. \quad (14.21)$$

To be more precise, $M$ accepts every string in $A$ and rejects every string in $\overline{A}$.

The specific way that the DTM $M$ works can be summarized as follows. The DTM $M$ starts out with its tape head scanning the blank symbol immediately to the left of its input. It moves the tape head right, and if it sees a 1 it rejects: the input string must not be of the form $0^n1^n$ if this happens. On the other hand, if it sees another blank symbol, it accepts: the input must be the empty string, which corresponds to the $n = 0$ case in the description of $A$. Otherwise, it must have seen the symbol 0, and in this case the 0 is erased (meaning that it replaces it with the blank symbol), the tape head repeatedly moves right until a blank is found, and then it moves one square back to the left. If a 1 is not found at this location the DTM rejects: there weren’t enough 1s at the right end of the string. Otherwise, if a 1 is found, it is erased, and the tape head moves all the way back to the left, where we essentially recurse on a slightly shorter string.

Of course, the summary just suggested doesn’t tell you precisely how the DTM works—but if you didn’t already have the state diagram from Figure 14.3, the summary would probably be enough to give you a good idea for how to come up with the state diagram (or perhaps a slightly different one operating in a similar way).

In fact, an even higher-level summary would probably be enough. For instance, we could describe the functioning of the DTM $M$ as follows:

1. Accept if the input is the empty string.
2. Check that the left-most non-blank symbol on the tape is a 0 and that the right-most non-blank symbol is a 1. Reject if this is not the case, and otherwise erase these symbols and goto 1.

There will, of course, be several specific ways to implement this algorithm with a DTM, with the DTM $M$ from Figure 14.3 being one of them.

Because state diagrams for DTMs tend to be complicated (and often completely incomprehensible) for all but the simplest of DTMs, it is very common that DTMs are described in a high level way, as in the last description above.
14.2 Equivalence of DTMs and DSMs

We will now argue that deterministic Turing machines and deterministic stack machines are equivalent computational models. This will require that we establish two separate facts:

1. Given a DTM $M$, there exists a DSM $K$ that simulates $M$.
2. Given a DSM $M$, there exists a DTM $K$ that simulates $M$.

Here we have used the term simulate, as we quite frequently will throughout the remainder of the course. It refers to the situation in which one machine (the simulator) mimics another machine (which we’ll call the original machine, for lack of a better name). Note that this does not necessarily mean that one step of the original machine corresponds to a single step of the simulator: the simulator might require many steps to simulate one step of the original machine. A consequence of both facts listed above is that, for every input string $w$, $K$ accepts $w$ whenever $M$ accepts $w$, $K$ rejects $w$ whenever $M$ rejects $w$, and $K$ runs forever on $w$ whenever $M$ runs forever on $w$.

The two simulations are described in the subsections that follow. These descriptions are not intended to be formal proofs, but they should provide enough information to convince you that the two models are indeed equivalent.

Simulating a DTM with a DSM

First we will discuss how a DSM can simulate a DTM. To simulate a given a DTM $M$, we will define a DSM $K$ having two stacks, called $L$ and $R$ (for “left” and “right,” respectively). The stack $L$ will represent the contents of the tape of $M$ to the left of the tape head (in reverse order, so that the topmost symbol of $L$ is the symbol immediately to the left of the tape head of $M$) while $R$ will represent the contents of the tape of $M$ to the right of the tape head. The symbol in the tape square of $M$ that is being scanned by its tape head will be stored in the internal state of $K$, so this symbol does not need to be stored on either stack. Our main task will be to define $K$ so that it pushes and pops symbols to and from $L$ and $R$ in a way that mimics the behavior of $M$.

To be more precise, suppose that $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ is the DTM to be simulated. The DSM $K$ will require a collection of states for every state/symbol pair $(p, a) \in Q \times \Gamma$. Figure 14.4 illustrates these collections of states in the case that $p$ is a non-halting state. If it is the case that $\delta(p, a) = (q, b, \leftarrow)$, then the states and transitions on the left-hand side of Figure 14.4 mimic the actions of $M$ in this way:
Lecture 14

Figure 14.4: For each state/symbol pair \((p, a)\) ∈ \((Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma\) of \(M\), there are two possibilities: if \(\delta(p, a) = (q, b, \leftarrow)\), then \(K\) includes the states and transitions in the left-hand diagram, and if \(\delta(p, a) = (q, b, \rightarrow)\), then \(K\) includes the states and transitions in the right-hand diagram.

1. The symbol \(b\) gets written to the tape of \(M\) and the tape head moves left, so \(K\) pushes the symbol \(b\) onto \(R\) to record the fact that the symbol \(b\) is now to the right of the tape head of \(M\).

2. The symbol that was one square to the left of the tape head of \(M\) becomes the symbol that \(M\) scans because the tape head moved left, so \(K\) pops a symbol off of \(L\) in order to learn what this symbol is and stores it in its finite state memory. In case \(K\) pops the bottom-of-the-stack marker, it pushes this symbol back, pushes a blank, and tries again; this has the effect of inserting extra blank symbols as \(M\) moves to previously unvisited tape squares.

3. As \(K\) pops the top symbol off of \(L\), as described in the previous item, it transitions to the new state \((q, c)\), for whatever symbol \(c\) it popped. This sets up \(K\) to simulate the next step of \(M\).

The situation is analogous in the case \(\delta(p, a) = (q, b, \rightarrow)\), with left and right (and the stacks \(L\) and \(R\)) swapped.

For each pair \((p, a)\) where \(p \in \{q_{\text{acc}}, q_{\text{rej}}\}\), there is no next-step of \(M\) to simulate, so \(K\) simply transitions to its accept or reject state accordingly, as illustrated in Figure 14.5. Note that if we only care about whether \(M\) accepts or rejects, as opposed to what is left on its tape in case it halts, we could alternatively eliminate all states
Figure 14.5: For each state/symbol pair $(p, a) \in \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma$ of $M$, the DSM $K$ simply transitions to its accept or reject state accordingly. The symbol $a$ stored in the finite state memory of $K$ is pushed onto the stack $L$ as this transition is followed. (The choice to push this symbol onto $L$ rather than $R$ is more or less arbitrary, and this operation is not important if one is only interested in whether $M$ accepts, rejects, or runs forever; this operation only has relevance when the contents of the tape of $M$ when it halts are of interest.)

of the form $(q_{\text{acc}}, a)$ and $(q_{\text{rej}}, a)$, and replace transitions to these eliminated states with transitions to the accept or reject state of $K$.

The start state of $K$ is the state $(q_0, \omega)$ and it is to be understood that stack $R$ is stack 0 (and therefore contains the input along with the bottom-of-the-stack marker) while $L$ is stack 1. The initial state of $K$ therefore represents the initial state of $M$, where the tape head scans a blank symbol and the input is written in the tape squares to the right of this blank tape square.

**Simulating a DSM with a DTM**

Now we will explain how a DSM can be simulated by a DTM. The idea behind this simulation is fairly straightforward: the DTM will use its tape to store the contents of all of the stacks of the DSM it is simulating, and it will update this information appropriately so as to mimic the DSM. This will require many steps in general, as the DTM will have to scan back and forth on its tape to manipulate the information representing the stacks of the original DSM.

In greater detail, suppose that $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is an $r$-DSM. The DTM $K$ that we will define to simulate $M$ will have a large tape alphabet:

$$\Gamma = (\Delta \cup \{\#, \omega\})^r.$$  \hspace{1cm} (14.22)

Here, we assume that $#$ is a special symbol that is not contained in the stack alphabet $\Delta$ of $M$, and that the blank symbol $\omega$ is also not contained in $\Delta$. A helpful way to think about a DTM whose tape alphabet is a Cartesian product like this is to imagine that its tape is divided into *tracks*, as Figure 14.6 suggests. Note that we
Figure 14.6: An example of a DTM whose tape has 6 tracks, each representing a stack. This figure is consistent with the stack alphabet of the DSM that is being simulated being $\Delta = \{0, 1, \Diamond\}$; the situation pictured is that the DSM stacks $0$ through $5$ store the strings $01100, \varepsilon, 011100, 00010, 1,$ and $\varepsilon$, respectively.

are not modifying the definition of a DTM at all when we speak of separate tracks on a tape like this—it's just a way of thinking about the tape alphabet $\Gamma$. It is to be understood that the true blank symbol of $K$ is the symbol $(\varepsilon, \ldots, \varepsilon)$, and that an input string $a_1 \cdots a_n \in \Sigma^*$ of $M$ is to be identified with the string of tape symbols

$$(a_1, \varepsilon, \ldots, \varepsilon) \cdots (a_n, \varepsilon, \ldots, \varepsilon) \in \Gamma^*.$$ (14.23)

The purpose of the symbol # is to mark a position on the tape of $K$; the contents of the stacks of $M$ will always be to the left of these # symbols. The first thing that $K$ does, before any steps of $M$ are simulated, is to scan the tape (from left to right) to find the end of the input string. In the first tape square after the input string, it places the bottom-of-the-stack marker $\Diamond$ in every track, and in the next square to the right it places the # symbol in every track. Once these # symbols are written to the tape, they will remain there for the duration of the simulation. The DTM then moves its tape head to the left, so that it is positioned over the # symbols, and begins simulating steps of the DSM $M$.

The DTM $K$ will store the current state of $M$ in its internal memory, and one way to think about this is to imagine that $K$ has a collection of states for every state $q \in Q$ of $M$ (which is similar to the simulation in the previous subsection, except there we had a collection of states for every state/symbol pair rather than just for each state). The DTM $K$ is defined so that this state will initially be set to $q_0$ (the start state of $M$) when it begins the simulation.
There are two possibilities for each non-halting state $q \in Q$ of $M$: it is either a push state or a pop state. In either case, there is a stack index $k$ that is associated with this state. The behavior of $K$ is as follows for these two possibilities:

1. If $q$ is a push state, then there must be a symbol $a \in \Gamma$ that is to be pushed onto stack $k$. The DTM $K$ scans left until it finds a blank symbol on track $k$, overwrites this blank with the symbol $a$, and changes the state of $M$ stored in its internal memory exactly as $M$ does.

2. If $q$ is a pop state, then $K$ needs to find out what symbol is on the top of stack $k$. It scans left to find a blank symbol on track $k$, moves right to find the symbol on the top of stack $k$, changes the state of $M$ stored in its internal memory accordingly, and overwrites this symbol with a blank. Naturally, in the situation where $M$ attempts to pop an empty stack, $K$ will detect this (as there will be no non-blank symbols to the left of the # symbols), and it immediately transitions to its reject state.

In both cases, after the push or pop operation was simulated, $K$ scans its tape head back to the right to find the # symbols, so that it can simulate another step of $M$.

Finally, if $K$ stores a halting state of $M$ when it would otherwise begin simulating a step of $M$, it accepts or rejects accordingly. In a situation in which the contents of the tape of $K$ after the simulation are important, such as when $M$ computes a function rather than simply accepting or rejecting, one may of course define $K$ so that it first removes the # symbols and $\Diamond$ symbols from its tape prior to accepting or rejecting.
Lecture 15

Encodings; examples of decidable languages

In this lecture we will discuss some examples of decidable languages that are based on finite automata and context-free grammars. These languages will have a somewhat different character from most of the languages we discussed previously in the course; their definitions are centered on fundamental mathematical concepts, as opposed to simple syntactic patterns.

Before doing this, however, we will discuss the notion of encodings, which allow us to represent complicated mathematical objects using strings over a given alphabet. We will make extensive use of this notion for the remainder of the course.

15.1 Encodings of interesting mathematical objects

As we continue to discuss stack machines and computability, it will be necessary for us to consider ways that interesting mathematical objects can be represented as strings. For example, we may wish to consider a DSM that takes as input a number, a graph, a DFA, a CFG, another DSM (maybe even a description of itself), or a list of objects of multiple types.

Of course, the idea that we can encode different sorts of objects as strings will be familiar to students of computer science, and for this reason we will not belabor this issue—but it will nevertheless be helpful to establish a few conventions and introduce useful ideas concerning the encoding of different objects of interest.

Encoding multiple strings into one

Let us begin by considering the following task that concerns two hypothetical individuals: Alice and Bob. Alice has two binary strings $x \in \{0, 1\}^*$ and $y \in \{0, 1\}^*$,
and she would like to communicate these two strings to Bob. However, Alice is only allowed to transmit a single binary string to Bob, so somehow $x$ and $y$ must be packed into a single string $z \in \{0, 1\}^*$ from which Bob can recover both $x$ and $y$. The two of them may agree ahead of time on a method through which this will be done, but naturally the method must be agreed upon prior to Alice knowing which strings $x$ and $y$ are to be communicated. That is, the method must work for an arbitrary choice of binary strings $x$ and $y$. There are different methods through which this task may be accomplished, but let us just describe one method.

The first step is to introduce a new symbol, which we will call #. We then choose to encode the pair $(x, y)$ into a single string $x#y \in \{0, 1, \#\}^*$. Obviously, if Alice were to send this string to Bob, he could recover $x$ and $y$ without difficulty, so it is a good method in that sense—but unfortunately it does not solve the original problem because it makes use of the alphabet $\{0, 1, \#\}$ rather than $\{0, 1\}$.

The second step of the method will take us back to the binary alphabet: we can encode the string $x#y$ as a binary string by substituting the individual symbols according to this pattern:

\[
\begin{align*}
0 & \rightarrow 00 \\
1 & \rightarrow 01 \\
\# & \rightarrow 1
\end{align*}
\] 

(15.1)

The resulting binary string will be the encoding of the two strings $x$ and $y$ that Alice sends to Bob.

For example, if the two strings are $x = 0110$ and $y = 01111$, we first consider the string $0110#01111$, and then perform the substitution suggested above to obtain

\[
\langle 0110, 01111 \rangle = 001010010001010101.
\] 

(15.2)

Here we have used a notation that we will use repeatedly throughout much of the remainder of the course: whenever we have some object $X$, along with an encoding scheme that encodes a class of objects that includes $X$ as strings, we write $\langle X \rangle$ to denote the string that encodes $X$. In the equation above, the notation $\langle 0110, 01111 \rangle$ therefore refers to the encoding of the two strings 0110 and 01111, viewed as an ordered pair.

Let us make a few observations about the encoding scheme just described:

1. The scheme works not only for two strings $x$ and $y$, but for any finite number of binary strings $x_1, \ldots, x_n$; such a list of strings may be encoded by first forming

---

1 There are other patterns that would work equally well. The one we have selected is an example of a **prefix-free code**; because none of the strings appearing on the right-hand side of (15.1) is a prefix of any of the other strings, we are guaranteed that by concatenating together a sequence of these strings we can recover the original string without ambiguity.
the string \( x_1 \# x_2 \# \cdots \# x_n \in \{0, 1, \#\}^* \), and then performing the substitutions described above to obtain \( \langle x_1, \ldots, x_n \rangle \in \{0, 1\}^* \).

2. Every \( n \)-tuple \((x_1, \ldots, x_n)\) of binary strings has a unique encoding \( \langle x_1, \ldots, x_n \rangle \), but it is not the case that every binary string encodes an \( n \)-tuple of binary strings. In other words, the encoding is one-to-one but not onto. For instance, the string 10 does not decode to any string over the alphabet \( \{0, 1, \#\} \), and therefore does not encode an \( n \)-tuple of binary strings. This is not a problem; most of the encoding schemes we will consider in this course have the same property that not every string is a valid encoding of some object of interest.

3. You could easily generalize this scheme to larger alphabets by adding a new symbol to mark the division between strings over the original alphabet, and then choosing a suitable encoding in place of (15.1).

This one method of encoding multiple strings into one turns out to be incredibly useful, and by using it repeatedly we can devise encoding schemes for highly complex mathematical objects.

**Encoding strings over arbitrary alphabets using a fixed alphabet**

Next, let us consider the task of encoding a string over an alphabet \( \Gamma \) whose size we do not know ahead of time by a string over a fixed alphabet \( \Sigma \). In the interest of simplicity, let us take \( \Sigma = \{0, 1\} \) to be the binary alphabet. (We will also consider the unary alphabet \( \Sigma = \{0\} \) toward the end of this subsection.)

Before we discuss a particular scheme through which this task can be performed, let us take a moment to clarify the task at hand. In particular, it should be made clear that we are not looking for a way to encode strings over any possible alphabet \( \Gamma \) that you could ever imagine. For instance, consider the alphabet

\[
\Gamma = \{\text{\textcircled{a}}, \text{\textcircled{b}}, \text{\textcircled{c}}, \text{\textcircled{d}}\},
\]

which was mentioned in the very first lecture of the course. Some might consider this to be an interesting alphabet, but in some sense there is nothing special about it—all that is really relevant about it from the viewpoint of the theory of computing is that it has four symbols, so there is little point in differentiating it from the alphabet \( \Gamma = \{0, 1, 2, 3\} \). That is, when we think about models of computation, all that really matters is the number of symbols in our alphabet, and sometimes the order we choose to put them in, but not the size, shape, or color of the symbols.

\[\text{[2]} \text{You could of course consider encoding schemes that represent the shapes and sizes of different alphabet symbols—the symbols appearing in (15.3), for example, are actually the result of a binary string encoding obtained from an image compression algorithm—but this is not what we’re talking about in this lecture.}\]
With this understanding in place, we will make the assumption that our encoding task is to be performed for an alphabet of the form

\[ \Gamma = \{0, 1, \ldots, n - 1\} \tag{15.4} \]

for some positive integer \( n \), where we are imagining that each integer between 0 and \( n - 1 \) is a single symbol. By the way, if a situation does arise where it is confusing to think of the integers \( 0, \ldots, n - 1 \) as individual alphabet symbols, we may instead name the symbols of \( \Gamma \) as \( \tau_0, \ldots, \tau_{n-1} \) rather than \( 0, \ldots, n - 1 \).

The method from the previous subsection provides a simple means through which the task at hand can be accomplished. First, for every nonnegative integer \( k \in \mathbb{N} \), let us decide that the encoding \( \langle k \rangle \) of this number is given by its representation using binary notation:

\[ \langle 0 \rangle = 0, \quad \langle 1 \rangle = 1, \quad \langle 2 \rangle = 10, \quad \langle 3 \rangle = 11, \quad \langle 4 \rangle = 100, \quad \text{etc.} \tag{15.5} \]

Then, to encode a given string \( k_1 k_2 \cdots k_m \), we simply encode the \( m \) binary strings \( \langle k_1 \rangle, \langle k_2 \rangle, \ldots, \langle k_m \rangle \) into a single binary string

\[ \langle \langle k_1 \rangle, \langle k_2 \rangle, \ldots, \langle k_m \rangle \rangle \tag{15.6} \]

using the method from the previous subsection.

For example, let us consider the string 001217429, which we might assume is over the alphabet \( \{0, \ldots, 9\} \) (although this assumption will not influence the encoding that is obtained). The method from the previous subsection suggests that we first form the string

\[ 0\#0\#1\#10\#1\#111\#100\#10\#1001 \tag{15.7} \]

and then encode this string using the substitutions (15.1). The binary string we obtain is

\[ \langle 001217429 \rangle = 00100101100101101010100000100100001. \tag{15.8} \]

Finally, let us briefly discuss the possibility that the alphabet \( \Sigma \) is the unary alphabet \( \Sigma = \{0\} \). You can still encode strings over any alphabet \( \Gamma = \{0, \ldots, n - 1\} \) using this alphabet, although (not surprisingly) it will be extremely inefficient. One way to do this is to first encode strings over \( \Gamma \) as strings over the binary alphabet, exactly as discussed above, and then to encode binary strings as unary strings with
Lecture 15

respect to the lexicographic ordering:

\[ \begin{align*}
\epsilon & \rightarrow \epsilon \\
0 & \rightarrow 0, \\
1 & \rightarrow 00, \\
10 & \rightarrow 000, \\
11 & \rightarrow 0000, \\
100 & \rightarrow 00000,
\end{align*} \] (15.9)

and so on.

This means that you could, in principle, encode an entire book in unary. Think of an ordinary book as a string over the alphabet that includes upper- and lowercase letters, spaces, and punctuation marks, and imagine encoding this string over the unary alphabet as just described. You open the unary-encoded book and see that every page is filled with 0s, and as you’re reading the book you have absolutely no idea what it’s about. All you can do is to eliminate the possibility that the book corresponds to a shorter string of 0s than the number you’ve seen so far, just like when you rule out the possibility that it’s 3 o’clock when the bells at City Hall ring four times. Finally you finish the book and in an instant it all becomes clear, and you say “Wow, what a great book!”

Numbers, vectors, and matrices

We already used the elementary fact that nonnegative integers can be encoded as binary strings using binary notation in the previous subsection. One can also encode arbitrary integers using binary notation by interpreting the first bit of the encoding to be a sign bit. Rational numbers can be encoded as pairs of integers (representing the numerator and denominator), by first expressing the individual integers in binary, and then encoding the two strings into one using the method from earlier in the lecture. One could also consider floating point representations, which are of course very common in practice, but also have the disadvantage that they only represent rational numbers for which the denominator is a power of two.

With a method for encoding numbers as binary strings in mind, one can represent vectors by simply encoding the entries as strings, and then encoding these multiple strings into a single string using the method described at the start of the lecture. Matrices can be represented as lists of vectors. Indeed, once you know how to encode lists of strings as strings, you can very easily devise encoding schemes for highly complex mathematical objects.
An encoding scheme for NFAs and DFAs

Now let us devise an encoding scheme for NFAs and DFAs. We’ll start with DFAs, and once we’re finished we will observe how the scheme can be easily modified to obtain an encoding scheme for NFAs.

What we are aiming for is a way to encode every possible DFA

\[ M = (Q, \Gamma, \delta, q_0, F) \]  

as a binary string \( \langle M \rangle \). Intuitively speaking, given the binary string \( \langle M \rangle \in \Sigma^* \), it should be possible to recover a description of exactly how \( M \) operates without difficulty. There are, of course, many possible encoding schemes that one could devise—we’re just choosing one that works but otherwise isn’t particularly special.

Along similar lines to the discussion above concerning the encoding of strings over arbitrary alphabets, we will make the assumption that the alphabet \( \Gamma \) of \( M \) takes the form

\[ \Gamma = \{0, \ldots, n - 1\} \]  

for some positive integer \( n \). For the same reasons, we will assume that the state set of \( M \) takes the form

\[ Q = \{q_0, \ldots, q_{m-1}\} \]  

for some positive integer \( m \).

There will be three parts of the encoding:

1. A positive integer \( n \) representing \( |\Gamma| \). This number will be represented using binary notation.

2. A specification of the set \( F \) together with the number of states \( m \). These two things together can be described by a binary string \( s \) of length \( m \). Specifically, the string

\[ s = b_0 b_1 \cdots b_{m-1} \]  

specifies that

\[ F = \{q_k : k \in \{0, \ldots, m-1\}, b_k = 1\}, \]  

and of course the number of states \( m \) is given by the length of the string \( s \). Hereafter we will write \( \langle F \rangle \) to refer to the encoding of the subset \( F \) that is obtained in this way.

---

3 Notice that we’re taking the alphabet of \( M \) to be \( \Gamma \) rather than \( \Sigma \) to be consistent with the conventions used in the previous subsections: \( \Gamma \) is an alphabet having an arbitrary size, and we cannot assume it is fixed as we devise our encoding scheme, while \( \Sigma = \{0, 1\} \) is the alphabet we’re using for the encoding.
3. The transition function $\delta$ will be described by listing all of the inputs and outputs of this function, in the following way. First, for $j, k \in \{0, \ldots, m - 1\}$ and $a \in \Gamma = \{0, \ldots, n - 1\}$, the string
\[
\langle \langle j \rangle, \langle a \rangle, \langle k \rangle \rangle
\]
(15.15)
specifies that
\[
\delta(q_j, a) = q_k.
\]
(15.16)
Here, $\langle j \rangle$, $\langle k \rangle$, and $\langle a \rangle$ refer to the strings obtained from binary notation, which makes sense because $j$, $k$, and $a$ are all nonnegative integers. We then encode the list of all of these strings, in the natural ordering that comes from iterating over all pairs $(j, a)$, into a single string $\langle \delta \rangle$.

For example, the DFA depicted in Figure 15.1 has a transition function $\delta$ whose encoding is
\[
\langle \delta \rangle = \langle \langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 1 \rangle, \langle 1, 1, 0 \rangle \rangle.
\]
(15.17)
(We’ll leave it in this form rather than expanding it out as a binary string in the interest of clarity.)

Finally, the encoding of a given NFA $M$ is just a list of the three parts just described:
\[
\langle M \rangle = \langle \langle n \rangle, \langle F \rangle, \langle \delta \rangle \rangle.
\]
(15.18)

This encoding scheme can easily be modified to obtain an encoding scheme for NFAs. This time, the values the transition function takes are subsets of $Q$ rather than elements of $Q$, and we must also account for the possibility of $\varepsilon$-transitions. Fortunately, we already know how to encode subsets of $Q$; we did this for the set $F$, and exactly the same method can be used to specify any of the values $\delta(q_j, a)$. That is, the string
\[
\langle \langle j \rangle, \langle a \rangle, \langle \delta(q_j, a) \rangle \rangle
\]
(15.19)
describes the value of the transition function for the pair \((q_j, a)\). To specify the \(\varepsilon\)-transitions of \(M\), we may use the string

\[
\langle \langle j \rangle, \varepsilon, \delta(q_j, \varepsilon) \rangle, \quad (15.20)
\]

which takes advantage of the fact that we never have \(\langle a \rangle = \varepsilon\) for any symbol \(a \in \Gamma\). As before, we simply list all of the strings corresponding to the different inputs of \(\delta\) in order to encode \(\delta\).

**Encoding schemes for regular expressions, CFGs, PDAs, etc.**

We could continue on and devise encoding schemes through which regular expressions, CFGs, PDAs, stack machines, and Turing machines can be specified. Because of its importance, we will in fact return to the case of DSMs in the next lecture, but for the others I will leave it to you to think about how you might design encoding schemes. There are countless specific ways to do this, but it turns out that the specifics aren’t really all that important—the reason why we did this carefully DFAs and NFAs is to illustrate how it can be done for those models, with the principal aim being to clarify the concept rather than to create an encoding scheme we will actually use in an operational sense.

**15.2 Decidability of formal language problems**

Now let us turn our attention toward languages that concern the models of computation we have studied previously in the course.

**Languages based on DFAs, NFAs, and regular expressions**

The first language we will consider is this one:

\[
A_{\text{DFA}} = \{ \langle \langle D \rangle, \langle w \rangle \rangle : D \text{ is a DFA and } w \in L(D) \}. \quad (15.21)
\]

Here we assume that \(\langle D \rangle\) is the encoding of a given DFA \(D\), \(\langle w \rangle\) is the encoding of a given string \(w\), and \(\langle \langle D \rangle, \langle w \rangle \rangle\) is the encoding of the two strings \(\langle D \rangle\) and \(\langle w \rangle\), all as described earlier in the lecture. Thus, \(\langle D \rangle\), \(\langle w \rangle\), and \(\langle \langle D \rangle, \langle w \rangle \rangle\) are all binary strings. It could be the case, however, that the alphabet of \(D\) is any alphabet of the form \(\Gamma = \{0, \ldots, n - 1\}\), and likewise for the string \(w\).

At this point it is a natural question to ask whether or not the language \(A_{\text{DFA}}\) is decidable. Certainly it is. For a given input string \(x \in \{0, 1\}^*\), one can easily check that it takes the form \(x = \langle \langle D \rangle, \langle w \rangle \rangle\) for a DFA \(D\) and a string \(w\), and then check whether or not \(D\) accepts \(w\) by simply *simulating*, just like you would do
The DSM $M$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle \langle D \rangle, \langle w \rangle \rangle$ for $D$ being a DFA and $w$ being a string over the alphabet of $D$, then reject.

2. Simulate $D$ on input $w$; accept if $D$ accepts $w$ and reject if $D$ rejects $w.$

Figure 15.2: A high-level description of a DSM $M$ that decides the language $A_{\text{DFA}}$.

with a piece of paper and a pencil if you were asked to make this determination for yourself. Figure 15.2 gives a high-level description of a DSM $M$ that decides the language $A_{\text{DFA}}$ along these lines.

Now, you might object to the claim that Figure 15.2 describes a DSM that decides $A_{\text{DFA}}$. It does describe the main idea of how $M$ operates, which is that it simulates $D$ on input $w$, but it offers hardly any detail at all. Compared with the DSM descriptions we have seen thus far in the course, it seems more like a suggestion for how to design a DSM than an actual description of a DSM.

This is a fair criticism, but as we move forward with the course, we will need to make a transition along these lines. The computations we will consider will become more and more complicated, and in the interest of both time and clarity we must abandon the practice of describing the DSMs that perform these computations explicitly. Hopefully our discussions and development of the DSM model have convinced you that the process of taking a high-level description of a DSM, such as the one in Figure 15.2, and producing an actual DSM that performs the computation described is a routine task.

Because the description of the DSM $M$ suggested by Figure 15.2 is our first example of such a high-level DSM description, let us take a moment to consider in greater detail how it could be turned into a formal specification of a DSM.

1. The input $x$ to the DSM $M$ is initially stored in stack 0, which we might instead choose to name $X$ for clarity. The first step of $M$ checks to see that the input takes the form $x = \langle \langle D \rangle, \langle w \rangle \rangle$. Assuming that the input does take this form, it is convenient for the sake of the second step of $M$ (meaning the simulation of $D$ on input $w$) that the input is split into two parts, with the string $\langle D \rangle$ being stored in a stack called $D$ and $\langle w \rangle$ being stored in a stack called $W$. This splitting could easily be done as a part of the check that the input does take the form $x = \langle \langle D \rangle, \langle w \rangle \rangle$.

2. To simulate $D$ on input $w$, the DSM $M$ will need to keep track of the current state of $D$, so it is natural to introduce a new stack $Q$ for this purpose. At
the start of the simulation, Q is initialized it so that it stores 0 (which is the encoding of the state $q_0$).

3. The actual simulation proceeds in the natural way, which is to examine the encodings of the symbols of $w$ stored in $W$, one at a time, updating the state contained in Q accordingly. While an explicit description of the DSM states and transitions needed to do this would probably look rather complex, it could be done in a conceptually simple manner. In particular, each step of the simulation would presumably involve $M$ searching through the transitions of $D$ stored in $D$ to find a match with the current state encoding stored in $Q$ and the next input symbol encoding stored in $W$, after which $Q$ is updated. Naturally, $M$ can make use of additional stacks and make copies of strings as needed so that the encoding $⟨D⟩$ is always available at the start of each simulation step.

4. Once the simulation is complete, an examination of the state stored in $Q$ and the encoding $⟨F⟩$ of the accepting states of $D$ leads to acceptance or rejection appropriately.

All in all, it would be a tedious task to write down the description of a DSM $M$ that behaves in the manner just described—but I hope you will agree that with a bit of time, patience, and planning, it would be feasible to do this. An explicit description of such a DSM $M$ would surely be made more clear if a thoughtful use of subroutines was devised (not unlike the analogous task of writing a computer program to perform such a simulation).

Next let us consider a variant of the language $A_{DFA}$ for NFAs in place of DFAs:

$$A_{NFA} = \{⟨⟨N⟩, ⟨w⟩⟩ : N \text{ is an NFA and } w \in L(N)\}.$$

Again, it is our assumption that the encodings with respect to which this language is defined are as discussed earlier in the lecture. The language $A_{NFA}$ is also decidable. This time, however, it would not be reasonable to simply describe a DSM that “simulates $N$ on input $w$,” because it isn’t at all clear how a deterministic stack machine can simulate a nondeterministic finite automaton computation. What we can do instead is to make use of the process through which NFAs are converted to DFAs that we discussed in Lecture 3; this is a well-defined deterministic procedure, and it can certainly be performed by a DSM. Figure 15.3 gives a high-level description of a DSM $M$ that decides $A_{NFA}$. Once again, although it would be a time-consuming task to explicitly describe a DSM that performs this computation, it is reasonable to view this as a straightforward task in a conceptual sense.

One can also define a language similar to $A_{DFA}$ and $A_{NFA}$, but for regular expressions in place of DFAs and NFAs:

$$A_{REX} = \{⟨⟨R⟩, ⟨w⟩⟩ : R \text{ is a regular expression and } w \in L(R)\}.$$

170
Lecture 15

The DSM $M$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = (\langle N \rangle, \langle w \rangle)$ for $N$ being an NFA and $w$ being a string over the alphabet of $N$, then reject.

2. Convert $N$ into an equivalent DFA $D$ using the subset construction described in Lecture 3.

3. Simulate $D$ on input $w$; accept if $D$ accepts $w$ and reject if $D$ rejects $w$.

Figure 15.3: A high-level description of a DSM $M$ that decides the language $A_{NFA}$.

We did not actually discuss an encoding scheme for regular expressions, so it will be left to you to devise or imagine your own encoding scheme—but as long as you picked a reasonable one, the language $A_{REX}$ would be decidable. In particular, given a reasonable encoding scheme for regular expressions, a DSM could first convert this regular expression into an equivalent DFA, and then simulate this DFA on the string $w$.

Here is a different example of a language, which we will argue is also decidable:

$$E_{DFA} = \{ \langle D \rangle : D \text{ is a DFA and } L(D) = \emptyset \}. \quad (15.24)$$

In this case, one cannot decide this language simply by “simulating the DFA $D$,“ because a priori there is no particular string on which to simulate it; we care about every possible string that could be given as input to $D$ and whether or not $D$ accepts any of them. Deciding the language $E_{DFA}$ is therefore not necessarily a straightforward simulation task.

What we can do instead is to treat the decision problem associated with this language as a graph reachability problem. The DSM $M$ suggested by Figure 15.4 takes this approach and decides $E_{DFA}$. By combining this DSM with ideas from the previous examples, one can prove that analogously defined languages $E_{NFA}$ and $E_{REX}$ are also decidable:

$$E_{NFA} = \{ \langle N \rangle : N \text{ is an NFA and } L(N) = \emptyset \},$$
$$E_{REX} = \{ \langle R \rangle : R \text{ is a regular expression and } L(R) = \emptyset \}. \quad (15.25)$$

One more example of a decidable language concerning DFAs is this language:

$$E_{DFA} = \{ \langle \langle A \rangle, \langle B \rangle \rangle : A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}. \quad (15.26)$$

Figure 15.5 gives a high-level description of a DSM that decides this language. One natural way to perform the construction in step 2 is to use the Cartesian product construction described in Lecture 4.
The DSM $M$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle \langle D \rangle \rangle$ for $D$ being a DFA, then reject.
2. Set $S \leftarrow \{0\}$.
3. Set $a \leftarrow 1$.
4. For every pair of integers $j, k \in \{0, \ldots, m - 1\}$, where $m$ is the number of states of $D$, do the following:
   4.1 If $j \in S$ and $k \not\in S$, and $D$ includes a transition from $q_j$ to $q_k$, then set $S \leftarrow S \cup \{k\}$ and $a \leftarrow 0$.
5. If $a = 0$ then goto step 3.
6. Reject if there exists $k \in S$ such that $q_k \in F$ (i.e., $q_k$ is an accept state of $D$), otherwise accept.

Figure 15.4: A high-level description of a DSM $M$ that decides the language $E_{\text{DFA}}$.

The DSM $M$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle \langle A \rangle, \langle B \rangle \rangle$ for $A$ and $B$ being DFAs, then reject.
2. Construct a DFA $C$ for which $L(C) = L(A) \Delta L(B)$.
3. Accept if $\langle C \rangle \in E_{\text{DFA}}$ and otherwise reject.

Figure 15.5: A high-level description of a DSM $M$ that decides the language $E_{\text{DFA}}$.

Languages based on CFGs

Next let us turn to a couple of examples of decidable languages concerning context-free grammars. Following along the same lines as the examples discussed above, we may consider these languages:

$$A_{\text{CFG}} = \{ \langle \langle G \rangle, \langle w \rangle \rangle : G \text{ is a CFG and } w \in L(G) \},$$
$$E_{\text{CFG}} = \{ \langle G \rangle : G \text{ is a CFG and } L(G) = \emptyset \}. \quad (15.27)$$

Once again, although we have not explicitly described an encoding scheme for context-free grammars, it is not difficult to come up with such a scheme (or to
The DSM $M$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle \langle G \rangle \rangle$ for $G$ a CFG and $w$ a string, then reject.

2. Convert $G$ into an equivalent CFG $H$ in Chomsky normal form.

3. If $w = \varepsilon$ then accept if $S \rightarrow \varepsilon$ is a rule in $H$ and reject otherwise.

4. Search over all possible derivations by $H$ having $2|w| - 1$ steps (of which there are finitely many). Accept if a valid derivation of $w$ is found, and reject otherwise.

Figure 15.6: A high-level description of a DSM $M$ that decides the language $A_{CFG}$. This DSM is ridiculously inefficient, but there are more efficient ways to decide this language.

just imagine that such a scheme has been defined). A DSM that decides the first language is described in Figure 15.6. It is worth noting that this is a ridiculously inefficient way to decide the language $A_{CFG}$, but right now we don’t care! We’re just trying to prove that this language is decidable. There are, in fact, much more efficient ways to decide this language, but we will not discuss them now.

Finally, the language $E_{CFG}$ can be decided using a variation on the reachability technique. In essence, we keep track of a set containing variables that generate at least one string, and then test to see if the start variable is contained in this set. A DSM that decides this language is described in Figure 15.7.

Now, you may be wondering about this next language, as it is analogous to one concerning DFAs from above:

\[
EQ_{CFG} = \{ \langle \langle G \rangle, \langle H \rangle \rangle : G \text{ and } H \text{ are CFGs and } L(G) = L(H) \}. \tag{15.28}
\]

As it turns out, this language is not decidable. (We won’t go through the proof, because it would take us a bit too far off the path of the rest of the course, but it would not be too difficult to prove sometime after the lecture following this one.) Some other examples of undecidable languages concerning context-free grammars are as follows:

\[
\{ \langle G \rangle : G \text{ is a CFG that generates all strings over its alphabet} \},
\{ \langle G \rangle : G \text{ is an ambiguous CFG} \}, \tag{15.29}
\{ \langle G \rangle : G \text{ is a CFG and } L(G) \text{ is inherently ambiguous} \}.
\]
The DSM $M$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle G \rangle$ for $G$ a CFG, then reject.
2. Set $T \leftarrow \Sigma$ (for $\Sigma$ being the alphabet of $G$).
3. Set $a \leftarrow 1$.
4. For each rule $X \rightarrow w$ of $G$ do the following:
   4.1 If $X$ is not contained in $T$, and every variable and every symbol of $w$ is contained in $T$, then set $T \leftarrow T \cup \{X\}$ and $a \leftarrow 0$.
5. If $a = 0$ then goto 2.
6. Reject if the start variable of $G$ is contained in $T$, otherwise accept.

Figure 15.7: A high-level description of a DSM $M$ that decides the language $E_{CFG}$.
Lecture 16

Universal stack machines and a non-semidecidable language

In this lecture we will describe a universal stack machine. This is a stack machine that, when given the encoding of an arbitrary stack machine, can simulate that machine on a given input. To describe such a machine, we must naturally consider encodings of stack machines, and this will be the first order of business for the lecture.

Once we are done discussing universal stack machines, we will encounter our first example of a language that is not semidecidable (and is therefore not decidable). By using the non-semidecidability of this language, many other languages can be shown to be either undecidable or non-semidecidable, as we will see in the lecture following this one.

16.1 An encoding scheme for DSMs

In the previous lecture we discussed in detail an encoding scheme for DFAs, and we observed that this scheme is easily adapted to obtain an encoding scheme for NFAs. While we did not discuss specific encoding schemes for regular expressions and context-free grammars, we made use of the fact that one can devise encoding schemes for these models without difficulty.

We could follow a similar route for DSMs, as there are no new conceptual difficulties that arise for this model in comparison to the other models just mentioned. However, given the high degree of importance that languages involving encodings of DSMs will have in the remainder of the course, it is fitting to take a few moments to be careful and precise about this notion. As is the case for just about every encoding scheme we consider, there are many alternatives to the encoding scheme for DSMs we will define—our focus on the specifics of this encoding scheme is done
in the interest of clarity and precision, and not because the specifics themselves are essential to the study of computability.

Throughout the discussion that follows, we will assume that

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}) \]  

(16.1)

is a given DSM whose encoding is to be described. We will assume that \( M \) has \( r \) stacks, for \( r \) being an arbitrary positive integer (which our encoding scheme must specify). Along similar lines to the discussion of string encodings, as well as the encodings of other models discussed in the previous lecture, we will make the assumption that the state set \( Q \) of \( M \) takes the form

\[ Q = \{q_0, \ldots, q_{m-1}\} \]  

(16.2)

for some positive integer \( m \), and that the input and stack alphabets of \( M \) take the form

\[ \Sigma = \{0, \ldots, k-1\} \quad \text{and} \quad \Gamma = \{0, \ldots, n-1\} \]  

(16.3)

for positive integers \( k \) and \( n \) satisfying \( k < n \). It is necessarily the case that \( k < n \), as the bottom-of-the-stack marker \( \diamond \) is contained in \( \Gamma \) but not \( \Sigma \), and hereafter we will identify the bottom-of-the-stack marker \( \diamond \) with the last symbol \( n-1 \) of \( \Gamma \). The encoding scheme we will define will encode \( M \) as a binary string \( \langle M \rangle \in \{0, 1\}^* \).

We will first describe how the possible actions that \( M \) makes on each individual non-halting state will be encoded. Once this is done, an encoding of the transition function will be obtained by simply encoding an ordered list of the encodings that describe the actions that \( M \) makes on its individual non-halting states. Throughout this process, let us agree that each state \( q \in Q \) is to be encoded as the binary string \( \langle q \rangle \) that is obtained by encoding that state’s index in binary notation (so that \( \langle q_0 \rangle = 0, \langle q_1 \rangle = 1, \langle q_2 \rangle = 10, \) etc.), and that stack symbols are encoded in a similar way (so that \( \langle 0 \rangle = 0, \langle 1 \rangle = 1, \langle 2 \rangle = 10, \) etc.).

There are two possibilities for a given non-halting state \( q \in Q \setminus \{q_{acc}, q_{rej}\} \):

1. The state \( q \) is a **push state**. This means that there is a single transition that originates from state \( q \); this transition must be labeled by a stack symbol \( a \in \Gamma \) and must lead to some state \( p \in Q \).

2. The state \( q \) is a **pop state**. This means that there is one transition originating from the state \( q \) for each stack symbol \( a \in \Gamma \). We will write \( p_a \) to denote the state to which \( M \) transitions when it pops \( a \) off of the stack. (Note that this is a context-dependent notation that only makes sense when we have in mind a particular choice of \( q \).)
Figure 16.1: An example of a DSM whose encoding will be calculated. This DSM appeared in Lecture 12; it erases the stack $X$ and accepts. Here, however, the reject state has been explicitly included in the diagram, and the states are clearly labeled by their names $q_0$, $q_1$, $q_2$, and $q_3$. For the sake of this example, we will assume that there is just one stack, so that $X$ refers to stack number 0.

In both cases, one of the stacks, indexed by $s \in \{0, \ldots, r-1\}$, is associated with the state, and once again we will encode this stack index as a string $\langle s \rangle$ using binary notation. For the first case, in which $q$ is a push state, we will encode the information summarized above as follows:

$$\langle \langle q \rangle, \langle s \rangle, 0, \langle a \rangle, \langle p \rangle \rangle.$$ \hspace{1cm} (16.4)

For the second case, in which $q$ is a pop state, we will encode the information summarized above as follows:

$$\langle \langle q \rangle, \langle s \rangle, 1, \langle p_0 \rangle, \ldots, \langle p_{n-1} \rangle \rangle.$$ \hspace{1cm} (16.5)

In the first case, the 0 in the third position indicates that the state $q$ is a push state, while the 1 in the third position indicates that the state $q$ is a pop state in the second case.

For example, consider the DSM whose state diagram is pictured in Figure 16.1. The state $q_0$ is a pop state, and by following the prescription above, we encode the actions corresponding to this state by the binary string

$$\langle 0, 0, 1, 0, 0, 1 \rangle = 00100101100100101.$$ \hspace{1cm} (16.6)

The state $q_1$, on the other hand, is a push state, and the actions associated with this state are encoded as

$$\langle 1, 0, 0, 10, 10 \rangle = 011001001010010100.$$ \hspace{1cm} (16.7)
These are the only two non-halting states of $M$, and therefore the transition function of this DSM is encoded as follows:

$$\langle \delta \rangle = \langle \langle 0, 0, 1, 0, 0, 1 \rangle, \langle 1, 0, 0, 10, 10 \rangle \rangle. \quad (16.8)$$

(As a binary string, this string’s length is about the text-width of this page; there’s not much point in writing it out explicitly.)

Aside from the transition function, we just need to specify these things (which are all represented by nonnegative integers) to complete the specification of a DSM $M$:

1. The number of stacks $r$.
2. The number of states $m$.
3. The number of input symbols $k$.
4. The number of stack symbols $n$.
5. Which state is the accept state.
6. Which state is the reject state.

The specific ordering we choose doesn’t really matter as long as we pick an ordering and stick to it, so let us decide that the encoding of the entire DSM $M$ is as follows:

$$\langle M \rangle = \langle \langle r \rangle, \langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \langle q_{\text{acc}} \rangle, \langle q_{\text{rej}} \rangle \rangle. \quad (16.9)$$

For example, the complete DSM $M$ illustrated in Figure 16.1 is encoded as

$$\langle M \rangle = \langle 1, 100, 10, 11, \langle \delta \rangle, 10, 11 \rangle, \quad (16.10)$$

where $\langle \delta \rangle$ is as in (16.8).

### 16.2 A universal stack machine

Now that we have defined an encoding scheme for DSMs, we can consider the computational task of simulating a given DSM on a given input. A *universal stack machine* is a stack machine that can perform such a simulation—it is universal in the sense that it is one single DSM that is capable of simulating all other DSMs.

Recall from Lecture 13 that a *configuration* of an $r$-DSM

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \quad (16.11)$$

is a tuple

$$c = (q, x_0, \ldots, x_{r-1}) \in Q \times (\Gamma^*)^r. \quad (16.12)$$
Such a configuration indicates that the current state of $M$ is $q$ and that the contents of the $r$ stacks of $M$ are described by the strings $x_0, \ldots, x_{r-1}$. Assuming that $Q = \{q_0, \ldots, q_{m-1}\}$ and $\Gamma = \{0, \ldots, n-1\}$, there is an obvious way that such a configuration can be encoded into a binary string: simply take this string to be

$$\langle c \rangle = \langle \langle q \rangle, \langle x_0 \rangle, \ldots, \langle x_{r-1} \rangle \rangle. \quad (16.13)$$

As before, $\langle q \rangle$ is the encoding of the nonnegative integer index of $q$ using binary notation and $\langle x_0 \rangle, \ldots, \langle x_{r-1} \rangle$ are binary strings encoding the strings $x_0, \ldots, x_{r-1}$ using the method we have been discussing for the last two lectures.

Now, if we wish to simulate the computation of a given DSM $M$ on some input string $w$, a natural approach is to keep track of the configurations of $M$. Specifically, we will begin with the initial configuration of $M$ on input $w$, which is

$$(q_0, w\Diamond, \Diamond, \ldots, \Diamond), \quad (16.14)$$

and then repeatedly compute the next configuration of $M$, over and over until perhaps we eventually reach a configuration whose state is $q_{\text{acc}}$ or $q_{\text{rej}}$, at which point we can stop. Of course, we might never reach such a configuration—if $M$ runs forever on input $w$, our simulation will also run forever. As it turns out (and as we will see later), there is no way to know whether or not the simulation will eventually stop, but this is OK—we’re just looking for a simulation that directly mimics $M$ on input $w$, including the possibility that the simulation runs forever when the same is true of $M$ on input $w$.

With this approach in mind, let us focus on the task of simply determining the next configuration, meaning the one that results from one computational step, for a given DSM $M$ and a given configuration of $M$. That is, we can focus on the function $f$ having the form

$$f : \{0, 1\}^* \times \{0, 1\}^* \to \{0, 1\}^* \quad (16.15)$$

that is defined as follows. For every DSM

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \quad (16.16)$$

and every configuration

$$c = (q, x_0, \ldots, x_{r-1}) \quad (16.17)$$

of $M$, the function $f$ is defined so that

$$f(\langle M \rangle, \langle c \rangle) = \langle d \rangle, \quad (16.18)$$

where $d$ is the configuration obtained by running $M$ for one step starting from the configuration $c$ (as specified by Definition 13.2).
The function $f$ is actually not all that difficult to compute. For example, suppose that $c = (p, x_0, \ldots, x_{r-1})$, where $p$ happens to be a push state of $M$ that pushes $a$ onto stack number 0 and transitions to the state $q$. The function $f$ must then satisfy

$$f(\langle M \rangle, \langle p \rangle, \langle x_0 \rangle, \langle x_1 \rangle, \ldots, \langle x_{r-1} \rangle) = \langle \langle q \rangle, \langle ax_0 \rangle, \langle x_1 \rangle, \ldots, \langle x_{r-1} \rangle \rangle. \quad (16.19)$$

If instead it were the case that $p$ was a pop state of $M$ that, when $a$ is popped off of stack number 1, transitions to the state $q$, then we would have

$$f(\langle M \rangle, \langle p \rangle, \langle ax_1 \rangle, \ldots, \langle x_{r-1} \rangle) = \langle \langle q \rangle, \langle x_0 \rangle, \langle x_1 \rangle, \ldots, \langle x_{r-1} \rangle \rangle. \quad (16.20)$$

We also need to worry about the special case in which an empty stack is popped, and for any input to $f$ that does not take the form $(\langle M \rangle, \langle c \rangle)$ for $M$ being a DSM and $c$ a valid configuration of $M$, we could simply define the output of $f$ to be $\varepsilon$ (which is an arbitrary choice that doesn’t really matter for the purposes of the simulation). It is also convenient to define

$$f(\langle M \rangle, \langle c \rangle) = \langle c \rangle \quad (16.21)$$

whenever $c$ is a halting configuration of $M$.

The difficulty in computing the function $f$ is, naturally, that one needs to examine the encoding $\langle M \rangle$ in order to determine how the encoding $\langle c \rangle$ of each configuration is to be transformed into the encoding $\langle d \rangle$ of the configuration that results from running $M$ for one step. It would be a time-consuming process to explicitly describe a DSM that computes $f$, but at a conceptual level it would not be unreasonable to describe this task as being fairly straightforward. If we were to do this carefully, perhaps we would start by defining a subroutine that searches through the encoding $\langle \delta \rangle$ of the transition function of $M$ to find the instruction corresponding to a given state, as well as a subroutine that applies a given instruction to a given configuration. Here is a rather high-level description how the required computation might be performed:

1. Test to see that $x = \langle M \rangle$ and $y = \langle c \rangle$ for some choice of a DSM $M$ and a valid configuration $c$ of $M$. Output $\varepsilon$ and halt if this is not the case.
2. Check if $c$ is a halting configuration of $M$. Output $\langle c \rangle$ and halt if this is the case.
3. Supposing that $c = (q, x_0, \ldots, x_{r-1})$ for $q$ being a non-halting state of $M$, process the encoding $\langle \delta \rangle$ of the transition function of $M$ to obtain the instructions corresponding to the state $q$ of $M$. This will be a string of the form

$$\langle \langle q \rangle, \langle s \rangle, 0, \langle a \rangle, \langle p \rangle \rangle \quad (16.22)$$
The DSM $U$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle \langle M \rangle , \langle w \rangle \rangle$ for $M$ being a DSM and $w$ being a string over the alphabet of $M$, then reject.

2. Set $Y \leftarrow \langle M \rangle$ and $Z \leftarrow \langle c \rangle$, for $c$ being the initial configuration of $M$ on input $w$.

3. Accept if $Z$ stores an accepting configuration of $M$ and reject if $Z$ stores a rejecting configuration of $M$. (If $Z$ stores a non-halting configuration of $M$, then continue to the next step.)

4. Compute $Z \leftarrow f(Y, Z)$, where $f$ is the next-configuration function described previously, and goto step 3.

Figure 16.2: A high-level description of a DSM $U$ that recognizes the language $A_{DSM}$.

if $q$ is a push state, or a string of the form

$$\langle \langle q \rangle , \langle s \rangle , 1, \langle p_0 \rangle , \ldots, \langle p_{n-1} \rangle \rangle$$

(16.23)

if $q$ is a pop state.

4. Modify $\langle c \rangle$ according to the instructions corresponding to the state $q$ obtained in the previous step to obtain the encoding $\langle d \rangle$ that results from running $M$ for one step on configuration $c$. Output $\langle d \rangle$ and halt.

With the function $f$ in hand, one can simulate the computation of a given DSM $M$ on a given input $w$ in the manner suggested above, by starting with the initial configuration of $M$ on $w$ and repeatedly applying $f$.

Now consider the following language, which is the natural DSM analogue of the languages $A_{DFA}$, $A_{NFA}$, $A_{REG}$, and $A_{CFG}$ discussed in the previous lecture:

$$A_{DSM} = \{ \langle \langle M \rangle , \langle w \rangle \rangle : M \text{ is a DSM and } w \in L(M) \}.$$  (16.24)

We conclude that $A_{DSM}$ is semidecidable: the DSM $U$ described in Figure 16.2 is such that $L(U) = A_{DSM}$. This DSM has been named $U$ to reflect the fact that it is a universal DSM.

**Proposition 16.1.** The language $A_{DSM}$ is semidecidable.
16.3 A non-semidecidable language

It is natural at this point to ask whether or not $A_{DSM}$ is decidable, given that it is semidecidable. It is not decidable, as we will soon prove. Before doing this, however, we will consider a different language and prove that this language is not even semidecidable. Here is the language:

$$\text{DIAG} = \{ \langle M \rangle : M \text{ is a DSM and } \langle M \rangle \not\in L(M) \}. \quad (16.25)$$

That is, the language DIAG contains all binary strings $\langle M \rangle$ that, with respect to the encoding scheme we discussed at the start of the lecture, encode a DSM $M$ that does not accept this encoding of itself. (Note that if it so happens that the string $\langle M \rangle$ encodes a DSM whose input alphabet has just one symbol, then it will indeed be the case that $\langle M \rangle \not\in L(M)$.)

**Theorem 16.2.** The language $\text{DIAG}$ is not semidecidable.

**Proof.** Assume toward contradiction that the language $\text{DIAG}$ is semidecidable. There must therefore exist a DSM $M$ such that $L(M) = \text{DIAG}$.

Now, consider the encoding $\langle M \rangle$ of $M$. By the definition of the language DIAG one has

$$\langle M \rangle \in \text{DIAG} \iff \langle M \rangle \not\in L(M). \quad (16.26)$$

On the other hand, because $M$ recognizes DIAG, it is the case that

$$\langle M \rangle \in \text{DIAG} \iff \langle M \rangle \in L(M). \quad (16.27)$$

Consequently,

$$\langle M \rangle \not\in L(M) \iff \langle M \rangle \in L(M), \quad (16.28)$$

which is a contradiction. We conclude that DIAG is not semidecidable. \hfill \Box

**Remark 16.3.** Note that this proof is very similar to the proof that $\mathcal{P}(\mathbb{N})$ is not countable from the very first lecture of the course. It is remarkable how simple this proof of the non-semidecidability of DIAG is; it has used essentially none of the specifics of the DSM model or the encoding scheme we defined.

Now that we know DIAG is not semidecidable, we prove that $A_{DSM}$ is not decidable.

**Proposition 16.4.** The language $A_{DSM}$ is undecidable.
The DSM $K$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle M \rangle$ for $M$ being a DSM, then reject.
2. Run $T$ on input $\langle \langle M \rangle, \langle M \rangle \rangle$. If $T$ accepts, then reject, otherwise accept.

**Proof.** Assume toward contradiction that $A_{\text{DSM}}$ is decidable. There must therefore exist a DSM $T$ that decides $A_{\text{DSM}}$. Define a new DSM $K$ as described in Figure 16.3.

For a given DSM $M$, we may now ask ourselves what $K$ does on the input $\langle M \rangle$. If it is the case that $\langle M \rangle \in \text{DIAG}$, then by the definition of DIAG it is the case that $\langle M \rangle \not\in L(M)$, and therefore $\langle \langle M \rangle, \langle M \rangle \rangle \not\in A_{\text{DSM}}$ (because $M$ does not accept $\langle M \rangle$). This implies that $T$ rejects the input $\langle \langle M \rangle, \langle M \rangle \rangle$, and so $K$ must accept the input $\langle M \rangle$. If, on the other hand, it is the case that $\langle M \rangle \not\in \text{DIAG}$, then $\langle M \rangle \in L(M)$, and therefore $\langle \langle M \rangle, \langle M \rangle \rangle \in A_{\text{DSM}}$. This implies that $T$ accepts the input $\langle \langle M \rangle, \langle M \rangle \rangle$, and so $K$ must reject the input $\langle M \rangle$. One final possibility is that $K$ is run on an input string that does not encode a DSM at all, and in this case it rejects.

Considering these possibilities, we find that $K$ decides DIAG. This, however, is in contradiction with the fact that DIAG is non-semidecidable (and is therefore undecidable). Having obtained a contradiction, we conclude that $A_{\text{DSM}}$ is undecidable, as required. \qed
Lecture 17

Undecidable languages

This lecture focuses on techniques for proving that certain languages are undecidable (or, in some cases, non-semidecidable). The lecture will be divided into two main parts: the first part focuses on undecidability proofs by contradiction, and the second part discusses the notion of a reduction and how undecidability results may be established through them. Along the way we will discuss some useful tricks that can be applied in both settings.

Before proceeding to the first part of the lecture, let us recall that we have fixed an encoding scheme for DSMs, whereby any given DSM $M$ is encoded as a binary string $\langle M \rangle$. In the previous lecture we proved that the language

$$\text{DIAG} = \{ \langle M \rangle : M \text{ is a DSM and } \langle M \rangle \notin L(M) \} \quad (17.1)$$

is non-semidecidable, and we then used this fact to conclude that the language

$$\text{A_{DSM}} = \{ \langle \langle M \rangle, \langle w \rangle \rangle : M \text{ is a DSM and } w \in L(M) \} \quad (17.2)$$

is undecidable (although it is semidecidable). All of the undecidability proofs that appear in this lecture are, in some sense, anchored by the diagonalization proof that DIAG is not semidecidable.

17.1 Undecidability proofs through contradiction

In this section we will see a few more examples of undecidability proofs that have a similar style to the proof we saw in at the end of the previous lecture, through which we concluded that $\text{A_{DSM}}$ is undecidable. More specifically, we assumed toward contradiction that $\text{A_{DSM}}$ is decidable, and based on that assumption we constructed a DSM that decided a language (specifically, the language DIAG) that we already knew to be undecidable.
This is the same general pattern that will be used in this section when we wish to prove that a chosen language $A$ is undecidable:

1. Assume toward contradiction that $A$ is decidable.
2. Use that assumption to construct a DSM that decides a language $B$ that we already know to be undecidable.
3. Having obtained a contradiction from the assumption that $A$ is decidable, we conclude that $A$ is undecidable.

A similar approach can sometimes be used to prove that a language $A$ is non-semidecidable, and in both cases we might potentially obtain a contradiction by using our assumption toward contradiction about $A$ to semidecide a language $B$ that we already know to be non-semidecidable (as opposed to deciding a language $B$ that we already know to be undecidable).

Here is an example. Define a language $HALT$ as follows:

$$HALT = \{ \langle \langle M \rangle, \langle w \rangle \rangle : M \text{ is a DSM that halts on input } w \}.$$

(17.3)

To say that $M$ halts on input $w$ means that it stops, either by accepting or rejecting. Let us agree that the statement “$M$ halts on input $w$” is false in case $w$ contains symbols not in the input alphabet of $M$, just as a matter of terminology.

We will prove that $HALT$ is undecidable, but before we do this let us observe that $HALT$ is semidecidable (just like $A_{DSM}$). In particular, this language can be semi-decided by a modified version of the universal stack machine $U$ from the previous lecture; the modification is that it accepts both in the case that $M$ accepts $w$ and in the case that $M$ rejects $w$. Of course, when it is the case that $M$ runs forever on $w$, the same will be true of $U$ running on input $\langle \langle M \rangle, \langle w \rangle \rangle$.

**Proposition 17.1.** The language $HALT$ is undecidable.

*Proof.* Assume toward contradiction that $HALT$ is decidable, so that there exists a DSM $T$ that decides it. Define a new DSM $K$ as described in Figure 17.1.

We will conclude that $K$ decides $A_{DSM}$. Note first that if $K$ is given an input that is not of the form $\langle \langle M \rangle, \langle w \rangle \rangle$, for $M$ a DSM and $w$ a string over the input alphabet of $M$, then it rejects (as a DSM for $A_{DSM}$ should). Otherwise, when the input to $K$ does take the form $\langle \langle M \rangle, \langle w \rangle \rangle$, for $M$ a DSM and $w$ a string over the input alphabet of $M$, there are three possible cases:

1. If it is the case that $M$ accepts $w$, then $T$ will accept $\langle \langle M \rangle, \langle w \rangle \rangle$ (because $M$ halts on $w$), and the simulation of $M$ on input $w$ will result in acceptance.
2. If it is the case that $M$ rejects $w$, then $T$ will accept $\langle \langle M \rangle, \langle w \rangle \rangle$ (again because $M$ halts on $w$), and the simulation of $M$ on input $w$ will result in rejection.
The DSM $K$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle \langle M \rangle, \langle w \rangle \rangle$ for $M$ being a DSM and $w$ being a string over the alphabet of $M$, then reject.
2. Run $T$ on input $\langle \langle M \rangle, \langle w \rangle \rangle$ and reject if $T$ rejects. Otherwise, continue to the next step.
3. Simulate $M$ on input $w$; accept if $M$ accepts and reject if $M$ rejects.

This, however, is in contradiction with the fact that $A_{\text{DSM}}$ is undecidable. Having obtained a contradiction, we conclude that $\text{HALT}$ is undecidable. \qed

Next we will consider this language, which is a DSM variant of the languages $E_{\text{DFA}}$ and $E_{\text{CFG}}$ from Lecture 15:

$$E_{\text{DSM}} = \{ \langle M \rangle : M \text{ is a DSM with } L(M) = \emptyset \}. \quad (17.4)$$

We will prove that this language is undecidable, but in order to do this we need to make use of the very simple but remarkably useful trick of hard-coding inputs into DSMs.

Here is the idea. Suppose that we have a DSM $M$ along with a fixed string $w$ over the input alphabet of $M$. Consider a new DSM, which we will call $M_w$, that operates as described in Figure 17.2. (Figure 17.3 illustrates what a state diagram of $M_w$ might look like, assuming $w = a_1 \cdots a_n$.) This may seem like a curious way to define a DSM; the DSM $M_w$ runs the same way regardless of its actual input string $x$, as it always discards this string and runs $M$ on the string $w$, which is “hard-coded” directly into its description. We will see, however, that it is sometimes very useful to consider a DSM defined like this. Let us also note that given an encoding $\langle \langle M \rangle, \langle w \rangle \rangle$ of a DSM $M$ and a string $w$ over the input alphabet of $M$, it is possible to compute an encoding $\langle M_w \rangle$ of the DSM $M_w$ without difficulty.

**Proposition 17.2.** The language $E_{\text{DSM}}$ is undecidable.
The DSM $M_w$ operates as follows on input $x$:

1. Ignore the input string $x$ and run $M$ on $w$.

Figure 17.2: For any DSM $M$ and a fixed string $w$, the DSM $M_w$ ignores its input and runs $M$ on the string $w$ (which is hard-coded into $M_w$).

```
\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node [input, name=X] {X \leftarrow \epsilon};
  \node [push] (push) {push X};
  \node [state] (M) {$M$};
  \node [state] (a_n) [right of=push] {$a_n$};
  \node [state] (a_n-1) [right of=a_n] {$a_{n-1}$};
  \node [state] (a_2) [right of=a_n-1] {$a_2$};
  \node [state] (a_1) [right of=a_2] {$a_1$};

  \path (X) edge [->] (push)
  (push) edge [->] (a_n)
  (a_n) edge [->] (a_n-1)
  (a_n-1) edge [->] (a_2)
  (a_2) edge [->] (a_1)
  (a_1) edge [->] (M);
\end{tikzpicture}
\end{figure}
```

Figure 17.3: An illustration of a state diagram for $M_w$, assuming $w = a_1 \cdots a_n$. We are also assuming that $X$ refers to the input stack of both $M_w$ and $M$, that the node labeled $X \leftarrow \epsilon$ refers to the subroutine discussed in Lecture 12, and that the node labeled $M$ refers to the entire description of $M$. Thus, the action of this machine is to delete whatever input string it is given, replace this string with $w$, and allow control to pass to the start state of $M$.

The DSM $K$ operates as follows on input $x$:

1. If it is not the case that $x = \langle \langle M \rangle, \langle w \rangle \rangle$ for $M$ being a DSM and $w$ being a string over the alphabet of $M$, then reject.
2. Compute an encoding $\langle M_w \rangle$ of the DSM $M_w$ described in Figure 17.2.
3. Run $T$ on input $\langle M_w \rangle$. If $T$ accepts $\langle M_w \rangle$, then reject, and otherwise accept.

Figure 17.4: A high-level description of a DSM $K$ that decides $A_{DSM}$, assuming the existence of a DSM $T$ that decides $E_{DSM}$.

**Proof.** Assume toward contradiction that $E_{DSM}$ is decidable, so that there exists a DSM $T$ that decides this language. Define a new DSM $K$ as described in Figure 17.4. We can see from the description of $K$ that it will immediately reject when its input does not have the form $\langle \langle M \rangle, \langle w \rangle \rangle$, for $M$ a DSM and $w$ a string over the input alphabet of $M$. Let us consider what happens for inputs that are of the form
\langle \langle M \rangle, \langle w \rangle \rangle$, where $M$ a DSM and $w$ is a string over the input alphabet of $M$.

First, if $w \in L(M)$, then the DSM $M_w$ will accept every string over its alphabet; no matter what string it receives as an input, it just erases this string and runs $M$ on $w$, leading to acceptance. It is therefore certainly not the case that $L(M_w) = \emptyset$. This implies that $T$ must reject the string $\langle M_w \rangle$, and therefore $K$ accepts $\langle \langle M \rangle, \langle w \rangle \rangle$.

On the other hand, if $w \not\in L(M)$, then $M_w$ must either reject every string or run forever on every string for the same reason as before; $M_w$ always discards its input and runs $M$ on $w$, which either rejects or runs forever. It is therefore the case that $L(M_w) = \emptyset$. The DSM $T$ therefore accepts $\langle M_w \rangle$, so $K$ rejects $\langle \langle M \rangle, \langle w \rangle \rangle$.

Considering the possibilities just analyzed, we find that $K$ decides $A_{\text{DSM}}$, which contradicts the fact that this language is undecidable. We conclude that $E_{\text{DSM}}$ is undecidable, as required.

\[ \square \]

### 17.2 Proving undecidability through reductions

The second method through which languages may be proved to be undecidable or non-semidecidable makes use of the notion of a *reduction*.

#### Reductions

The notion of a reduction is, in fact, very general, and many different types of reductions are considered in theoretical computer science—but for now we will consider just one type of reduction (sometimes called a *mapping reduction* or *many-to-one reduction*), which is defined as follows.

**Definition 17.3.** Let $\Sigma$ and $\Gamma$ be alphabets and let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be languages. It is said that $A$ reduces to $B$ if there exists a computable function $f : \Sigma^* \to \Gamma^*$ such that

$$w \in A \iff f(w) \in B$$

(17.5)

for all $w \in \Sigma^*$. One writes

$$A \leq_m B$$

(17.6)

to indicate that $A$ reduces to $B$, and any function $f$ that establishes that this is so may be called a *reduction* from $A$ to $B$.

Figure 17.5 illustrates the action of a reduction. Intuitively speaking, a reduction is a way of transforming one computational decision problem into another. Imagine that you receive an input string $w \in \Sigma^*$, and you wish to determine whether or not $w$ is contained in some language $A$. Perhaps you do not know how to make this determination, but you happen to have a friend who is able to tell you whether
or not a particular string \( y \in \Gamma^* \) is contained in a different language \( B \). If you have a reduction \( f \) from \( A \) to \( B \), then you can determine whether or not \( w \in A \) using your friend’s help: you compute \( y = f(w) \), ask your friend whether or not \( y \in B \), and take their answer as your answer to whether or not \( w \in A \).

The following theorem has a simple and direct proof, but it will nevertheless have central importance with respect to the way that we use reductions to reason about decidability and semidecidability.

**Theorem 17.4.** Let \( \Sigma \) and \( \Gamma \) be alphabets, let \( A \subseteq \Sigma^* \) and \( B \subseteq \Gamma^* \) be languages, and assume \( A \leq_m B \). The following two implications hold:

1. If \( B \) is decidable, then \( A \) is decidable.
2. If \( B \) is semidecidable, then \( A \) is semidecidable.

**Proof.** Let \( f : \Sigma^* \rightarrow \Gamma^* \) be a reduction from \( A \) to \( B \). We know that such a function exists by the assumption \( A \leq_m B \).

We will first prove the second implication. Because \( B \) is semidecidable, there must exist a DSM \( M_B \) such that \( B = L(M_B) \). Define a new DSM \( M_A \) as described in Figure 17.6. It is possible to define a DSM in this way because \( f \) is a computable function.

For a given input string \( w \in A \), we have that \( y = f(w) \in B \), because this property is guaranteed by the reduction \( f \). When \( M_A \) is run on input \( w \), it will therefore accept because \( M_B \) accepts \( y \). Along similar lines, if it is the case that \( w \notin A \), then \( y = f(w) \notin B \). When \( M_A \) is run on input \( w \), it will therefore not accept because \( M_B \) does not accepts \( y \). (It may be that these machines reject or run forever, but we do not care which.) It has been established that \( A = L(M_A) \), and therefore \( A \) is semidecidable.
The DSM $M_A$ operates as follows on input $w \in \Sigma^*$:

1. Compute $y = f(w)$.
2. Run $M_B$ on input $y$.

Figure 17.6: Given a reduction $f$ from $A$ to $B$, and assuming the existence of a DSM $M_B$ that either decides or semidecides $B$, the DSM $M_A$ described either decides or semidecides $A$.

The proof for the first implication is almost identical, except that we take $M_B$ to be a DSM that decides $B$. The DSM $M_A$ defined in Figure 17.6 then decides $A$, and therefore $A$ is decidable. \qed

We will soon use this theorem to prove that certain languages are undecidable (or non-semidecidable), but let us first take a moment to observe two useful facts about reductions.

**Proposition 17.5.** Let $\Sigma, \Gamma, \text{ and } \Delta$ be alphabets and let $A \subseteq \Sigma^*, B \subseteq \Gamma^*, \text{ and } C \subseteq \Delta^*$ be languages. If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$. (In words, $\leq_m$ is a transitive relation among languages.)

**Proof.** As $A \leq_m B$ and $B \leq_m C$, there must exist computable functions $f : \Sigma^* \rightarrow \Gamma^*$ and $g : \Gamma^* \rightarrow \Delta^*$ such that

$$w \in A \iff f(w) \in B \quad \text{and} \quad y \in B \iff g(y) \in C \quad (17.7)$$

for all $w \in \Sigma^*$ and $y \in \Gamma^*$.

Define a function $h : \Sigma^* \rightarrow \Delta^*$ as $h(w) = g(f(w))$ for all $w \in \Sigma^*$. It is evident that $h$ is a computable function: if we have DSMs $M_f$ and $M_g$ that compute $f$ and $g$, respectively, then we can obtain a DSM $M_h$ that computes $h$ by first running $M_f$ and then running $M_g$.

It remains to observe that $h$ is a reduction from $A$ to $C$. If $w \in A$, then $f(w) \in B$, and therefore $h(w) = g(f(w)) \in C$; and if $w \notin A$, then $f(w) \notin B$, and therefore $h(w) = g(f(w)) \notin C$. \qed

**Proposition 17.6.** Let $\Sigma \text{ and } \Gamma$ be alphabets and let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be languages. It is the case that $A \leq_m B$ if and only if $\overline{A} \leq_m \overline{B}$.

**Proof.** For a given function $f : \Sigma^* \rightarrow \Gamma^*$ and a string $w \in \Sigma^*$, the statements $w \in A \iff f(w) \in B$ and $w \in \overline{A} \iff f(w) \in \overline{B}$ are logically equivalent. If we have a reduction $f$ from $A$ to $B$, then the same function also serves as a reduction from $\overline{A}$ to $\overline{B}$, and vice versa. \qed
The DSM $K_M$ operates as follows on input $w \in \Sigma^*$:

1. Run $M$ on input $w$.
   1.1 If $M$ accepts $w$ then accept.
   1.2 If $M$ rejects $w$, then run forever.

Figure 17.7: Given a DSM $M$, we can easily obtain a DSM $K_M$ that behaves as described by replacing any transitions to the accept state of $M$ with transitions to a state that intentionally causes an infinite loop.

Undecidability through reductions

It is possible to use Theorem 17.4 to prove that certain languages are either decidable or semidecidable, but we will focus mainly on using it to prove that languages are either undecidable or non-semidecidable. When using the theorem in this way, we consider the two implications in the contrapositive form. That is, if two languages $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ satisfy $A \leq_m B$, then the following two implications hold:

1. If $A$ is undecidable, then $B$ is undecidable.
2. If $A$ is non-semidecidable, then $B$ is non-semidecidable.

So, if we want to prove that a particular language $B$ is undecidable, then it suffices to pick any language $A$ that we already know to be undecidable, and then prove $A \leq_m B$. The situation is similar for proving languages to be non-semidecidable. The examples that follow illustrate how this may be done.

Example 17.7 ($A_{DSM} \leq_m \text{HALT}$). The first thing we will need to consider is a simple way of modifying an arbitrary DSM $M$ to obtain a slightly different one. In particular, for an arbitrary DSM $M$, let us define a new DSM $K_M$ as described in Figure 17.7. The idea behind the DSM $K_M$ is very simple: if $M$ accepts a string $w$, then so does $K_M$, if $M$ rejects $w$ then $K_M$ runs forever on $w$, and of course if $M$ runs forever on input $w$ then so does $K_M$. Thus, $K_M$ halts on input $w$ if and only if $M$ accepts $w$. Note that if you are given a description of a DSM $M$, it is very easy to come up with a description of a DSM $K_M$ that operates as suggested: just replace the reject state of $M$ with a new state that purposely causes an infinite loop (by repeatedly pushing a symbol onto some stack, for instance).
Now let us define a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ as follows:

$$f(x) = \begin{cases} 
\langle\langle K_M \rangle, \langle w \rangle \rangle & \text{if } x = \langle\langle M \rangle, \langle w \rangle \rangle \text{ for a DSM } M \text{ and a string } w \\
\varepsilon & \text{otherwise.}
\end{cases} \quad (17.8)$$

The function $f$ is computable: all it does is that it essentially looks at an input string, determines whether or not this string is an encoding $\langle\langle M \rangle, \langle w \rangle \rangle$ of a DSM $M$ and a string $w$ over the alphabet of $M$, and if so it replaces the encoding of $M$ with the encoding of the DSM $K_M$ suggested above.

Now let us check to see that $f$ is a reduction from $A_{\text{DSM}}$ to $\text{HALT}$. Suppose first that we have an input $\langle\langle M \rangle, \langle w \rangle \rangle \in A_{\text{DSM}}$. These implications hold:

$$\langle\langle M \rangle, \langle w \rangle \rangle \in A_{\text{DSM}} \Rightarrow M \text{ accepts } w \Rightarrow K_M \text{ halts on } w$$
$$\Rightarrow \langle\langle K_M \rangle, \langle w \rangle \rangle \in \text{HALT} \Rightarrow f(\langle\langle M \rangle, \langle w \rangle \rangle) \in \text{HALT}. \quad (17.9)$$

We therefore have

$$\langle\langle M \rangle, \langle w \rangle \rangle \in A_{\text{DSM}} \Rightarrow f(\langle\langle M \rangle, \langle w \rangle \rangle) \in \text{HALT}, \quad (17.10)$$

which is half of what we need to verify that $f$ is indeed a reduction from $A_{\text{DSM}}$ to $\text{HALT}$. It remains to consider the output of the function $f$ on inputs that are not contained in $A_{\text{DSM}}$, and here there are two cases: one is that the input takes the form $\langle\langle M \rangle, \langle w \rangle \rangle$ for a DSM $M$ and a string $w$ over the alphabet of $M$, and the other is that it does not. For the first case, we have these implications:

$$\langle\langle M \rangle, \langle w \rangle \rangle \notin A_{\text{DSM}} \Rightarrow M \text{ does not accept } w$$
$$\Rightarrow K_M \text{ runs forever on } w \Rightarrow \langle\langle K_M \rangle, \langle w \rangle \rangle \notin \text{HALT}$$
$$\Rightarrow f(\langle\langle M \rangle, \langle w \rangle \rangle) \notin \text{HALT}. \quad (17.11)$$

The key here is that $K_M$ is defined so that it will definitely run forever in case $M$ does not accept (regardless of whether that happens by $M$ rejecting or running forever). The remaining case is that we have a string $x \in \Sigma^*$ that does not take the form $\langle\langle M \rangle, \langle w \rangle \rangle$ for a DSM $M$ and a string $w$ over the alphabet of $M$, and in this case it trivially holds that $f(x) = \varepsilon \notin \text{HALT}$ (because $\varepsilon$ does not encode any element of $\text{HALT}$). We have therefore proved that

$$x \in A_{\text{DSM}} \iff f(x) \in \text{HALT}, \quad (17.12)$$

and therefore $A_{\text{DSM}} \leq_m \text{HALT}$. 

193
We already proved that HALT is undecidable, but the fact that $A_{\text{DSM}} \leq_m \text{HALT}$ provides an alternative proof: because we already know that $A_{\text{DSM}}$ is undecidable, it follows that HALT is also undecidable.

It might not seem that there is any advantage to this proof over the proof we saw in the previous lecture that HALT is undecidable (which wasn’t particularly difficult). We have, however, established a closer relationship between $A_{\text{DSM}}$ and HALT than we did previously. In general, using a reduction is sometimes an easy shortcut to proving that a language is undecidable (or non-semidecidable).

Example 17.8 (DIAG $\leq_m E_{\text{DSM}}$). Recall this language, which was defined earlier in the lecture:

$$E_{\text{DSM}} = \{ \langle M \rangle : M \text{ is a DSM and } L(M) = \emptyset \}. \quad (17.13)$$

We will now prove that DIAG $\leq_m E_{\text{DSM}}$. Because we already know that DIAG is non-semidecidable, we conclude from this reduction that $E_{\text{DSM}}$ is not just undecidable, but in fact it is also non-semidecidable.

For this one we will use the same hardcoding trick that we used earlier in the lecture: for a given DSM $M$, let us define a new DSM $M_{\langle M \rangle}$ just like in Figure 17.2, for the specific choice of the string $w = \langle M \rangle$. This actually only makes sense if the input alphabet of $M$ includes the symbols $\{0, 1\}$ used in the encoding $\langle M \rangle$, so let us agree that $M_{\langle M \rangle}$ immediately rejects if this is not the case.

Now let us define a function $f : \{0, 1\}^* \to \{0, 1\}^*$ as follows:

$$f(x) = \begin{cases} \langle M_{\langle M \rangle} \rangle & \text{if } x = \langle M \rangle \text{ for a DSM } M \\ \epsilon & \text{otherwise.} \end{cases} \quad (17.14)$$

If you think about it for a few moments, it should not be hard to convince yourself that $f$ is computable. It remains to verify that $f$ is a reduction from DIAG to $E_{\text{DSM}}$.

For any string $x \in \text{DIAG}$ we have that $x = \langle M \rangle$ for some DSM $M$ that satisfies $\langle M \rangle \notin L(M)$. In this case we have that $f(x) = \langle M_{\langle M \rangle} \rangle$, and because $\langle M \rangle \notin L(M)$ it must therefore be that $M_{\langle M \rangle}$ never accepts, and so $f(x) = \langle M_{\langle M \rangle} \rangle \in E_{\text{DSM}}$.

Now suppose that $x \notin \text{DIAG}$. There are two cases: either $x = \langle M \rangle$ for a DSM $M$ such that $\langle M \rangle \in L(M)$, or $x$ does not encode a DSM at all. If it is the case that $x = \langle M \rangle$ for a DSM $M$ such that $\langle M \rangle \in L(M)$, we have that $M_{\langle M \rangle}$ accepts every string over its alphabet, and therefore $f(x) = \langle M_{\langle M \rangle} \rangle \notin E_{\text{DSM}}$. If it is the case that $x$ does not encode a DSM, then it trivially holds that $f(x) = \epsilon \notin E_{\text{DSM}}$.

We have proved that

$$x \in \text{DIAG} \iff f(x) \in E_{\text{DSM}}, \quad (17.15)$$

so the proof that DIAG $\leq_m E_{\text{DSM}}$ is complete.
Example 17.9 \((A_{DSM} \leq_m A_E)\). Define a language

\[ AE = \{ \langle M \rangle : M \text{ is a DSM that accepts } \varepsilon \}. \]  

(17.16)

The name AE stands for “accepts the empty string.”

To prove this reduction, we can use exactly the same hardcoding trick that we’ve now used twice already. For every DSM \(M\) and every string \(w\) over the alphabet of \(M\), define a new DSM \(M_w\) as in Figure 17.2, and define a function \(f : \{0, 1\}^* \rightarrow \{0, 1\}^*\) as follows:

\[
f(x) = \begin{cases} 
\langle M_w \rangle & \text{if } x = \langle \langle M \rangle, \langle w \rangle \rangle \text{ for a DSM } M \text{ and a string } w \\
\varepsilon & \text{otherwise.}
\end{cases}
\]  

(17.17)

Now let us check that \(f\) is a valid reduction from \(A_{DSM}\) to \(AE\).

First, for any string \(x \in A_{DSM}\) we have \(x = \langle \langle M \rangle, \langle w \rangle \rangle\) for a DSM \(M\) that accepts the string \(w\). In this case, \(f(x) = \langle M_w \rangle\). We have that \(M_w\) accepts every string, including the empty string, because \(M\) accepts \(w\). Therefore \(f(x) = \langle M_w \rangle \in AE\).

Now consider any string \(x \notin A_{DSM}\). Again there are two cases: either \(x = \langle \langle M \rangle, \langle w \rangle \rangle\) for some DSM \(M\) and a string \(w\) over the alphabet of \(M\), or this is not the case. If it is the case that \(x = \langle \langle M \rangle, \langle w \rangle \rangle\) for a DSM \(M\) and \(w\) a string over the alphabet of \(M\), then \(x \notin A_{DSM}\) implies that \(M\) does not accept \(w\). In this case we have \(f(x) = \langle M_w \rangle \notin AE\), because \(M_w\) does not accept any strings at all (including the empty string). If \(x \neq \langle \langle M \rangle, \langle w \rangle \rangle\) for a DSM \(M\) and string \(w\) over the alphabet of \(M\), then \(f(x) = \varepsilon \notin AE\) (again because \(\varepsilon\) does not encode a DSM, and therefore cannot be included in the language \(AE\)).

We have shown that \(x \in A_{DSM} \iff f(x) \in AE\) holds for every string \(x \in \{0, 1\}^*\), and therefore \(A_{DSM} \leq_m AE\), as required.

Example 17.10 \((E_{DSM} \leq_m R)\). The last example for the lecture will be a tough one. Define a language as follows:

\[ REG = \{ \langle M \rangle : M \text{ is a DSM such that } L(M) \text{ is regular} \}. \]  

(17.18)

We will prove \(E_{DSM} \leq_m REG\).

We will need to make use of a strange way to modify DSMs in order to do this one. Given an arbitrary DSM \(M\), let us define a new DSM \(K_M\) as in Figure 17.8. This is indeed a strange way to define a DSM, but there’s nothing wrong with strange DSMs—we’re just proving a reduction.

Now let us define a function \(f : \{0, 1\}^* \rightarrow \{0, 1\}^*\) as

\[
f(x) = \begin{cases} 
\langle K_M \rangle & \text{if } x = \langle M \rangle \text{ for a DSM } M \\
\varepsilon & \text{otherwise.}
\end{cases}
\]  

(17.19)
The DSM $K_M$ operates as follows on input $x \in \{0, 1\}^*$:

1. Set $t \leftarrow 1$.
2. For every string $w$ over the input alphabet of $M$ satisfying $|w| \leq t$:
   2.1 Run $M$ for $t$ steps on input $w$.
   2.2 If $M$ accepts $w$ within $t$ steps, goto 4.
3. Set $t \leftarrow t + 1$ and goto 2.
4. Accept if $x \in \{0^n1^n : n \in \mathbb{N}\}$, reject otherwise.

Figure 17.8: The DSM $K_M$ in Example 17.10.

This is a computable function, and it remains to verify that it is a reduction from $E_{DSM}$ to REG.

Suppose $\langle M \rangle \in E_{DSM}$. We therefore have that $L(M) = \emptyset$; and by considering the way that $K_M$ behaves we see that $L(K_M) = \emptyset$ as well (because we never get to step 4 if $M$ never accepts). The empty language is regular, and therefore $f(\langle M \rangle) = \langle K_M \rangle \in REG$.

On the other hand, if $M$ is a DSM and $\langle M \rangle \notin E_{DSM}$, then $M$ must accept at least one string. This means that $L(K_M) = \{0^n1^n : n \in \mathbb{N}\}$, because $K_M$ will eventually find a string accepted by $M$, reach step 4, and then accept or reject based on whether the input string $x$ is contained in the nonregular language

\[ \{0^n1^n : n \in \mathbb{N}\}. \quad (17.20) \]

Therefore $f(\langle M \rangle) = \langle K_M \rangle \notin REG$. The remaining case, in which $x$ does not encode a DSM, is straightforward as usual: we have $f(x) = \varepsilon \notin REG$ in this case.

We have shown that $x \in E_{DSM} \iff f(x) \in REG$ holds for every string $x \in \{0, 1\}^*$, and therefore $E_{DSM} \leq_m REG$, as required. We conclude that the language REG is non-semidecidable, as we already know that $E_{DSM}$ is non-semidecidable.
Lecture 18

Further discussion of computability

In this lecture we will discuss a few aspects of decidable and semidecidable languages that were not mentioned in previous lectures. In particular, we will discuss closure properties of these classes of languages and prove a useful alternative characterization of semidecidable languages.

18.1 Closure properties of decidable and semidecidable languages

The decidable and semidecidable languages are closed under many of the operations on languages that we’ve considered thus far in the course (although not all). While we won’t go through every operation we’ve discussed, it is worthwhile to mention some basic examples.

Closure properties of decidable languages

First let us observe that the decidable languages are closed under the regular operations as well as complementation. In short, if $A$ and $B$ are decidable, then there is no difficulty in deciding the languages $A \cup B$, $AB$, $A^*$, and $\overline{A}$ in a straightforward way.

**Proposition 18.1.** Let $\Sigma$ be an alphabet and let $A, B \subseteq \Sigma^*$ be decidable languages. The languages $A \cup B$, $AB$, and $A^*$ are decidable.

**Proof.** Because the languages $A$ and $B$ are decidable, there must exist a DSM $M_A$ that decides $A$ and a DSM $M_B$ that decides $B$. The DSMs described in Figures 18.1, 18.2, and 18.3 decide the languages $A \cup B$, $AB$, and $A^*$, respectively. It follows that these languages are all decidable. \qed
The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. Run $M_A$ on input $w$. If $M_A$ accepts, then accept.
2. Run $M_B$ on input $w$. If $M_B$ accepts, then accept.
3. Reject.

Figure 18.1: A DSM $M$ that decides $A \cup B$, given DSMs $M_A$ and $M_B$ that decide $A$ and $B$, respectively.

The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. For every choice of strings $u, v \in \Sigma^*$ satisfying $w = uv$:
   1.1 Run $M_A$ on input $u$ and run $M_B$ on input $v$.
   1.2 If both $M_A$ and $M_B$ accept, then accept.
2. Reject.

Figure 18.2: A DSM $M$ that decides $AB$, given DSMs $M_A$ and $M_B$ that decide $A$ and $B$, respectively.

The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. If $w = \varepsilon$, then accept.
2. For every way of writing $w = u_1 \cdots u_m$ for nonempty strings $u_1, \ldots, u_m$:
   2.1 Run $M_A$ on each of the strings $u_1, \ldots, u_m$.
   2.2 If $M_A$ accepts all of the strings $u_1, \ldots, u_m$, then accept.
3. Reject.

Figure 18.3: A DSM $M$ that decides $A^*$, given a DSM $M_A$ that decides $A$. 

198
The decidable languages are also closed under complementation, as the next proposition states. Perhaps we don’t even need to bother writing a proof for this one: if a DSM $M$ decides $A$, then one can obtain a new DSM $K$ deciding $\overline{A}$ by defining $K$ so that it runs $M$ on a given string and accepts if and only if $M$ rejects.

**Proposition 18.2.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a decidable language. The language $\overline{A}$ is decidable.

There are a variety of other operations under which the decidable languages are closed. For example, because the decidable languages are closed under union and complementation, we immediately have that they are closed under intersection and symmetric difference. Another example is string reversal: if a language $A$ is decidable, then $A^R$ is also decidable, because a DSM can decide $A^R$ simply by reversing the input string, then deciding whether the string that is obtained is contained in $A$.

There are, however, some natural operations under which the decidable languages are not closed. The following example shows that this is the case for the prefix operation.

**Example 18.3.** The language $\text{Prefix}(A)$ might not be decidable, even if $A$ is decidable. To construct an example that illustrates that this is so, let us first take $B \subseteq \{0, 1\}^*$ to be any language that is semidecidable but not decidable (such as $\text{HALT}$).

Let $M_B$ be a DSM such that $L(M_B) = B$, and define a language $A \subseteq \{0, 1, \#\}^*$ as follows:

\[
A = \{w\#0^t : M_B \text{ accepts } w \text{ within } t \text{ steps}\}. \tag{18.1}
\]

This is a decidable language, but $\text{Prefix}(A)$ is not—for if $\text{Prefix}(A)$ were decidable, then one could easily decide $B$ by using the fact that a string $w \in \{0, 1\}^*$ is contained in $B$ if and only if $w\# \in \text{Prefix}(A)$. (That is, $w \in B$ and $w\# \in \text{Prefix}(A)$ are both equivalent to the existence of a positive integer $t$ such that $w\#0^t \in A$.)

**Closure properties of semidecidable languages**

The semidecidable languages are also closed under a variety of operations, although not precisely the same operations under which the decidable languages are closed.

Let us begin with the regular operations, under which the semidecidable languages are indeed closed. In this case, one needs to be a bit more careful than was sufficient when proving the analogous property for decidable languages, as the stack machines that recognize these languages might run forever.
The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. Set $t \leftarrow 1$.
2. Run $M_A$ on input $w$ for $t$ steps. If $M_A$ accepts $w$ within $t$ steps, then accept.
3. Run $M_B$ on input $w$ for $t$ steps. If $M_B$ accepts $w$ within $t$ steps, then accept.
4. Set $t \leftarrow t + 1$ and goto 2.

Figure 18.4: A DSM $M$ that semidecides $A \cup B$, given DSMs $M_A$ and $M_B$ that semidecide $A$ and $B$, respectively.

The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. Set $t \leftarrow 1$.
2. For every choice of strings $u, v$ satisfying $w = uv$:
   1.1 Run $M_A$ on input $u$ for $t$ steps and run $M_B$ on input $v$ for $t$ steps.
   1.2 If both $M_A$ and $M_B$ have accepted within $t$ steps, then accept.
3. Set $t \leftarrow t + 1$ and goto 2.

Figure 18.5: A DSM $M$ that semidecides $AB$, given DSMs $M_A$ and $M_B$ that semidecide $A$ and $B$, respectively.

**Proposition 18.4.** Let $\Sigma$ be an alphabet and let $A, B \subseteq \Sigma^*$ be semidecidable languages. The languages $A \cup B$, $AB$, and $A^*$ are semidecidable.

**Proof.** Because the languages $A$ and $B$ are semidecidable, there must exist DSMs $M_A$ and $M_B$ such that $L(M_A) = A$ and $L(M_B) = B$. The DSMs described in Figures 18.4, 18.5, and 18.6 semidecide the languages $A \cup B$, $AB$, and $A^*$, respectively. It follows that these languages are all semidecidable.

The semidecidable languages are also closed under intersection. This can be proved through a similar method to closure under union, but in fact this is a situation in which we don’t actually need to be as careful about running forever.
Lecture 18

The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. If $w = \varepsilon$, then accept.
2. Set $t \leftarrow 1$.
3. For every way of writing $w = u_1 \cdots u_m$ for nonempty strings $u_1, \ldots, u_m$:
   3.1 Run $M_A$ on each of the strings $u_1, \ldots, u_m$ for $t$ steps.
   3.2 If $M_A$ accepts all of the strings $u_1, \ldots, u_m$ within $t$ steps, then accept.
4. Set $t \leftarrow t + 1$ and goto 3.

Figure 18.6: A DSM $M$ that semidecides $A^*$, given a DSM $M_A$ that semidecides $A$.

The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. Run $M_A$ on input $w$. If $M_A$ rejects $w$, then reject.
2. Run $M_B$ on input $w$. If $M_B$ rejects $w$, then reject.
3. Accept.

Figure 18.7: A DSM $M$ that semidecides $A \cap B$, given DSMs $M_A$ and $M_B$ that semidecide $A$ and $B$, respectively. Note that if either $M_A$ or $M_B$ runs forever on input $w$, then so does $M$, but this does not change the fact that $M$ semidecides $A \cap B$.

**Proposition 18.5.** Let $\Sigma$ be an alphabet and let $A, B \subseteq \Sigma^*$ be semidecidable languages. The language $A \cap B$ is semidecidable.

**Proof.** Because the languages $A$ and $B$ are semidecidable, there must exist DSMs $M_A$ and $M_B$ such that $L(M_A) = A$ and $L(M_B) = B$. The DSM $M$ described in Figure 18.7 semidecides $A \cap B$, which implies that $A \cap B$ is semidecidable. \qed

It turns out that the semidecidable languages are not closed under complementation. We will be able to conclude this from the following theorem, which is both interesting in its own right and useful in other situations.
The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. Set $t \leftarrow 1$.
2. Run $M_0$ for $t$ steps on input $w$. If $M_0$ accepts within $t$ steps, then accept.
3. Run $M_1$ for $t$ steps on input $w$. If $M_1$ accepts within $t$ steps, then reject.
4. Set $t \leftarrow t + 1$ and goto 2.

Figure 18.8: A DSM $M$ that decides $A$, given DSMs $M_0$ and $M_1$ that semidecide $A$ and $\overline{A}$, respectively.

**Theorem 18.6.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a language such that both $A$ and $\overline{A}$ are semidecidable. The language $A$ is decidable.

**Proof.** Because $A$ and $\overline{A}$ are semidecidable languages, there must exist DSMs $M_0$ and $M_1$ such that $A = L(M_0)$ and $\overline{A} = L(M_1)$. Define a new DSM $M$ as described in Figure 18.8.

Now let us consider the behavior of the DSM $M$ on a given input string $w$. If it is the case that $w \in A$, then $M_0$ eventually accepts $w$, while $M_1$ does not. (It could be that $M_1$ either rejects or runs forever, but it cannot accept $w$.) It is therefore the case that $M$ accepts $w$. On the other hand, if $w \notin A$, then $M_1$ eventually accepts $w$ while $M_0$ does not, and therefore $M$ rejects $w$. Consequently, $M$ decides $A$, so $A$ is decidable. \qed

We know that there exist languages, such as HALT, that are semidecidable but not decidable—so it cannot be that the semidecidable languages are closed under complementation.

Finally, there are some operations under which the semidecidable languages are closed, but under which the decidable languages are not. For example, if $A$ is semidecidable, then so are the languages Prefix($A$), Suffix($A$), and Substring($A$).

### 18.2 The range of a computable function

We will now consider an alternative characterization of semidecidable languages (with the exception of the empty language), which is that they are precisely the languages that are equal to the range of a computable function. Recall that the range of a function $f : \Gamma^* \rightarrow \Sigma^*$ is defined as follows:

$$\text{range}(f) = \{ f(w) : w \in \Gamma^* \}. \quad (18.2)$$
The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. Set $x \leftarrow \epsilon$.
2. Compute $y = f(x)$, and accept if $w = y$.
3. Increment $x$ with respect to the lexicographic ordering of $\Gamma^*$ and goto 2.

Figure 18.9: A DSM $M$ that semidecides $A = \text{range}(f)$ for a computable function $f : \Gamma^* \rightarrow \Sigma^*$.

**Theorem 18.7.** Let $\Sigma$ and $\Gamma$ be alphabets and let $A \subseteq \Sigma^*$ be a nonempty language. The following two statements are equivalent:

1. $A$ is semidecidable.
2. There exists a computable function $f : \Gamma^* \rightarrow \Sigma^*$ such that $A = \text{range}(f)$.

**Proof.** Let us first prove that the second statement implies the first. That is, we will prove that if there exists a computable function $f : \Gamma^* \rightarrow \Sigma^*$ such that $A = \text{range}(f)$, then $A$ is semidecidable. Consider the DSM $M$ described in Figure 18.9. In essence, this DSM searches over $\Gamma^*$ to find a string that $f$ maps to a given input string $w$. If it is the case that $w \in \text{range}(f)$, then $M$ will eventually find $x \in \Gamma^*$ such that $f(x) = w$ and accept, while $M$ will certainly not accept if $w \notin \text{range}(f)$. Thus, we have $L(M) = \text{range}(f) = A$, which implies that $A$ is semidecidable.

Now suppose that $A$ is semidecidable, so that there exists a DSM $M$ such that $L(M) = A$. We will also make use of the assumption that $A$ is nonempty—there exists at least one string in $A$, so we may take $w_0$ to be such a string. (If you like, you may define $w_0$ more concretely as the first string in $A$ with respect to the lexicographic ordering of $\Sigma^*$, but it is not important for the proof that we make this particular choice.) Define a function $f : \Gamma^* \rightarrow \Sigma^*$ as follows:

$$f(x) = \begin{cases} 
  w & \text{if } x = \langle w, 0^t \rangle, \text{ for } w \in \Sigma^* \text{ and } t \in \mathbb{N}, \text{ and } M \text{ accepts } w \text{ within } t \text{ steps} \\
  w_0 & \text{otherwise.}
\end{cases} \quad (18.3)$$

Here we assume that $\langle w, 0^t \rangle$ refers to any encoding scheme through which the strings $w \in \Sigma^*$ and $0^t \in \{0\}^*$ may be encoded into a single string $\langle w, 0^t \rangle \in \Gamma^*$. (As we discussed earlier in the course, this is possible even if $\Gamma$ contains only a single symbol.) It is evident that the function $f$ is computable: a DSM $M_f$ can compute $f$ by checking to see if the input has the form $\langle w, 0^t \rangle$, simulating $M$ for $t$ steps on
input \( w \) if so, and then outputting either \( w \) or \( w_0 \) depending on the outcome. If \( M \) accepts a particular string \( w \), then it must be that \( w = f(\langle w, 0^t \rangle) \) for some sufficiently large natural number \( t \), so \( A \subseteq \text{range}(f) \). On the other hand, every output of \( f \) is either a string \( w \) accepted by \( M \) or the string \( w_0 \), and therefore \( \text{range}(f) \subseteq A \). It therefore holds that \( A = \text{range}(f) \), which completes the proof.

Remark 18.8. The assumption that \( A \) is nonempty is essential in the previous theorem because it cannot be that \( \text{range}(f) = \emptyset \) for a computable function \( f \). Indeed, it cannot be that \( \text{range}(f) = \emptyset \) for any function whatsoever.

Theorem 18.7 provides a useful characterization of semidecidable languages. For instance, you can use this theorem to come up with alternative proofs for all of the closure properties of the semidecidable languages stated in the previous section.

For example, suppose that \( A, B \subseteq \Sigma^* \) are nonempty semidecidable languages. By the theorem above, there must exist computable functions \( f : \Sigma^* \rightarrow \Sigma^* \) and \( g : \Sigma^* \rightarrow \Sigma^* \) such that \( \text{range}(f) = A \) and \( \text{range}(g) = B \). Define a new function \( h : (\Sigma \cup \{\#\})^* \rightarrow \Sigma^* \) as follows:

\[
g(x) = \begin{cases} f(y)f(z) & \text{if } x = y\#z \text{ for } y, z \in \Sigma^* \\ f(\varepsilon)g(\varepsilon) & \text{otherwise.} \end{cases} \tag{18.4}
\]

(We are assuming \( \# \notin \Sigma \), but if it were, \( \# \) could of course be replaced by any other choice of a symbol that is not contained in \( \Sigma \).) One sees that \( h \) is computable, and \( \text{range}(h) = AB \), which implies that \( AB \) is semidecidable.

Here is another example of an application of Theorem 18.7.

Corollary 18.9. Let \( \Sigma \) be an alphabet and let \( A \subseteq \Sigma^* \) be any infinite semidecidable language. There exists an infinite decidable language \( B \subseteq A \).

Proof. Because \( A \) is infinite (and therefore nonempty), there exists a computable function \( f : \Sigma^* \rightarrow \Sigma^* \) such that \( A = \text{range}(f) \).

We will define a language \( B \) by first defining a DSM \( M \) and then taking \( B = L(M) \). In order for us to be sure that \( B \) satisfies the requirements of the corollary, it will need to be proved that \( M \) never runs forever (so that \( B \) is decidable), that \( M \) only accepts strings that are contained in \( A \) (so that \( B \subseteq A \)), and that \( M \) accepts infinitely many different strings (so that \( B \) is infinite). The DSM \( M \) is described in Figure 18.10.

The fact that \( M \) never runs forever follows from the assumption that \( A \) is infinite. That is, because \( A \) is infinite, the function \( f \) must output infinitely many different strings, so regardless of what input string \( w \) is input into \( M \), the loop will
Lecture 18

The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. Set $x \leftarrow \epsilon$.
2. Compute $y \leftarrow f(x)$.
3. If $y = w$ then accept.
4. If $y > w$ (with respect to the lexicographic ordering of $\Sigma^*$) then reject.
5. Increment $x$ with respect to the lexicographic ordering of $\Sigma^*$ and goto 2.

Figure 18.10: A DSM $M$ for Corollary 18.9.

eventually reach a string $x$ so that $f(x) = w$ or $f(x) > w$, either of which causes $M$ to halt.

The fact that $M$ only accepts strings in $A$ follows from the fact that the condition for acceptance is that the input string $w$ is equal to $y$, which is contained in $\text{range}(f) = A$.

Finally, let us observe that $M$ accepts precisely the strings in this set:

$$\left\{ w \in \Sigma^* : \text{there exists } x \in \Sigma^* \text{ such that } w = f(x) \text{ and } w > f(z) \text{ for all } z < x \right\}. \quad (18.5)$$

The fact that this set is infinite follows from the assumption that $A = \text{range}(f)$ is infinite—for if the set were finite, there would necessarily be a maximal output of $f$ with respect to the lexicographic ordering of $\Sigma^*$, contradicting the assumption that $\text{range}(f)$ is infinite.

The language $B = L(M)$ therefore satisfies the requirements of the corollary, which completes the proof. \hfill \square
Lecture 19

Time-bounded computations

In the final few lectures of the course, we will discuss the topic of computational complexity theory, which is concerned with the inherent difficulty (or hardness) of computational problems and the effect of resource constraints on models of computation. We will only have time to scratch the surface; complexity theory is a rich subject, and many researchers around the world are engaged in a study of this field. Unlike formal language theory and computability theory, many of the central questions of complexity theory remain unanswered to this day.

In this lecture we will focus on the most important resource (from the view of computational complexity theory), which is time. The motivation is, in some sense, obvious: in order to be useful, computations generally need to be performed within a reasonable amount of time. In an extreme situation, if we have some computational task that we would like to have performed, and someone gives us a computational device that will perform this computational task, but only after running for one million years or more, it is practically useless. One can of course consider other resources besides time, such as space (or memory usage), communication in a distributed scenario, or a variety of more abstract notions concerning resource usage.

We will start with a definition of the running time of a DSM, assuming that the DSM never runs forever.

**Definition 19.1.** Let $M$ be a DSM with input alphabet $\Sigma$ that halts on every input. For each string $w \in \Sigma^*$, let $T(w)$ denote the number of steps for which $M$ runs on input $w$. The running time of $M$ is the function $t : \mathbb{N} \rightarrow \mathbb{N}$ defined as

$$t(n) = \max\{T(w) : w \in \Sigma^*, |w| = n\} \quad (19.1)$$

for every $n \in \mathbb{N}$. In words, $t(n)$ is the maximum number of steps required for $M$ to halt, over all input strings of length $n$. 

207
19.1 DTIME and time-constructible functions

Deterministic time complexity classes

For every function $f : \mathbb{N} \to \mathbb{N}$, we define a class of languages called $\text{DTIME}(f)$, which represents those languages decidable in time $O(f(n))$.

**Definition 19.2.** Let $f : \mathbb{N} \to \mathbb{N}$ be a function. A language $A$ is contained in the class $\text{DTIME}(f)$ if there exists a DSM $M$ that decides $A$ and whose running time $t$ satisfies $t(n) = O(f(n))$.

We define $\text{DTIME}(f)$ in this way, using $O(f(n))$ rather than $f(n)$, because we are generally not interested in constant factors or in what might happen in finitely many special cases. One fact that motivates this choice is that it is usually possible to “speed up” a DSM by defining a new DSM, having a larger stack alphabet than the original, that succeeds in simulating multiple computation steps of the original DSM with each step it performs.

When it is reasonable to do so, we generally reserve the variable name $n$ to refer to the input length for whatever language or DSM we are considering. So, for example, we may refer to a DSM that runs in time $O(n^2)$ or refer to the class of languages $\text{DTIME}(n^2)$ with the understanding that we are speaking of the function $f(n) = n^2$, without explicitly saying that $n$ is the input length.

**Example 19.3.** The language $\{w\#w : w \in \{0, 1\}^*\}$ is contained in $\text{DTIME}(n)$. The example of a DSM for deciding this language from Lecture 12, for instance, runs in time $O(n)$ on inputs of length $n$.

We also sometimes refer to classes such as

$$\text{DTIME}(n^{\sqrt{n}}) \quad \text{or} \quad \text{DTIME}(n^2 \log(n)), \quad (19.2)$$

where the function $f$ that we are implicitly referring to appears to take non-integer values for some choices of $n$. This is done in an attempt to keep the expressions of these classes simple and intuitive, and you can interpret these things as referring to functions of the form $f : \mathbb{N} \to \mathbb{N}$ obtained by rounding up to the next nonnegative integer. For instance, $\text{DTIME}(n^2 \log(n))$ means $\text{DTIME}(f)$ for

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \lceil n^2 \log(n) \rceil & \text{otherwise.} \end{cases} \quad (19.3)$$
Time-constructible functions

The complexity class \( \text{DTIME}(f) \) has been defined for an arbitrary function of the form \( f : \mathbb{N} \to \mathbb{N} \), but there is a sense in which most functions of this form are uninteresting from the viewpoint of computational complexity—because they have absolutely nothing to do with the running time of any DSM.

There are, in fact, some choices of functions \( f : \mathbb{N} \to \mathbb{N} \) that are so strange that they lead to highly counter-intuitive results. For example, there exists a function \( f \) such that

\[
\text{DTIME}(f) = \text{DTIME}(g), \quad \text{for } g(n) = 2^{f(n)}; \quad (19.4)
\]

even though \( g \) is exponentially larger than \( f \), they both result in exactly the same deterministic time complexity class. This doesn’t necessarily imply anything important about time complexity, it’s more a statement about the strangeness of the function \( f \).

For this reason we define a collection of functions, called \textit{time-constructible functions}, that represent well-behaved upper bounds on the possible running times of DSMs. Here is a precise definition.

\textbf{Definition 19.4.} Let \( f : \mathbb{N} \to \mathbb{N} \) be a function satisfying \( f(n) = \Omega(n) \). The function \( f \) is said to be \textit{time constructible} if there exists a DSM \( M \) that operates as follows:

1. On each input \( 0^n \) the DSM \( M \) outputs \( 0^{f(n)} \), for every \( n \in \mathbb{N} \). In other words, \( M \) computes the function \( f \) with respect to unary notation.
2. \( M \) runs in time \( O(f(n)) \).

It might not be clear why we would define a class of functions in this particular way, but the essence is that these are functions that can serve as upper bounds for DSM computations. That is, a DSM can compute \( f(n) \) on any input of length \( n \), and doing this doesn’t take more than \( O(f(n)) \) steps—and then it has the number \( f(n) \) stored in unary notation so that it can then use this number to limit some subsequent part of its computation (perhaps the number of steps for which it runs during a second phase of its computation).

As it turns out, just about any reasonable function \( f \) with \( f(n) = \Omega(n) \) that you are likely to care about as a bound on running time is time constructible. Examples include the following:

1. For any choice of an integer \( k \geq 1 \), the function \( f(n) = n^k \) is time constructible.
2. For any choice of an integer \( k \geq 2 \), the function \( f(n) = k^n \) is time constructible.
The DSM $K$ operates as follows on input $w \in \{0, 1\}^*$:

1. If the input $w$ does not take the form $w = \langle M \rangle 01^k$ for a DSM $M$ with input alphabet $\{0, 1\}$ and $k \in \mathbb{N}$, then reject.
2. Compute $t = f(|w|)$.
3. Simulate $M$ on input $w$ for $t$ steps. If $M$ has rejected $w$ within $t$ steps, then accept, otherwise reject.

Figure 19.1: This DSM decides a language that cannot be decided in time $o(f(n))$.

3. For any choice of an integer $k \geq 1$, the functions

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \lceil n^k \log(n) \rceil & \text{otherwise} \end{cases} \quad (19.5)$$

and

$$f(n) = \lceil n^k \sqrt{n} \rceil \quad (19.6)$$

are time constructible.

4. If $f$ and $g$ are time-constructible functions, then the functions

$$h_1(n) = f(n) + g(n), \quad h_2(n) = f(n)g(n), \quad \text{and} \quad h_3(n) = f(g(n)) \quad (19.7)$$

are also time constructible.

### 19.2 The time-hierarchy theorem

What we will do next is to discuss a fairly intuitive theorem concerning time complexity. A highly informal statement of the theorem is this: more languages can be decided with more time. While this is indeed an intuitive idea, it is not obvious how a formal version of this statement is to be proved. We will begin with a somewhat high-level discussion of how the theorem is proved, and then state the strongest-known form of the theorem (without going through the low-level details needed to obtain the stronger form).

Suppose that a time-constructible function $f : \mathbb{N} \to \mathbb{N}$ has been selected, and define a DSM $K$ as described in Figure 19.1. It is not immediately clear what the running time is for $K$, because this depends on precisely how the simulation of $M$ is done; different ways of performing the simulation could of course lead to different
running times. For the time being, let us take \( g : \mathbb{N} \to \mathbb{N} \) to be the running time of \( K \), and we’ll worry later about how specifically \( g \) relates to \( f \).

Next, let us think about the language \( L(K) \) decided by \( K \). This is a language over the binary alphabet, and it is obvious that \( L(K) \in \text{DTIME}(g) \), because \( K \) itself is a DSM that decides \( L(K) \) in time \( g(n) \). What we will show is that \( L(K) \) cannot possibly be decided by a DSM that runs in time \( o(f(n)) \).

To this end, assume toward contradiction that there does exist a DSM \( M \) that decides \( L(K) \) in time \( o(f(n)) \). Because the running time of \( M \) is \( o(f(n)) \), we know that there must exist a natural number \( n_0 \) such that, for all \( n \geq n_0 \), the DSM \( M \) halts on all inputs of length \( n \) in strictly fewer than \( f(n) \) steps. Choose \( k \) to be large enough so that the string \( w = \langle M \rangle 01^k \) satisfies \( |w| \geq n_0 \), and (as always) let \( n = |w| \). Because \( M \) halts on input \( w \) after fewer than \( f(n) \) steps, we find that

\[
w \in L(K) \iff w \notin L(M).
\]  

(19.8)

The reason is that \( K \) simulates \( M \) on input \( w \), it completes the simulation because \( M \) runs for fewer than \( f(n) \) step, and it answers opposite to the way \( M \) answers (i.e., if \( M \) accepts, then \( K \) rejects; and if \( M \) rejects, then \( K \) accepts). This contradicts the assumption that \( M \) decides \( L(K) \). We conclude that no DSM whose running time is \( o(f(n)) \) can decide \( L(K) \).

It is natural to wonder what the purpose is for taking the input to \( K \) to have the form \( \langle M \rangle 01^k \), as opposed to just \( \langle M \rangle \) (for instance). The reason is pretty simple: it’s just a way of letting the length of the input string grow, so that the asymptotic behavior of the function \( f \) and the running time of \( M \) take over (even though we’re really interested in fixed choices of \( M \)). If we were to change the language, so that the input takes the form \( w = \langle M \rangle \) rather than \( \langle M \rangle 01^k \), we would have no way to guarantee that \( K \) is capable of finishing the simulation of \( M \) on input \( \langle M \rangle \) within \( f(|\langle M \rangle|) \) steps—for it could be that the running time of \( M \) on input \( \langle M \rangle \) exceeds \( f(|\langle M \rangle|) \) steps, even though the running time of \( M \) is small compared with \( f \) for significantly longer input strings.

What we have proved is that, for any choice of a time-constructible function \( f : \mathbb{N} \to \mathbb{N} \), the proper subset relation

\[
\text{DTIME}(h) \subsetneq \text{DTIME}(g)
\]

holds whenever \( h(n) = o(f(n)) \), where \( g \) is the running time of \( K \) (which depends somehow on \( f \)).

**Remark 19.5.** There is an aspect of the argument just presented that is worth noting. We obtained the language \( L(K) \), which is contained in \( \text{DTIME}(g) \) but not \( \text{DTIME}(h) \) assuming \( h(n) = o(f(n)) \), not by actually describing the language explicitly, but by simply describing the DSM \( K \) that decides it. Indeed, in this case it
is hard to imagine a description of the language \( L(K) \) that would be significantly more concise than the description of \( K \) itself. This technique can be useful in other situations. Sometimes, when you wish to prove the existence of a language having a certain property, rather than explicitly defining the language, it is possible to define a DSM \( M \) that operates in a particular way, and then take the language you are looking for to be \( L(M) \).

If you work very hard to make \( K \) run as efficiently as possible, the following theorem can be obtained.

**Theorem 19.6** (Time-hierarchy theorem). If \( f, g : \mathbb{N} \to \mathbb{N} \) are time-constructible functions for which \( f(n) = o(g(n)/\log(g(n))) \), then

\[
\text{DTIME}(f) \subsetneq \text{DTIME}(g).
\]

The main reason that we will not go through the details required to prove this theorem is that optimizing \( K \) to simulate a given DSM as efficiently as possible gets very technical. For the sake of this course, it is enough that you understand the basic idea of the proof. In particular, notice that it is another example of a proof that uses the diagonalization technique; while it is a bit more technical, it has a very similar flavor to the proof that \( \text{DIAG} \) is not semidecidable, and to the proof that \( \mathcal{P}(\mathbb{N}) \) is uncountable.

From the time-hierarchy theorem, one can conclude the following down-to-earth corollary.

**Corollary 19.7.** For all \( k \in \mathbb{N} \) with \( k \geq 1 \), it is the case that

\[
\text{DTIME}(n^k) \subsetneq \text{DTIME}(n^{k+1}).
\]

### 19.3 Polynomial and exponential time

We'll finish off the lecture by introducing a few important notions based on deterministic time complexity. First, let us define two complexity classes, known as \( \mathcal{P} \) and \( \text{EXP} \), as follows:

\[
P = \bigcup_{k \geq 1} \text{DTIME}(n^k) \quad \text{and} \quad \text{EXP} = \bigcup_{k \geq 1} \text{DTIME}(2^{n^k}).
\]

In words, a language \( A \) is contained in the complexity class \( \mathcal{P} \) if there exists a DSM \( M \) that decides \( A \) and has *polynomial running time*, meaning a running time that is \( O(n^k) \) for some fixed choice of \( k \geq 1 \); and a language \( A \) is contained in the
complexity class $\text{EXP}$ if there exists a DSM $M$ that decides $A$ and has exponential running time, meaning a running time that is $O(2^{n^k})$ for some fixed choice of $k \geq 1$.

As a very rough but nevertheless useful simplification, we often view the class $P$ as representing languages that can be efficiently decided by a DSM, while $\text{EXP}$ contains languages that are decidable by a brute force approach. These are undoubtedly over-simplifications in some respects, but for languages that correspond to “natural” computational problems that arise in practical settings, this is a reasonable picture to keep in mind.

By the time-hierarchy theorem, we can conclude that $P \subsetneq \text{EXP}$. In particular, if we take $f(n) = 2^n$ and $g(n) = 2^{2n}$, then the time hierarchy theorem establishes the middle (proper) inclusion in this expression:

$$P \subseteq \text{DTIME}(2^n) \subsetneq \text{DTIME}(2^{2n}) \subseteq \text{EXP}.$$  \hspace{1cm} (19.13)

There are many examples of languages contained in the class $P$. If we restrict our attention to languages we have discussed thus far in the course, we may say the following.

- The languages $A_{\text{DFA}}$, $E_{\text{DFA}}$, $E_{\text{EQDFA}}$, and $E_{\text{CFG}}$ from Lecture 15 are all certainly contained in $P$; if you analyzed the running times of the DSMs we described for those languages, you would find that they run in polynomial time.

- The languages $A_{\text{NFA}}$, $A_{\text{REX}}$, $E_{\text{NFA}}$, and $E_{\text{REX}}$ are also in $P$, but the DSMs we described for these languages in Lecture 15 do not actually show this. Those DSMs have exponential running time, because the conversion of an NFA or regular expression to a DFA could result in an exponentially large DFA. It is, however, not too hard to decide these languages in polynomial time through different methods.

  In particular, we can decide $A_{\text{NFA}}$ in polynomial time through a more direct simulation in which we keep track of the set of all states that a given NFA could be in when reading a given input string, and we can decide $A_{\text{REX}}$ by performing a polynomial-time conversion of a given regular expression into an equivalent NFA, effectively reducing the problem in polynomial time to $A_{\text{NFA}}$. The language $E_{\text{NFA}}$ can be decided in polynomial time by treating it as a graph reachability problem, and $E_{\text{REX}}$ can be reduced to $E_{\text{NFA}}$ in polynomial time.

- The language $A_{\text{CFG}}$ is also contained in $P$, but once again, the DSM for this language that we discussed in Lecture 15 does not establish this. A more sophisticated approach based on the algorithmic technique of dynamic programming does, however, allow one to decide $A_{\text{CFG}}$ in polynomial time. This fact allows one to conclude that every context-free language is contained in $P$.  


There does not currently exist a proof that the languages $\text{EQ}_{\text{NFA}}$ and $\text{EQ}_{\text{REX}}$ fall outside of the class $P$, but this is conjectured to be the case. This is because these languages are complete for the class $\text{PSPACE}$ of languages that are decidable within a polynomial amount of space. If $\text{EQ}_{\text{NFA}}$ and $\text{EQ}_{\text{REX}}$ are in $P$, then it would then follow that $P = \text{PSPACE}$, which seems highly unlikely. It is the case, however, that $\text{EQ}_{\text{NFA}}$ and $\text{EQ}_{\text{REX}}$ are contained in $\text{EXP}$, for in exponential time one can afford to perform a conversion of NFAs or regular expressions to DFAs and then test the equivalence of the two (possibly exponential size) DFAs in the same way that we considered earlier.

Finally, let us observe that one may consider not only languages that are decided by DSMs having bounded running times, but also functions that can be computed by time-bounded DSMs. It will be enough for the purposes of the remaining lectures of this course to consider the class of polynomial-time computable functions, which are functions that can be computed by a DSM with running time $O(n^k)$ for some fixed positive integer $k$. An algorithms course such as CS 341 discusses numerous practical examples of polynomial-time computable functions.\footnote{Algorithms courses generally consider computational models that represent machines having random access memory, as opposed to stack machines. However, because a stack machine can simulate such a model with no more than a polynomial slowdown, the class of polynomial-time computable functions is the same for the two types of models.}
Lecture 20

NP, polynomial-time mapping reductions, and NP-completeness

In the previous lecture we discussed deterministic time complexity, along with the time-hierarchy theorem, and introduced two complexity classes: P and EXP. In this lecture we will introduce another complexity class, called NP, and study its relationship to P and EXP. In addition, we will define a polynomial-time variant of mapping reductions along with the notion of completeness for the class NP.

20.1 The complexity class NP

There are two equivalent ways to define the complexity class NP that we will cover in this lecture. The first way is arguably more intuitive and more closely connected with the way in which the complexity class NP is typically viewed. The second way directly connects the class with the notion of nondeterminism, and leads to a more general notion (which we will not have time to explore further).

NP as certificate verification

We adopt the following definition as our principal definition for the complexity class NP. In this definition, and throughout the remainder of this lecture, when we refer to a *polynomially bounded time-constructible function* $f$, we mean a time-constructible function $f$ for which $f(n) = O(n^k)$ for some fixed positive integer $k$.

**Definition 20.1.** Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language $A$ is contained in NP if there exists a polynomially bounded time-constructible function $f$, an alphabet $\Gamma$ with $\# \notin \Gamma$, and a language $B \in P$ such that

$$A = \{ x \in \Sigma^* : \text{there exists } y \in \Gamma^* \text{ such that } |y| \leq f(|x|) \text{ and } x\#y \in B \}. \quad (20.1)$$
The essential idea that this definition expresses is that \( A \in \text{NP} \) means that membership in \( A \) is efficiently verifiable. The string \( y \) in the definition plays the role of a proof that a string \( x \) is contained \( A \), while the language \( B \) represents an efficient verification procedure that checks the validity of this proof of membership for \( x \). The terms certificate and witness are alternatives (to the term proof) that are often used to describe the string \( y \).

NP as polynomial-time nondeterministic computations

As suggested above, there is a second way to define the class \( \text{NP} \) that is equivalent to the way just described. The way that this definition works is that we first define \( \text{NTIME}(f) \), for every function \( f : \mathbb{N} \to \mathbb{N} \), to be the complexity class of all languages that are decided by a nondeterministic stack machine running in time \( O(f(n)) \). To say that a nondeterministic stack machine \( M \) runs in time at most \( t(n) \) means that for every nonnegative integer \( n \in \mathbb{N} \), every input string \( w \in \Sigma^* \) with \( |w| = n \), and every possible sequence of transitions that \( M \) can make on the input \( w \), it is the case that \( M \) halts within \( t(n) \) steps. In other words, running time for nondeterministic stack machines is understood to refer to the worst-case running time, taken over all possible nondeterministic choices the machine can make.

Once we have defined \( \text{NTIME}(f) \) for every function \( f : \mathbb{N} \to \mathbb{N} \), we define \( \text{NP} \) in a similar way to what we did for \( \text{P} \):

\[
\text{NP} = \bigcup_{k \geq 1} \text{NTIME}(n^k).
\]  

(20.2)

This definition is where \( \text{NP} \) gets its name: \( \text{NP} \) is short for nondeterministic polynomial time.

Equivalence of the two definitions of \( \text{NP} \)

The equivalence of the two definitions of \( \text{NP} \) just suggested is not too difficult to establish, but we won’t go through it in detail. The basic ideas needed to prove the equivalence are (i) a nondeterministic stack machine can first guess a polynomial-length string \( y \) over an alphabet \( \Gamma \) and then verify in polynomial time that \( x \# y \in B \), and (ii) a polynomial-length string \( y \in \Gamma^* \) can represent the nondeterministic moves of a polynomial-time nondeterministic stack machine, and it could then be verified deterministically in polynomial time that this sequence of nondeterministic moves would lead the original NSM to acceptance.
Relationships among $P$, $NP$, and $EXP$

Let us now observe the following inclusions:

$$P \subseteq NP \subseteq EXP. \quad (20.3)$$

The first of these inclusions, $P \subseteq NP$, is straightforward. Suppose $A \subseteq \Sigma^*$ is a language over an alphabet $\Sigma$, and assume $A \in P$. We may then define a language $B$ as follows:

$$B = \{ x\# : x \in A \}. \quad (20.4)$$

It is evident that $B \in P$; if we have a DSM $M_A$ that decides $A$ in polynomial time, we can easily decide $B$ in polynomial time by deleting the symbol # from the right-hand side of the input string $x\#$, and then running $M_A$ on the resulting string $x$. For $f(n) = n$, or any other polynomially bounded, time-constructible function, it is the case that

$$A = \{ x \in \Sigma^* : \text{there exists } y \in \Gamma^* \text{ such that } |y| \leq f(|x|) \text{ and } x\#y \in B \}, \quad (20.5)$$

with $y = \epsilon$ being the only choice of $y$ that actually fulfills the required condition, and therefore $A \in NP$.

With respect to the alternative definition of $NP$, as the union of $NTIME(n^k)$ over all $k \geq 1$, the inclusion $P \subseteq NP$ is also straightforward (or perhaps obvious). Every DSM is equivalent to an NSM that happens to obey restrictions that make it deterministic, so

$$DTIME(f) \subseteq NTIME(f) \quad (20.6)$$

for all functions $f : \mathbb{N} \to \mathbb{N}$. This implies that

$$DTIME(n^k) \subseteq NTIME(n^k) \quad (20.7)$$

for all $k \geq 1$, and therefore $P \subseteq NP$.

Now let us observe that $NP \subseteq EXP$. Suppose $A \subseteq \Sigma^*$ is language over an alphabet $\Sigma$, and assume $A \in NP$. This implies that there exists a polynomially bounded time-constructible function $f$ and a language $B \in P$ such that

$$A = \{ x \in \Sigma^* : \text{there exists } y \in \Sigma^* \text{ such that } |y| \leq f(|x|) \text{ and } x\#y \in B \}. \quad (20.8)$$

Define a DSM $M$ as described in Figure 20.1.

It is evident that $M$ decides $A$, as it simply searches over the set of all strings $y \in \Gamma^*$ with $|y| \leq f(|x|)$ to find if there exists one such that $x\#y \in B$. It remains to consider the running time of $M$.

Let us first consider step 2, in which $M$ tests whether $x\#y \in B$ for an input string $x \in \Sigma^*$ and a string $y \in \Gamma^*$ satisfying $|y| \leq f(|x|)$. This test takes a number
The DSM $M$ operates as follows on input $x \in \Sigma^*$:

1. Set $y \leftarrow \varepsilon$.
2. If $x \# y \in B$, then accept.
3. Increment $y$ with respect to the lexicographic ordering of $\Gamma^*$.
4. If $|y| > f(|x|)$ then reject, else goto 2.

Figure 20.1: A DSM $M$ that decides a given NP-language in exponential time.

of steps that is polynomial in $|x|$, and the reason why is as follows. First, we have $|y| \leq f(|x|)$, and therefore the length of the string $x \# y$ is polynomially bounded (in the length of $x$). Now, because $B \in P$, we have that membership in $B$ can be tested in polynomial time. Because the input in this case is $x \# y$, this means that the time required to test membership in $B$ is polynomial in $|x \# y|$. However, because the composition of two polynomially bounded functions is another polynomially bounded function, we find that the time required to test whether $x \# y \in B$ is polynomial in the length of $x$. Step 3 can also be performed in time polynomial in the length of $x$, as can the test $|y| > f(|x|)$ in step 4.

Finally, again using the assumption that $f$ is polynomially bounded, so that $f(n) = O(n^k)$ for some positive integer $k$, we find that the total number of times the steps just considered are executed is at most

$$|\Gamma|^{f(n)+1} - 1 = O\left(2^{n^{k+1}}\right).$$  \hspace{1cm} (20.9)

Using the rather coarse upper-bound that every polynomially bounded function $g$ satisfies $g(n) = O(2^n)$, we find that the entire computation runs in time

$$O\left(2^{n^{k+2}}\right).$$ \hspace{1cm} (20.10)

We have established that $M$ runs in exponential time, so $A \in \text{EXP}$.

Now we know that

$$P \subseteq \text{NP} \subseteq \text{EXP},$$ \hspace{1cm} (20.11)

and we also know that

$$P \subset \text{EXP}$$ \hspace{1cm} (20.12)

by the time-hierarchy theorem. Of course this means that one (or both) of the following proper containments must hold: (i) $P \subset \text{NP}$, or (ii) $\text{NP} \subset \text{EXP}$. Neither one
has yet been proved, and a correct proof of either one would be a major breakthrough in complexity theory. Indeed, determining whether or not \( P = NP \) is viewed by many as being among the greatest unsolved mathematical challenges of our time.

### 20.2 Polynomial-time reductions and NP-completeness

We discussed reductions in Lecture 17 and used them to prove that certain languages are undecidable or non-semidecidable. *Polynomial-time reductions* are defined similarly, except that we add the condition that the reductions themselves must be given by polynomial-time computable functions.

**Definition 20.2.** Let \( \Sigma \) and \( \Gamma \) be alphabets and let \( A \subseteq \Sigma^* \) and \( B \subseteq \Gamma^* \) be languages. It is said that \( A \) *polynomial-time reduces* to \( B \) if there exists a polynomial-time computable function \( f : \Sigma^* \rightarrow \Gamma^* \) such that

\[
w \in A \iff f(w) \in B \tag{20.13}
\]

for all \( w \in \Gamma^* \). One writes

\[
A \leq_p B \tag{20.14}
\]

to indicate that \( A \) polynomial-time reduces to \( B \), and any function \( f \) that establishes that this is so may be called a *polynomial-time reduction* from \( A \) to \( B \).

Polynomial-time reductions of this form are sometimes called *polynomial-time mapping reductions* (and also *polynomial-time many-to-one reductions*) to differentiate them from other types of reductions that we will not consider—but we will stick with the term *polynomial-time reductions* for simplicity. They are also sometimes called *Karp reductions*, named after Richard Karp, one of the pioneers of the theory of NP-completeness.

With the definition of polynomial-time reductions in hand, we can now define NP-completeness. You may already be familiar with this notion from another course, such as CS 341, which discusses NP-completeness in depth.

**Definition 20.3.** Let \( \Sigma \) be an alphabet and let \( B \subseteq \Sigma^* \) be a language.

1. It is said that \( B \) is NP-hard if \( A \leq_p B \) for every language \( A \in \text{NP} \).
2. It is said that \( B \) is NP-complete if \( B \) is NP-hard and \( B \in \text{NP} \).
The idea behind this definition is that the NP-complete languages represent the hardest languages to decide in NP; every language in NP can be polynomial-time reduced to an NP-complete language, so if we view the difficulty of performing a polynomial-time reduction as being negligible, the ability to decide any one NP-complete language would give us a key to unlocking the computational difficulty of the class NP in its entirety. Figure 20.2 illustrates the relationship among the classes P and NP, and the NP-hard and NP-complete languages, under the assumption that P $\neq$ NP.

Now, it is not at all obvious from the definition that there should exist any NP-complete languages at all, for it is a strong condition that every language in NP must polynomial-time reduce to such a language. There are, in fact, thousands of known NP-complete languages that correspond to natural computational problems of interest, and many examples are discussed in CS 341.

We will finish off the lecture by listing several properties of polynomial-time reductions, NP-completeness, and related concepts. For all of these facts, it is to be assumed that $A$, $B$, and $C$ are languages.

1. If $A \leq^p_m B$ and $B \leq^p_m C$, then $A \leq^p_m C$.
2. If $A \leq^p_m B$ and $B \in P$, then $A \in P$.
3. If $A \leq^p_m B$ and $B \in NP$, then $A \in NP$.
4. If $A$ is NP-hard and $A \leq^p_m B$, then $B$ is NP-hard.
5. If \( A \) is NP-complete, \( B \in \text{NP} \), and \( A \leq_p m B \), then \( B \) is NP-complete.

6. If \( A \) is NP-hard and \( A \in \text{P} \), then \( \text{P} = \text{NP} \).

We typically use statement 5 when we wish to prove that a certain language \( B \) is NP-complete: we first prove that \( B \in \text{NP} \) (which is often easy) and then look for a known NP-complete language \( A \) for which we can prove \( A \leq_p m B \).

The proofs of the statements listed above are all fairly straightforward, and you might try proving them for yourself if you are interested. Let us pick just one of the statements and prove it.

**Proposition 20.4.** Let \( A \subseteq \Sigma^* \) and \( B \subseteq \Gamma^* \) be languages, for alphabets \( \Sigma \) and \( \Gamma \), and assume \( A \leq_p m B \) and \( B \in \text{NP} \). It is the case that \( A \in \text{NP} \).

**Proof.** Let us begin by gathering some details concerning the assumptions of the proposition.

First, because \( A \leq_p m B \), we know that there exists a polynomial-time computable function \( f : \Sigma^* \rightarrow \Gamma^* \) such that
\[
x \in A \iff f(x) \in B
\] (20.15)
for all \( x \in \Sigma^* \). Because \( f \) is polynomial-time computable, there must exist a polynomially bounded time-constructible function \( g \) such that \( |f(x)| \leq g(|x|) \) for all \( x \in \Sigma^* \).

Second, by the assumption that \( B \in \text{NP} \), there must exist an alphabet \( \Delta \) (which does not include \#), a polynomially bounded time-constructible function \( h \), and a language \( C \in \text{P} \) for which
\[
B = \{ x \in \Gamma^* : \text{there exists } y \in \Delta^* \text{ such that } |y| \leq h(|x|) \text{ and } x#y \in C \}.
\] (20.16)

Now, define a new language
\[
D = \{ x#y : f(x)#y \in C \}.
\] (20.17)
It is evident that \( D \in \text{P} \) because one may simply compute \( f(x)#y \) from \( x#y \) in polynomial time (given that \( f \) is polynomial-time computable), and then test if \( f(x)#y \in C \), which requires polynomial time because \( C \in \text{P} \).

Finally, observe that
\[
A = \left\{ x \in \Sigma^* : \text{there exists } y \in \Delta^* \text{ such that } |y| \leq h(g(|x|)) \text{ and } x#y \in D \right\}.
\] (20.18)
As the composition \( h \circ g \) is a polynomially bounded time-constructible function and \( D \in \text{P} \), it follows that \( A \in \text{NP} \), as required.

Let us conclude the lecture with the following corollary, which is meant to be a fun application of the previous proposition along with the time-hierarchy theorem.
The DSM $M$ operates as follows on input $w \in \Sigma^*$:

1. If $w$ does not take the form $w = xx$ for some string $x \in \Sigma^*$, then reject.
2. Accept if $x \in A$, otherwise reject.

Figure 20.3: The DSM $M$ from the proof of Corollary 20.5.

**Corollary 20.5.** $\text{NP} \neq \text{DTIME}(2^n)$.

**Proof.** Assume toward contradiction that $\text{NP} = \text{DTIME}(2^n)$. Let $\Sigma$ be any alphabet, let $A \subseteq \Sigma^*$ be an arbitrarily chosen language in $\text{DTIME}(4^n)$, and define

$$B = \{xx : x \in A\}.$$  

First we observe that $B \in \text{DTIME}(2^n)$. In particular, the DSM $M$ described in Figure 20.3 decides $B$ in time $O(2^n)$. The reason why $M$ runs in time $O(2^n)$ is as follows: the first step can easily be performed in polynomial time, and the second step requires $O(4^n/2) = O(2^n)$ steps, as $A$ can be decided in time $O(4^n)$ on inputs of length $m$, and here we are deciding membership in $A$ on a string of length $m = n/2$. The running time of $M$ is therefore $O(2^n)$. As we have assumed that $\text{NP} = \text{DTIME}(2^n)$, it follows that $B \in \text{NP}$.

Now define a function $f : \Sigma^* \rightarrow \Sigma^*$ as

$$f(x) = xx$$

for all $x \in \Sigma^*$. The function $f$ can easily be computed in polynomial time, and it is immediate from the definition of $B$ that

$$x \in A \iff f(x) \in B.$$  

We therefore have that $A \leq_{m} B$. By Proposition 20.4, it follows that $A \in \text{NP}$, and given the assumption $\text{NP} = \text{DTIME}(2^n)$, it follows that $A \in \text{DTIME}(2^n)$.

However, as $A$ was an arbitrarily chosen language in $\text{DTIME}(4^n)$, we conclude that $\text{DTIME}(4^n) \subseteq \text{DTIME}(2^n)$. This contradicts the time hierarchy theorem, so our assumption $\text{NP} = \text{DTIME}(2^n)$ was incorrect. \[\square\]
Lecture 21

Ladner’s theorem

The purpose of this lecture is to prove a theorem called Ladner’s theorem. This theorem establishes, under the assumption that $P \neq NP$, that there exist languages inside NP that are neither in P nor are they NP-complete.

**Theorem 21.1 (Ladner’s theorem).** *If $P \neq NP$, then there exists a language $A \in NP$ such that*

1. $A \notin P$, and
2. $A$ is not NP-complete.

Throughout the remainder of this lecture, we will take $\Sigma = \{0, 1\}$ to be the binary alphabet for simplicity. We could consider other alphabets besides this one, but it is enough to restrict our attention to the binary alphabet in order to prove Theorem 21.1.

### 21.1 Some useful concepts and ideas

In this section we will discuss a few of the concepts and ideas that are used in the proof of Ladner’s theorem without worrying about how exactly they relate to the theorem itself.

**Computable functions on natural numbers**

Whenever we have a function of the form $f : \mathbb{N} \to \mathbb{N}$, let us agree that when we say that $f$ is computable, we mean that there exists a DSM $M$ with input alphabet $\{0\}$ that, when given an input $0^n$, halts and outputs $0^{f(n)}$, for all $n \in \mathbb{N}$. In other words, $M$ computes $f$ with respect to unary notation.

Note that it would not actually make any difference if we were to choose different ways to represent the input and output, so long as we chose reasonable ways
The DSM $K$ works as follows on input $0^n$:

1. Repeat the following steps until $M$ halts:
   1.1 Run $M$ for one step.
   1.2 Push 0 onto $Y$.
2. Repeat the following steps until $Y$ stores $\varepsilon$:
   2.1 Pop 0 off of $Y$.
   2.2 Push 0 onto $X$.
3. Halt and output the string stored by $X$.

Figure 21.1: The DSM $K$ for the proof of Proposition 21.2, for a given DSM $M$ that computes a function $f$ with respect to unary notation. It is to be assumed that $X$ denotes the input/output stack of both $M$ and $K$, and also that $Y$ denotes a stack that is not used by $M$. (As is our default assumption, the stack $Y$ stores the empty string at the start of the computation of $K$.)

of encoding natural numbers; we would end up with an equivalent definition for which functions of the form $f : \mathbb{N} \to \mathbb{N}$ are computable. We’re basically just picking a way of encoding the inputs and outputs of such functions so that it is easily comparable to the definition of time-constructibility.

We will make use of the following simple proposition that relates computable functions of the form $f : \mathbb{N} \to \mathbb{N}$ with time-constructible functions. In short, the proposition establishes that every computable function is upper-bounded by some time-constructible function.

**Proposition 21.2.** Suppose $f : \mathbb{N} \to \mathbb{N}$ is a computable function. There exists a time-constructible function $g : \mathbb{N} \to \mathbb{N}$ such that $g(n) \geq f(n)$ for every $n \in \mathbb{N}$.

**Proof.** The simple idea behind the proof is to define $g(n)$ to be equal to $f(n)$ plus the number of steps required by the DSM that computes $f$ on input $n$.

In more detail, under the assumption that $f$ is computable, there exists a DSM $M$ that outputs $0^{f(n)}$ on input $0^n$ for each $n \in \mathbb{N}$. Let $K$ be the DSM described in Figure 21.1, for any fixed choice of such a DSM $M$. As suggested above, this DSM $K$ computes a function $g$ that takes the value $f(n)$ plus the number of steps for which $M$ runs on input $0^n$. It is therefore the case that $g(n) \geq f(n)$ for every $n \in \mathbb{N}$. It is also evident from its description that $K$ runs in time $O(g(n))$, and therefore $g$ is time-constructible.

\[\square\]
Lecture 21

Gap languages

Suppose that $f : \mathbb{N} \to \mathbb{N}$ is a time-constructible function with $f(n) > n$ for every $n \in \mathbb{N}$. Think about what happens when we repeatedly compose this function with itself, starting with the value 0. Hereafter we will use the notation $f^{(n)}$ to refer to the function obtained by composing $f$ with itself $n$ times. If we compose the function with itself zero times, then it’s like we’re doing nothing at all, so we have

$$f^{(0)}(0) = 0.$$  \hfill (21.1)

Continuing on, we have

$$f^{(1)}(0) = f(0), \quad f^{(2)}(0) = f(f(0)), \quad f^{(3)}(0) = f(f(f(0))),$$ \hfill (21.2)

and so on. Because $f(n) > n$, we know that if we keep on composing $f$ with itself like this, we’ll get larger and larger values:

$$f^{(n)}(0) < f^{(n+1)}(0)$$ \hfill (21.3)

for every $n \in \mathbb{N}$.

Now, for any fixed choice of an alphabet $\Sigma$, define the language

$$G_f = \bigcup_{n \in \mathbb{N}} \left\{ x \in \Sigma^* : f^{(2n)}(0) \leq |x| < f^{(2n+1)}(0) \right\}. \hfill (21.4)$$

We will call such a language a gap language. Whether or not a string is in this language only depends on how long the string is, and not on the particular way the symbols are selected. Because the values

$$f^{(0)}(0) < f^{(1)}(0) < f^{(2)}(0) < f^{(3)}(0) < \cdots$$ \hfill (21.5)

are different and keep getting bigger and bigger, every string $x \in \Sigma^*$ is such that $|x|$ will lie between two of these values—and we simply include $x$ in $G_f$ if it is the case that

$$f^{(n)}(0) \leq |x| < f^{(n+1)}(0)$$ \hfill (21.6)

for an even value of $n$ (which is equivalent to $f^{(2n)}(0) \leq |x| < f^{(2n+1)}(0)$ for some $n \in \mathbb{N}$).

Later on we will refer to the gap language $G_f$ for a particular time-constructible function $f$ of our choosing. Irrespective of this particular choice of $f$, we can prove that $G_f$ is always polynomial-time decidable; the only assumptions needed to prove this are that $f$ is time-constructible and satisfies $f(n) > n$ for all $n \in \mathbb{N}$.
Proposition 21.3. Let \( f : \mathbb{N} \rightarrow \mathbb{N} \) be a time-constructible function such that \( f(n) > n \) for all \( n \in \mathbb{N} \), let \( \Sigma \) be an alphabet, and let \( G_f \subseteq \Sigma^* \) be as defined in (21.4). It is the case that \( G_f \in P \).

Proof. Because \( f \) is time constructible, there must be a DSM \( M \) that computes \( f(n) \) given \( 0^n \) in time \( O(f(n)) \). For a given input string \( x \in \Sigma^* \), we can therefore compute \( f^{(0)}(0) \), \( f^{(1)}(0) \), \( f^{(2)}(0) \), etc., by simply running \( M \) iteratively, until a value is reached that exceeds \( |x| \). Note that for each run of \( M \), we can limit \( M \) to \( O(|x|) \) steps, for if this number of steps is exceeded, then the output of \( M \) must exceed \( |x| \).

As the values \( f^{(0)}(0) \), \( f^{(1)}(0) \), \( f^{(2)}(0) \) are increasing, we conclude that at most \( |x| \) iterations are required, and consequently one can find the smallest integer \( m \) for which \( |x| < f^{(m)}(0) \) in polynomial time. By accepting if this value of \( m \) is odd and rejecting if it is even, the language \( G_f \) has been decided in polynomial time. \( \square \)

Uniformly decidable languages

Suppose that \( M \) is a DSM with input alphabet \( \Sigma \cup \{\#\} \) that halts on all input strings. For each string \( y \in \Sigma^* \), define a language \( A_y \subseteq \Sigma^* \) as follows:

\[
A_y = \{ x \in \Sigma^* : x\#y \in L(M) \}.
\]  

We can describe the collection of languages \( \{A_y : y \in \Sigma^* \} \) as being uniformly decidable. Not only are each of the languages \( A_y \) decidable, but in fact there is a single DSM, namely \( M \), that is ready to decide them all: to check if \( x \in A_y \), just run \( M \) on input \( x\#y \).

In the theorem that follows, we will make use of the notation \( B \equiv_{m}^{p} C \), which is a shorthand way to express that both \( B \leq_{m}^{p} C \) and \( C \leq_{m}^{p} B \). What the theorem states is that the set of all languages \( B \subseteq \Sigma^* \) satisfying \( B \equiv_{m}^{p} C \) is uniformly decidable, assuming \( C \) is decidable. Notice that by selecting \( C \) to be any NP-complete language, we may conclude that the set of all NP-complete languages over the alphabet \( \Sigma \) is uniformly decidable; and that by selecting \( C \) to be any nontrivial language in \( P \) (meaning \( C \) is neither \( \emptyset \) nor \( \Sigma^* \)), we may conclude that the set of all nontrivial languages in \( P \) over the alphabet \( \Sigma \) is uniformly decidable.

Theorem 21.4. Let \( C \subseteq \Sigma^* \) be a decidable language. There exists a DSM \( M \) having input alphabet \( \Sigma \cup \{\#\} \) that halts on all input strings such that the languages

\[
A_y = \{ x \in \Sigma^* : x\#y \in L(M) \}
\]  

satisfy

\[
\{A_y : y \in \Sigma^* \} = \{ B \subseteq \Sigma^* : B \equiv_{m}^{p} C \}.
\]
The DSM $M$ works as follows on input $w \in (\Sigma \cup \{\#\})^*$:

1. If it is not the case that $w = x\#y$ for strings $x, y \in \Sigma^*$, then reject.
2. If it is not the case that $y$ is an encoding $y = \langle \langle R \rangle, \langle S \rangle, \langle m \rangle \rangle$, (21.10)
   where $R$ and $S$ are DSMs with input alphabet $\Sigma$ and $m$ is a positive integer, then accept if $x \in C$ and reject if $x \not\in C$.

Hereafter we will let $f$ and $g$ be the functions defined as follows:

2.1 $f : \Sigma^* \rightarrow \Sigma^*$ is the function obtained by simulating $R$ on a given input string $z \in \Sigma^*$ for $|z|^m + m$ steps and outputting the contents of its first stack projected onto the alphabet $\Sigma$.

2.2 $g : \Sigma^* \rightarrow \Sigma^*$ is the function obtained by simulating $S$ on a given input string $z \in \Sigma^*$ for $|z|^m + m$ steps and outputting the contents of its first stack projected onto the alphabet $\Sigma$.

3. If there exists a string $z \in \Sigma^*$ with $|z| \leq |x|$ for which the statement
   
   \[ [z \in C] \iff [f(g(z))] \in C \]  
   (21.11)

   is false, then accept if $x \in C$ and reject if $x \not\in C$.

4. Accept if $f(x) \in C$ and reject if $f(x) \not\in C$.

Figure 21.2: The DSM $M$ for the proof of Theorem 21.4.

Proof. Consider the DSM $M$ described in Figure 21.2. In order to prove (21.9), it suffices to prove two statements:

1. For any language $B \subseteq \Sigma^*$ satisfying $B \equiv_m^p C$, it is the case that $B = A_y$ for some string $y \in \Sigma^*$.
2. $A_y \equiv_m^p C$ for every string $y \in \Sigma^*$.

As we prove these two statements, it may help to have in your mind that the function $f$ described in step 2.1 might possibly be a reduction from $A_y$ to $C$, while the function $g$ described in step 2.2 might possibly be a reduction from $C$ back to $A_y$. The specific way that $M$ operates will depend upon whether or not these functions really are reductions as just suggested.
Let us start with the first statement. Suppose that $B \subseteq \Sigma^*$ is a language for which $B \equiv_p^m C$. This implies that $B \leq_p^m C$, so there must exist a polynomial-time computable function $f$ such that

$$[x \in B] \Leftrightarrow [f(x) \in C];$$

(21.12)

as well as $C \leq_p^m B$, so there must exist a polynomial-time computable function $g$ such that

$$[x \in C] \Leftrightarrow [g(x) \in B].$$

(21.13)

If we take $R$ and $S$ to be DSMs that compute $f$ and $g$, respectively, in polynomial time, and take $m$ to be a sufficiently large integer so that $R$ and $S$ always finish their computations within $n^m + m$ steps on inputs of length $n$, then an examination of $M$ reveals that $B = A_y$ for $y = \langle\langle R\rangle, \langle S\rangle, \langle m\rangle\rangle$. That is, for this selection of $y$, the check performed in step 3 will never be false, so $M$ accepts if $f(x) \in C$ and rejects if $f(x) \not\in C$, which is equivalent to $M$ accepting if $x \in B$ and rejecting if $x \not\in B$.

Now we will move on to the second statement. Let $y \in \Sigma^*$ be any string. We wish to prove that $A_y \equiv_p^m C$, i.e., $A_y \leq_p^m C$ and $C \leq_p^m A_y$. There are a few different cases to consider:

Case 1. The check in step 2 fails: $y$ is not an encoding of $R$, $S$, and $m$ as described in Figure 21.2. In this case $M$ gives up on the $y$ part of the input and tests whether or not $x \in C$. This implies $A_y = C$, and therefore $A_y \equiv_p^m C$ holds trivially.

Case 2. The check in step 2 passes, but there exists a string $z \in \Sigma^*$ for which (21.11) is false. Let us fix $z$ to be the lexicographically first string for which (21.11) is false. For every string $x \in \Sigma^*$ satisfying $|x| \geq |z|$, the DSM $M$ will realize that (21.11) is false for the string $z$, and will accept if $x \in C$ and reject if $x \not\in C$. This means that $A_y$ and $C$ agree for all strings having length at least $|z|$, and therefore $A_y \equiv_p^m C$.

Case 3. The check in step 2 passes, and (21.11) is true for every string $z \in \Sigma^*$. This implies that

$$A_y = \{x \in \Sigma^* : f(x) \in C\},$$

(21.14)

as $M$ always reaches step 4 in this case. We see immediately that $f$ is a polynomial-time mapping reduction from $A_y$ to $C$. We also have that $g$ is a polynomial-time mapping reduction from $C$ to $A_y$, due to the fact that

$$[z \in C] \Leftrightarrow [f(g(z)) \in C] \Leftrightarrow [g(z) \in A_y]$$

(21.15)

for all $z \in \Sigma^*$. Therefore $A_y \equiv_p^m C$.

As it has been proved that $A_y \equiv_p^m C$ in all three cases, the theorem follows. □
21.2 Proof of Ladner’s theorem

Now we will prove Ladner’s theorem, using the concepts and ideas introduced in the previous section.

Main diagonalization lemma

Suppose that $M$ is a DSM with input alphabet $\Sigma \cup \{\#\}$ that halts on all input strings, and define a language $A_y \subseteq \Sigma^*$ as

$$A_y = \{x \in \Sigma^* : x\#y \in L(M)\} \quad (21.16)$$

for each string $y \in \Sigma^*$, just like we did in the previous section in the discussion of uniformly decidable languages. Also suppose that, in addition to $M$, a decidable language $A \subseteq \Sigma^*$ has been selected, and assume that the symmetric difference $A \Delta A_y$ is infinite for every string $y \in \Sigma^*$. In other words, not only is $A$ not in the collection $\{A_y : y \in \Sigma^*\}$, but it is different from each of these languages on infinitely many strings.

Now define a function $f : \mathbb{N} \to \mathbb{N}$ as follows.

$$f(n) = \max\left\{\min\{ |x| : x \in A \Delta A_y \text{ and } |x| \geq n \} : y \in \Sigma^* \text{ and } |y| \leq n \right\}. \quad (21.17)$$

For a given choice of $n$, this function tells us something about the lengths of the strings on which $A$ and $A_y$ differ for varying choices of $y$, but it seems a bit strange how $n$ is being used as a lower bound on the length of $x$ and an upper bound on the length of $y$. The simple explanation is that we’re setting ourselves up for a diagonalization argument using this function, and the function $f$ is defined the way it is defined because that’s what makes the proof work. For now, though, it is not necessary to understand why $f$ is defined the way it is, but instead to focus on the fact that $f$ is well-defined and can be computed. To be more precise, let us argue that there exists a stack machine $K$ with input alphabet $\{0\}$ that outputs $0^{f(n)}$ on input $0^n$ for each $n \in \mathbb{N}$.

To see that $f$ is indeed computable, first observe that if you are given two strings $x, y \in \Sigma^*$, you can easily decide whether or not $x \in A \Delta A_y$. In greater detail, one can first decide whether or not $x \in A$, which is possible because $A$ is decidable, and then use the DSM $M$ to decide whether or not $x \in A_y$. Once it is known whether or not $x \in A$ and whether or not $x \in A_y$, determining whether or not $x \in A \Delta A_y$ is easy.

Next, if you are given a natural number $n$ and a string $y \in \Sigma^*$, you can determine the value

$$\min\{ |x| : x \in A \Delta A_y \text{ and } |x| \geq n \} \quad (21.18)$$
by simply searching. More specifically:

1. Initialize a stack \( X \) so that it stores the lexicographically first string \( x \in \Sigma^* \) satisfying \( |x| \geq n \).

2. Test if the string \( x \) stored by \( X \) is included in \( A \triangle A_y \) as described above. If \( x \in A \triangle A_y \) then output \( |x| \) and halt.

3. Increment \( X \) with respect to the lexicographic ordering of \( \Sigma^* \) and go back to step 2.

Note that this computation will always halt because we have assumed that \( A \triangle A_y \) is infinite for every string \( y \in \Sigma^* \); there will always be a string \( x \in A \triangle A_y \) with \( |x| \geq n \) for any given value of \( n \) (and in fact there will be infinitely many such \( x \)), and the computation will eventually find one of these strings \( x \) and halt.

Now, to compute \( f(n) \), simply perform the computation described above for every string \( y \in \Sigma^* \) with \( |y| \leq n \) and output the maximum value that is obtained. Notice that the definition of \( f \) immediately implies that \( f(n) \geq n \), irrespective of the choice of \( A \) and \( M \).

At this point we are ready for the following lemma, which includes the main diagonalization argument needed to prove Ladner’s theorem.

**Lemma 21.5.** Let \( \Sigma = \{0, 1\} \), let \( A, B \subseteq \Sigma^* \) be decidable languages, and let \( M \) and \( N \) be DSMs with input alphabet \( \Sigma \cup \{\#\} \) that halt on all input strings. For each string \( y \in \Sigma^* \) define languages \( A_y, B_y \subseteq \Sigma^* \) as follows:

\[
\begin{align*}
A_y &= \{ x \in \Sigma^* : x \# y \in L(M) \}, \\
B_y &= \{ x \in \Sigma^* : x \# y \in L(N) \},
\end{align*}
\] (21.19)

and assume that for every string \( y \in \Sigma^* \) the languages \( A \triangle A_y \) and \( B \triangle B_y \) are infinite. There exists a time-constructible function \( f \), with \( f(n) > n \) for all \( n \in \mathbb{N} \), so that the language

\[
C = (A \cap G_f) \cup (B \cap \overline{G_f})
\] (21.20)

is such that \( C \triangle A_y \) and \( C \triangle B_y \) are infinite for all \( y \in \Sigma^* \).

**Proof.** Define functions \( g \) and \( h \) along similar lines to the discussion above:

\[
\begin{align*}
g(n) &= \max \left\{ \min \left\{ |x| : x \in A \triangle A_y \text{ and } |x| \geq n \right\} : y \in \Sigma^*, |y| \leq n \right\}, \\
h(n) &= \max \left\{ \min \left\{ |x| : x \in B \triangle B_y \text{ and } |x| \geq n \right\} : y \in \Sigma^*, |y| \leq n \right\}.
\end{align*}
\] (21.21)
As per that discussion, these are computable functions that necessarily satisfy \( g(n) \geq n \) and \( h(n) \geq n \) for all \( n \in \mathbb{N} \). The function \( \max\{g(n), h(n)\} + 1 \) is therefore also computable, so by Proposition 21.2, it follows that there exists a time-constructible function \( f \) satisfying

\[
f(n) \geq \max\{g(n), h(n)\} + 1 \tag{21.22}\]

for every \( n \in \mathbb{N} \).

Now choose an arbitrary string \( y \in \Sigma^* \) as well as any even natural number \( k \) that satisfies

\[
f(k)(0) \geq |y| \tag{21.23}\]

Notice that there are infinitely many such choices of \( k \), as every sufficiently large even number satisfies this inequality. Let \( x \in \Sigma^* \) be any minimal-length string such that \( |x| \geq f(k)(0) \) and \( x \in A \triangle A_y \). In other words, choose \( x \in A \triangle A_y \) such that

\[
|x| = \min\{|z| : z \in A \triangle A_y \text{ and } |z| \geq f(k)(0)\} \tag{21.24}\]

There must exist such an \( x \), given the assumption that \( A \triangle A_y \) is infinite. By the definition of the function \( g \), we see that

\[
|x| \leq g(f(k)(0)) < f^{(k+1)}(0), \tag{21.25}\]

and therefore

\[
f(k)(0) \leq |x| < f^{(k+1)}(0). \tag{21.26}\]

As \( k \) is even, it follows that \( x \in G_h \). Given that \( x \in G_h \) and \( x \in A \triangle A_y \), we conclude that

\[
x \in C \triangle A_y. \tag{21.27}\]

Because we can repeat this argument for infinitely many choices of \( k \), we conclude that \( C \triangle A_y \) is infinite for every \( y \in \Sigma^* \).

By applying the same argument to \( N \) and \( h \) in place of \( M \) and \( g \), and changing the requirement that \( k \) is even to \( k \) is odd, we conclude that \( C \triangle B_y \) is infinite for every \( y \in \Sigma^* \), which completes the proof. \( \square \)

**Finishing off the proof**

Finally, we will apply the main diagonalization lemma from above to prove Ladner’s theorem.

231
Proof of Theorem 21.1. Let \( A \subseteq \Sigma^* \) be any language in P such that \( A \not\in \{ \emptyset, \Sigma^* \} \) and let \( B \) be any NP-complete language. By Theorem 21.4, there exist DSMs \( M \) and \( K \), both having input alphabet \( \Sigma \cup \{ \# \} \), such that the languages
\[
A_y = \{ x \in \Sigma^* : x\#y \in L(M) \}, \quad B_y = \{ x \in \Sigma^* : x\#y \in L(K) \}
\]
satisfy
\[
\{ A_y : y \in \Sigma^* \} = \{ C \subseteq \Sigma^* : C \equiv_m^p B \}, \quad \{ B_y : y \in \Sigma^* \} = \{ D \subseteq \Sigma^* : D \equiv_m^p A \}.
\]
Observe that \( \{ A_y : y \in \Sigma^* \} \) contains every NP-complete language over the alphabet \( \Sigma \) while \( \{ B_y : y \in \Sigma^* \} \) contains every nontrivial language in P over the alphabet \( \Sigma \) (i.e., every language over \( \Sigma \) in P besides the trivial languages \( \emptyset \) and \( \Sigma^* \)).

Now we will finally use the assumption that \( P \neq NP \), which we have not used up to this point. Under this assumption, we see not only that \( A \not\in \{ A_y : y \in \Sigma^* \} \) and \( B \not\in \{ B_y : y \in \Sigma^* \} \), but in fact \( A \triangle A_y \) and \( B \triangle B_y \) are infinite for every string \( y \in \Sigma^* \).

We may therefore apply Lemma 21.5 to obtain a language
\[
C = (A \cap G_f) \cup (B \cap \overline{G_f})
\]
such that \( C \triangle A_y \) and \( C \triangle B_y \) are infinite for every \( y \in \Sigma^* \). This implies that \( C \) is neither in P nor is it NP-complete. The language \( C \) is, however, in NP: the language \( G_f \) is in P, and therefore \( \overline{G_f} \) is also in P, and thus \( A, B, G_f, \) and \( \overline{G_f} \) are in NP, so that \( C \) is in NP because NP is closed under finite unions and intersections.

\( \square \)

Intuition

The proof we just discussed was technically involved, and you should not feel that it is essential that you understand all of the details. There is, however, something intuitive about the way the language
\[
C = (A \cap G_f) \cup (B \cap \overline{G_f})
\]
is defined that puts it strictly between the P languages and the NP-complete languages. In essence, \( C \) alternates back and forth between \( A \) (which is in P) and \( B \) (which is NP-complete) for long stretches of input lengths. It isn’t NP-complete because it’s too easy on some input lengths and it isn’t in P because it’s too hard on other input lengths.