Theory of Computation

Hikmat Farhat
Notre Dame University

April 16, 2018

Contents

1 Introduction **1**
- 1.1 Sets . . . 1
- 1.2 Relations . . . 2
- 1.3 Proof techniques . . . 4
  - 1.3.1 Proof by Contradiction . . . 4
  - 1.3.2 Proof by Induction . . . 5
  - 1.3.3 The Pigeonhole Principle . . . 5
  - 1.3.4 Diagonalization Principle . . . 6
- 1.4 Closures and Algorithms . . . 7
- 1.5 Alphabets, strings, and languages . . . 8
  - 1.5.1 Alphabets . . . 8
- 1.6 State machines . . . 9
  - 1.6.1 Examples of DFAs . . . 10

2 Deterministic Finite Automata **15**
- 2.1 Formal Definition of a DFA . . . 15
- 2.2 Configuration . . . 15
- 2.3 Formal Definition of Computation . . . 16
  - 2.3.1 Extended Function . . . 16
- 2.4 Regular Operations . . . 17
- 2.5 Closure Under the Intersection Operation . . . 17
- 2.6 Closure Under the Complement Operation . . . 20
- 2.7 Closure Under the Union Operation . . . 21

3 Non Deterministic Finite Automata **25**
- 3.1 Non-deterministric Finite Automata . . . 25
- 3.2 Example . . . 25
- 3.3 Formal Definition of a Nondeterministric Automaton . . . 26
- 3.4 Computation With NFA . . . 26
- 3.5 Examples . . . 27

4 Equivalence of DFA and NFA **29**
- 4.1 Introduction . . . 29
- 4.2 Subset Construction . . . 31

4.3 Subset Construction Examples 32

5 Non Deterministic Finite Automata with ϵ Transitions 35
5.1 Introduction 35
5.2 Formal Definition of an ϵ-NFA 36
5.3 Epsilon Closure 37
5.4 Computing With ϵ-NFA's 38
5.5 Equivalence of ϵ-NFA and DFA 40
5.6 Closure Properties 44
5.6.1 Closure under Union 44
5.6.2 Closure under Concatenation 45
5.6.3 Closure under Kleene star 47

6 Regular Expressions 51
6.1 Introduction 51
6.2 Formal Definition of a Regular Expression 51
6.3 Equivalence of RE and DFA 52
6.4 Generalized Non Deterministic Finite Automata 53
6.4.1 Converting a DFA to a GNFA 53
6.4.2 From GNFA to Regular Expression 54
6.5 From Regular Expression to NFA 57

7 Non-Regular Languages 61
7.1 Pumping Lemma for Regular Languages 61

8 DFA Minimization 65
8.1 Introduction 65
8.2 Equivalent States 65
8.3 Quotient Construction 71
8.4 Nondeterminisitic Automata 72
8.5 Myhill-Nerode Relations 72
8.6 From Myhill-Nerode to DFA 73
8.7 From DFA to Myhill-Nerode 74
8.8 Myhill-Nerode Theorem 75
8.9 Examples 76

9 Context-free Grammars 77
9.1 Introduction 77
9.2 Formal Definition 77
9.3 Derivations 78
9.3.1 Leftmost and rightmost derivations 82
9.4 Parse Trees 83
9.4.1 Ambiguous Grammar 83
9.5 Non Context-free Languages 85

10 Pushdown Automata **89**
10.1 Introduction 89
10.2 Formal Definition of PDA 89
10.3 Instantaneous Descriptions of a PDA 91

11 CYK Algorithm **93**
11.1 Introduction 93
11.2 Examples 93
11.3 Complexity 96

12 Turing Machines **97**
12.1 Introduction 97
12.2 Formal Definition 97
12.3 Configurations 98
12.4 Multitape Turing Machine 101
12.4.1 Storing State Information 102
12.4.2 Simulating a Multitape Machine 104
12.4.3 Running Time 105
12.5 Representing TM with Strings 105
12.6 Universal Turing Machine 106
12.7 Nondeterministic Turing Machine 107
12.8 Simulating a non-deterministic Turing machine 109

13 Undecidability **113**
13.1 Introduction 113
13.2 Decidable Languages 113
13.3 Diagonalization Method 114
13.4 Universal and Diagonal Languages 115
13.5 Halting Problem 117
13.6 Reduction 118
13.7 Rice's Theorem 122
13.8 RE Completeness 123

14 Post Correspondence Problem **125**
14.1 Introduction 125
14.2 Modified PCP is Undecidable 126

15 Complexity **129**
15.1 Class P 129
15.2 Class NP 130

16 Lambda Calculus **133**
16.1 Free and Bound Variables 133
16.2 Substitution 133
16.3 Reductions 134
16.4 Programming 134
16.4.1 Booleans 134

16.4.2 Natural Numbers 135
16.4.3 Recursion 136

Lecture 1

Introduction

1.1 Sets

A **set** is a collection of objects. A set could be defined by enumerating the objects of a set, called elements or members. For example, the letters a, b, c, and d form a set $S = \{a, b, c, d\}$. Sometimes a is in S, is written as $a \in S$. Similarly, f is not in S is written $f \notin S$. In a set the order of the elements does not matter, so $\{a, b, c, d\} = \{b, c, d, a\}$. The repetition of an element does not change the set, so $\{a, a, b\} = \{a, b\}$. A set with one element is called a **singleton**. For example $\{a\}$ is the set with one element a. Note that a and $\{a\}$ are different objects. The set with no elements is called the empty set and is denoted by $\emptyset$.

A different way of defining sets is to specify a property that all elements have or do not have. For example, if $\mathbf{N}$ is the set of natural numbers we can define the set of even numbers as

$$E = \{x : x \in \mathbf{N} \text{ and } x \text{ is divisible by } 2\}$$

A set A is a **subset** of a set B, written as $A \subseteq B$, if each element $a \in A$ is also an element of B. As in arithmetic there are various operations that act on sets. The **union** of two sets is a set having the elements of both sets (obviously without repetition). In symbols, the union operation is written as $\cup$

$$A \cup B = \{x : x \in A \text{ or } x \in B\}$$

For example

$$\{1, 3, 9\} \cup \{3, 5, 7\} = \{1, 3, 5, 7, 9\}$$

The **intersection** of two sets A and B, denoted by $A \cap B$, is a set whose elements are the common elements of A and B

$$A \cap B = \{x : x \in A \text{ and } x \in B\}$$

For example

$$\{1, 3, 9\} \cap \{3, 5, 7\} = \{3\}$$

The difference of two sets A and B, denoted by $A - B$, is the set of elements in A but not in B

$$A - B = \{x : x \in A \text{ and } x \notin B\}$$

Then operations of union, intersection and difference have the following properties

Commutativity	$A \cup B = B \cup A$
	$A \cap B = B \cap A$
Associativity	$(A \cup B) \cup C = A \cup (B \cup C)$
	$(A \cap B) \cap C = A \cap (B \cap C)$
Distributivity	$(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$
	$(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
DeMorgan's laws	$A - (B \cup C) = (A - B) \cap (A - C)$
	$A - (B \cap C) = (A - B) \cup (A - C)$

To prove that two sets A and B are equal we show that $A \subseteq B$ and $B \subseteq A$. We prove the first DeMorgan law as an example. Let $x \in A - (B \cup C)$ then $x \in A$ and $x \notin B$ and $x \notin C$. Since $x \in A$ and $x \notin B$ then $x \in A - B$. Similarly, $x \in A - C$ therefore $x \in (A - B) \cap (A - C)$ and $A - (B \cap C) \subseteq (A - B) \cap (A - C)$. To prove the other direction let $x \in (A - B) \cap (A - C)$ then $x \in A - B$ and $x \in A - C$. $x \in A - B \Rightarrow x \in A$ and $x \notin B$. Similarly, $x \in A$ and $x \notin C$ therefore x is not in either B or C so $x \notin B \cup C$. Finally, since $x \in A$ we get that $x \in A - (B \cup C)$ and $(A - B) \cup (A - C) \subseteq A - (B \cup C)$.

Given a set A then the set whose elements are all possible subsets (including $\emptyset$) of A is called the **power set** of A and is denoted 2^A.

1.2 Relations

An **ordered pair** is a sequence of two objects. Given two objects, x and y, we can form two ordered pairs: (x, y) and (y, x). The difference between an **ordered pair** and a set is that order matters: $(x, y) \neq (y, x)$. The **Cartesian product** of two sets A and B is the set of all **ordered pairs** (x, y) such that $x \in A$ and $y \in B$.

A **binary relation** R over two sets A and B is a subset of $A \times B$, $R \subseteq A \times B$.

Example 1.1. *Let $A = \{1, 3, 9\}$ and $B = \{b, c, d\}$. Then $A \times B = \{(1, b), (1, c), (1, d), (3, b), (3, c), (3, d), (9, b), (9, c), (9, d))\}$. An example binary relation over A and B is the following relation*

$$\{(1, b), (1, c), (3, d), (9, d)\}$$

The two sets need not be distinct. The following relation is a subset of $\mathbf{N} \times \mathbf{N}$

Example 1.2. *The less-than relation defined over the set of natural numbers:*

$$\{(i, j) : i, j \in \mathbf{N} \text{ and } i < j\}$$

We can generalize and define an **ordered n-tuple** $(a_1, \ldots, a_n)$ and **n-ary relation** on sets $A_1, \ldots, A_n$

A **function** from a set A to a set B is a binary relation R on A and B with the special property that any element of A occurs *exactly* once in R. This conforms with the usual definition that a function is *single* valued. Let C be the set of cities in the U.S. and let S be the set of states. Define the two relations

$$R_1 = \{(x, y) : x \in C, y \in S, \text{ and } x \text{ is a city in state y}\}$$
$$R_2 = \{(x, y) : x \in S, y \in C, \text{ and } y \text{ is a city in state x}\}$$

Clearly in the above example, R_1 is a function but R_2 is not a function since many states have more than one city. Usually, we denote a function on A and B by a letter such as f or g and write $f : A \to B$. A is the **domain** of f and B is the **range** of f. If $a \in A$ we write $f(a)$ to denote the element $b \in B$ such that $(a, b) \in f$. $b = f(a)$ is called the **image** of a under f. Clearly, we can specify a function f by enumerating $f(a)\ \forall a \in A$. A function $f : A \to B$ is **one-to-one** if $a \neq a' \Rightarrow f(a) \neq f(a')$. Using the example above where S is the set of states in the U.S. and C is the set of cities then the function $g : S \to C$, $g(s) =$ the capital of s is **one-to-one** because different states have different capitals. A function $f : A \to B$ is **onto** B if each element of B is the image of some element of A. In the example above, g is not **onto** since there are elements of C (cities) which are not the image of elements of S (are not capitals of states). Finally, a function $f : A \to B$ is called a **bijection** between A and B if it is both one-to-one and onto. Let C_0 be the set of capital cities in the U.S. then the following function is a bijection: $g(s) =$ the capital of state s.

The **inverse** of a binary relation $R \subseteq A \times B$, denoted $R^{-1} \subseteq B \times A$, is the relation $R^{-1} = \{(b, a) : (a, b) \in R\}$. The relation R_2 is the inverse of R_1. Obviously, the inverse of a function (in this case R_1) is not a function (in this case R_2). The reason in this case is because R_1 is not one-to-one and therefore the inverse will not be single valued. Also if a function $f : A \to B$ is not onto then the inverse is not a function since there exists $b \in B$ such that $f(a) \neq b$, $\forall a \in A$. When $f : A \to B$ is a bijection then f^{-1} is a function. Furthermore, $f^{-1}(f(a)) = a$ and $f(f^{-1}(b)) = b$, $\forall a \in A, b \in B$.

Sometimes when a simple bijection exists between two sets A and B then we can view $a \in A$ and $b = f(a) \in B$ as indistinguishable.

Example 1.3. *For any three sets A, B, and C there is a natural isomorphism of $A \times B \times C$ to $(A \times B) \times C$*

$$f(a, b, c) = ((a, b), c)$$

A **very important** isomorphism is the one from $2^{A \times B}$ to $A \times 2^B$. Given any relation R on A and B, then $R \in 2^{A \times B}$ and R is not necessarily a function. Recall that a function is a relation R with the special property that it is single valued so for any $a \in A$ there is **only one** $b \in B$ such that $(a, b) \in R$. The isomorphism we are proposing turns any relation R into a function. Suppose that R is not a function, i.e. there exists $a \in A$ and $b_1, \ldots, b_n$ such that $(a, b_i) \in R$, $\forall 1 \leq i \leq n$. Now since $\{b_1, \ldots, b_n\} \in 2^B$ rewrite the pairs $(a, b_1) \in R, \ldots, (a, b_n) \in R$ as a single pair $(a, \{b_1, \ldots, b_n\}) \in A \times 2^B$. This isomorphism will be useful when we introduce nondeterministic finite automata.

1.3 Proof techniques

Before listing the proof techniques a note about a statement that occurs often in this course. Given two statements P and Q, we need to show that "P if and only if Q". This statement can be divided into two parts. The first, P only if Q means that if P is true then Q is true, symbolically $P \Rightarrow Q$. The second part, P if Q means that if Q is true then P is true, symbolically $Q \Rightarrow P$.

We will study proof techniques that will be used in this course: *contradiction*, *induction*, *pigeonhole principle*, and *diagonalization*.

1.3.1 Proof by Contradiction

When using this type of proof technique we assume that the theorem we need to prove is false and then show that this assumption leads to a contradiction. We will use this technique to prove that $\sqrt{2}$ is an *irrational* number. First a useful lemma which uses the concept of contradiction itself!

Lemma 1.1. *An integer n is even if and only if n^2 is even.*

Proof. There are two parts to the proof. First "n is even only if n^2 is even", symbolically "n is even" $\Rightarrow$ "n^2 is even". Suppose than n is even then we can write $n = 2m$. Squaring both sides we get $n^2 = 4m^2 = 2(2m^2)$ thus n^2 is even. The second part, "n is even if n^2 is even", symbolically "n^2 is even" $\Rightarrow$ "n is even". Assume that it is otherwise. Then there exists an odd integer n such that n^2 is even. Since n is odd then we can write $n = 2m+1$. Squaring both sides we get $n^2 = (2m+1)^2 = 4m^2+4m+1 = 2(2m^2 + 2m) + 1$, an odd number which contradicts our assumption that n^2 is even.

Theorem 1.1. *$\sqrt{2}$ is irrational.*

Proof. Assume that $\sqrt{2}$ is rational thus $\sqrt{2} = \frac{m}{n}$ where m and n are integers. We assume that the greatest common divisor of m and n is one, gcd(m, n)=1. Otherwise we divide both of them by the gcd and the value of the fraction remains the same.Since gcd(m,n)=1 then either m or n is an odd number. Multiplying both sides of the fraction with n we get

$$n\sqrt{2} = m$$

Squaring both sides we get

$$2n^2 = m^2$$

Then m^2 is even and by lemma 1.1 so is m. We write $m = 2k$ and the above equation becomes

$$2n^2 = 4k^2$$

Dividing both sides by 2

$$n^2 = 2k^2$$

Therefore n^2 is an even number and consequently both m and n are even numbers which contradicts our starting point that either m or n is an odd number.

1.3.2 Proof by Induction

Proof by induction is a way of showing that all elements of an infinite set have a certain property. Let the desired property be $\mathcal{P}$. We want to prove that $\mathcal{P}(k)$ is true for all $k \in \mathbf{N}$. An induction proof has two parts, the **basis** and the **induction step**. In the **basis** we prove that $\mathcal{P}(1)$ is true and in the induction step we prove that if $\mathcal{P}(i)$ is true then $\mathcal{P}(i+1)$ is true for any $i \geq 1$. Once these two steps are done the result, that $\mathcal{P}(1), \mathcal{P}(2), \ldots$, are all true follows. This is because $\mathcal{P}$ is true from step one. Using that fact and the induction step proves that $\mathcal{P}(2)$ is true and so on.

Example 1.4. *For any* $n \geq 0$, $1 + 2 + \ldots, n = \frac{n(n+1)}{2}$

Proof. We proceed by induction
Basis. This is the case when $n = 1$. We have $1 = \frac{1(1+1)}{2} = 1$.
Induction Step. Suppose that the equality is true for $n-1$, then $1 + \ldots + (n-1) = \frac{(n-1)n}{2}$, and prove that it is true for n.

$$\begin{aligned} 1 + 2 + \ldots + (n-1) + n &= \frac{(n-1)n}{2} + n && \text{induction hypothesis} \\ &= \frac{(n-1)n + 2n}{2} \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Example 1.5. *For any finite set* A, $|2^A| = 2^{|A|}$

Proof. We prove the above by induction on the size of the set.
Basis. If $|A| = 1$ let $A = \{a\}$. Then $2^A = \{\{a\}, \emptyset\}$ thus $|2^A| = 2 = 2^1 = 2^{|A|}$.
Induction step. Assume that for $|A| = n$ and $|2^A| = 2^n$. Consider $B = A \cup \{b\}$. Clearly $|B| = n + 1$. The subsets of B can be divided into two groups: the ones that contain b and the ones that don't. The one that do not contain b are just the subsets of A since $B = A \cup \{b\}$. On the other hand the subsets that contain b are formed by taking the union of the subsets of A and $\{b\}$. Therefore we can write $2^B = 2^A \cup \{C \cup \{b\} : C \in 2^A\}$ and the cardinality of 2^B is the sum of the cardinality of the two sets. From the induction hypothesis $|2^A| = 2^n$. Also the second set has the same number of elements as 2^A. Combining the two we get that $|2^B| = 2 \cdot |2^A| = 2^{n+1} = 2^{|B|}$.

1.3.3 The Pigeonhole Principle

Theorem 1.2. *If* A *and* B *are finite sets and* $|A| > |B|$ *then there is no one-to-one function from* A *to* B.

Proof. we prove the theorem by induction on $|B|$.
Basis. $|B| = 1$. Then B contain one element, call it b. Since $|A| > |B| = 1$ then A contains at least two elements and any function has to map them to the same element b. Therefore there is no one-to-one function from A to B.
Induction hypothesis. Suppose that $|A| > |B| = n$ and there is no one-to-one function from A to B.
Induction step. Let f be any arbitrary function from A to B with $|A| > |B| = n + 1$. Choose an element $a \in A$ and consider the two sets $A-\{a\}$ and $B-\{f(a)\}$. Define the function g such that $g(x) = f(x)\ \forall x \in A - \{a\}$. Now, $|A - \{a\}| > |B - \{f(a)\} = n$ and by the induction hypothesis g is not one-to-one which means there are two elements of $A - \{a\}$ mapped by g to the same element of $B - \{f(a)\}$. Since $g(x) = f(x)$ $\forall x \in A - \{a\}$ then f is not one-to-one.

Definition 1.1. *A **path** of length n in a binary relation R is a sequence $a_1, \ldots, a_n$ such that $(a_i, a_{i+1}) \in R$ for $i = 1, \ldots, n-1$.*

Example 1.6. *Let R be a binary relation on a finite set A, and let $a, b \in A$. If there is a path from a to b in R, then there is a path of length at most $|A|$.*

Assume by way of contradiction that there are paths in R of size greater than $|A|$ and let $(a_1, \ldots, a_n) > |A|$ be the shortest such path. Define the function $f : N \longrightarrow A$ as $f(i) = a_i$. The path $(a_1, \ldots, a_n)$ of length n, is just the image of the set $\{1, \ldots, n\}$ under the function f. Now since $n > |A|$, by the pigeonhole principle f is not one-to-one and therefore there exists $i, j < n$ such that $f(i) = a_i = f(j) = a_j$. We can rewrite the path as $(a_1, \ldots, a_i, \ldots, a_j, a_{j+1}, \ldots, a_n)$ but since $a_i = a_j$ we get $(a_1, \ldots, a_i, \ldots, a_i, a_{j+1}, \ldots, a_n)$. The last path can be shortened by removing the subpath from a_i back to itself which leads to a contradiction because we assumed that $(a_1, \ldots, a_n)$ is the shortest path.

1.3.4 Diagonalization Principle

Theorem 1.3. *Let R be a binary relation on a set A, and define the "rows" of the relation as $R_a = \{b \mid b \in A \text{ and } (a, b) \in R\}$. The "diagonal" of the relation R is the set $D = \{a \mid a \in A \text{ and } (a, a) \notin R\}$. Then D is distinct from $R_a\ \forall a \in A$.*

Proof. Consider R_a for a given $a \in A$. There are two cases. If $a \in R_a$ then by construction $(a, a) \in R$ therefore $a \notin D$. On the other hand if $a \notin R_a$ then $(a, a) \notin R$ and also by construction $a \in D$. Therefore in the two cases D and R_a have at least the element a not in common.

Theorem 1.4. *The set 2^N is uncountable.*

Proof. Suppose that 2^N is countable. That is we assume that there is a way of enumerating all members of 2^N.

$$2^N = \{R_0, R_1, \ldots\}$$

The above enumeration induces a relation R over $N \times N$ with $(i, j) \in R$ iff $i \in R_j$. $R_i = \{j \in N : (i, j) \in R\}$ and consider the set

$$D = \{n \in N : n \notin R_n\}$$

By the diagonalization principle $D \neq R_i$ for all i. But D is a subset of N which is a contradiction and therefore 2^N is uncountable.

1.4 Closures and Algorithms

Given a set A and a relation R on A we define the reflexive transitive closure of R as

$$R^* = \{(a,b) : \text{ there is a path from } a \text{ to } b \text{ in } R\}$$

The transitive closure of R can be computed as follows

```
Initially R* = R ∪ {(a_i, a_i) : a_i ∈ A}
    for i = 1, ..., n do
        foreach i-tuple (b_1, ..., b_i) in A^i do
        if (b_1, ..., b_i) is a path in R then add (b_1, b_i) to R*
```

The first question that comes to mind is that why we stopped at $i = n$? Recall from the result of example 1.6 that in a set A with $|A| = n$ and a relation $R \subseteq A^2$ if there is a path between two elements a and b the length of the path is at most n. While the above algorithm is correct it is not efficient at all. Let $|A| = n$ then the $|A^i| = n^i$ and therefore in each iteration of the "for" loop there are n^i tuples to consider. For each tuple $(b_1, \ldots, b_i)$, we need at most i operations to check if there is a path from b_1 to b_i and then add (b_1, b_i) to R^*. Therefore the cost of the algorithm is $\sum_{i=1}^{n} i \times n^i = O(n^{n+1})$. A better algorithm to compute the reflexive transitive closure is the following

```
Initially R* = R ∪ {(a_i, a_i) : a_i ∈ A};
changed=True;
while changed do
    changed=False;
    forall the a_i, a_j, a_k do
        if (a_i, a_j) ∈ R* ∧ (a_j, a_k) ∈ R* ∧ (a_i, a_k) ∉ R* then
            R* = R* ∪ {(a_i, a_k)};
            changed=true;
        end
    end
end
```

The above algorithm terminates after at most n^2 steps since the while loop adds a pair at each step and there are at most n^2 pairs to be added to R^*. On the other hand, each step costs at most n^3 which is the number of triplets (a_i, a_j, a_k). Therefore the total cost is $O(n^5)$

1.5 Alphabets, strings, and languages

1.5.1 Alphabets

An *alphabet* is a finite set of symbols or characters. For example, the set $\Sigma = \{a, \ldots, z\}$ is the Roman alphabet. A useful alphabet that we will use often is the set $\Sigma = \{0, 1\}$. Usually we use the letter Σ to denote an alphabet.

A *string* over an alphabet Σ is a finite sequence of symbols from Σ. For example, abc is a string over $\Sigma = \{a, \ldots, z\}$. The *length* of a string is the number of symbols in it. Thus, the length of the string $w = abcdef$, denoted $|w|$ is 6. There is a special string that has no symbols at all called the *empty string* and denoted by ϵ, and it has length 0. The set of all strings, including the empty string, over an alphabet Σ is denoted by Σ^*. For example if $\Sigma = \{a, b\}$ then $\Sigma^* = \{\epsilon, a, b, aa, ab, bb, aaa, \ldots\}$. This is referred to as **lexicographic** ordering, listing shorter strings first and strings having the same length are listed alphabetically. The *concatenation* of two strings x and w is denoted by xw and it is the string formed by the string x followed by the string w. As a concrete example, consider : $x = cat$, $w = dog$ and the concatenated strings $xw = catdog$, $wx = dogcat$.

Concatenating with the empty string results in no change in the string. That is for any string x, we have that $x\epsilon = \epsilon x = x$. A string x is a *substring* of y (x occurs consecutively in y) if there exists strings u and v (possibly empty) such that $y = uxv$. If u is the empty string then x is called a prefix. Similarly, if v is the empty string then x is called a suffix. Given a string w and $i \in \mathbf{N}$ we define w^i as

$$
\begin{aligned}
w^0 &= \epsilon \\
w^{i+1} &= w^i w \qquad\qquad \text{for all } i \geq 0
\end{aligned}
$$

The reversal of a string w, denoted by w^R, has the same sequence of symbols but enumerated in reverse. For example if $w = theory$ then $w^R = yroeht$. A formal definition is given by induction

$$
\begin{aligned}
&\text{if } |w| = 0 \text{ then } w^R = w = \epsilon \\
&\text{if } |w| = n + 1 \text{ then } w = ua \text{ for some } a \in \Sigma \text{ and } w^R = au^R
\end{aligned}
$$

Now we will use the above inductive definition to show that if u and v are strings then $(uv)^R = v^R u^R$.

Proof. We proceed by induction on $|v|$.
Basis. If $v = \epsilon$ then $(uv)^R = (u\epsilon)^R = u^R = \epsilon u^R = \epsilon^R u^R$.
Induction step. Suppose that $|v| = n$ then $(uv)^R = v^R u^R$. Let $|v| = n + 1$ then we can write $v = xa$ with $x \in \Sigma^*$, $a \in \Sigma$ and $|x| = n$.

$$
\begin{aligned}
(uv)^R &= (uxa)^R \\
&= a(ux)^R && \text{from definition of string reversal} \\
&= ax^R u^R && \text{from induction hypothesis} \\
&= (xa)^R u^R && \text{from definition of string reversal} \\
&= v^R u^R
\end{aligned}
$$

A set of strings over an alphabet Σ is called a **language**. Obviously an language is a subset of Σ^*. Since languages are sets we define them in similar fashion. For finite languages it is usually enough to enumerate the strings in the language. Most languages of interest are infinite. Such languages are specified as follows

$$L = \{w \in \Sigma^* : w \text{ has property } P\}$$

For example the language $\{w \in \{0,1\}^* : w \text{ has an equal number of 0's and 1's}\}$. Since languages are sets they can be combined by the usual set operations of union, intersection and difference. In addition some operations are defined only for languages. Given two languages L_1 and L_2, the **concatenation**, L, of L_1 and L_2, denoted $L = L_1L_2$, is defined as

$$L = \{w \in \Sigma^* : w = uv \text{ for some } u \in L_1 \text{ and } v \in L_2\}$$

For example, let $L_1 = \{w \in \Sigma^* : w \text{ has an even number of 0's}\}$ and $L_2 = \{w \in \Sigma^* : w \text{ starts with a 0 and the remaining symbols are 1's}\}$ then $L = L_1L_2 = \{w \in \Sigma^* : w \text{ has an odd number of 0's}\}$. Finally, the Kleene star of a language L is defined as $L^* = \{w \in \Sigma^* : w = w_1w_2 \ldots w_k \text{ for some } k \text{ and some } w_1 \ldots w_k \in L\}$. Note that this definition is consistent with the definition of Σ^* if we regard Σ as a language of strings of length 1 over the alphabet Σ.

1.6 State machines

We can define a Finite Automaton (FA) or a Finite State Machine (FSM) informally using a **state diagram**. A **state diagram** is a set of states connected by labeled arrows. It is easier to introduce automaton by an example.

Given an alphabet $\Sigma = \{0,1\}$ the FA shown in Figure 1.1 **recognizes** all strings that end with a 1. The main parts of a state diagram are: the states and the labeled arrows. In this example we have four states, q_0 is the start state (arrow coming in), q_3 is the final state (double circle). We use the FSM to decide if a string is recognized by starting at the start state. On each input character, we follow the corresponding arc. When we run out of input characters, we answer “yes” or “no”, depending on whether we are in the final state.

The language of a machine M is the set of strings it accepts, written $L(M)$. In this case $L(M) = \{1, 01, 11, 001, 011, 101, 111, \ldots\}$.

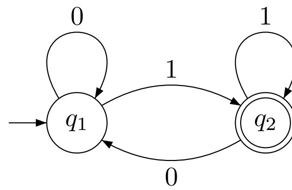

Figure 1.1: First Automaton Example

1.6.1 Examples of DFAs

Example 1.7. *In Figure 1.2 we show a DFA that recognize the language* L=$\{w \in \{0,1\}^* :$ *w contains at least 1 one and has an even number of 0s*$\}$

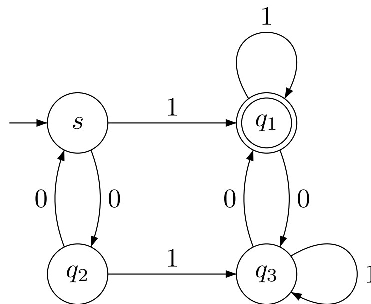

Figure 1.2: L(M)=has at least 1 one and number of zero is even

Example 1.8. *In Figure 1.3 we show a DFA that recognizes the language* $L = \{w \in \{0,1\}^* :$ *w ends with 1* $\}$

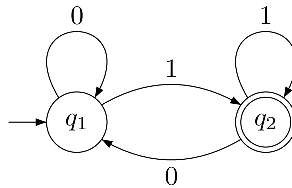

Figure 1.3: L(M) all strings that end with 1

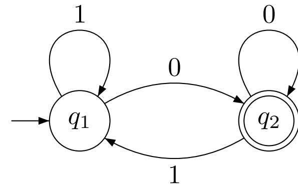

Figure 1.4: L(M) all strings that end with 0

Example 1.9. *In Figure 1.4 we show a DFA that recognizes the language* $L = \{w \in \{0,1\}^* : w$ *ends with 0* $\}$.

Example 1.10. *In Figure 1.5 we show a DFA that recognizes the language* $L = \{w \in \{a,b\}^* : w$ *ends with the same character it starts with* $\}$.

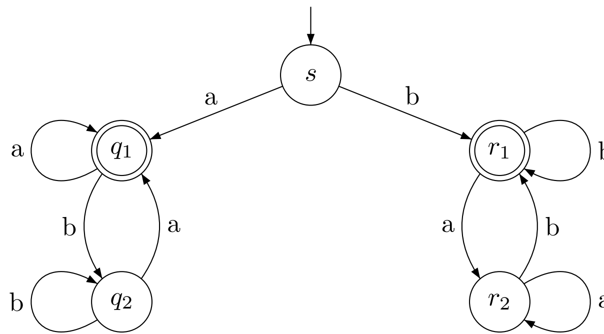

Figure 1.5: L(M) all strings that end with the same character it starts with

Example 1.11. *In Figure 1.6 we show a DFA that recognizes the language* $L = \{w \in \{0,1\}^* : w$ *has an even number of characters* $\}$.

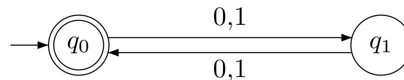

Figure 1.6: L(M) all strings in which the number of characters is even

Example 1.12. *In Figure 1.7 we show a DFA that recognizes the language* $L = \{w \in \{0,1\}^* : w$ *has a number of characters divisible by 3*$\}$

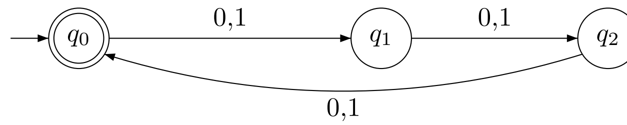

Figure 1.7: L(M) all strings in which the number of characters is divisible by 3

Example 1.13. *In Figure 1.8 we show a DFA that recognizes the language* $L = \{w \in \{0,1\}^* : w \textit{has an even number of 1's}\}$.

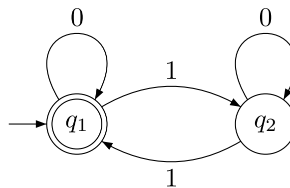

Figure 1.8: L(M) all strings that have even number of 1's

Example 1.14. *Strings where the sum of digits is a multiple of 3*

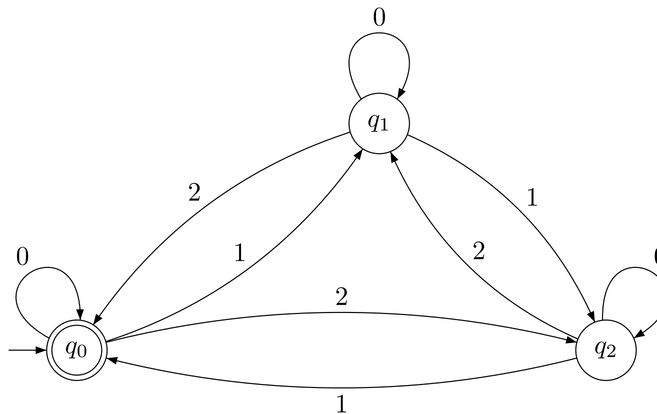

Figure 1.9: L(M) all strings in which the sum is multiple of 3

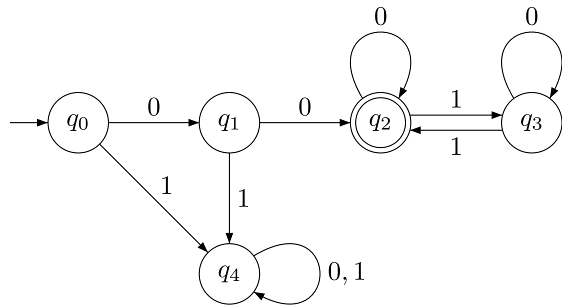

Figure 1.10: L(M)

Example 1.15. *In Figure 1.10 we show a DFA that recognizes the language* $L = \{w \in \{0,1\}^* : w = 00v$ *where* v *contains an even number of ones*

Notre Dame University
Computer Science Department

CSC 311 Theory of Computation

Homework 1

For each of the following languages over $\Sigma = \{0, 1\}$ build the DFA that recognizes it. Each exercise is worth 10 pts.

1. $L = \{w \in \{0,1\}^* : w$ begins with a 1 and ends with a $0\}$.
2. $L = \{w \in \{0,1\}^* : w$ contains at least two 1's$\}$. Note: Not necessarily consecutive.
3. $L = \{w \in \{0,1\}^* : w$ contains the substring $0110\}$.
4. $L = \{w \in \{0,1\}^* :$ every odd position of w is a $1\}$.
5. $L = \{w \in \{0,1\}^* :$ every 1 in w is preceded and followed by a $0\}$.
6. $L = \{w \in \{0,1\}^* : w$ does not contain 001 as substring$\}$.
7. $L = \{w \in \{0,1\}^* : w$ contains at least two 1's not followed immediately by a $0\}$.
8. $L = \{w \in \{0,1\}^* : w$ ends in $00\}$.
9. $L = \{w \in \{0,1\}^* : w$ has three consecutive 0's$\}$.
10. $L = \{w \in \{0,1\}^* :$ the number of 1's in w is divisible by $3\}$.

Lecture 2

Deterministic Finite Automata

2.1 Formal Definition of a DFA

A Deterministic Finite Automaton (DFA) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of called states.
- Σ is a finite set of called the alphabet.
- $\delta : Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the starting state.
- $F \subseteq Q$ is the set of final or accepting states.

A few remarks regarding the definition. First there is only one starting state whereas there could be many accepting states. Also since $\phi \subset Q$ we could have 0 final states. More importantly δ is a function in the mathematical sense which means it is *single* valued: for every pair $(q_i, s) \in Q \times \Sigma$ there is a unique transition.

If A is the set of all strings that machine M accepts, we say that M recognizes the language A.

2.2 Configuration

A **configuration** of a deterministic automaton is any element of $Q \times \Sigma^*$. The binary relation $\vdash_M$ between two configurations is defined as follows

$$(q, w) \vdash_M (q', w') \text{ iff } \; w = aw' \text{ for some } a \in \Sigma \text{ and } \delta(q, a) = q'$$

We define the multistep transition recursively as follows

1. $\vdash_M \; \equiv \vdash_M^1$ i.e. yields in one step
2. $(q, w) \vdash_M^n (q', w')$ iff $(q, w) \vdash_M^{n-1} (q'', w'') \vdash_M^1 (q', w')$
3. $(q, w) \vdash_M^* (q', w')$ iff $(q, w) \vdash_M^n (q', w')$ for some n

2.3 Formal Definition of Computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \ldots w_n$ be a string in Σ^*. Then M **accepts** w iff $(q_0, w) \vdash_M^* (q, \epsilon)$ for some $q \in F$. A language is called a **regular language** if some DFA recognizes it.

Example 2.1. *So far we have always defined the transition function δ using a transition diagram. In this example we give the definition using a formula. Let $\Sigma_i = \{0, 1, \ldots, i-1\}$ and the language $A_i = \{w : \text{sum of terms in } w \text{ is multiple of } i\}$. In other words, if $w = w_1 w_2 \ldots w_n$, with $w_i \in \Sigma_i$, $\forall i$, then $w_1 + w_2 + \ldots + w_n$ is multiple of i. The automaton $M_i(Q_i, \Sigma_i, \delta_i, q_0, \{q_0\})$* ***recongnizes*** *$A_i$ with $Q_i = \{q_0, q_1, \ldots, q_{i-1}\}$ and given $c \in \Sigma$*

$$\delta_i(q_j, c) = q_k \qquad \text{with } k = j + c \mod i$$

We will use the definition $\vdash_M^*$ to prove that the automaton M_i accepts the language A_i. If $w = w_1 w_2 \ldots w_n$ with $w_i \in \Sigma = \{0, 1, \ldots, i-1\}$ then define $sum(w) = w_1 + w_2 + \ldots + w_n$. To prove that M_i recognizes A_i we proceed by induction. It is sufficient to show that$(q_0, w) \vdash_M^* (q_0, \epsilon)$ iff $w \in A_i$, i.e. $sum(w) = 0 \mod i$. A stronger property to show would be

$$(q_0, w) \vdash_M^* (q_k, \epsilon) \qquad k = sum(w) \mod i \tag{2.1}$$

We prove equation (2.1) by induction on $|w|$. The induction basis, for $|w| = 0$, can be obtained directly from the definition $\vdash_M^*$ since $(q_0, \epsilon) \vdash_M^* (q_0, \epsilon)$ and $sum(\epsilon) = 0 \mod i$. Now assume that equation (2.1) is true for $x \in \{0, 1, i-1\}^*$ with $|x| = n$, and we show it is true for $|xc| = n + 1$ where $c \in \{0, 1, \ldots, i-1\}$.

$$(q_0, xc) \vdash_M^* (q_k, \epsilon) \Rightarrow (q_0, xc) \vdash_M^* (q_l, c) \vdash_M (q_k, \epsilon)$$

Since $(q_0, xc) \vdash_M^* (q_l, c)$ it follows that $(q_0, x) \vdash_M^* (q_l, \epsilon)$ and therefore $l = sum(x) \mod i$ from the induction hypothesis. Also $\delta_i(q_l, c) = q_k$ with $k = l + c \mod i$ from the definition of δ_i. Combining both results we obtain

$$\begin{aligned} k =& l + c \mod i \\ =& (sum(x) \mod i) + c \mod i \\ =& sum(x) + c \mod i \\ =& sum(xc) \mod i \end{aligned}$$

■

2.3.1 Extended Function

We will find it useful to use an extended function $\hat{\delta} : Q \times \Sigma^* \to Q$ with the following properties:

$$\hat{\delta}(p, \epsilon) = p$$
$$\hat{\delta}(p, wc) = \delta(\hat{\delta}(p, w), c)$$

An important property is that $(p, w) \vdash_M^* (q, \epsilon)$ iff $\hat{\delta}(p, w) = q$.

2.4 Regular Operations

Let Σ be an alphabet, A and B languages, $A, B \subseteq \Sigma^*$ define

- Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- Intersection: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
- complement: $\sim A = \{x \in \Sigma^* \mid x \notin A\}$
- Concatenation: $AB = \{xy \mid x \in A \text{ and } y \in B\}$
- Star: $A^* = \{x_1 x_2 \dots x_n \mid x_i \in A\}$

2.5 Closure Under the Intersection Operation

In this section we show that if A_1 and A_2 are regular languages so is $A_1 \cap A_2$. Since A_1 and A_2 are regular then there are automata $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ such that $L(M_1) = A_1$ and $L(M_2) = A_2$. Now define the automaton $M = (Q, \Sigma, \delta, q, F)$ as

- $Q = Q_1 \times Q_2$.
- $F = F_1 \times F_2$.
- $q_0 = (q_1, q_2)$.
- and

$$\delta((p,q),a) = (\delta_1(p,a), \delta_2(q,a)) \tag{2.2}$$

Define

$$\begin{aligned} \hat{\delta}((p,q),\epsilon) &= (p,q) \\ \hat{\delta}((p,q),xa) &= \delta(\hat{\delta}((p,q),x),c) \end{aligned} \tag{2.3}$$

Lemma 2.1. *For* $x \in \Sigma^*$.

$$\hat{\delta}((p,q),x) = (\hat{\delta}_1(p,x), \hat{\delta}_2(q,x))$$

Proof. By induction over $|x|$. The basis of the induction $x = \epsilon$ is true since

$$\hat{\delta}((p,q),\epsilon) = (p,q) = (\hat{\delta}_1(p,\epsilon), \hat{\delta}_2(q,\epsilon))$$

We will assume that it is true for x and show that it is true for xc where $c \in \Sigma$.

$$\begin{aligned} \hat{\delta}((p,q),xc) &= \delta(\hat{\delta}((p,q),x),c) && \text{from the definition eq (2.3)} \\ &= \delta((\hat{\delta}_1(p,x), \hat{\delta}_2(q,x)),c) && \text{from the induction hypothesis} \\ &= (\delta_1(\hat{\delta}_1(p,x),c), \delta_2(\hat{\delta}_2(q,x),c)) && \text{from the defintion in eq (2.2)} \\ &= (\hat{\delta}_1(p,xc), \hat{\delta}_2(q,xc)) && \text{from the definition of } \hat{\delta}_1 \text{ and } \hat{\delta}_2 \end{aligned}$$

Theorem 2.1. $L(M) = A_1 \cap A_2$.

Proof.

$$
\begin{aligned}
x \in L(M) & \\
& \Leftrightarrow \hat{\delta}(q_0, x) \in F \\
& \Leftrightarrow \hat{\delta}((q_1, q_2), x) \in F_1 \times F_2 \\
& \Leftrightarrow (\hat{\delta}_1(q_1, x), \hat{\delta}_2(q_2, x)) \in F_1 \times F_2 \\
& \Leftrightarrow \hat{\delta}_1(q_1, x) \in F_1 \text{ and } \hat{\delta}_2(q_2, x) \in F_2 \\
& \Leftrightarrow x \in L(M_1) \text{ and } x \in L(M_2) \\
& \Leftrightarrow x \in L(M_1) \cap L(M_2)
\end{aligned}
$$

Example 2.2. *Consider the language $L = \{w \mid w$ contains at least two 0s and at least two 1s$\}$. We can write $L = L_1 \cap L_2$ where $L_1 = \{w \mid w$ contains at least two 0s$\}$ and $L_2 = \{w \mid w$ contains at least two 1s $\}$. Construct a DFA M to recognize L by starting from the DFAs M_1 and M_2 with $L_1 = L(M_1)$ and $L_2 = L(M_2)$.*

We know from section 2.5 that once we construct $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ then $M = (Q, \Sigma, \delta, s, F)$ can be obtained as follows

- $Q = Q_1 \times Q_2$
- $\delta((p, q), a) = (\delta(p, a), \delta(q, a))$
- $s = (s_1, s_2)$
- $F = F_1 \times F_2$

We start with M_1. The DFA shown in Figure 2.1 recognizes L_1

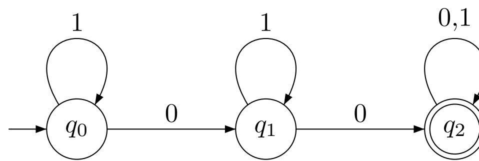

Figure 2.1: $L(M) = \{w \mid$ contains at least two 0s$\}$

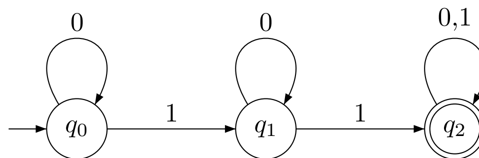

Figure 2.2: $L(M) = \{w \mid$ contains at least two 1s$\}$

M_2 is easy, we just replace 1 with 0 and 0 with 1 in Figure 2.1 with the result shown in Figure 2.2.

We combine M_1 and M_2 to get M

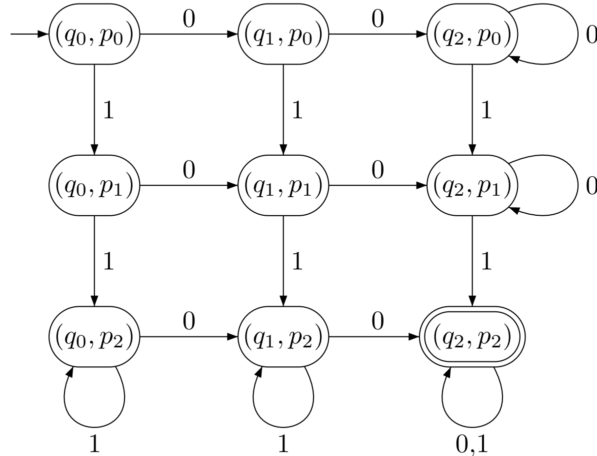

Figure 2.3: $L(M) = \{w \mid w$ contains at least two 1s and at least two 0s$\}$

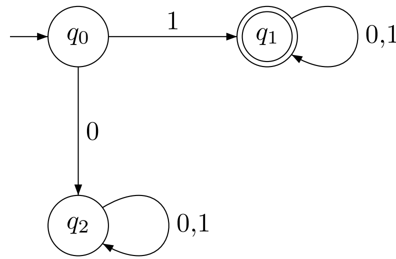

Figure 2.4: $L(M) = \{w \mid w$ begins with a 1$\}$

Example 2.3. *Construct a DFA that recognizes the language $L = \{w \mid w$ begins with a 1 and ends with a 0 $\}$*

Let $L_1 = \{w \mid w$ begins with a 1$\}$ and $L_2 = \{w \mid w$ ends with a 0$\}$. The DFAs that recognize L_1 and L_2 are shown in Figures 2.4 and 2.5 respectively. We use the same procedure to construct the DFA that recognizes $L_1 \cap L_2$. The result is shown in Figure 2.6. It should be noted that the DFA in Figure 2.6 can be simplified further. First the state (q_0, p_1) is unreachable from the starting state. Furthermore, the state (q_2, p_0) is not really needed and we can replace it by having (q_2, p_1) transition to itself on reading a 1 since anyway if we have reached the state (q_2, p_1) it means we will never reach an accept state.

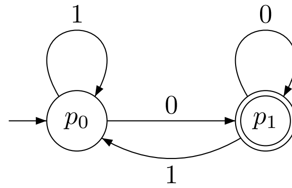

Figure 2.5: $L(M) = \{w \mid w \text{ ends with a } 0\}$

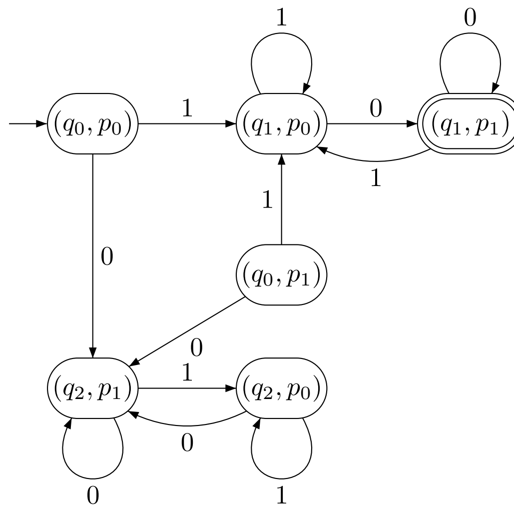

Figure 2.6: $L(M) = \{w \mid w \text{ begins with 1 and ends with a } 0\}$

2.6 Closure Under the Complement Operation

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA with $L(M) = L_1$. Define $\overline{M} = (Q, \Sigma, \delta, s, Q - F)$ to be the complement DFA and $L(\overline{M}) = L_2$.

Theorem 2.2. *The regular languages are closed under the complement operation:* $\overline{L_1} = L_2$.

Proof.

$$\begin{aligned} x \in \overline{L_1} &\Leftrightarrow x \notin L_1 \\ &\Leftrightarrow \hat{\delta}(s, x) \notin F \\ &\Leftrightarrow \hat{\delta}(s, x) \in Q - F \\ &\Leftrightarrow x \in L(\overline{M}) \\ &\Leftrightarrow x \in L_2 \end{aligned}$$

Example 2.4. *Construct a DFA that recognizes the language* $L_2 = \{w \mid w$ *contains maximum one 0* $\}$.

Clearly L_2 is the complement of the language $L_1 = \{w \mid w \text{ contains at least two 0s}\}$ whose DFA is shown in Figure 2.1. Therefore to construct the DFA for L_2 we just turn accepting states into non-accepting states in Figure 2.1 and vice versa. The result is shown in Figure 2.7.

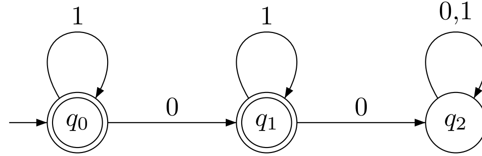

Figure 2.7: $L(M) = \{w \mid w \text{ contains max one zero}\}$

2.7 Closure Under the Union Operation

The class of regular languages is closed under the union operation. If A_1 and A_2 are regular languages so is $A_1 \cup A_2$.

Proof Since A_1 and A_2 are regular then there exits some automata M_1 and M_2 such that $L(M_1) = A_1$ and $L(M_2) = A_2$. To prove that $A_1 \cup A_2$ is regular we construct an automaton M such that $L(M) = A_1 \cup A_2$. Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ then $M = (Q, \Sigma, \delta, q_0, F)$ is defined by

1. $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$. i.e $Q = Q_1 \times Q_2$.
2. $q_0 = (q_1, q_2)$.
3. For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$$

4. $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$. It should be noted hat $F \neq F_1 \times F_2$.

Let $x \in A_1 = L(M_1)$. Because M_1 accepts x then $\hat{\delta}_1(q_1, x) = u$ is an accepting state, i.e. $u \in F_1$. Also let $\hat{\delta}_2(q_2, x) = v$ where $v \in Q_2$ could be any arbitrary state, not necessarily in F_2.

$$\begin{aligned} \hat{\delta}(q_0, x) &= \hat{\delta}((q_1, q_2), x) && \text{from definition} \\ &= (\hat{\delta}_1(q_1, x), \hat{\delta}_2(q_2, x)) && \text{from lemma 1} \\ &= (u, v) \in F && \text{because } u \in F_1, v \in Q_2 \end{aligned}$$

■

Example 2.5. *Construct a DFA that recognizes the language $L = \{w \mid w$ ends with a 0 or contains at least two 1s.*

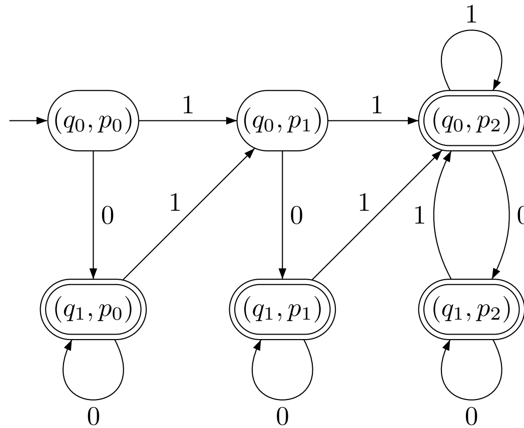

Figure 2.8: $L(M) = \{w \mid w$ ends with a 0 or contains at least two 1s$\}$

We can write $L = L_1 \cup L_2$ where $L_1 = \{w \mid w$ ends with a 0 $\}$ and $L_2 = \{w \mid w$ contains at least two 1s$\}$. We follow the construction shown in section 2.7 which is similar, except for the accept states, to the construction for the intersection operation. Thee DFAs for L_1 and L_2 were already constructed and are shown in Figures 2.5 and 2.2 respectively. We relabel the states in Figure 2.2 as p_0, p_1, p_2 instead of q_0, q_1, q_2. The resulting DFA is shown in Figure 2.8. Note that in the case of intersection (ends with a 0 **and** contains at least two 1s), the only accepting state would be (q_1, p_2).

---

1. Use the closure properties of regular languages to give the state diagram of DFAs recognizing the following languages

   (a) $L = \{w \in \{0,1\}^* : w$ begins with 1 and ends with 0$\}$

   (b) $L = \{w \in \{0,1\}^* : w$ has exactly two 0's and at least two 1's$\}$

   (c) $L = \{w \in \{0,1\}^* : w$ has an even number of 0's and each 0 is followed by a 1$\}$

   (d) $L = \{w \in \{0,1\}^* : w$ starts with a 0 and has odd length or starts with a 1 and has even length$\}$

   (e) $L = \{w \in \{0,1\}^* : w$ does not contain 1010$\}$

2. Construct a DFA to recognize the language of all binary numbers which are multiple of 5. In other words, $L = \{w \in \{0,1\}^* : w$ is a binary number and it is multiple of 5$\}$. Example: $101 \in L$ and $1010 \in L$ but $110 \notin L$.

3. Give the state diagram of NFAs recognizing the following languages

   (a) $L = \{w \in \{0,1\}^* : w$ ends with 00$\}$

   (b) $L = \{w \in \{0,1\}^* : w$ contains the substring 0101$\}$

   (c) $L = \{w \in \{0,1\}^* : w$ does not contain 1$\}$. The NFA should contain **one state only**.

   (d) $L = \{w \in \{a,b\}^* : w$ contains 0 or more a's followed by 0 or more b's$\}$

   (e) $L = \{w \in \{0,1\}^* :$ the final symbol of w has occurred at least once before in $w\}$

4. Use the subset construction to construct DFA's that recognize the same language as in exercises 3a and 3b.

5. Prove that if $(p, wc) \vdash_M^* (q, c)$ then $(p, w) \vdash_M^* (q, \epsilon)$.

6. Let Σ be an alphabet and $D = (Q, \Sigma, \delta, s, F)$ be a finite automaton. Use the definition of the extended function $\hat{\delta}$ to show that $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$, $a \in \Sigma, x \in \Sigma^*, q \in Q$

7. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA such that $\exists a \in \Sigma$ with the property $\delta(q, a) = q$ **for all** $q \in Q$. Show that either $\{a\}^* \subseteq L(M)$ or $\{a\}^* \cap L(M) = \emptyset$.

8. Consider the language $L_k = \{w \in \{0,1\}^* :$ the kth symbol of w from the right is a 1$\}$. Prove that any DFA that recognizes L_k must have at least 2^k states. (Hint: do it by contradiction).

Lecture 3

Non Deterministic Finite Automata

3.1 Non-deterministric Finite Automata

A non-deterministic finite automaton (NFA) is similar to NFA with three additional properties:

1. Epsilon transitions.Unlike a DFA, an NFA can change from one state to another **without** reading an input.
2. Missing transitions. An NFA does not necessarily have a transition on **every** input. In that case the value of the transition function is the empty set ϕ
3. Multiple transitions. An NFA can have **multiple** transitions on the **same** input.

3.2 Example

An example NFA is shown in Figure 3.1. The only difference in this case between an NFA and DFA is that it has multiple transitions from state q_1 when the DFA reads "1". The NFA in Figure 3.1 accepts all strings where the 3rd digit from the right is a "1". The definition for accepting in the case of NFA is slightly different from the DFA case. We say that an NFA accepts a string x if there is at least one path that leads from a starting state to an accepting state. Using the example in Figure 3.1 and the input string "0101" we see that there are three possible paths for the NFA:

- $q_1 \stackrel{0}{\to} q_1 \stackrel{1}{\to} q_2 \stackrel{0}{\to} q_3 \stackrel{1}{\to} q_4$
- $q_1 \stackrel{0}{\to} q_1 \stackrel{1}{\to} q_1 \stackrel{0}{\to} q_1 \stackrel{1}{\to} q_2$
- $q_1 \stackrel{0}{\to} q_1 \stackrel{1}{\to} q_1 \stackrel{0}{\to} q_1 \stackrel{1}{\to} q_1$

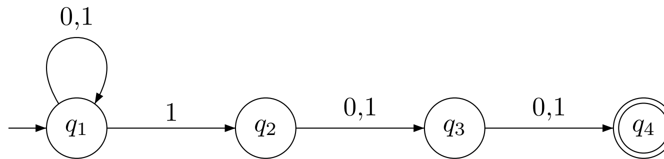

Figure 3.1: Example NFA

Since one of the paths (first one) leads to an accepting state therefore the NFA accepts "0101".

We can construct a DFA to accept the same language. Such a DFA needs to "remember" the last 3 digits and therefore will have 8 states.

3.3 Formal Definition of a Nondeterministic Automaton

Let $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and 2^Q the **power set** of Q, i.e the set of subsets of Q. A nondeterministic finite automaton is a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$ where

1. Q is a finite set of states.
2. Σ is a finite alphabet.
3. $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation.
4. $q_0 \in Q$ is the start state.
5. $F \subseteq Q$ is the set of accept states.

We can take advantage of the isomorphism mentioned in chapter 1 between the set of relations and set of functions. Then we can define Δ as a function:

$$\Delta : Q \times \Sigma \to 2^Q$$

It should be noted that if for a given state and input the transition is missing then it should be the empty set ϕ. For example, in Figure 3.1 there are no transitions from state q_4 on any input this implies $\Delta(q_4, 0) = \Delta(q_4, 1) = \phi$. This should be contrasted with the case for DFAs where there is a transition for every input from any state.

3.4 Computation With NFA

The formal definition of computation for an NFA is similar to DFA. Let $N = (Q, \Sigma, \Delta, q_0, F)$ be an NFA and w a string over the alphabet Σ. We say that N **accepts** w if $w = w_1 w_2 \ldots w_n$ where each $w_i \in \Sigma_\epsilon$ and there is a sequence of states $r_0 r_1 \ldots r_n$ in Q such that

1. $r_0 = q_0$
2. $r_{i+1} \in \Delta(r_i, w_{i+1})$
3. $r_n \in F$

Alternatively, define the multistep transition function

$$\hat{\Delta} : 2^Q \times \Sigma^* \to 2^Q$$

With the properties

$$\hat{\Delta}(R, \epsilon) = R \tag{3.1}$$

$$\hat{\Delta}(R, xa) = \bigcup_{q \in \hat{\Delta}(R,x)} \Delta(q, a) \tag{3.2}$$

We say that N accepts a string w if $\hat{\Delta}(q_0, x) \cap F \neq \phi$. Intuitively, this means that there is an accept state reachable from the start state under the input string x.

A useful property:

$$\begin{aligned}\hat{\Delta}(R, a) &= \bigcup_{q \in \hat{\Delta}(R,\epsilon)} \Delta(q, a) \\ &= \bigcup_{q \in R} \Delta(q, a) \end{aligned} \tag{3.3}$$

3.5 Examples

Example 3.1. *Give an NFA that recognizes the following language:* $L = \{w \in \{a, b\}^* : w =$ *contains an occurrence of bb*$\}$.

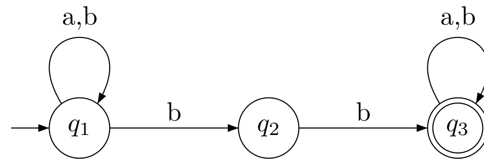

Figure 3.2: Example NFA

If the reflexive transition, $q_1 \to q_1$ and $q_3 \to q_3$, did not exist then the NFA will accept the string *bb* only not *an occurrence* of *bb*. For example the string *babb* which clearly is in L, would be rejected since it reads the first b, transitions to q_2 and ”dies” since there is no transition out of q_2 upon reading a. It is **important** to note also that when the NFA reads the first b it **does not** choose the transition $q_1 \to q_2$. Therefore the NFA ”guesses” correctly which possibility to choose. In general, to build an NFA that matches the occurrence of a string x in an alphabet Σ we just build an NFA that recognizes x **exactly** then we add reflexive transition to the initial and final states. The next example is another illustration of this idea.

Example 3.2. *Give an NFA that recognizes the following language:* $L = \{w \in \{a,b\}^* : w =$ *contains an occurrence of bab*$\}$.

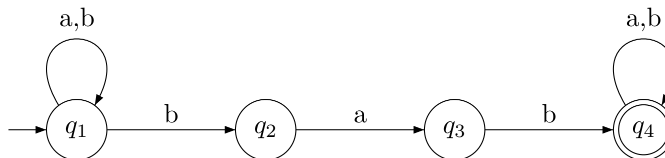

Figure 3.3: Example NFA

We can combine the two NFAs above to produce the following.

Example 3.3. *Give an NFA that recognizes the following language:* $L = \{w \in \{a,b\}^* : w$ *contains an occurrence of bb or bab*$\}$.

There are many possible NFA that recognize L, one of them is shown in Figure 3.4. Consider the string $w = bababab$ which is accepted by the presented NFA. Actually

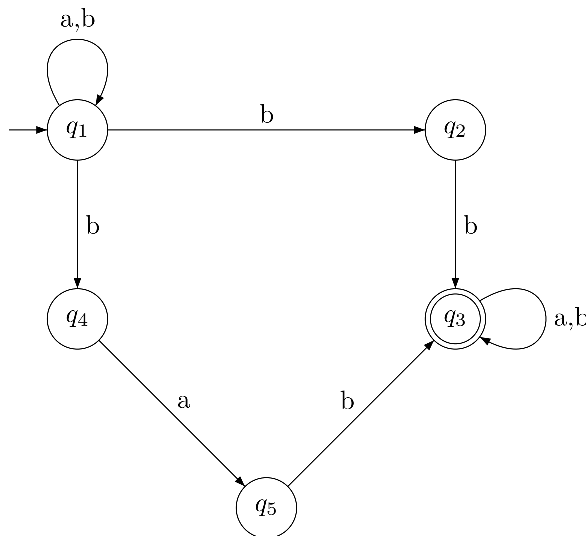

Figure 3.4: Example NFA

there are three ways in which the NFA accepts w. Can you find them?

Lecture 4

Equivalence of DFA and NFA

4.1 Introduction

In this lecture we show that given any NFA we can build a DFA that accepts the same language. It is useful to recall the definition of $\hat{\Delta}$:

$$\hat{\Delta}(R, \epsilon) = R \tag{4.1}$$

$$\hat{\Delta}(R, xa) = \bigcup_{q \in \hat{\Delta}(R,x)} \Delta(q, a) \tag{4.2}$$

$$\begin{aligned} \hat{\Delta}(R, a) &= \bigcup_{q \in \hat{\Delta}(R,\epsilon)} \Delta(q, a) \\ &= \bigcup_{q \in R} \Delta(q, a) \end{aligned} \tag{4.3}$$

A useful particular case of (4.3), if $R = \{p\}$ then

$$\begin{aligned} \hat{\Delta}(\{p\}, a) &= \bigcup_{q \in \{p\}} \Delta(q, a) \\ &= \Delta(p, a) \end{aligned} \tag{4.4}$$

Lemma 4.1. *For all $x, y \in \Sigma^*$,*

$$\hat{\Delta}(R, xy) = \hat{\Delta}(\hat{\Delta}(R, x), y)$$

Proof. We proceed by induction on $|y|$. Let $y = a_1 a_2 \ldots a_n$.
basis. For $y = \epsilon$

$$\begin{aligned} \hat{\Delta}(R, x\epsilon) &= \hat{\Delta}(R, x) \\ &= \hat{\Delta}(\hat{\Delta}(R, x), \epsilon) && \text{from (4.1)} \end{aligned}$$

For $y = a_1$.

$$\begin{aligned}\hat{\Delta}(R, xa_1) &= \bigcup_{q \in \hat{\Delta}(R,x)} \Delta(q, a_1) && \text{from (4.2)} \\ &= \hat{\Delta}(\hat{\Delta}(R, x), a_1) && \text{from (4.3)}\end{aligned}$$

Assume that the relation is true for $a_1 a_2 \ldots a_{n-1}$, i.e. suppose that

$$\hat{\Delta}(R, xa_1 a_2 \ldots a_{n-1}) = \hat{\Delta}(\hat{\Delta}(R, x), a_1 a_2 \ldots a_{n-1})$$

Now the result of the lemma

$$\begin{aligned}\hat{\Delta}(R, xy) &= \hat{\Delta}(R, xa_1 a_2 \ldots a_{n-1} y_n) \\ &= \bigcup_{q \in \hat{\Delta}(R, xa_1 a_2 \ldots a_{n-1})} \Delta(R, a_n) && \text{from (4.2)} \\ &= \bigcup_{q \in \hat{\Delta}(\hat{\Delta}(R,x), a_1 a_2 \ldots a_{n-1})} \Delta(R, a_n) && \text{from induction step} \\ &= \hat{\Delta}(\hat{\Delta}(R, x), a_1 a_2 \ldots a_{n-1} a_n) && \text{from (4.2)} \\ &= \hat{\Delta}(\hat{\Delta}(R, x), y)\end{aligned}$$

Lemma 4.2. *The function $\hat{\Delta}$ commutes with set union: for any family of subsets R_i of Q and $x \in \Sigma^*$,*

$$\hat{\Delta}\left(\bigcup_i R_i, x\right) = \bigcup_i \hat{\Delta}(R_i, x)$$

Proof. By induction on $|x|$
Basis.

$$\begin{aligned}\hat{\Delta}(\bigcup_i R_i, \epsilon) &= \bigcup_i R_i && \text{from (4.1)} \\ &= \bigcup_i \hat{\Delta}(R_i, \epsilon) && \text{from (4.1)}\end{aligned}$$

induction step. Assume that the result is true for strings of length $|x|$, we need to show that it is true for $|xa|$ where $a \in \Sigma$

$$\begin{aligned}\hat{\Delta}(\bigcup_i R_i, xa) &= \bigcup_{p \in \hat{\Delta}(\bigcup_i R_i, x)} \Delta(p, a) && \text{from (4.2)} \\ &= \bigcup_{p \in \bigcup_i \hat{\Delta}(R_i, x)} \Delta(p, a) && \text{from induction hypothesis} \\ &= \bigcup_i \bigcup_{p \in \hat{\Delta}(R_i, x)} \Delta(p, a) && \text{basic set theory} \\ &= \bigcup_i \hat{\Delta}(R_i, xa) && \text{from (4.2)}\end{aligned}$$

A useful particular case of lemma 4.2 is

$$\hat{\Delta}(\{p\}, a) = \hat{\Delta}(p, a) = \Delta(p, a)$$

4.2 Subset Construction

Given an arbitrary NFA, $N = (Q_N, \Sigma, \Delta, S_N, F_N)$. Let $R \subseteq Q_N$ and $a \in \Sigma$,we produce an equivalent DFA, $M = (Q_M, \Sigma, \delta, S_M, F_M)$ as follows

$$\begin{aligned} Q_M &= 2^{Q_N} \\ \delta(R, a) &= \hat{\Delta}(R, a) \\ S_M &= \{S_N\} \\ F_M &= \{R \subseteq Q_N \mid R \cap F_N \neq \emptyset\} \end{aligned}$$

Lemma 4.3. *For any $R \subseteq Q_N$ and $x \in \Sigma^*$, $\hat{\delta}(R, x) = \hat{\Delta}(R, x)$.*

Proof. By induction on $|x|$
Basis.

$$\begin{aligned} \hat{\delta}(R, \epsilon) &= R && \text{from the def. of } \hat{\delta} \\ &= \hat{\Delta}(R, \epsilon) && \text{from the def. of } \hat{\Delta} \end{aligned}$$

induction step. Assume that $\hat{\delta}(R, x) = \hat{\Delta}(R, x)$. Then

$$\begin{aligned} \hat{\delta}(R, xa) &= \delta(\hat{\delta}(R, x), a) && \text{from def. of } \hat{\delta} \\ &= \hat{\Delta}(\hat{\delta}(R, x), a) && \text{by construction} \\ &= \hat{\Delta}(\hat{\Delta}(R, x), a) && \text{induction hypothesis} \\ &= \hat{\Delta}(R, xa) && \text{from lemma 4.1} \end{aligned}$$

Theorem 4.1. *The automata M and N accept the same language*

Proof. For any $x \in \Sigma^*$

$$\begin{aligned} x \in L(M) &\Leftrightarrow \hat{\delta}(s_M, x) \in F_M && \text{definition of acceptance for M} \\ &\Leftrightarrow \hat{\Delta}(s_M, x) \in F_M && \text{lemma 4.3} \\ &\Leftrightarrow \hat{\Delta}(\{s_N\}, x) \in F_M && \text{definition of } s_M \\ &\Leftrightarrow \hat{\Delta}(s_N, x) \in F_M && \text{lemma 4.2} \\ &\Leftrightarrow \hat{\Delta}(s_N, x) \cap F_N \neq \emptyset && \text{definition of } F_M \\ &\Leftrightarrow x \in L(N) && \text{definition of acceptance for N} \end{aligned}$$

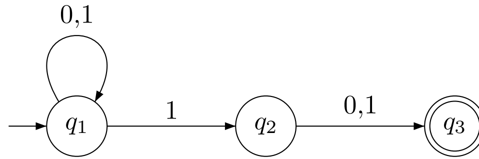

Figure 4.1: NFA that accepts the language defined in Example 11.1

4.3 Subset Construction Examples

Example 4.1. *Consider the language*

$$L = \{x \in \{0,1\}^* \mid \textit{the second symbol from the right is 1}\}$$

We will use the subset construction to build the equivalent DFA. First there will be $2^3 = 8$ subsets

$$\emptyset, \{q_1\}, \{q_2\}, \{q_3\}, \{q_1, q_2\}, \{q_1, q_3\}, \{q_2, q_3\}, \{q_1, q_2, q_3\}$$

Then we use Eq. (4.3), to compute the transition function for the subsets. For a given subset R we compute $\hat{\Delta}(R, a) = \bigcup_{q \in R} \Delta(q, a)$. The results are shown in Table 4.1

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_1\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_2\}$	$\{q_3\}$	$\{q_3\}$
$\{q_3\}$	$\emptyset$	$\emptyset$
$\{q_1, q_2\}$	$\{q_1, q_3\}$	$\{q_1, q_2, q_3\}$
$\{q_1, q_3\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_2, q_3\}$	$\{q_3\}$	$\{q_3\}$
$\{q_1, q_2, q_3\}$	$\{q_1, q_3\}$	$\{q_1, q_2, q_3\}$

Table 4.1: Transition table for DFA that accepts language defined in Example 11.1

If we follow the transitions from state $\{q_1\}$ we see that states $\emptyset$,$\{q_2\}$, $\{q_3\}$, $\{q_2, q_3\}$ are inaccessible and therefore can be omitted. The trimmed down version of the transition table is shown in Table 4.2. Also, if we relabel the states in the trimmed down versions as follows

	0	1
$\rightarrow \{q_1\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	$\{q_1, q_3\}$	$\{q_1, q_2, q_3\}$
$\{q_1, q_3\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_1, q_2, q_3\}$	$\{q_1, q_3\}$	$\{q_1, q_2, q_3\}$

Table 4.2: Trimmed down transition table for DFA for Example 11.1

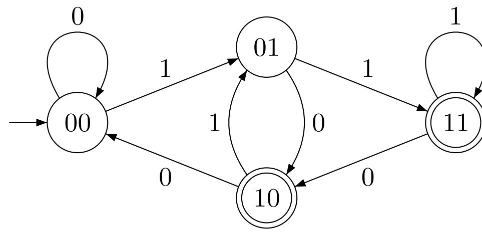

Figure 4.2: DFA that accepts the language defined in Example 11.1

$$\begin{aligned}\{q_1\} &\leftrightarrow 00\\ \{q_1, q_2\} &\leftrightarrow 01\\ \{q_1, q_3\} &\leftrightarrow 10\\ \{q_1, q_2, q_3\} &\leftrightarrow 11\end{aligned}$$

we get exactly the DFA we would have designed originally to recognize the language in Example 11.1 and shown in Figure 4.2.

Example 4.2. *Consider the language*

$$L = \{x \in \{0,1\}^* \mid x \textit{ ends with 1}\}$$

The language in this example is accepted by the NFA shown in Figure 4.3.

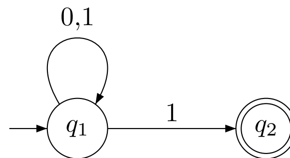

Figure 4.3: NFA that accepts the language defined in Example 11.2

We use the subset construction to build the equivalent DFA. First there will be $2^2 = 4$ subsets

$$\emptyset, \{q_1\}, \{q_2\}, \{q_1, q_2\}$$

Then we use Eq. (4.3), to compute the transition function for the subsets. For a given subset R we compute $\hat{\Delta}(R, a) = \bigcup_{q \in R} \Delta(q, a)$. The results are shown in Table 4.3 Again we see that state $\{q_2\}$ and $\emptyset$ are not reachable therefore can be omitted and the DFA reduces to having only 2 states. The (expected) result is shown in Figure 4.4

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_1\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_1, q_2\}$	$\{q_1\}$	$\{q_1, q_2\}$

Table 4.3: Transition table for DFA that accepts language defined in Example 11.2

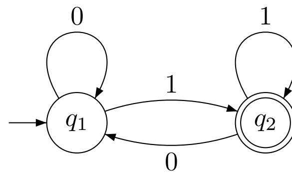

Figure 4.4: DFA that accepts the language defined in Example 11.2

Lecture 5

Non Deterministic Finite Automata with ϵ Transitions

5.1 Introduction

In this section we introduce another "feature" for NFAs. we allow transition on the empty string ϵ. An NFA can make a transition spontaneously without reading any input. This new capability while useful does not add any power to NFAs.

Example 5.1. *The language $L = \{a^n \mid n \text{ is even or multiple of 3}\}$ is recognized by the ϵ-NFA shown in Figure 5.1.*

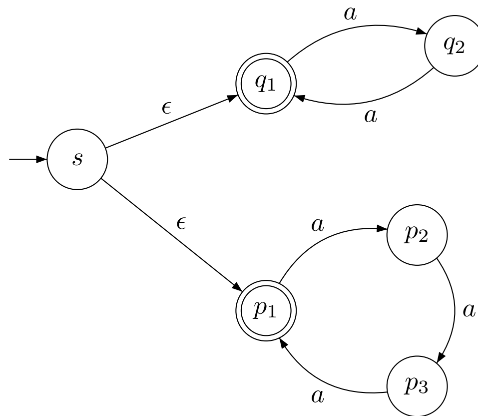

Figure 5.1: ϵ-NFA to recognize language of example 5.1

The ϵ-NFA "guesses" which path to take depending whether the input has an even number of a's or multiple of 3. This is done without "consuming" any input, hence the ϵ-transitions.

5.2 Formal Definition of an ϵ-NFA

An ϵ-NFA A is represented by $A = (Q, \Sigma, \Delta, q_0, F)$ where all the components have the same meaning as for NFAs except that the domain of Δ is $\Sigma \cup \{\epsilon\}$. We require that ϵ cannot be a member of Σ.

Example 5.2. *Figure 5.2 accepts decimal numbers consisting of an optional +/- sign followed by a sequence of digits, a dot then a second sequence of digits. One of the sequence of digits can be empty but not both.*

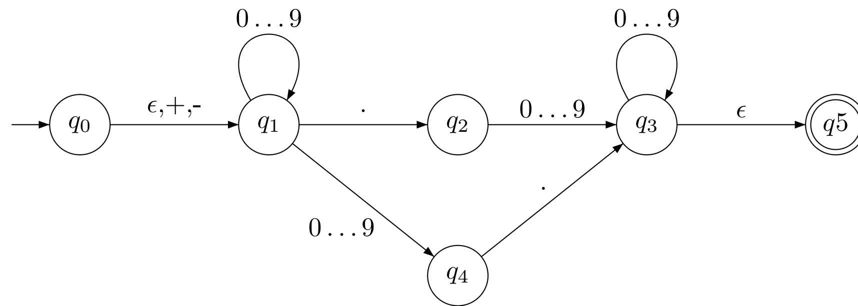

Figure 5.2: ϵ-FNA to recoginize the language of example 5.2

The ϵ-NFA of example 5.2 can be represented formally as

$$E = (\{q_1, \dots, q_5\}, \{\cdot, +, -, 0, \dots, 9\}, \Delta, q_0, \{q_5\})$$

where the transition table for Δ is shown in Table 5.1.

	ϵ	+,-	$\cdot$	$0 \dots 9$
q_0	$\{q_1\}$	$\{q_1\}$	ϕ	ϕ
q_1	ϕ	ϕ	$\{q_2\}$	$\{q_1, q_4\}$
q_2	ϕ	ϕ	ϕ	$\{q_3\}$
q_3	$\{q_5\}$	ϕ	ϕ	$\{q_3\}$
q_4	ϕ	ϕ	$\{q_3\}$	ϕ
q_5	ϕ	ϕ	ϕ	ϕ

Table 5.1: Transition table for example 5.2

Example 5.3. *Let $\Sigma = \{a_1, \dots, a_n\}$ be alphabet with n symbols and let $L = \{w \in \Sigma :$ there is a symbol $a_i \in \Sigma$ not appearing in $w\}$.*

Here also we use ϵ-transitions to solve the problem. First define $\bar{a}_i = \Sigma - \{a_i\}$. then an NFA that accepts all the strings that do not contain a_i is shown in Figure 5.3. We build n such NFAs, one for every different symbol, and combine them using ϵ-transition. The resulting NFA, shown in Figure 5.4, would "guess" which symbol is missing uses the appropriate ϵ-transition.

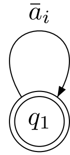

Figure 5.3: Strings that do not contain a_i

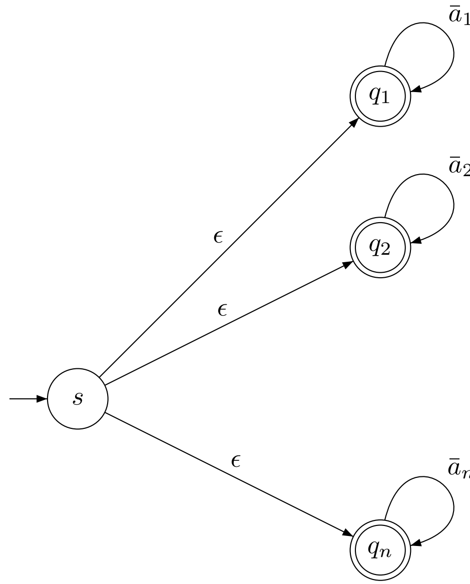

Figure 5.4: Strings that do not contain all $a_i \in \Sigma$

5.3 Epsilon Closure

Before we proceed with the study of the properties of ϵ-NFAs we need to define an important concept called the *ϵ-closure* of a state. Informally, the *ϵ-closure* of a state q is a set containing q and all states that can be reached from q along a path containing ϵ-transitions only. Formally, the *ϵ-closure* of a state q, $\mathcal{E}(q)$ is defined recursively as follows

Basis: state $q \in \mathcal{E}(q)$
Induction: if $p \in \mathcal{E}(q)$ then $\Delta(p, \epsilon) \subseteq \mathcal{E}(q)$. In other words, if $p \in \mathcal{E}(q)$ then for all $s \in \Delta(p, \epsilon)$, $s \in \mathcal{E}(q)$.

Example 5.4. *As an example of ϵ-closure consider the ϵ-NFA in Figure 5.5 and compute $\mathcal{E}(q_1)$ we get*

$$\mathcal{E}(q_1) = \{q_1, q_2, q_3, q_4, q_6\}$$

This is because each of the states listed can be reached from q_1 along a path containing

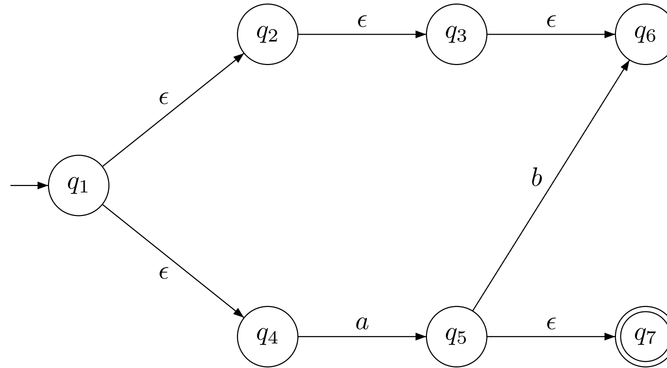

Figure 5.5: Example for ϵ-*closure*

ϵ-transitions only. For example, state q_6 is reached from q_1 along $q_1 \to q_2 \to q_3 \to q_6$ while $q_7 \notin \mathcal{E}(q_1)$ because to reach q_7 from q_1 we must use the transition $q_4 \to q_5$ which is not an ϵ-transition.

It should be noted that the ϵ-*closure* of a set is the union of the e-*closure* of its members:

$$\mathcal{E}(R) = \bigcup_{p \in R} \mathcal{E}(p)$$

5.4 Computing With ϵ-NFA's

Let $E = (Q, \Sigma, \Delta, q_0, F)$ be an ϵ-NFA. We need to define what happens when E reads a sequence of inputs. For $R \subseteq Q$, $a \in \Sigma$, $x \in \Sigma^*$ The extended transition function $\hat{\Delta}$ is defined recursively as follows

$$\hat{\Delta}(R, \epsilon) = \mathcal{E}(R) \tag{5.1}$$

$$\hat{\Delta}(R, xa) = \mathcal{E}\left(\bigcup_{p \in \hat{\Delta}(R,x)} \Delta(p, a)\right) \tag{5.2}$$

In particular, if we set $x = \epsilon$ and $R = \{q\}$ in (5.2) we get

$$\hat{\Delta}(q, a) = \mathcal{E}\left(\bigcup_{p \in \hat{\Delta}(q,\epsilon)} \Delta(p, a)\right)$$

$$= \mathcal{E}\left(\bigcup_{p \in \mathcal{E}(q)} \Delta(p, a)\right) \tag{5.3}$$

In Equation (5.3) the starting state of $\hat{\Delta}(q, a)$ is not the state q but the ϵ-*closure* of q, i.e. all the states that can be reached from q by ϵ-transitions. Also, $\hat{\Delta}(q, a)$ includes the states that can be reached by ϵ-transitions from all states in $\Delta(q, a)$.

Example 5.5. *We illustrate the use of equations (5.1- 5.3) by computing $\hat{\Delta}(q_0, 1.2)$ using the ϵ-NFA of example 5.2.*

$$
\begin{aligned}
\hat{\Delta}(q_0, 1) &= \mathcal{E}\left(\bigcup_{p\in\mathcal{E}(1)} \Delta(p, 1)\right) \\
&= \mathcal{E}\left(\bigcup_{p\in\{q_0,q_1\}} \Delta(p, 1)\right) \\
&= \mathcal{E}\left(\Delta(q_0, 1) \cup \Delta(q_1, 1)\right) \\
&= \mathcal{E}\left(\phi \cup \{q_1, q_4\}\right) \\
&= \{q_1, q_4\}
\end{aligned}
$$

$$
\begin{aligned}
\hat{\Delta}(q_0, 1.) &= \mathcal{E}\left(\bigcup_{p\in\hat{\Delta}(q_0,1)} \Delta(p, \cdot)\right) \\
&= \mathcal{E}\left(\Delta(q_1, \cdot) \cup \Delta(q_4, \cdot)\right) \\
&= \mathcal{E}\left(\{q_2, q_3\}\right) \\
&= \{q_2, q_3, q_5\}
\end{aligned}
$$

Note that $\hat{\Delta}(q_0, 1.) \cap \{q_5\} \neq \phi$ and therefore by the definition of acceptance the string "1." is recognized by the ϵ-NFA. Continuing with the string

$$
\begin{aligned}
\hat{\Delta}(q_0, 1.2) &= \mathcal{E}\left(\bigcup_{p\in\hat{\Delta}(q_0,1.)} \Delta(p, 2)\right) \\
&= \mathcal{E}\left(\Delta(q_2, 2) \cup \Delta(q_3, 2) \cup \Delta(q_5, 2)\right) \\
&= \mathcal{E}\left(\{q_3\} \cup \{q_3\} \cup \phi\right) \\
&= \{q_3, q_5\}
\end{aligned}
$$

Since $\hat{\Delta}(q_0, 1.2) \cap \{q_5\} \neq \phi$, the ϵ-NFA accepts the string "1.2". As a counter example if we consider the string "1.." (with 2 dots) we would get

$$
\begin{aligned}
\hat{\Delta}(q_0, 1..) &= \mathcal{E}\left(\bigcup_{p\in\hat{\Delta}(q_0,1.)} \Delta(p, .)\right) \\
&= \mathcal{E}\left(\Delta(q_2, .) \cup \Delta(q_3, .) \cup \Delta(q_5, .)\right) \\
&= \mathcal{E}\left(\phi \cup \phi \cup \phi\right) \\
&= \phi
\end{aligned}
$$

5.5 Equivalence of ϵ-NFA and DFA

Given an ϵ-NFA E we can construct a DFA D that accepts the same language, $L(E) = L(D)$. The construction is very similar to the subset construction for NFAs. Let $E = (Q_E, \Sigma, \Delta, s_E, F_E)$ then the equivalent DFA $D = (Q_D, \Sigma, \delta, s_D, F_D)$ is defined as follows:

1. Q_D is the set of subsets of Q_E: $Q_D = 2^{Q_E}$.
2. $s_D = \mathcal{E}(s_E)$. The starting state of D is the ϵ-*closure* of the starting state of E.
3. $F_D \subseteq Q_D$ is a set of subsets of Q_E where each set has at least one accepting state of E. $F_D = \{S \mid S \subseteq Q_D \text{ and } S \cap F_E \neq \emptyset\}$
4. $\delta(R, a) = \hat{\Delta}(R, a)$.

As an example we construct a DFA from the ϵ-NFA discussed previous lectures and shown in Figure 5.6. The corresponding DFA is shown in Figure 5.7.

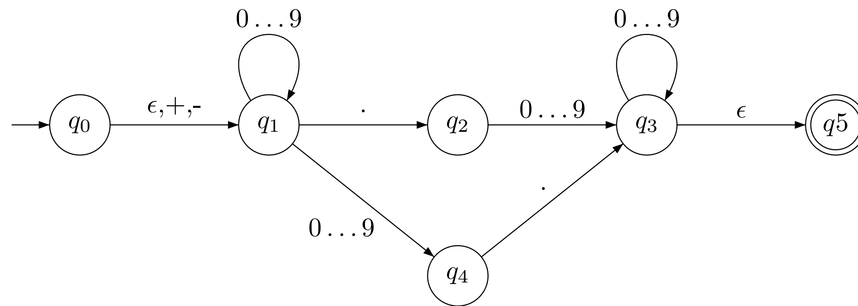

Figure 5.6: ϵ-NFA that accepts decimal numbers

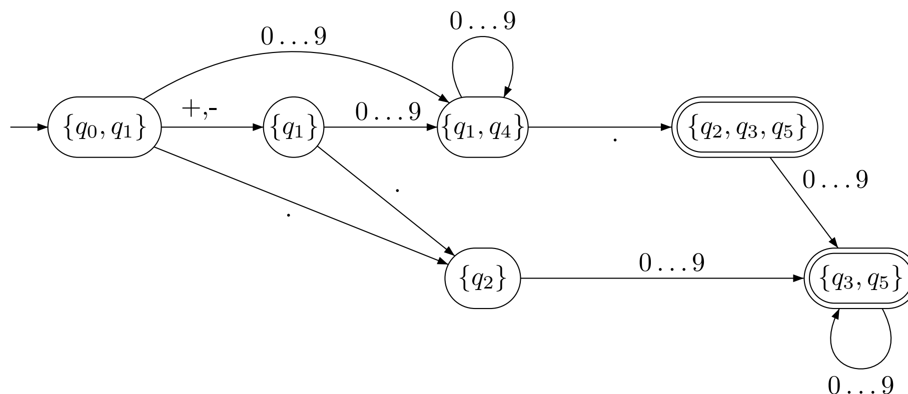

Figure 5.7: A DFA equivalent to the ϵ-NFA in Figure 5.6

©Hikmat Farhat

In should be noted that we have omitted the state $\emptyset$ and all transitions to it from the DFA in Figure 5.7. For any state in Figure 5.7 there is a transition to state $\emptyset$ for any input that is not shown in the figure. For example, $\Delta(\{q_1\},+) = \Delta(\{q_1\},-) = \emptyset$. The DFA in Figure 5.7 was obtained as follows. First, the start state of D is $\mathcal{E}(q_0) = \{q_0, q_1\}$. We will generate the states of D starting from $\{q_0, q_1\}$.

$$\begin{aligned}
\delta(\{q_0,q_1\},+) &= \mathcal{E}(\{q_1\} \cup \emptyset) = \{q_1\} \\
\delta(\{q_0,q_1\},-) &= \mathcal{E}(\{q_1\} \cup \emptyset) = \{q_1\} \\
\delta(\{q_0,q_1\},.) &= \mathcal{E}(\emptyset \cup \{q_2\}) = \{q_2\} \\
\delta(\{q_0,q_1\},0\ldots 9) &= \mathcal{E}(\emptyset \cup \{q_1,q_4\}) = \{q_1,q_4\}
\end{aligned}$$

We have obtained three new states: $\{q_1\}$, $\{q_2\}$ and $\{q_1, q_4\}$. We compute the transitions for each state on all possible inputs. For $\{q_1\}$:

$$\begin{aligned}
\delta(\{q_1\},+) &= \mathcal{E}(\emptyset) = \phi \\
\delta(\{q_1\},-) &= \mathcal{E}(\emptyset) = \phi \\
\delta(\{q_1\},.) &= \mathcal{E}(\emptyset \cup \{q_2\}) = \{q_2\} \\
\delta(\{q_1\},0\ldots 9) &= \mathcal{E}(\{q_1,q_4\}) = \{q_1,q_4\}
\end{aligned}$$

No new states were generated by the above computation. We proceed to compute the same for $\{q_2\}$.

$$\begin{aligned}
\delta(\{q_2\},+) &= \mathcal{E}(\emptyset) = \emptyset \\
\delta(\{q_2\},-) &= \mathcal{E}(\emptyset) = \emptyset \\
\delta(\{q_2\},.) &= \mathcal{E}(\emptyset) = \emptyset \\
\delta(\{q_2\},0\ldots 9) &= \mathcal{E}(\{q_3\}) = \{q_3,q_5\}
\end{aligned}$$

A new state, $\{q_3, q_5\}$ was generated and we include it in our computation. For $\{q_1, q_4\}$

$$\begin{aligned}
\delta(\{q_1,q_4\},+) &= \mathcal{E}(\emptyset \cup \emptyset) = \emptyset \\
\delta(\{q_1,q_4\},-) &= \mathcal{E}(\emptyset) = \emptyset \\
\delta(\{q_1,q_4\},.) &= \mathcal{E}(\{q_2\} \cup \{q_3\}) = \{q_2,q_3,q_5\} \\
\delta(\{q_1,q_4\},0\ldots 9) &= \mathcal{E}(\emptyset \cup \emptyset) = \emptyset
\end{aligned}$$

A new state, $\{q_2, q_3, q_5\}$ was generated and we include it in our computation. We proceed with $\{q_3, q_5\}$

$$\begin{aligned}
\delta(\{q_3,q_5\},+) &= \mathcal{E}(\emptyset \cup \emptyset) = \emptyset \\
\delta(\{q_3,q_5\},-) &= \mathcal{E}(\emptyset \cup \emptyset) = \emptyset \\
\delta(\{q_3,q_5\},.) &= \mathcal{E}(\emptyset \cup \emptyset) = \emptyset \\
\delta(\{q_3,q_5\},0\ldots 9) &= \mathcal{E}(\{q_3\} \cup \emptyset) = \{q_3,q_5\}
\end{aligned}$$

Finally

$$\delta(\{q_2, q_3, q_5\}, +) = \mathcal{E}(\emptyset \cup \emptyset \cup \emptyset) = \emptyset$$
$$\delta(\{q_2, q_3, q_5\}, -) = \mathcal{E}(\emptyset \cup \emptyset \cup \emptyset) = \emptyset$$
$$\delta(\{q_2, q_3, q_5\}, .) = \mathcal{E}(\emptyset \cup \emptyset \cup \emptyset) = \emptyset$$
$$\delta(\{q_2, q_3, q_5\}, 0 \ldots 9) = \mathcal{E}(\{q_3\} \cup \{q_3\} \cup \emptyset) = \{q_3, q_5\}$$

Lemma 5.1. $\mathcal{E}(\mathcal{E}(q)) = \mathcal{E}(q)$

Proof. Let $\mathcal{E}(q) = \{q, p_1, p_2, \ldots, p_n\}$. Then $\mathcal{E}(\mathcal{E}(q)) = \mathcal{E}(q) \cup \mathcal{E}(p_1) \cup \mathcal{E}(p_2) \ldots \cup \mathcal{E}(p_n)$. Clearly $\mathcal{E}(q) \subseteq \mathcal{E}(\mathcal{E}(q))$. To show that $\mathcal{E}(\mathcal{E}(q)) \subseteq \mathcal{E}(q)$ we pick an arbitrary element in $\mathcal{E}(\mathcal{E}(q))$ and show that it is in $\mathcal{E}(q)$. Let $s \in \mathcal{E}(p_i)$ then s is reachable from p_i along a path which contains ϵ-transitions exclusively, we write this symbolically as $p_i \overset{\epsilon}{\leadsto} s$. On the other hand $p_i \in \mathcal{E}(q)$ so $q \overset{\epsilon}{\leadsto} p_i$, therefore $q \overset{\epsilon}{\leadsto} s$ which implies that $s \in \mathcal{E}(q)$.

■

Lemma 5.2. *Let $E = (Q_E, \Sigma, \Delta, s_E, F_E)$ be an ϵ-NFA. For any $R \subseteq Q_E$ and $x \in \Sigma^*$*

$$\mathcal{E}(\hat{\Delta}(R, x)) = \hat{\Delta}(R, x) \tag{5.4}$$
$$\hat{\Delta}(\mathcal{E}(R), x) = \hat{\Delta}(R, x) \tag{5.5}$$

Proof. Let $x = ya$, $y \in \Sigma^*$, $a \in \Sigma$. Then

$$\begin{aligned}
\mathcal{E}(\hat{\Delta}(R, x)) &= \mathcal{E}\left(\hat{\Delta}(R, ya)\right) \\
&= \mathcal{E}\left(\mathcal{E}\left(\bigcup_{p \in \hat{\Delta}(R,y)} \Delta(p, a)\right)\right) \\
&= \mathcal{E}\left(\bigcup_{p \in \hat{\Delta}(R,y)} \Delta(p, a)\right) && \text{from lemma 5.1} \\
&= \hat{\Delta}(R, x)
\end{aligned}$$

Lemma 5.3. *Let $E = (Q_E, \Sigma, \Delta, s_E, F_E)$ be an ϵ-NFA. For any $R \subseteq Q_E$ and $x, y \in \Sigma^*$*

$$\hat{\Delta}(R, xy) = \hat{\Delta}(\hat{\Delta}(R, x), y) \tag{5.6}$$

Proof. By induction on $\mid x \mid$.
Basis.

$$\begin{aligned}
\hat{\Delta}(R, x\epsilon) &= \hat{\Delta}(R, x) \\
&= \mathcal{E}\left(\hat{\Delta}(R, x)\right) && \text{from lemma 5.2} \\
&= \hat{\Delta}(\hat{\Delta}(R, x), \epsilon) && \text{from definition of } \hat{\Delta}
\end{aligned}$$

induction step. Suppose that $\hat{\Delta}(R, xy) = \hat{\Delta}(\hat{\Delta}(R, x), y)$ and let $a \in \Sigma$

$$
\begin{aligned}
\hat{\Delta}(R, xya) &= \mathcal{E}\left(\bigcup_{p\in\hat{\Delta}(R,xy)} \Delta(p, a)\right) \\
&= \mathcal{E}\left(\bigcup_{p\in\hat{\Delta}(\hat{\Delta}(R,x),y)} \hat{\Delta}(p, a)\right) && \text{induction hypothesis} \\
&= \hat{\Delta}(\hat{\Delta}(R, x), ya) && \text{from definition}
\end{aligned}
$$

Theorem 5.1. *A string w is accepted by an ϵ-NFA $E = (Q_E, \Sigma, \Delta, s_E, F_E)$ if and only if it is accepted by some DFA $D = (Q_D, \Sigma, \delta, s_D, F_D)$.*

Proof. We need to prove that for any $x \in \Sigma^*$, $\hat{\delta}(s_D, x) = \hat{\Delta}(s_E, x)$. We proceed by induction on $| x |$.
Basis.

$$
\begin{aligned}
\hat{\delta}(s_D, \epsilon) &= s_D \\
&= \mathcal{E}(s_E) \\
&= \hat{\Delta}(s_E, \epsilon)
\end{aligned}
$$

induction step. Assume that $\hat{\delta}(s_D, x) = \hat{\Delta}(s_E, x)$ and let $a \in \Sigma$.

$$
\begin{aligned}
\hat{\delta}(s_D, xa) &= \delta(\hat{\delta}(s_D, x), a) && \text{from definition of } \hat{\delta} \\
&= \delta(\hat{\Delta}(s_E, x), a) && \text{induction hypothesis} \\
&= \hat{\Delta}(\hat{\Delta}(s_E, x), a) && \text{from definition of } \delta \\
&= \hat{\Delta}(s_E, xa) && \text{from lemma 5.3}
\end{aligned}
$$

■

The following is a property that we will use later.

Lemma 5.4. *The function $\hat{\Delta}$ commutes with set union: for any sets A_i and $x \in \Sigma^*$*

$$\hat{\Delta}(\bigcup_i A_i, x) = \bigcup_i \hat{\Delta}(A_i, x)$$

Proof. By induction on $|x|$.
Basis. From the definition of ϵ-closure we have

$$
\begin{aligned}
\hat{\Delta}(\bigcup_i A_i, \epsilon) &= \mathcal{E}(\bigcup_i A_i) && \text{from definition of } \hat{\Delta} \\
&= \bigcup_i \mathcal{E}(A_i) && \text{from the definition of } \mathcal{E} \\
&= \bigcup_i \hat{\Delta}(A_i, \epsilon)
\end{aligned}
$$

Induction step. Assume that $\hat{\Delta}(\bigcup_i A_i, x) = \bigcup_i \hat{\Delta}(A_i, x)$ for $|x| = n$ and let $a \in \Sigma$.

$$
\begin{aligned}
\hat{\Delta}(\bigcup_i A_i, xa) &= \mathcal{E}\left(\bigcup_{p\in\hat{\Delta}(\bigcup_i A_i, x)} \Delta(p,a)\right) && \text{from definition of } \hat{\Delta} \\
&= \mathcal{E}\left(\bigcup_{p\in\bigcup_i \hat{\Delta}(A_i, x)} \Delta(p,a)\right) && \text{from induction hypothesis} \\
&= \mathcal{E}\left(\bigcup_i \bigcup_{p\in\hat{\Delta}(A_i, x)} \Delta(p,a)\right) && \text{basic set theory} \\
&= \bigcup_i \mathcal{E}\left(\bigcup_{p\in\hat{\Delta}(A_i, x)} \Delta(p,a)\right) && \text{definition of } \mathcal{E} \\
&= \bigcup_i \hat{\Delta}(A_i, xa)
\end{aligned}
$$

■

5.6 Closure Properties

The introduction of ϵ-NFAs makes it much easier to prove the remaining closure properties of regular languages: concatenation and Kleene start. To illustrate the power of the method we again prove the closure under the union operation.

5.6.1 Closure under Union

Assume that L_1 and L_2 are regular languages and $N_1 = (Q_1, \Sigma, \Delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Delta_2, q_2, F_2)$ are ϵ-NFAs such that $L_1 = L(N_1)$ and $L_2 = L(N_2)$. We build an ϵ-NFAs $N = L(L_1 \cup L_2)$ as shown in Figure 5.8. The construction below provides a formal proof.

Proof. Define $N = (Q, \Sigma, \Delta, q_0, F)$ with

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$. With q_0 a new state.
- There are ϵ-transitions from q_0 to q_1 and to q_2. In other words $\mathcal{E}(q_0) = \{q_0, q_1, q_2\}$.
- $F = F_1 \cup F_2$.
- $\Delta(q,a) = \begin{cases} \Delta_1(q,a) & \text{if } q \in Q_1 \\ \Delta_2(q,a) & \text{if } q \in Q_2 \end{cases}$

TODO show that with the exception of empty string $\hat{\Delta}(q_0, x) = \hat{\Delta}(q_1, x) \bigcup \hat{\Delta}(q_2, x)$

Let $x \in L_1 \cup L_2$. There are two identical cases: either $x \in L_1$ or $x \in L_2$ (or both).

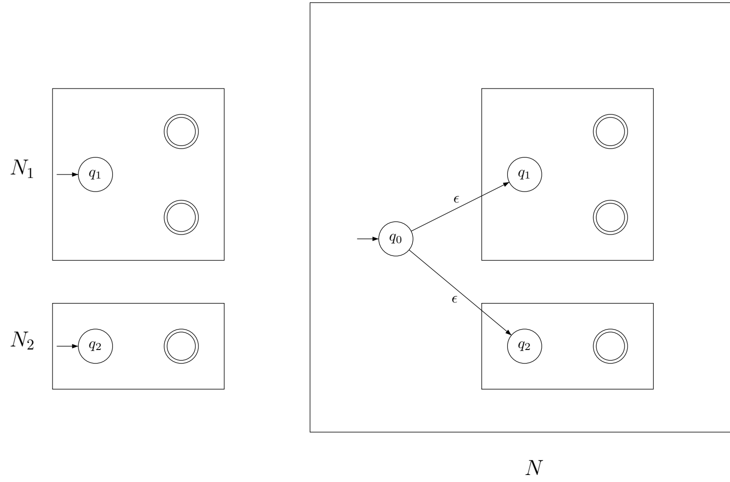

Figure 5.8: ϵ-NFA to accept $L_1 \cup L_2$

Assume that $x \in L_1$ (the other case is similar).

$$\begin{aligned}
\hat{\Delta}(q_0, x) &= \hat{\Delta}(q_0, \epsilon x) \\
&= \hat{\Delta}(\hat{\Delta}(q_0, \epsilon), x) \\
&= \hat{\Delta}(\mathcal{E}(q_0), x) && \text{from definition} \\
&= \hat{\Delta}(\{q_0, q_1, q_2\}, x) && \text{by construction} \\
&= \hat{\Delta}(q_0, x) \cup \hat{\Delta}(q_1, x) \cup \hat{\Delta}(q_2, x) && \text{by lemma 5.4}
\end{aligned}$$

therefore

$$\begin{aligned}
\hat{\Delta}(q_0, x) \cap F_1 &= \Big(\hat{\Delta}(q_0, x) \cup \hat{\Delta}(q_1, x) \cup \hat{\Delta}(q_2, x)\Big) \cap F_1 \\
&\neq \emptyset
\end{aligned}$$

Because $\hat{\Delta}(q_1, x) \cap F_1 \neq \emptyset$ by definition of x begin accepted by N_1.

5.6.2 Closure under Concatenation

Assume that L_1 and L_2 are regular languages and $N_1 = (Q_1, \Sigma, \Delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Delta_2, q_2, F_2)$ are ϵ-NFAs such that $L_1 = L(N_1)$ and $L_2 = L(N_2)$. We build an ϵ-NFAs $N = L(L_1 L_2)$ as shown in Figure 5.9. The construction below provides a formal proof.

- $Q = Q_1 \cup Q_2$.
- For every $p \in F_1$, $\hat{\Delta}(p, \epsilon) = q_2$.
- $F = F_2$.

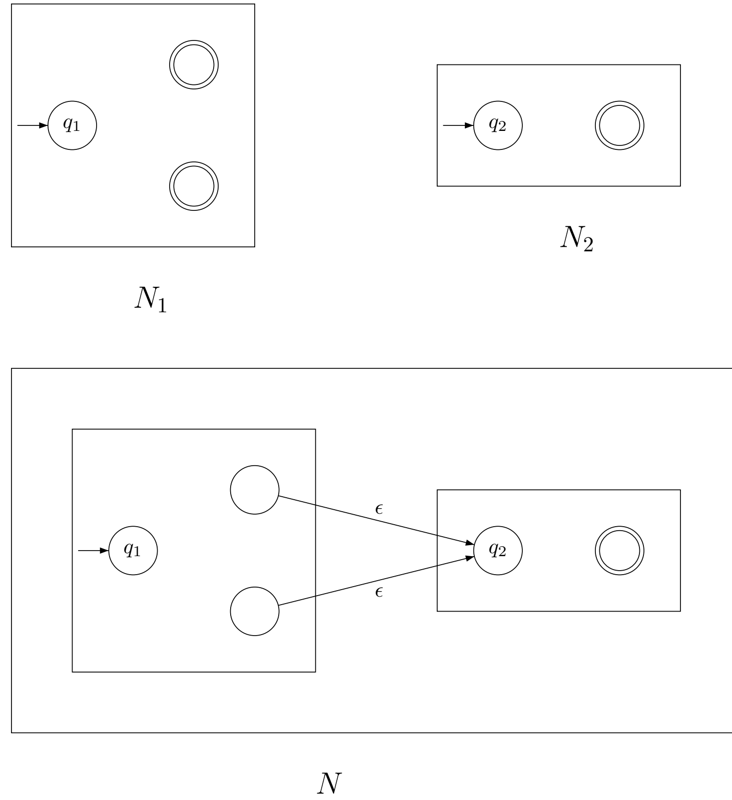

Figure 5.9: ϵ-NFA to accept L_1L_2

- $\Delta(q,a) = \begin{cases} \Delta_1(q,a) & \text{if } q \in Q_1 - F_1 \\ \Delta_1(q,a) & \text{if } q \in F_1 \text{ and } a \neq \epsilon \\ \Delta_1(q,a) \cup \{q_2\} & \text{if } q \in F_1 \text{ and } a = \epsilon \\ \Delta_2(q,a) & \text{if } q \in Q_2 \end{cases}$

Let $w \in L_2$ then we can write $w = xy$ where $x \in L_1$ and $y \in L_2$.

$$\begin{aligned}
\hat{\Delta}(q_1, w) &= \hat{\Delta}(q_0, xy) \\
&= \hat{\Delta}(\hat{\Delta}(q_1, x), y) \\
&= \hat{\Delta}(f, y) && f \in F_1 \text{ because } x \in L_1 \\
&= \hat{\Delta}(f, \epsilon y) \\
&= \hat{\Delta}(\hat{\Delta}(f, \epsilon), y) && \text{from lemma 5.3} \\
&= \hat{\Delta}(\mathcal{E}(f), y) && \text{from definition}
\end{aligned}$$

From the definition of Δ and the fact that $f \in F_1$ implies that $q_2 \in \Delta(f, \epsilon)$ and therefore $q_2 \in \mathcal{E}(f)$. Let $S = \mathcal{E} - \{q_2\}$ then we can write $\mathcal{E} = \{q_2\} \cup S$. Continuing the derivation above we get

$$\begin{aligned}
\hat{\Delta}(q_1, w) &= \hat{\Delta}(S \cup \{q_2\}, y) \\
&= \hat{\Delta}(S, y) \cup \hat{\Delta}(q_2, y) && \text{from lemma 5.4}
\end{aligned}$$

therefore

$$\begin{aligned}
\hat{\Delta}(q_1, w) \cap F_2 &= \Big(\hat{\Delta}(S, y) \cup \hat{\Delta}(q_2, y)\Big) \cap F_2 \\
&\neq \emptyset
\end{aligned}$$

Because $\hat{\Delta}(q_2, y) \neq \emptyset$ by definition of y being accepted by N_2.

5.6.3 Closure under Kleene star

Assume that L_1 is a regular languages and $N_1 = (Q_1, \Sigma, \Delta_1, q_1, F_1)$ is an ϵ-NFAs such that $L_1 = L(N_1)$. We build an ϵ-NFAs $N = L(L_1)^*$ as shown in Figure 5.10. The construction below provides a formal proof.

- $Q = Q_1 \cup \{q_0\}$.
- q_0 is the start state of N.
- $F = F_1 \cup \{q_0\}$.
- $\Delta(q, a) = \begin{cases} \Delta_1(q, a) & \text{if } q \in Q_1 - F_1 \\ \Delta_1(q, a) & \text{if } q \in F_1 \text{ and } a \neq \epsilon \\ \Delta_1(q, a) \cup \{q_1\} & \text{if } q \in F_1 \text{ and } a = \epsilon \\ \{q_1\} & \text{if } q = q_0 \text{ and } a = \epsilon \\ \emptyset & \text{if } q = q_0 \text{ and } a \neq \epsilon \end{cases}$

Proof. If $x \in L_1^*$ then we can write $x = x_1 \ldots x_n$ for some $n \geq 0$ and $x_i \in L_1$ for all i. We will prove that N accepts x by induction on n.
Basis. $n = 0$ then $x = \epsilon$ and $\hat{\Delta}(q_0, \epsilon) = \mathcal{E}(q_0) \cap F \neq \emptyset$ because $q_0 \in \mathcal{E}(q_0)$ and $q_0 \in F$.
Induction hypothesis. Assume that $x = x_1 \ldots x_n$ and $\hat{\Delta}(q_0, x) \cap F \neq \emptyset$.
Induction step. Let $f \in \hat{\Delta}(q_0, x) \cap F$ be an arbitrary accepting state and let $S = \hat{\Delta}(q_0, x) - \{f\}$. Then

$$\begin{aligned}
\hat{\Delta}(q_0, x_1 x_2 \ldots x_{n+1}) &= \hat{\Delta}(\hat{\Delta}(q_0, x_1 \ldots x_n), x_{n+1}) \\
&= \hat{\Delta}(S \cup \{f\}, x_{n+1}) \\
&= \hat{\Delta}(S, x_{n+1}) \cup \hat{\Delta}(f, x_{n+1}) && \text{by lemma 5.4}
\end{aligned}$$

Now $\hat{\Delta}(f, x_{n+1}) = \hat{\Delta}(\hat{\Delta}(f, \epsilon), x_{n+1}) = \hat{\Delta}(\mathcal{E}(f), x_{n+1})$ and by construction $q_1 \in \mathcal{E}(f)$ for all $f \in F_1$ therefore $\hat{\Delta}(q_1, x_{n+1}) \subseteq \hat{\Delta}(f, x_{n+1})$. Continuing the derivation above we get

$$\begin{aligned}
\hat{\Delta}(q_0, x_1 x_2 \ldots x_{n+1}) &= \hat{\Delta}(S, x_{n+1}) \cup \hat{\Delta}(f, x_{n+1}) \\
&\supseteq \hat{\Delta}(q_1, x_{n+1}) && \text{from discussion above} \\
&\cap F_1 \neq \emptyset && \text{because } x_{n+1} \in L_1 \\
&\cap F \neq \emptyset && \text{because } F = F_1 \cup \{q_0\}
\end{aligned}$$

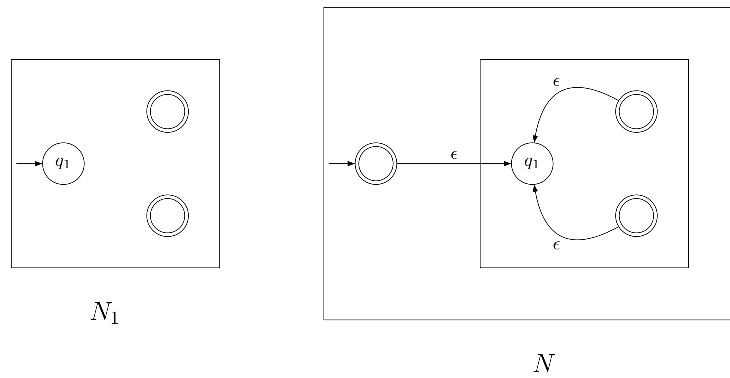

Figure 5.10: ϵ-NFA to accept L^*

Note 5.1. *Given a DFA that recognizes L one might be tempted to construct a DFA that recognizes L^* by linking the final states to the start state by an ϵ-transition and make the start state accepting. In fact this does not always work. Consider the NFA that recongnizes $L = \{a^n b \,|n \geq 0\}$ shown below in Figure 5.11.*

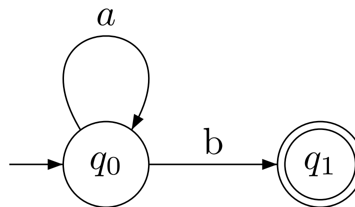

Figure 5.11: ϵ-NFA to accept $a^n b$

If we use the above strategy we get the result shown in Figure 5.12 which is obviously wrong.
The NFA in Figure 5.12 actually accepts $\{a, b\}^*$. The proper construction is shown in Figure 5.13.

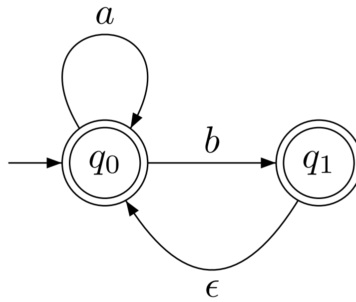

Figure 5.12: $WRONG \epsilon$-NFA to accept $(a^n b)^*$

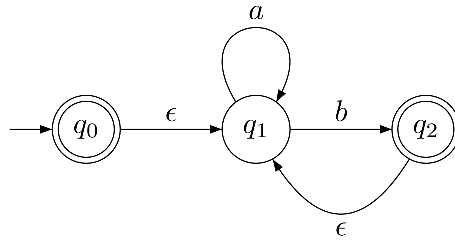

Figure 5.13: $CORRECT \epsilon$-NFA to accept $(a^n b)^*$

Lecture 6

Regular Expressions

6.1 Introduction

Regular expressions are similar to arithmetic expression. In the same way that arithmetic expression are built from numbers and operators, regular expression are built from strings and operators. The result of an arithmetic expression is a number while the result of a regular expression is a language. As an example the regular expression

$$(0+1)0^*$$

produces the language $L = \{w \mid w$ starts with a 0 or 1 followed by any number of 0s$\}$.

6.2 Formal Definition of a Regular Expression

R is said to be a regular expression if one of the following is true

1. $R = a$ for some $a \in \Sigma$. In this case $L(R) = \{a\}$.
2. $R = \epsilon$. $L(R) = \{\epsilon\}$.
3. $R = \phi$. $L(\phi) = \phi$.
4. $R = R_1 \cup R_2$ where R_1 and R_2 are regular expressions.
   $L(R) = L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$.
5. $R = R_1 R_2$ where R_1 and R_2 are regular expressions.
   $L(R) = L(R_1 R_2) = L(R_1)L(R_2)$.
6. $R = R_1^*$ where R_1 is a regular expressions. $L(R_1^*) = (L(R_1))^*$.

Example 6.1. *Write a regular expression for the set of strings that consist of alternating 0s and 1s.*

First we note that any string consisting of alternating 0s and 1s should have as a substring zero or more repetitions of the string 01. There are four possibilities: $(\mathbf{01})^*$, $\mathbf{0}(\mathbf{10})^*$, $\mathbf{1}(\mathbf{01})^*$, $(\mathbf{10})^*$. Therefore the resulting regular expression is

$$(\mathbf{01})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^* + (\mathbf{10})^*$$

Another approach would be to treat the set of alternating 0s and 1s as $(\mathbf{01})^*$ with optional 1 to the left and optional 0 to the right which gives an alternative regular expression for the same language

$$(\mathbf{1} + \epsilon)(\mathbf{01})^*(\mathbf{0} + \epsilon)$$

We can see why the two expressions are the same by expanding the last expression

$$\begin{aligned}(1+\epsilon)(01)^*(0+\epsilon) &= \left(1(01)^* + (01)^*\right)(0+\epsilon)\\ &= 1(01)^*0 + (01)^*0 + 1(01)^* + (01)^*\end{aligned}$$

Now since $1(01)^*0 = (10)^*$ and $(01)^*0 = 0(10)^*$ we get

$$(1+\epsilon)(01)^*(0+\epsilon) = (10)^* + 0(10)^* + 1(01)^* + (01)^*$$

Example 6.2. *The following are examples of regular expressions over the alphabet* $\Sigma = \{0, 1\}$.

- $L = \{w : w \text{ contains exactly one } 1\}$. The strings in L will have the form $10\ldots0$, $0\ldots01$, or $0\ldots010\ldots0$ without forgetting that 1 by itself is also in the language. Thus the regular expression we are looking for is 0^*10^*.
- $\Sigma^*1\Sigma^* = \{w : w \text{ contains at least one } 1\}$.
- $\Sigma^*001\Sigma^* = \{w : w \text{ contains 001 as substring}\}$. In fact we can generalize and say if s is a string and we need a regular expression for all the strings w that have s as a substring, it is $\Sigma^*s\Sigma^*$.
- $L = \{w \in \{a,b\}^* : w \text{ starts and ends with the same symbol }\}$. The string can be of the two forms $a\ldots a$ or $\ldots b$ therefore it is the union of two regular expression $(a\{a,b\}^*a) \cup (b\{a,b\}^*b)$. There are two exceptions: the strings a and b also start and end with the same symbol. Thus the expression is $(a\{a,b\}^*a) \cup (b\{a,b\}^*b) \cup a \cup b$.

6.3 Equivalence of RE and DFA

Even though Finite Automata and Regular Expressions are two different ways of dealing with languages, in fact they are equivalent. In this chapter we show that for any DFA we can construct a Regular Expression that describes the same language, we call this from DFA to Regular Expressions. The reverse, from Regular Expressions to DFA, is left to the next lecture.

6.4 Generalized Non Deterministic Finite Automata

A Generalized Nondeterministic Finite Automaton GNFA is simply an NFA except that the labels of the transitions are regular expressions instead of symbols in an alphabet and ϵ. The GNFA reads a string or a sequence of symbols in one step instead of reading one symbol at a time as in an NFA. Figure 6.1 shows an example GNFA. For example, all strings matched by the regular expression ab^*, will make the GNFA go from state q_0 to state $q1$.
Formally we define a GNFA as a 5-tuple $G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ with

1. Q is a finite set of states.
2. Σ is the alphabet.
3. The transition function $\delta : (Q - q_{accept}) \times (Q - q_{start}) \rightarrow \mathcal{R}$.
4. q_{start} is the start state and q_{accept} is the accept state.

From the above definition we deduce the following properties:

- There is only a single accept state.
- Because the domain of δ is $(Q - q_{accept}) \times (Q - q_{start})$ then
  - There is a transition from the any state (except q_{accept}) to **all** (except q_{start}) other states. This includes a transition from a state to itself.
  - There is **no** transition from any state to the start state.
  - There is **no** transition from the accept state to any other state.

A GNFA accepts a string $w \in \Sigma^*$ if $w = w_1 w_2 \dots w_n$ where $w_i \in \Sigma^*$ and there exists states $q_0, q_1, \dots, q_n$ such that

1. $q_0 = q_{start}$ and $q_n = q_{accept}$.
2. $w_i \in L(\delta(q_{i-1}, q_i))$. In other words, the substring w_i is "matched" by the regular expression labeling the arrow from q_i to q_j.

6.4.1 Converting a DFA to a GNFA

Given an arbitrary DFA $D = (Q_D, \Sigma, \delta_D, q_0, F)$ can be converted to a GNFA $G = (Q_G, \Sigma, \delta_G, q_{start}, \{q_{accept}\})$ as follows

1. Add a new start state q_{start} with the ϵ-transition from q_{start} to q_0. This is to make sure that there is no transition coming into the start state.
2. Add a new accept state q_{accept} with ϵ-transitions from all $q_f \in F$ to q_{accept}. This is to make sure that there is a single accept state and no transition coming out of the accept state.

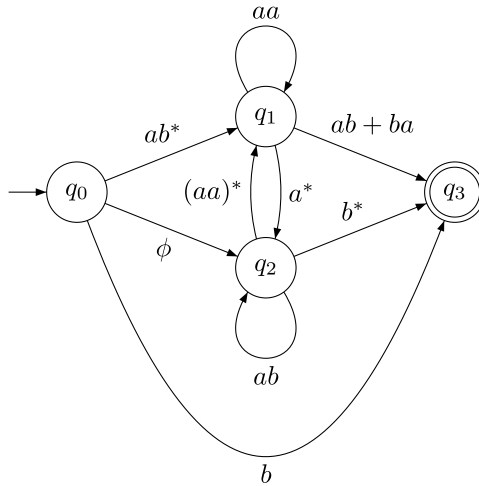

Figure 6.1: Example GNFA

3. Items 1 and 2 imply that $Q_G = Q_D \cup \{q_{start}\} \cup \{q_{accept}\}$.

4. If for any two states $q, p \in Q$ there is no $a \in \Sigma$ such that $\delta_D(q, a) = p$ then $\delta_G(q, p) = \emptyset$. Note that in a GNFA a transition labeled $\emptyset$ cannot be used since $L(\emptyset) = \emptyset$.

5. Multiple transitions from state q to state p are replaced by a single transition whose label is the union of all the previous labels.

Example 6.3. *As an example, we show in Figure 6.2 a conversion from a DFA to an GNFA.*

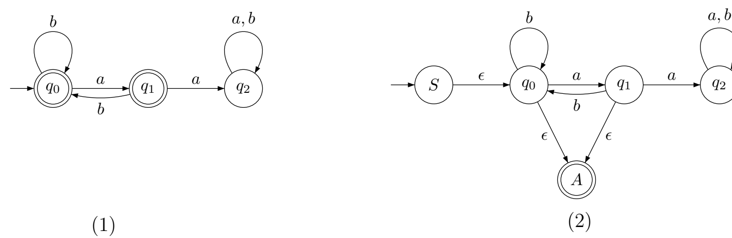

Figure 6.2: From DFA to GNFA

6.4.2 From GNFA to Regular Expression

We build a regular expression from a GNFA recursively by reducing the number of states in the GNFA by one each step until we arrive at a GNFA with only two states:

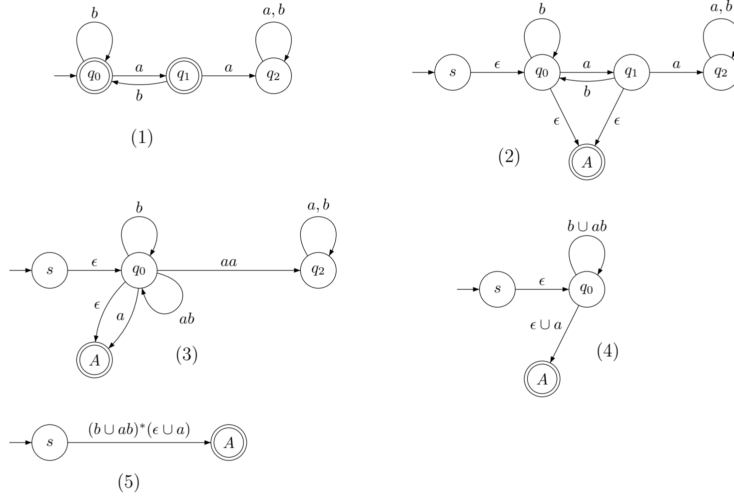

Figure 6.3: From GNFA to RegExp 1

the start and accept states. The label on the transition from the start to the accept state is the equivalent regular expression. Given a GNFA with k states we construct an **equivalent** GNFA with $k-1$ elements by removing an arbitrary state and modifying the transition so that the same language is recognized. Let
$G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ be a GNFA and q_{rip} be the removed state. Construct a new GNFA $G'(Q', \Sigma, \delta', q_{start}, q_{accept})$ having one state less, with $Q' = Q - \{q_{rip}\}$ and for every q_i, q_j, $\delta'(q_i, q_j)$ takes into account the missing state q_{rip} as follows:

$$\delta\prime(q_i, q_j) = \delta(q_i, q_j) \cup \delta(q_i, q_{rip})\delta(q_{rip}, q_{rip})^*\delta(q_{rip}, q_j) \qquad (6.1)$$

Obviously many q_i, q_j are such that $\delta(q_i, q_{rip}) = \emptyset$ or $\delta(q_{rip}, q_j) = \emptyset$, i.e. q_i and q_{rip} are not "connected" or q_{rip} and q_j are not "connected". In all those cases
$\delta\prime(q_i, q_j) = \delta(q_i, q_j)$.
Figure 6.3 shows an example of the construction described above. The initial DFA recognizes all the strings that do not contain aa as a substring. Another example that recognizes all strings containing at least one b is shown in Figure 6.4. It should be noted that in both examples the transitions labeled with the regular expression $\emptyset$ are not shown.

Theorem 6.1. *The GNFAs G and G' as defined above are equivalent.*

Proof. To show that G and G' are equivalent we need to show that any string accepted by one is accepted by the other. Let $w \in \Sigma^*$ be string accepted by G. By the definition of acceptance, we can write $w = w_1 w_2 \ldots w_n$, $w_i \in \Sigma^*$, and there exists $q_0, q_1, \ldots, q_n$ all in Q such that $w_i \in L(\delta(q_{i-1}, q_i)$ for all i. If $q_{rip} \notin \{q_0, q_1, \ldots, q_n\}$ then $q_0, q_1, \ldots, q_n \in Q'$ and $\delta'(q_{i-1}, q_i) = \delta(q_{i-1}, q_i)$, therefore G' accepts w. The other case, suppose that $q_{rip} \in \{q_0, q_1, \ldots, q_n\}$. Let $q_{rip} = q_i$ for some i. We

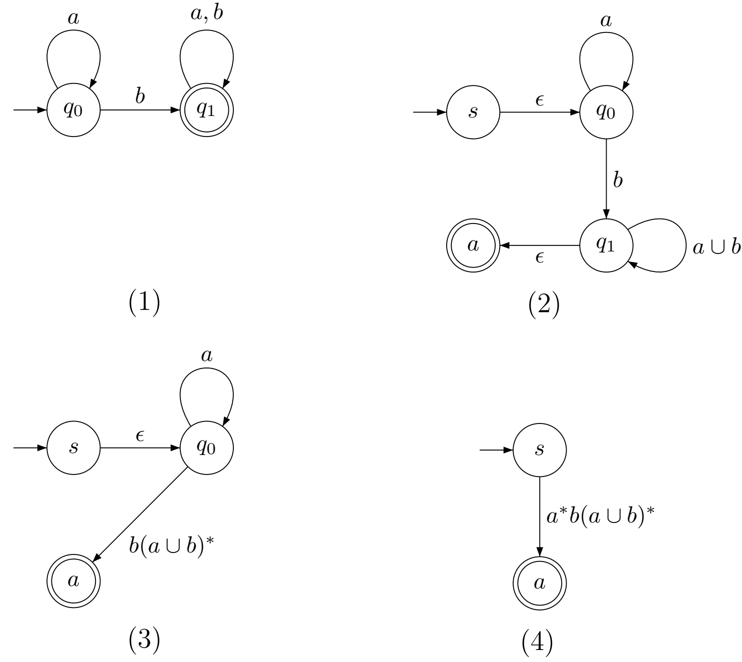

Figure 6.4: From GNFA to RegExp 2

consider the most general case where $w_{i+1} = w_{i+2} = \ldots = w_{i+k}$ and
$q_i = q_{i+1} = \ldots = q_{i+k}$ (i.e. state q_i has a self transition that consumes w_{i+1} k times). Clearly k could be zero.
Figure 6.5 illustrates this case. Since w is accepted by G then the following is true

$$\begin{aligned}
w_j &\in L(\delta(q_{j-1}, q_j)) & 0 \le j < i \\
w_i &\in L(\delta(q_{i-1}, q_i) & \\
w_{i+j} &\in L(\delta(q_i, q_i)) & 0 \le j \le k \\
w_{i+k+1} &\in L(\delta(q_i, q_{i+k+1})) & \\
w_j &\in L(\delta(q_{j-1}, q_j)) & i+k+1 < j \le n
\end{aligned} \tag{6.2}$$

In the construction of G', only the state $q_i = q_{rip}$ is removed and therefore the first and last lines in (6.2) do not change. This means that $\delta'(q_{j-1}, q_j) = \delta(q_{j-1}, q_j)$, and $w_j \in L(\delta'(q_{j-1}, q_j))$for $0 \le j < i$ and $i + k + 1 < j \le n$. The remaining three lines in (6.2) are removed and replaced, according to the transition function defined in (6.1), by:

$$\delta'(q_i, q_{i+k+1}) = \delta(q_i, q_{i+k+1}) \cup \delta(q_i, q_i)\delta(q_i, q_i)^*\delta(q_i, q_{i+k+1})$$

Now we show that w is accepted by G'. From (6.2) we have

$$w_i w_{i+1} w_{i+2} \ldots w_{i+k} w_{i+k+1} \in L(\delta(q_{i-1}, q_i))L(\delta(q_i, q_i))^k L(\delta(q_i, q_{i+k+1})$$

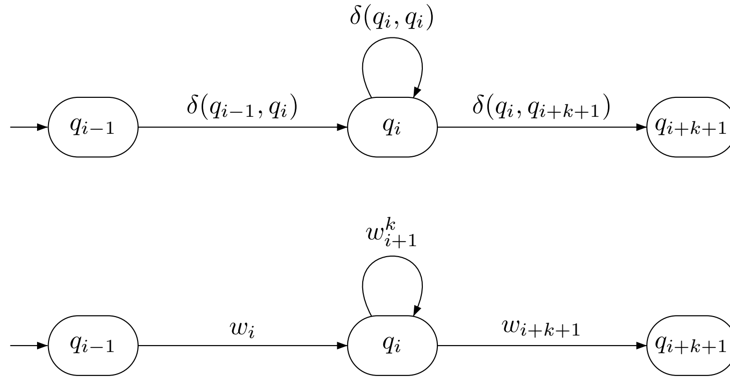

Figure 6.5: Matching the string $w_i w_{i+1}^k w_{i+k+1}$

Using the rules of regular expressions we get

$$\begin{aligned}
w_i w_{i+1} w_{i+2} \ldots w_{i+k} w_{i+k+1} &\in L(\delta(q_{i-1},q_i)\delta(q_i,q_i)^k\delta(q_i,q_{i+k+1})) \\
&\in L(\delta(q_{i-1},q_i)\delta(q_i,q_i)^*\delta(q_i,q_{i+k+1}) \cup \delta(q_{i-1},q_{i+k+1})) \\
&\in L(\delta'(q_{i-1},q_{i+k+1}))
\end{aligned}$$

Combining the above with the fact that $w_j \in L(\delta'(q_{j-1},q_j))$for $0 \leq j < i$ and $i+k+1 < j \leq n$ then for $w_1 w_2 \ldots w_{n-k}$ there exists states $q_0, q_1, \ldots, q_{n-k}$ in G' such that $w_j \in L(\delta'(q_{j-1},q_j))$, $0 \leq j \leq n-k$, therefore G' accepts w.

6.5 From Regular Expression to NFA

We show that given a regular expression we can construct an NFA that accepts the same language. We will prove the equivalence by considering all the cases of a regular expression in the formal definition. Let R be a regular expression.

1. $R = a$ for some $a \in \Sigma$. The NFA $N = (\{q_1,q_2\},\Sigma,\delta,q_1,\{q_2\})$ with $\delta(q_1,a) = \{q_2\}$ and $\delta(q,c) = \emptyset$ for $q \neq q_1$ or $c \neq a$, recognizes $L(R) = \{a\}$.
2. $R = \epsilon$. The NFA $N = (\{q_1\},\Sigma,\delta,q_1,\{q_1\})$ with $\delta(q,c) = \emptyset$ for all q and c, recognizes $L(R) = \{\epsilon\}$.
3. $R = \emptyset$. The NFA $N = (\{q_1\},\Sigma,\delta,q_1,\emptyset)$ with $\delta(q,c) = \emptyset$ for all q and c, recognizes $L(R) = \emptyset$
4. $R = R_1 \cup R_2$ or $R = R_1 R_2$ or $R = R_1^*$. We build an NFA for R_1 and another NFA for R_2 and then combine them by using the construction used in proving that the regular languages are closed under union, concatenation and Kleene star.

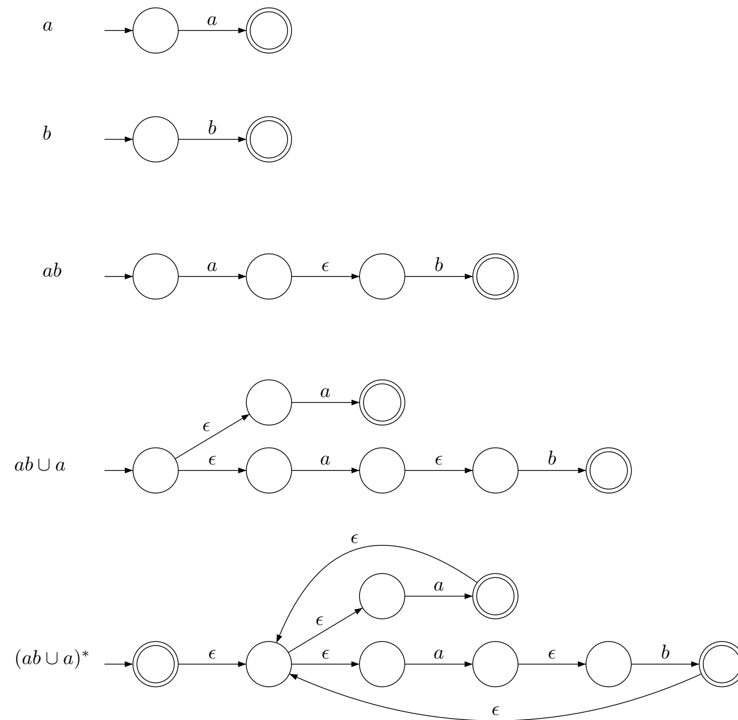

Figure 6.6: NFA equivalent to $(a \cup ab)^*$

Example 6.4. *Convert* $(ab \cup a)^*$ *to an NFA.*

The solution is shown in Figure 6.6.

Example 6.5. *Convert* $(a \cup b)^*ab$ *to an NFA.*

The solution is shown in Figure 6.7.

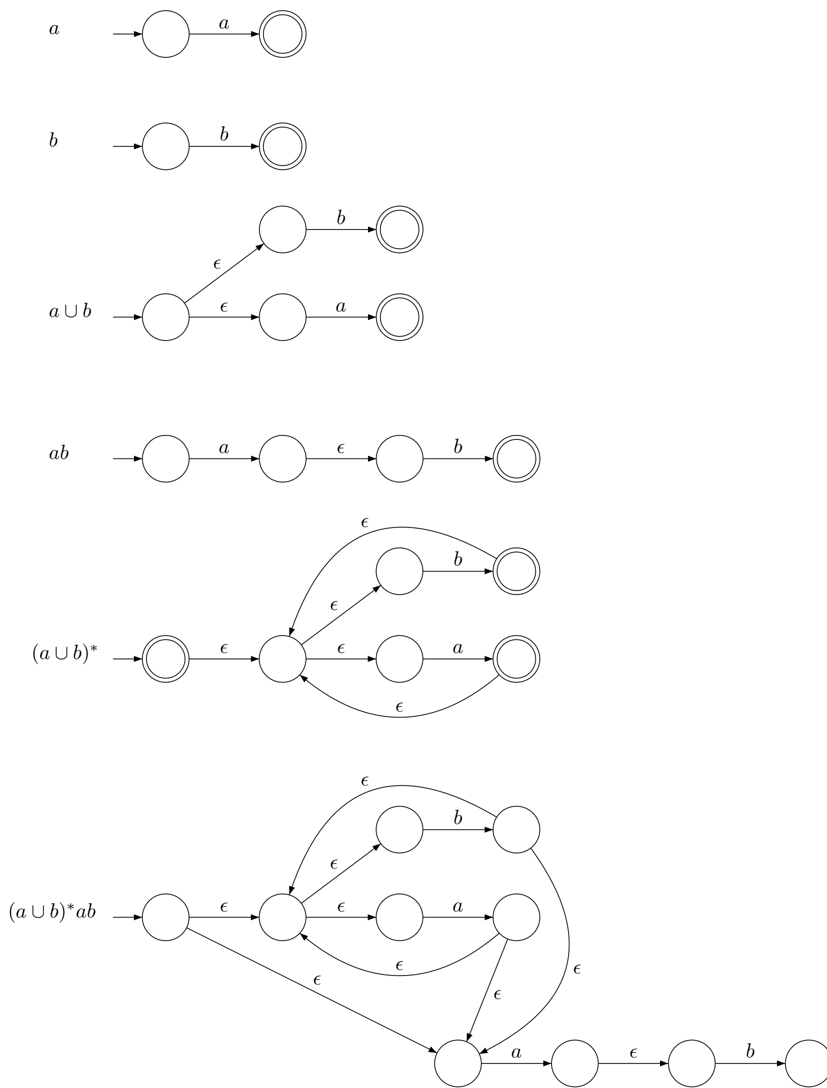

Figure 6.7: NFA equivalent to $(a \cup b)^*ab$

Lecture 7

Non-Regular Languages

So far we have seen that the class of regular languages can be described by DFA's, NFA's, ϵ-NFA's and regular expressions. Not all languages, however, are regular. The following is a powerful result that can be used as a tool to show that certain language are not regular.

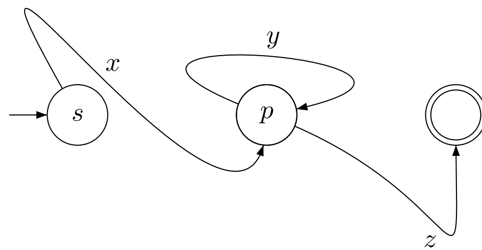

Figure 7.1: If the input is long enough a state is visited more than once, in this case p

7.1 Pumping Lemma for Regular Languages

Theorem 7.1 (Pumping Lemma)**.** *Let L be a regular language. There exists a constant n such that for every $w \in L$ and $|w| \geq n$ we can write $w = xyz$ with the following properties:*

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. $xy^k z \in L$, $\forall k \geq 0$

Proof idea: The basic idea is that a DFA has a finite number of states, n, and when the DFA computes on any string of length bigger than n, it must visit certain states

more than once. The idea is illustrated in Figure 7.1 where the state p can be visited more than once. This means that if xyz is accepted as shown in the figure any string xy^kz will be accepted for any k, including $k = 0$, (i.e. xz is accepted). □

Proof. Because L is regular then there exists a DFA $M = (Q, \Sigma, \delta, s, F)$ that recognizes L, i.e. $L = L(M)$. Let $n = |Q|$ and consider a string $w \in L$ with $|w| = m \geq n$. We can write $w = w_1w_2 \dots w_n$ and define the states $p_i = \hat{\delta}(s, w_1w_2 \dots w_i)$, in particular $p_0 = \hat{\delta}(s, \epsilon) = s$. Since w has $m \geq n$ symbols then it "visits" $m + 1$ states: $s = p_0, p_1, \dots, p_m$. Now M has n states so by the pigeonhole principle there exists $i < j \leq n$ such that $p_i = \hat{\delta}(s, w_1 \dots w_i) = p_j = \hat{\delta}(s, w_1 \dots w_iw_{i+1} \dots w_j)$. This means that state p_i is visited at least twice. Let $x = w_1 \dots w_i$, $y = w_{i+1} \dots w_j$ and $z = w_{j+1} \dots w_m$. First, we observe that since the shortest w has size n i.e. $w = w_1 \dots w_n$ this makes $\hat{\delta}(s, w)$ visit at least $n + 1$ states. In other words $\hat{\delta}(s, w_1 \dots w_{n-1})$ visits all the states and thus i is strictly less than j and subsequently $y \neq \epsilon$. We also have

$$\begin{aligned}
p_j &= \hat{\delta}(s, w_1, \dots, w_i, w_{i+1}, \dots w_j) \\
&= \hat{\delta}(s, xy) \\
&= \hat{\delta}(\hat{\delta}(s, x), y) \\
&= \hat{\delta}(p_i, y) \\
p_j &= \hat{\delta}(s, xy) = \hat{\delta}(s, x) = p_i \qquad (7.1)
\end{aligned}$$

From the discussion above it is clear that $j \leq n$ and therefore $|xy| = |w_1 \dots w_j| \leq n$ For property 3 we use the fact that for any $q \in Q$ and $x, y \in \Sigma^*$, $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ to get

$$\begin{aligned}
\hat{\delta}(s, xy^kz) &= \hat{\delta}(\hat{\delta}(s, xy^k), z) \\
&= \hat{\delta}(\hat{\delta}(\hat{\delta}(s, xy), y^{k-1}), z) \\
&= \hat{\delta}(\hat{\delta}(\hat{\delta}(s, x), y^{k-1}), z) && \text{from eq (7.1)} \\
&= \hat{\delta}(\hat{\delta}(s, xy^{k-1}), z) \\
&= \hat{\delta}(s, xy^{k-1}z)
\end{aligned}$$

Therefore for any $k \geq 0$ we can iterate the above equality to get $\hat{\delta}(s, xy^kz) = \hat{\delta}(s, xyz)$. Therefore $xy^kz \in L$ for any $k \geq 0$. ■

Example 7.1. *The language $L = \{w \in \{0, 1\}^* : w = 0^p1^p\}$ is not regular.*

Proof. Let n be the pumping length of L and $s = 0^n1^n$ be a string in L. Since $|s| \geq n$ then by the pumping lemma we can write $s = xyz$ with $|xy| \leq n$ which in this case means that x and y are all 0's. We can write $x = 0^i$, $y = 0^j$ and $z = 0^{n-i-j}1^n$. Again by the pumping lemma $xy^kz \in L$ and in particular $xyyz \in L$ and therefore $0^{n+j}1^n \in L$ which leads to a contradiction.

■

Example 7.2. *The language*
$L = \{w \in \{0,1\}^* : w$ *has an equal number of 0's and 1's*$\}$ *is not regular.*

Proof. The proof is the same as example 7.1. Let n be the pumping length of L and consider the string $s = 0^n 1^n$. We can write $s = xyz$ where by the pumping lemma x and y are all 0's and we can pump y therefore $xyyz \in L$ but $xyyz$ has more 0's than 1's which is a contradiction.

∎

Example 7.3. *The language* $L = \{w \in \{0,1\}^* : |w|$ *is prime*$\}$ *is not regular.*

Proof. Assume that L is regular and let p be the pumping length. Let $w \in L$ with $|w| = n \geq p$. Since $|w| = n \geq p$ then by the pumping lemma we can write $w = xyz$ with $|xyz| = n$ and $xy^k z \in L$ for any k. In particular choose $k = n + 1$ then $xy^{n+1}z \in L$ and $|xy^{n+1}z| = |xyz| + n|y| = n + n|y| = n(1 + |y|)$ but since $|y| > 1$ and $n > 1$ this is a contradiction because $xy^{n+1}z \in L$ and its length is not prime.

∎

Example 7.4. *The language* $L = \{w \in \{0,1\}^* : |w| = n^2\}$ *is not regular.*

Proof. Assume that L is regular and let p be the pumping length of L and choose $w \in L$ with $|w| = p^2$. By the pumping lemma we can write $w = xyz$ with $|xy| \leq p$. Consider the string $xyyz \in L$ by the pumping lemma. Now since $y \neq \epsilon$ and $|xy| \leq p$ then $p^2 < |xyyz| \leq p^2 + p < (p+1)^2$. Therefore $|xyyz|$ is not a square and $xyyz \notin L$ which is a contradiction.

∎

Example 7.5. *The language* $L = \{w \in \{0,1\}^* : w = w^R\}$ *is not regular.*

Proof. Assume that L is regular and let p be the pumping length of L. Let $w = 0^p 1 0^p$ which clearly is in L. By the pumping lemma we can write $w = xyz$ and $|xy| \leq p$ therefore x and y are all 0's. On the other hand $xyyz$ should be in L by the pumping lemma but since y is all 0's then $xyyz$ contains more 0's to the "left" of 1 than to the right which is a contradiction.

∎

Example 7.6. *The language* $L = \{w \in \{a,b\}^* : w = a^i b^j \text{ , } i < j\}$ *is not regular.*

Proof. Assume that L is regular and let p be the pumping length of L. Consider $w = a^p b^{p+1}$ which is clearly in L. By the pumping lemma we can write $w = xyz$ with $|xy| \leq p$ therefore x and y are all $a's$. On the other hand $xyyz$ should be in L, and $y \neq \epsilon$. Since y is all $a's$ then $xyyz$ contains more than p $a's$, i.e. greater or equal number of $b's$ which is a contradiction.

∎

Lecture 8

DFA Minimization

8.1 Introduction

We had enough experience with DFAs to realize that on many occasions different DFA, usually having a different number of states, recognize the same language. Alternatively, sometimes we can simplify a DFA by removing from it ”unnecessary” states. We already encountered one kind of simplification when we removed ”inaccessible” states. In addition to the removal of ”unnecessary” states, the question is : given a language L and a DFA M such that $L = L(M)$, is there a minimum number of states that M should have? In this lecture we will answer the question by showing how to minimize the number of states of a given DFA. To do so we first introduce the concept of equivalent states then use this equivalence to construct a ”quotient” DFA for the original DFA.

8.2 Equivalent States

Definition 8.1. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Two states p and q are said to be equivalent, denoted $p \sim q$, if for every $x \in \Sigma^*$, $\hat{\delta}(p, x) \in F \Leftrightarrow \hat{\delta}(q, x) \in F$.*

If two states (p, q) are not equivalent we say they are distinguishable and we write $p \nsim q$. If $p \nsim q$ then $\exists\, x$ such that $\hat{\delta}(p, x) \in F$ and $\hat{\delta}(q, x) \notin F$ **or** $\hat{\delta}(p, x) \notin F$ and $\hat{\delta}(q, x) \in F$. For the remainder of this lecture when we say that (p, q) are distinguishable we assume, without loss of generality, that $\exists\, x$ such that $\hat{\delta}(p, x) \in F$ and $\hat{\delta}(q, x) \notin F$.

Lemma 8.1. *If $p \sim q$ then $\delta(p, a) \sim \delta(q, a)$ for all $a \in \Sigma$. Equivalently, if for some $a \in \Sigma$ we have $\delta(p, a) \nsim \delta(q, a)$ then $p \nsim q$.*

Proof. Assume that $p \sim q$ and let y be an arbitrary string, then

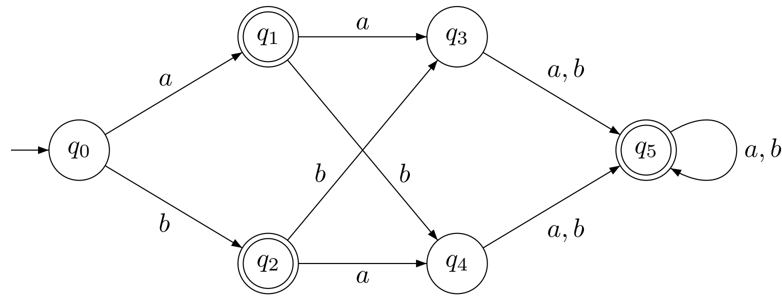

Figure 8.1: DFA containing equivalent states used in Example 8.1

$$
\begin{aligned}
\hat{\delta}(\delta(p,a),y) \in F &\Leftrightarrow \hat{\delta}(p,ay) \in F \\
&\Leftrightarrow \hat{\delta}(q,ay) \in F && \text{because } p \sim q \\
&\Leftrightarrow \hat{\delta}(\delta(q,a),y) \in F && \text{hw 2}
\end{aligned}
$$

Thus $\delta(p,q) \sim \delta(q,a)$.
The following algorithm discovers all distinguishable states recursively.

Marking Algorithm

Basis (step 1). If $p \in F$ and $q \notin F$ then (p,q) are marked as distinguishable.
Induction step (step 2). If the pair (r,s) are marked as distinguishable and $\delta(p,a) = r$ and $\delta(q,a) = s$ for some $a \in \Sigma$ then the pair (p,q) are marked as distinguishable. This is true because if (r,s) are distinguishable then $\exists\ x$ such that $\hat{\delta}(r,x) \in F$ and $\hat{\delta}(s,x) \notin F$ and therefore the string ax distinguishes (p,q):
$\hat{\delta}(p,ax) = \hat{\delta}(\delta(p,a),x) = \hat{\delta}(r,x) \in F$ and $\hat{\delta}(q,ax) = \hat{\delta}(\delta(q,a),x) = \hat{\delta}(s,x) \notin F$.

Example 8.1. *Use the marking algorithm to find all equivalent states for the DFA in Figure 8.1.*

Initially the table is empty.

q_0					
	q_1				
		q_2			
			q_3		
				q_4	
					q_5

Table 8.1: Initial marking for DFA in Figure 8.1

Now we fill all the pairs in which one of them is accepting
The first pass of the algorithm is as follows

q_0					
x	q_1				
x		q_2			
	x	x	q_3		
	x	x		q_4	
x			x	x	q_5

Table 8.2: Pass one of marking algorithm for Figure 8.1

- Pair $\{q_1, q_2\}$. We have
  - $\{q_1, q_2\} \xrightarrow{a} \{q_3, q_4\}$ and pair $\{q_3, q_4\}$ is not marked so do not mark $\{q_1, q_2\}$.
  - $\{q_1, q_2\} \xrightarrow{b} \{q_3, q_4\}$ and pair $\{q_3, q_4\}$ is not marked so do not mark $\{q_1, q_2\}$.
- Pair $\{q_0, q_3\}$. We have
  - $\{q_0, q_3\} \xrightarrow{a} \{q_1, q_5\}$ and pair $\{q_1, q_5\}$ is not marked so do not mark $\{q_0, q_3\}$.
  - $\{q_0, q_3\} \xrightarrow{b} \{q_2, q_5\}$ and pair $\{q_2, q_5\}$ is not marked so do not mark $\{q_0, q_3\}$.
- Pair $\{q_0, q_4\}$. We have
  - $\{q_0, q_4\} \xrightarrow{a} \{q_1, q_5\}$ and pair $\{q_1, q_5\}$ is not marked so do not mark $\{q_0, q_4\}$.
  - $\{q_0, q_4\} \xrightarrow{b} \{q_2, q_5\}$ and pair $\{q_2, q_5\}$ is not marked so do not mark $\{q_0, q_4\}$.
- Pair $\{q_3, q_4\}$. We have
  - $\{q_3, q_4\} \xrightarrow{a} \{q_5, q_5\}$ and pair $\{q_5, q_5\}$ is not marked so do not mark $\{q_0, q_4\}$.
  - $\{q_3, q_4\} \xrightarrow{b} \{q_5, q_5\}$ and pair $\{q_5, q_5\}$ is not marked so do not mark $\{q_0, q_4\}$.
- Pair $\{q_1, q_5\}$. We have
  - $\{q_1, q_5\} \xrightarrow{a} \{q_3, q_5\}$ and pair $\{q_3, q_5\}$ is marked so we mark $\{q_1, q_5\}$.
- Pair $\{q_2, q_5\}$. We have
  - $\{q_2, q_5\} \xrightarrow{a} \{q_4, q_5\}$ and pair $\{q_4, q_5\}$ is marked so we mark $\{q_2, q_5\}$.

After this pass the table looks like in
Another pass

q_0					
x	q_1				
x		q_2			
	x	x	q_3		
	x	x		q_4	
x	x	x	x	x	q_5

Table 8.3: Pass two of the marking algorithm for Figure 8.1

- Pair $\{q_1, q_2\}$. We have
  - $\{q_1, q_2\} \xrightarrow{a} \{q_3, q_4\}$ and pair $\{q_3, q_4\}$ is not marked so do not mark $\{q_1, q_2\}$.
  - $\{q_1, q_2\} \xrightarrow{b} \{q_3, q_4\}$ and pair $\{q_3, q_4\}$ is not marked so do not mark $\{q_1, q_2\}$.
- Pair $\{q_0, q_3\}$ we have
  - $\{q_0, q_3\} \xrightarrow{a} \{q_1, q_5\}$ and pair $\{q_1, q_5\}$ is marked so we mark $\{q_0, q_3\}$.
- Pair $\{q_0, q_4\}$ we have
  - $\{q_0, q_4\} \xrightarrow{a} \{q_1, q_5\}$ and pair $\{q_1, q_5\}$ is marked so we mark $\{q_0, q_4\}$.

After that no other state is marked and the final table is shown below. This means that $q_1 \sim q_2$ and $q_3 \sim q_4$.

q_0					
x	q_1				
x		q_2			
x	x	x	q_3		
x	x	x		q_4	
x	x	x	x	x	q_5

Table 8.4: Final result for DFA shown in Figure 8.1

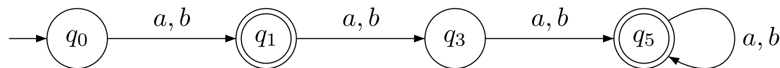

Figure 8.2: Resulting DFA equivalent to DFA in 8.1

Example 8.2. *Use the marking algorithm to find all equivalent states for the DFA in Figure 8.3.*

It should be noted that $\sim$ is reflexive and symmetric therefore the table will be symmetric. Since q_2 is the only accepting state then q_2 is not equivalent (is distinguishable) to any other state. Therefore, all pairs containing q_2 are marked. Now starting from the fact that (q_2, q_7) are distinguishable and $\delta(q_4, 0) = q_7$ and $\delta(q_5, 0) = q_2$ then (q_4, q_5) are distinguishable. Continuing with our marking algorithm we obtain the result shown in Table 14.1.

q_0							
x	q_1						
x	x	q_2					
x	x	x	q_3				
	x	x	x	q_4			
x	x	x		x	q_5		
x	x	x	x	x	x	q_6	
x		x	x	x	x	x	q_7

Table 8.5: Marking of distinguishable states for DFA in Figure 8.4

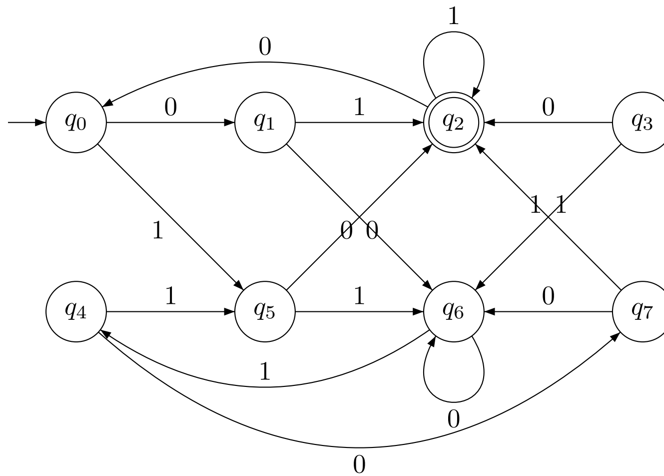

Figure 8.3: DFA containing equivalent states for example 8.2

Example 8.3. *Use the marking algorithm to find all equivalent states for the DFA in Figure 8.4.*

First, $q_0 \nsim q_4$, because q_0 is accepting and q_4 is not, with $\delta(q_3, a) = q_0$ and $\delta(q_4, a) = q_4$ leads to $q_3 \nsim q_4$. The same reasoning leads to $q_1 \nsim q_3$. Also $q_2 \nsim q_4$ because q_2 is accepting and q_4 is not, together with $\delta(q_1, b) = q_2$ and $\delta(q_5, b) = q_4$ leads to $q_1 \nsim q_5$. The same reasoning leads also to $q_1 \nsim q_4$. The final result, shown in Table 14.2, is that $q_0 \sim q_2$ and $q_3 \sim q_5$. This allows us to simplify the DFA in Figure 8.5 to the one shown in Figure

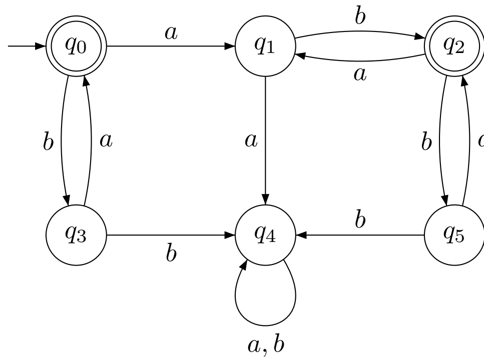

Figure 8.4: DFA containing equivalent states for example 8.3

q_0					
x	q_1				
	x	q_2			
x	x	x	q_3		
x	x	x	x	q_4	
x	x	x		x	q_5

Table 8.6: Marking of distinguishable states for DFA in Figure 8.4

Theorem 8.1. *A pair (p, q) is distinguishable if and only if it is discovered by the marking algorithm.*

Proof. First we show that if $p \nsim q$ then the pair (p, q) is marked by the marking algorithm. If $p \nsim q$ then $\exists\, x$ such that $\hat{\delta}(p, x) \in F$ and $\hat{\delta}(q, x) \notin F$. We prove by induction on $|x|$ that (p, q) is marked .
Basis. $|x| = 0$ then $\hat{\delta}(p, \epsilon) = p \in F$ and $\hat{\delta}(q, \epsilon) = q \notin F$ therefore (p, q) is marked by step 1 of the marking algorithm.
Induction hypothesis. Assume that all pairs distinguishable by strings x of length $|x|$ are marked by the algorithm.
Induction step. Suppose that the string ax, $a \in \Sigma$, distinguishes (p, q). Then by definition $\hat{\delta}(p, ax) \in F$ and $\hat{\delta}(q, ax) \notin F$. This implies that $\hat{\delta}(\delta(p, a), x) \in F$ and $\hat{\delta}(\delta(q, a), x) \notin F$ thus $\delta(p, a)$ and $\delta(q, a)$ are marked by the induction hypothesis. Finally, step 2 of the algorithm marks (p, q) because $\delta(p, a)$ and $\delta(q, a)$ are marked.
Now we show that if (p, q) are marked by the algorithm then $p \nsim q$. The proof is by induction on the number of iterations of the marking algorithm.
Basis. (p, q) are marked by the first iteration implies that $p \in F$ and $q \notin F$, thus $p \nsim q$ because $\delta(p, \epsilon) \in F$ and $\delta(q, \epsilon) \notin F$.
Induction hypothesis Assume that all pairs (p, q) marked by iteration n are distinguishable.
Induction step. If (p, q) is marked by the algorithm on iteration $n + 1$ then $\exists a \in \Sigma$ such that $(\delta(p, a), \delta(q, a))$ was marked on step n. By the induction hypothesis

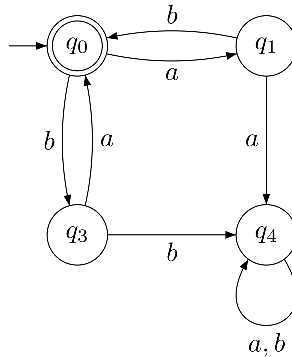

Figure 8.5: DFA equivalent to the DFA in Figure 8.4

$\delta(p,a) \nsim \delta(q,a)$ thus $p \nsim q$ by lemma 8.1.

8.3 Quotient Construction

The relation $\sim$ is an equivalence relation because it is reflexive, symmetric and transitive. Therefore $\sim$ partitions Q into disjoint equivalence classes. We use the following notation for the equivalence classes: $[p] = \{q : q \sim p\}$. We take advantage of equivalent classes and their properties to define a new kind of automaton: the **quotient** automaton. As we will see later the quotient DFA will be the minimal DFA we are seeking. Given a DFA $M = (Q, \Sigma, \delta, s, F)$ we construct the **quotient** DFA, sometimes denoted by $M/\sim$, $M' = (Q', \Sigma, \delta', s', F')$ as follows:

1. $Q' = \{[q] : q \in Q\}$.
2. $\delta'([q], a) = [\delta(q, a)]$.
3. $s' = [s]$.
4. $F' = \{[q] : q \in F\}$.

The first thing we have to do is to show that δ' given in 2 is well defined. In other words, suppose that $q \in [p]$ is $\delta'([p], a) = \delta'([q], a)$ as it should? The answer is yes and the proof is in lemma 8.1. $q \in [p]$ implies that $q \sim p$ and by lemma 8.1 $\delta(p,a) \sim \delta(q,a)$ therefore $[\delta(p,a)] = \delta'([p], a) = \delta'([q], a) = [\delta(q,a)]$.

Lemma 8.2. *For all* $x \in \Sigma^*$, $\hat{\delta}'([p], x) = [\hat{\delta}(p, x)]$.

Proof. By induction on $|x|$.
Basis. $x = 0$.

$$\begin{aligned} \hat{\delta}'([p], \epsilon) &= [p] && \text{def. } \hat{\delta}' \\ &= [\hat{\delta}(p, \epsilon)] && \text{def. } \hat{\delta} \end{aligned}$$

Induction step. Assume $\hat{\delta}'([p], x) = [\hat{\delta}(p, x)]$ then

$$\begin{aligned}
\hat{\delta}'([p], xa) &= \delta'(\hat{\delta}'([p], x), a) && \text{def. } \hat{\delta}' \\
&= \delta'([\hat{\delta}(p, x], a) && \text{ind. hyp.} \\
&= [\delta(\hat{\delta}(p, x), a)] && \text{def. } \delta' \\
&= [\hat{\delta}(p, xa)] && \text{def. } \hat{\delta}
\end{aligned}$$

Lemma 8.3. *If* $q \in F \Leftrightarrow [q] \in F'$.

Proof. The "if" part follows directly from the definition of F'. The "only if": assume that $[q] \in F'$ then by the definition of F', $\exists p$, $p \sim q$ and $p \in F$. But, $p \sim q$ implies that $\hat{\delta}(q, x) \in F \Leftrightarrow \hat{\delta}(p, x) \in F$. Replacing x by ϵ in the last relation leads to $\hat{\delta}(p, \epsilon) = p \in F \Leftrightarrow q = \hat{\delta}(q, \epsilon) \in F$ therefore $q \in F$.

Theorem 8.2. $L(M) = L(M/\sim)$.

Proof.

$$\begin{aligned}
x \in L(M/\sim) &\Leftrightarrow \hat{\delta}'(s', x) \in F' && \text{def. of acceptance} \\
&\Leftrightarrow \hat{\delta}'([s], x) \in F' && \text{def. of } s' \\
&\Leftrightarrow [\hat{\delta}(s, x)] \in F' && \text{lemma 8.2} \\
&\Leftrightarrow \hat{\delta}(s, x) \in F && \text{lemma 8.3} \\
&\Leftrightarrow x \in L(M) && \text{def. of acceptance}
\end{aligned}$$

8.4 Nondeterminisitic Automata

Can we minimize nondeterministic automata? The answer is yes but the minimal automaton is not necessarily unique as the figure below shows.
In this chapter we show the correspondence between a certain type of equivalence relations, called Myhill-Nerode relations and DFAs. Namely, given a DFA we can construct a Myhill-Nerode relation and vice versa, given a Myhill-Nerode relation we can construct a corresponding DFA.
Recall that a given language L (regular or not) defines an equivalence relation $x \sim y \Rightarrow (x \in L \Leftrightarrow y \in L)$. This relation partitions Σ^* into two partitions $x \in L$ and $x \notin L$.

8.5 Myhill-Nerode Relations

Definition 8.2. *A relation R_1 is said to refine another relation R_2 if $R_1 \subseteq R_2$ when R_1 and R_2 are regarded as sets of pairs. We also say that R_1 is finer than R_2 and R_2 is coarser than R_1.*

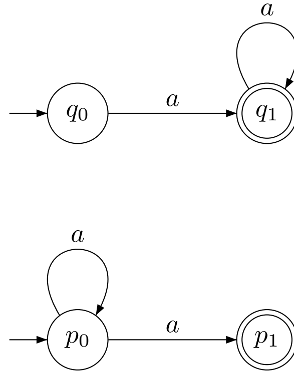

Figure 8.6: Two minimal NFAs for a^+ that are not isomorphic

Definition 8.3. *Given $L \subseteq \Sigma^*$ a regular language a Myhill-Nerode relation is an equivalence relation $\equiv$ over Σ^* with the following properties:*

1. *It is right congruent: $x \equiv y \Rightarrow xa \equiv ya$, $\forall a \in \Sigma$.*

2. *$\equiv$ refines L. This means that if $x \equiv y$ then either both are in L or none of them is in L. $\forall\ x, y \in \Sigma^*$*
$$x \equiv y \Rightarrow (x \in L \Leftrightarrow y \in L)$$

3. *$\equiv$ has a finite number of equivalence classes (finite index).*

The equivalence class of $x \in \Sigma^*$ is denoted b $[x]_\equiv = \{y \in \Sigma^* \mid x \equiv y\}$.

8.6 From Myhill-Nerode to DFA

Given a Myhill-Nerode relation $\equiv$ we can construct the automaton
$M_\equiv = (Q, \Sigma, \delta, s, F)$ with

- $Q = \{[x] \mid x \in \Sigma^*\}$
- $s = [\epsilon]$
- $\delta([x], a) = [xa]$
- $F = \{[x] \mid x \in L\}$

Note that $\delta([x], a)$ is well defined because if $x \in [x]$ and $y \in [x]$ then from the congruence defined in property 1 we have $\delta([x], x) = [xa] = [ya] = \delta([y], a)$.

Lemma 8.4. $\hat{\delta}([x], y) = [xy]$.

Proof. By induction on $|y|$.
basis:

$$\begin{aligned} \hat{\delta}([x],\epsilon) =& [x] && \text{def. of } \hat{\delta} \\ =& [x\epsilon] \end{aligned}$$

induction step:

$$\begin{aligned} \hat{\delta}([x],ya) =& \delta(\hat{\delta}([x],y),a) && \text{def. of } \hat{\delta} \\ =& \delta([xy],a) && \text{hyp.} \\ =& [xya] && \text{def. of } \delta \end{aligned}$$

Theorem 8.3. $L(M_{\equiv}) = L$

proof.

$$\begin{aligned} x \in L(M) \Leftrightarrow& \hat{\delta}(s,x) \in F && \text{def. of acceptance} \\ \Leftrightarrow& \hat{\delta}([\epsilon],x) \in F && \text{def. of } s \\ \Leftrightarrow& [\epsilon x] \in F && \text{lemma 8.4} \\ \Leftrightarrow& [x] \in F \\ \Leftrightarrow& x \in L && \text{def. of } F \end{aligned}$$

8.7 From DFA to Myhill-Nerode

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA such that $L(M) = L$ and define the equivalence relation $\equiv_M$ on Σ^* with

$$x \equiv_M y \Leftrightarrow \hat{\delta}(s,x) = \hat{\delta}(s,y)$$

Theorem 8.4. *$\equiv_M$ is a Myhill-Nerode relation.*

First it is easily shown that $\equiv_M$ is an equivalence relation (reflexive,symmetric and transitive). The remaining properties

1. $\equiv_M$ is a right congruence: for any $x, y \in \Sigma^*$ and $a \in \Sigma$ we have:

$$\begin{aligned} x \equiv_M y \Leftrightarrow& \hat{\delta}(s,x) = \hat{\delta}(s,y) = p \in Q \\ \Leftrightarrow& \delta(\hat{\delta}(s,x),a) = \delta(\hat{\delta}(s,y),a) \\ \Leftrightarrow& \hat{\delta}(s,xa) = \hat{\delta}(s,ya) \\ \Leftrightarrow& xa \equiv_M ya \end{aligned}$$

2. $\equiv_M$ refines L. Assume that $x \equiv_M y$ then

$$\begin{aligned} x \in L \Leftrightarrow& \hat{\delta}(s,x) \in F && \text{def. of acceptance} \\ \Leftrightarrow& \hat{\delta}(s,y) \in F && x \equiv_M y \\ \Leftrightarrow& y \in L \end{aligned}$$

3. $\equiv_M$ has a finite number of equivalence classes. This is because for any $x \in \Sigma^*$, if $\hat{\delta}(s,x) = p$ then $[x] = \{y \in \Sigma^* \mid \hat{\delta}(s,y) = p\}$ this means that the number of distinct classes is equal to the number of states which is finite.

8.8 Myhill-Nerode Theorem

First we define the relation $\equiv_L$ for a given language $L \subseteq \Sigma^*$

Definition 8.4. *Given a language $L \subseteq \Sigma^*$ define $\equiv_L$ on Σ^* as: $x \equiv_L y$ iff:*

$$\forall z(xz \in L \Leftrightarrow yz \in L)$$

First we need the following lemma

Lemma 8.5. *The relation $\equiv_L$ is a right congruence that refines L and is the coarsest such relation.*

Proof. Suppose that $x \equiv_L y$ then from the definition $\forall z, xz \in L \Leftrightarrow yz \in L$. Choose $z = aw$ where $a \in \Sigma$ and $w \in \Sigma^*$ then

$$\begin{aligned} x \equiv_L y &\Rightarrow \forall a \in \Sigma\ \forall w \in \Sigma^*\ (xaw \in L \Leftrightarrow yaw \in L) \\ &\Rightarrow \forall a \in \Sigma(\forall w \in \Sigma^*(xaw \in L \Leftrightarrow yaw \in L)) \\ &\Rightarrow \forall a \in \Sigma(xa \equiv_L ya) \end{aligned}$$

Next we prove that $\equiv_L$ refines L. Just take $z = \epsilon$ in the definition of $\equiv_L$ to get

$$x \equiv_L y \Rightarrow (x \in L \Leftrightarrow y \in L)$$

Finally we prove that any relation $\equiv$ that is a right congruence and refines L it refines $\equiv_L$. Let $x \equiv y$. Since it is a right congruence then
$x \equiv y \Rightarrow \forall z(xz \equiv yz)$ which can be shown by induction. It follows that
$\forall z(xz \in L \Leftrightarrow yz \in L)$ because $\equiv$ refines L. therefore $x \equiv_L y$. ■

Theorem 8.5 (Myhill-Nerode)**.** *For a language $L \subseteq \Sigma^*$ the following statements are equivalent*

(a) L is regular.

(b) There exists a Myhill-Nerode relation for L.

(c) $\equiv_L$ has finite index.

Proof.
$(a) \Rightarrow (b)$ this follows from Theorem 8.4.
$(b) \Rightarrow (a)$ this follows from Theorem 8.3.
$(b) \Rightarrow (c)$ because any Myhill-Nerode relation $\sim$ refines $\equiv_L$, i.e. $\sim \subseteq \equiv_L$ and since $\sim$ has a finite index then $\equiv_L$ has finite index.

$(c) \Rightarrow (a)$ because if $\equiv_L$ has finite index then it is a Myhill-Nerode relation and therefore L is regular. Note that the constructed DFA from $\equiv_L$ is the minimal DFA because $\equiv_L$ is the coarsest Myhill-Nerode relation.

An alternate statement
Let $S = \{[x]_{\equiv_L} \mid x \in \Sigma^*\}$. The Myhill-Nerode theorem states that L is regular iff S is finite. This implies that we can show that a language L is nonregular if we can find an infinite subset $E \subseteq \Sigma^*$ such that $\forall x, y \in E$ $x \not\equiv_L y$. This is true because it is obvious that $E \subseteq S$ and since E is infinite then S is infinite.

8.9 Examples

We will use the alternate statement of Myhill-Nerode theorem to prove some languages are not regular.

Example 8.4. *Let $L = \{a^n b^n\}$ and consider the infinite set $E = \{a^k\}$.*

For any $x = a^k$ and $y = a^m$ with $k \neq m$ then $x \not\equiv_L y$ because $a^k a^k \in L$ but $a^m a^k \notin L$ therefore L is not regular since $\equiv_L$ has an infinite number of equivalence classes.

Example 8.5. *Consider the language $L = \{x1y||x| = |y|\}$ and the infinite set $E = \{0^k\}$.*

For any $x = 0^k$ and $y = 0^m$ with $k \neq m$ then $x \not\equiv y$ because $x1x \in L$ and $y1x \notin L$ and therefore $\equiv_L$ has an infinite number of equivalence classes and thus L is not regular.

Example 8.6. *Consider the language $L = \{ww\}$ and let set $E = \{a^n\}$.*

Select two elements from E, say $u = a^k$ and $v = a^l$ with $k \neq l$. u and v are distinguishable by the string ba^kb because $uba^kb = a^kba^kb \in L$ and $vba^kb = a^lba^kb \notin L$.

Lecture 9

Context-free Grammars

9.1 Introduction

We introduce context-free grammars informally by an example. Let p_{pal} the language of palindromes over $\{0, 1\}$, i.e. $L_{pal} = \{w \in \{0, 1\}^* : w = w^R\}$. We have shown previously that L_{pal} is not regular but we can define it recursively as follows:
Basis: $\epsilon, 0, 1 \in L_{pal}$
Induction: If $w \in L_{pal}$ then $0w0 \in L_{pal}$ and $1w1 \in L_{pal}$.

The above rules can be write as

$$
\begin{aligned}
A &\to \epsilon \\
A &\to 0 \\
A &\to 1 \\
A &\to 0A0 \\
A &\to 1A1
\end{aligned}
$$

Figure 9.1:

9.2 Formal Definition

Definition 9.1. *A Context-Free grammar (CFG) is a quadruple $G = (V, T, P, S)$ where*

1. *V is a finite set of variables called nonterminals.*
2. *T is a finite set of symbols called terminals.*

3. *$S \in V$ represent the language being defined and called the start symbol.*
4. *P is a set of rules or productions that represent the recursive definition of the language. Each production consists of*
   (a) *A variable being defined by the production and called the head of the production.*
   (b) *The head is followed by the symbol $\rightarrow$.*
   (c) *The body of the production is a string of variables and terminals.*

The CGF for L_{pal} introduced in the previous section can be specified as $G = (\{P\}, \{0, 1\}, R, P)$ where R is the set of rules shown in Figure 9.1. We can simplify the notation of productions by combining them. For example the productions shown in Figure 9.1 can be written on a single line as $P \rightarrow \epsilon|0|1|0P0|1P1$.

Example 9.1. *Use CFG to define a simplified version of expressions in programming languages.*

Typically an expression contains identifiers and operators. In this simplified version we restrict the identifiers to begin with either a or b and could be followed with zero or more symbols from the set $\{a, b, 0, 1\}$. Also we restrict the operators to addition $+$ and multiplication $*$ only. This language *is* regular and represented by $\{a \cup b\}\{a \cup b \cup 0 \cup 1\}^*$. The CFG that describes the same language can be defined as $G = (\{E, I\}, \{+, *, a, (,), b, 0, 1\}, P, E)$ and P is the set of productions shown in Figure 9.2.

$$
\begin{aligned}
E &\rightarrow I \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow (E) \\
I &\rightarrow a|b|Ia|Ib|I0|I1
\end{aligned}
$$

Figure 9.2:

9.3 Derivations

Let $G = (V, T, P, S)$ be a CFG and $\alpha A \beta$ be a string such that $\alpha, \beta \in (V \cup T)^*$ and $A \in V$. Let $A \rightarrow \gamma$ be a production of G. We say $\alpha A \beta \stackrel{1}{\Rightarrow} \alpha \gamma \beta$, or $\alpha A \beta \Rightarrow \alpha \gamma \beta$ is a one step derivation which replaces A by the body of one of its productions. We can extend the above to multistep derivations, denoted by $\stackrel{*}{\Rightarrow}$, in a recursive manner.

- $\alpha \stackrel{0}{\Rightarrow} \alpha$ for any $\alpha \in (V \cup T)^*$.

- $\alpha \overset{n+1}{\Rightarrow} \beta$ if there exist γ such that $\alpha \overset{n}{\Rightarrow} \gamma$ and $\gamma \overset{1}{\Rightarrow} \beta$.
- $\alpha \overset{*}{\Rightarrow} \beta$ if $\alpha \overset{n}{\Rightarrow} \beta$ for some $n \geq 0$.

A string in $(V \cup T)^*$ derivable from the start symbol S is called a *sentential form*. If the *sentential form* contains only terminals then it is called a *sentence*. The language generated by a grammar G, denoted $L(G)$, is the set of all sentences

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

A language $L \subseteq T^*$ is said to be a *context-free language* if $L = L(G)$ for some CFG.

Example 9.2. *The language $L = \{a^n b^n : n \geq 0\}$, which is not regular, is a CFL.*

The following grammar generates L. $G = (V, T, P, S)$ with $V = \{S\}$, $T = \{a, b\}$, $P = \{S \to aSb|\epsilon\}$

Proof. We need to show that $w \in L$ iff $w \in L(G)$.
(If)$w \in L(G)$ then by definition $S \overset{*}{\Rightarrow} w$. We show by induction on the number of derivation steps that $w = a^n b^n$.
Basis. $S \overset{*}{\Rightarrow} w$ in one step. In this case $w = \epsilon = a^0 b^0 \in L$.
Hypothesis. Assume that if $S \overset{*}{\Rightarrow} w$ in n steps then $w \in L$. Consider an $n + 1$ derivation $S \overset{n+1}{\Rightarrow} w$. Since it is more than one step then we can write $S \Rightarrow aSb \overset{n}{\Rightarrow} w$. This means that w is of the form axb. We can deduce that $S \overset{n}{\Rightarrow} x$ and by the induction hypothesis $x = a^m b^m$ therefore $w = a^{m+1} b^{m+1}$ and $w \in L$.
(Only if) Assume that $w = a^n b^n$. We show by induction on $|w|$ that $S \overset{*}{\Rightarrow} w$.
Basis. Assume that $|w| = 0$ then $w = \epsilon$ and since $S \to \epsilon$ then $S \overset{*}{\Rightarrow} w$.
Hypothesis. Assume that $|w| = a^n b^n$ implies that $S \overset{*}{\Rightarrow} w$.
Consider $w = a^{n+1} b^{n+1}$. We can write $w = axb$ with $x = a^n b^n$. By the induction hypothesis we have that $S \overset{*}{\Rightarrow} x$. From the production rule $S \to aSb$ we can write $S \Rightarrow aSb$. But since $S \overset{*}{\Rightarrow} x$ then $S \Rightarrow aSb \overset{*}{\Rightarrow} axb$ therefore $S \overset{*}{\Rightarrow} axb = w$. ■

Example 9.3. *The language $L = \{a^n b^m : 0 \leq n \leq m \leq 2n\}$ is generated by the following grammar:*

$$S \to aSb|aSbb|\epsilon$$

Theorem 9.1. *$L(G_{pal})$ with $G_{pal}(\{S\}, \{0, 1\}, P, S)$ and P, the set of productions, as defined in Figure 9.1 is the set of palindromes over $\{0, 1\}$*

Proof. To prove the theorem we prove that $w \in L(G_{pal})$ if and only if $w = w^R$.
The If part. Suppose that $w = w^R$ then we prove by induction on $|w|$ that $S \overset{*}{\Rightarrow} w$. It is noted that L can be divided into odd and even strings. The proof for the even strings start from the basis of $w = \epsilon$ whereas the proof for the odd strings starts from the

basis of $w = 0$ or $w = 1$. In other respects the two cases are totally equivalent.
Basis. For the even class, $|w| = 0$ then $w = \epsilon$ and since there is a production $S \to \epsilon$, thus $S \overset{*}{\Rightarrow} w$. Similarly for the odd class, $|w| = 1$ then $w = 0$ or $w = 1$ since we have $S \to 0|1$ then $S \overset{*}{\Rightarrow} w$.
Induction hypothesis. Assume that for all w such that $|w| = n$ and $w = w^R$ then $S \overset{*}{\Rightarrow} w$.
Induction step. Let $w = w^R$ with $|w| = n + 2$. Using the fact that $w = w^R$ then w starts and ends with the same symbol and we can write $w = bvb$ with $v = v^R$ where $b = 0$ or $b = 1$. From the induction hypothesis we get $S \overset{*}{\Rightarrow} v$. Combining this fact with one of the production rules we get $S \Rightarrow bSb \overset{*}{\Rightarrow} bvb = w$.

The only if part. We need to show that if $S \overset{*}{\Rightarrow} w$ then $w = w^R$ which we proceed to prove by induction on the number of derivations.
Basis. Assume that $S \Rightarrow w$ then there is a production such that $S \to w$, then $w = \epsilon$ or $w = 0$ or $w = 1$ and therefore $w = w^R$.
Induction step. Assume that $S \overset{k}{\Rightarrow} x$ implies that $x = x^R$ for $0 \leq k \leq n$. Consider the string w such that $S \overset{n+1}{\Rightarrow} w$ which can be written as $S \overset{1}{\Rightarrow} 0A0 \overset{n}{\Rightarrow} w = 0x0$. By the induction hypothesis, $x = x^R$ therefore $w = w^R$. ∎

Given a grammar G and language L we would like to show that $L = L(G)$. From the above two examples we deduce the following general approach:

> **General Rule:** To show that $w \in L$ implies $S \overset{*}{\Rightarrow} w$ we use induction on $|w|$. To show that $S \overset{*}{\Rightarrow} w$ implies $w \in L$ we use induction on the number of derivation steps.

Example 9.4. *Given a string w define $\#a(w) =$ the number of a's in w. Prove that the language $L = \{w \in \{a, b\}^* | \#a(w) = \#b(w)\}$ is generated by the following grammar:*

$$S \to aSbS|bSaS|\epsilon$$

Proof. The fact that all strings generated by the above rules produce strings with equal number of a's and b's is obvious. We need to show that the grammar generates **all** of them.
Let $w \in L$ prove by induction that $S \overset{*}{\Rightarrow} w$.
base case: $w = \epsilon$ then clearly $S \overset{*}{\Rightarrow} \epsilon$ since $S \to \epsilon$ is a production.
hypothesis: Assume that if w is of length $2n$ and $w \in L$ then $S \overset{*}{\Rightarrow} w$.
induction step: Assume that x, of length $2(n + 1)$ is in L. The string x can be decomposed (in many ways) into a prefix and suffix: $x = ps$ where the number of a's in p is equal to the number of b's. Let u be the smallest such prefix then we can write $x = uv$ where $\#a(u) = \#b(u)$ and it follows that $\#a(v) = \#b(v)$. The prefix u can start with either a or b and the two cases are symmetric so we consider the first case, u starts with a. But being the smallest prefix that has equal number of a's and b's it **must** end with a **b** (do you see why?). Therefore we can write $x = aybv$ where y and

v have equal number of a's and b's and of length at most $2n$. By the induction hypothesis $S \stackrel{*}{\Rightarrow} y$ and $S \stackrel{*}{\Rightarrow} v$ then we can write

$$S \stackrel{*}{\Rightarrow} aSbS \stackrel{*}{\Rightarrow} aybv$$

which completes the proof ■

Example 9.5. *Let*
$L = \{w \in \{a,b\}^* |$ *every prefix of* w *has at least as many a's as b's*$\}$. *Prove that the following grammar generates* L*:*

$$S \to aS|aSbS|\epsilon$$

Proof. First we show that every string generated by the above grammar generates strings in L by induction over the number of derivations.
base case $S \stackrel{1}{\Rightarrow} w$ then $w = \epsilon$ which is clearly in L.
hypothesis Assume that if $S \stackrel{n}{\Rightarrow} w$ then $w \in L$.
induction step Let $S \stackrel{n+1}{\Rightarrow} x$. We can write

$$S \stackrel{1}{\Rightarrow} aS \stackrel{n}{\Rightarrow} w$$

or

$$S \stackrel{1}{\Rightarrow} aSbS \stackrel{n}{\Rightarrow} w$$

It is clear that in both cases w starts with an **a**. The first case gives

$$S \stackrel{1}{\Rightarrow} aS \stackrel{n}{\Rightarrow} au$$

which implies that

$$S \stackrel{n}{\Rightarrow} u$$

and by the hypothesis $u \in L$ (i.e. every prefix of u has at least as many a's as b's).
Since $x = au$ then $x \in L$.
The second case we have

$$S \stackrel{1}{\Rightarrow} aSbS \stackrel{n}{\Rightarrow} aubv$$

It follows that $S \stackrel{n}{\Rightarrow} u$ and $S \stackrel{n}{\Rightarrow} v$ and by the hypothesis $u, v \in L$ and therefore $x \in L$.
Conversely, we show that every strings $w \in L$ can be obtained from the grammar above.
base case: for $w = a$ and $w = ab$ can be derived by the sequence $S \stackrel{*}{\Rightarrow} aS \stackrel{*}{\Rightarrow} a$ and $S \stackrel{*}{\Rightarrow} aSbS \stackrel{*}{\Rightarrow} ab$.
hypothesis: assume that if $w \in L$ and $|w| = n$ then $S \stackrel{*}{\Rightarrow} w$.
induction step. Let $x \in L$ and $|x| = n+1$. Since $x \in L$ then it has to start with an **a** and we write $x = ay$. Because $x \in L$ then every prefix of y has **at most** one b more than a. If y contains no b's then $y \in L$ and by hypothesis $S \stackrel{n}{\Rightarrow} y$ thus $S \stackrel{1}{\Rightarrow} aS \stackrel{n}{\Rightarrow} ay = x$. If y contains at least one b, let u be a prefix of y containing all

a's (u could be ϵ) and write $y = ubv$. Now since every prefix of y contains at most one more b than a's it follows that $v \in L$ and thus we have $S \overset{*}{\Rightarrow} v$ and $S \overset{*}{\Rightarrow} u$. Finally

$$S \overset{1}{\Rightarrow} aSbS \overset{*}{\Rightarrow} aubv = ay = x$$

■

Example 9.6. *Give a grammar that generates the language* $L = \{w \in \{a,b\}^* | \#a(w) > \#b(w)\}$.

$$\begin{aligned} S &\to EaS|EaE \\ E &\to aEbE|bEaE|\epsilon \end{aligned}$$

The production rules for the variable E ensures that the generated string has an equal number of a's and b's. For a given number of b's, say k, we can generate as many a's more than b's as we want by repeatedly using the production $S \to EaS$. For example, to generate m more a's than b's we apply $S \to EaS$ $m - 1$ times to get $EaEa \ldots EaS$ then we apply the rule $S \to EaE$ once to get $EaEa \ldots EaEaE$ so we have m times Ea. Since each E produces a string of equal number of a's and b's the result is k b's and $m + k$ a's.

9.3.1 Leftmost and rightmost derivations

Sometimes it is useful to restrict the way in which we derive a given string. One way to do such a thing is a *leftmost* derivation in which at each step the leftmost variable is replaced by the body of one of its productions. As an example we show that the string $a * (a + b00)$ belongs to the language generated by the CFG defined in Example 9.1

$$\begin{aligned} &E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E \underset{lm}{\Rightarrow} \\ &a * (E) \underset{lm}{\Rightarrow} a * (E + E) \underset{lm}{\Rightarrow} a * (I + E) \\ &\underset{lm}{\Rightarrow} a * (a + E) \underset{lm}{\Rightarrow} a * (a + I) \underset{lm}{\Rightarrow} a * (a + I0) \\ &\underset{lm}{\Rightarrow} a * (a + I00) \underset{lm}{\Rightarrow} a * (a + b00) \end{aligned}$$

Similarly we can define a *rightmost* derivation and use it to show that $a * (a + b00)$ belongs to $L(G)$.

$$\begin{aligned} &E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E) \underset{rm}{\Rightarrow} E * (E + I) \\ &\underset{rm}{\Rightarrow} E * (E + I0) \underset{rm}{\Rightarrow} E * (E + I00) \underset{rm}{\Rightarrow} E * (E + b00) \\ &\underset{rm}{\Rightarrow} E * (I + b00) \underset{rm}{\Rightarrow} E * (a + b00) \underset{rm}{\Rightarrow} I * (a + b00) \underset{rm}{\Rightarrow} a * (a + b00) \end{aligned}$$

9.4 Parse Trees

Given a grammar $G = (V, T, P, S)$ the parse trees of G are all trees satisfying the following conditions

1. Each interior node is labeled by a variable in V.
2. Each leaf node is labeled by a variable, a terminal or ϵ. If it is ϵ then it is the only child of its parent.
3. If an interior node is labeled A and its children are labeled, from left to right, $X_1, \ldots, X_k$ then $A \longrightarrow X_1 \ldots X_k$ is a production of P.

Example 9.7. *An example parse tree is shown in Figure 9.3.*

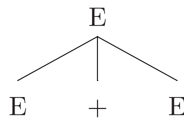

Figure 9.3: a parse tree for the grammar defined in example 9.1

Example 9.8. *An example parse tree is shown in Figure 9.4.*

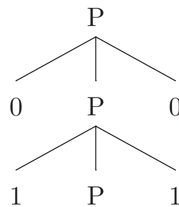

Figure 9.4: a parse tree for the grammar of palindromes

The yield of a parse tree is the concatenation of the labels, from left to right, of the leaves of the tree. Of special importance is the yield of a tree whose root is the start symbol of the grammar and the yield contains terminals only. In this case the yield is a string in the language of the grammar. The yield of the tree show in Figure 9.6 below is $a * (a + b00)$.

9.4.1 Ambiguous Grammar

The grammar shown in Figure 9.2 is ambiguous because, for example,the sentence $a + b * a$ has two different parse trees as shown in Figure 9.7 below. This leads us to the following definition

Definition 9.2. *A grammar $G = (V, T, P, S)$ is said to be ambiguous if $\exists w \in T^*$ for which we can find two different parse trees, each with root S and yield w.*

In the example above we are lucky since we can find a grammar for the same language which is not ambiguous

$$
\begin{aligned}
E &\to E + T|T \\
T &\to E * F|F \\
F &\to I|(E) \\
I &\to a|b|Ia|Ib|I0|I1
\end{aligned}
$$

Figure 9.5:

Not all languages, however, have an nonambiguous grammar. A context-free languages L is said to be inherently ambiguous if all its grammars are ambiguous. For example the language $L = \{a^i b^j c^k : i, j, k \geq 0, i = j$ or $j = k\}$ is inherently ambiguous.

9.5 Non Context-free Languages

Theorem 9.2. *If L is a context-free language then there exists a constant p such that if $s \in L$ and $|s| \geq p$ then we can write $s = uvwxy$ with*

1. $|vx| > 0$.
2. $|vwx| \leq p$.
3. $uv^i wx^i y \in L$.

Example 9.9. $L = \{a^n b^n c^n | n \geq 0\}$ *is not context-free.*

Proof. Assume that L is context-free and let p be the pumping length. Consider $s = a^p b^p c^p$. By the pumping lemma we can write $s = uvwxy$ with $|vwx| \leq p$. This means that vwx cannot contain both a's and c's. Also by the pumping lemma $uv^2wx^2y \in L$. There are two cases

1. vwx does not contain a the uv^2wx^2y contains less a's then b's and c's.
2. vwx does not contain c the uv^2wx^2y contains less c's then b's and a's.

Example 9.10. $L = \{ww | w \in \{0,1\}^*\}$ *is not context-free.*

Proof. Consider the string $s = 0^p 1^p 0^p 1^p \in L$. By the pumping lemma we can write $s = uvwxy$ and consider $uwy \in L$. There are four cases

1. Since $|vwx| \leq p$ then vwx could be contained in the first 0^p therefore $uwy = 0^{p-k}1^p 0^p 1^p$ and since $uwy \in L$ by P.L. then we can write $0^{p-k}1^p0^p1^p = tt$ thus $|t| = 2p - k/2$. On the other hand, since $k \leq p$ then $p - k/2 > 0$. Using the last inequality we get $2p - k < |t| < 3p - k$ then it follows that the first t in $tt = 0^{p-k}1^p0^p1^p$ ends with a 0. On the other hand, the second $tt = 0^{p-k}1^p0^p1^p$ ends with a 1 which is a contradiction. Similarly, if vwx is contained in the first and the second 1^p or the second 0^p the argument is the same because we can write, for example, $uwy = 0^p1^{p-k}0^p1^p = tt \in L$ and since $2p - k < |t| < 3p - k$ then the first t ends with a 0 whereas the second t ends with a 1, also a contradiction.
2. vwx is contained in the first 0^p1^p. Since $|vwx| \leq p$ then we can write $uwy = tt \in L$ and $uwy = 0^{p-i}1^{p-j}0^p1^p$ with $0 < k = i + j \leq p$. Now $|tt| = 4p - k$ so $2p - k < |t| = 2p - k/2 < 3p - k$. this means that the "first" t ends with a 0 and the "second" t ends with a 1, a contradiction.
3. vwx is contained in 1^p0^p. Since $|vwx| \leq p$ then we can write $vx = 1^i0^j$. Let $k = i + j$ from the P.L we know that $0 < k \leq p$, also $uwy \in L$ therefore $0^p1^{p-i}0^{p-j}1^p \in L$. So we can write $tt = 0^p1^{p-i}0^{p-j}1^p$ thus $2p - k < |t| = 2p - k/2 < 3p - k$. This means that the "first" t starts and ends with 0 but the "second" t ends with 1, a contradiction.

4. vwx is contained in the second 0^p1^p Since $|vwx| \leq p$ then we can write $vx = 0^i1^j$. Let $k = i + j$ from the P.L we know that $0 < k \leq p$, also $uwy \in L$ therefore $0^p1^p0^{p-i}1^{p-j} = tt \in L$. We have $2p - k < |t| < 3p - k$ which means that the "second" t starts with a 1 whereas the "first" t starts with a 0, a contradiction.

Alternative proof
Since $s = 0^p1^p0^p1^p = uvwxy$, by pumping lemma $uwy \in L$ which means that $uwy = tt$ for some string t. Let $|vx|= k$ then $|uwy|= 4p - k$, if we combine that with the fact that $|vx|\leq p$ we get $2p - k <|t|= 2p - k/2 < 3p - k$.
There are three cases: vx was "removed" from the first 0^p1^p of s, from the second or from both.
In the first case we get that $uwy = tt = \alpha 0^p1^p$ with $|\alpha|= 2p - k$. On the other hand, we know that $2p - k <|t|= 2p - k/2 < 3p - k$ then the first t ends with 0 while the second ends with 1 which is a contradiction.
In the second case we get that $uwy = tt = 0^p1^p\alpha$ and again $|\alpha|= 2p - k$. Using the size of t we get that the second t starts with 1 while the first starts with 0, again a contradiction.
The third and final case is when $uwy = tt = 0^p1^{p-i}0^{p-j}1^p$ with $i + j = k$. Note that $i \neq 0$, $j \neq 0$, otherwise it will fall back to the previous cases. Since $|t|> 2p - k > p$ then the only way for both t's to start with 0 and end with 1 is if $i = j = k/2$ but that case means the first t has p 0's and $p - k/2$ 1's while the second t has $p - k/2$ 0's and p 1's.

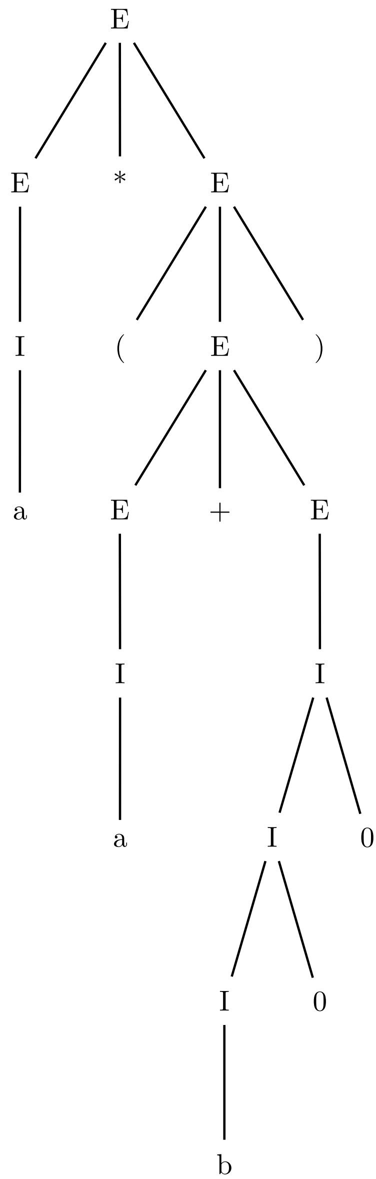

Figure 9.6: $a * (a + b00)$ is the yield of this parse tree

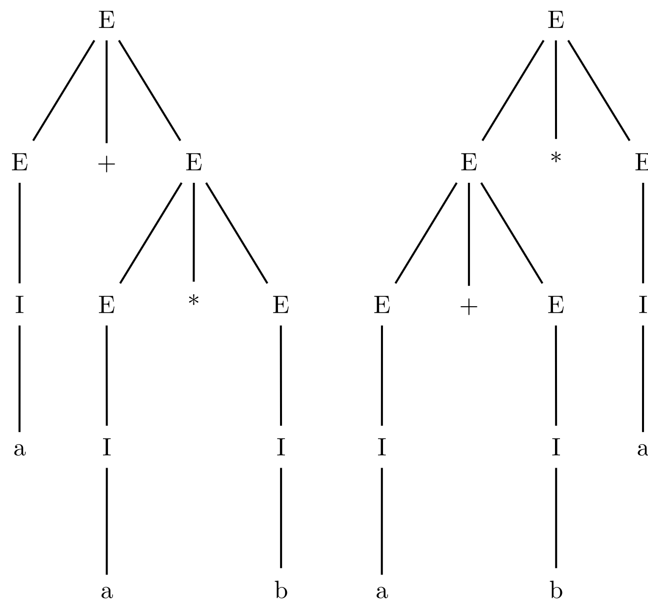

Figure 9.7: Two different parse trees for $a + b * a$

Lecture 10

Pushdown Automata

10.1 Introduction

A pushdown automata (PDA) is a nondeterministic finite automaton with ϵ-transitions with one extra feature: a stack on which it can store list of symbols. This way a PDA can remember an infinite number of information and access them in a last-in first-out fashion. A PDA bases its transitions on

1. The current state
2. The input symbol
3. The symbol on top of the stack.

In one transition the PDA

1. Consumes the input symbol.
2. Goes from the current state to a new state which could be the same.
3. Replaces the top of the stack with 0 or more symbols.

10.2 Formal Definition of PDA

A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ with

1. Q is a finite set of state.
2. Σ is a finite set of symbols that does not include ϵ.
3. Γ is a finite set of stack symbols, usually $\Gamma = \Sigma \cup Z_0$.
4. δ the transition function which takes a triple (q, a, X) where $a \in Q$, $a \in \Sigma$ or $a = \epsilon$ and $X \in \Gamma$. The output of δ is a pair p, γ where p is a state and γ is a string of stack symbols that replace X.

5. q_0 is the start state.
6. Z_0 is the start symbol on the stack.
7. F is the set of accepting states.

Example 10.1. *The language $L = \{ww^R : w \in \{0,1\}^*\}$ is accepted by the PDA $P = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$ with δ defined as follows:*

1. $\delta(q_0, a, X) = (q_0, aX)$ if $a \neq \epsilon$. The symbol a is pushed on the stack.
2. $\delta(q_0, \epsilon, X) = (q_1, X)$. The stack is left intact.
3. $\delta(q_1, 0, 0) = (q_1, \epsilon)$ and $\delta(q_1, 1, 1) = (q_1, \epsilon)$. If the input symbol matches the top of the stack then the PDA consumes the input and pops the stack.
4. $\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$. We the PDA encounters the bottom of the stack it moves to q_2.

Some notes on the behaviour of the PDA. First, the PDA ”guesses” when it reaches the ”middle” of the string and it moves ”spontaneously” to state q_1 and starts ”matching” the string stored on the stack with the input. An equivalent way of looking at this is that for a given string w the PDA pushes n symbols of w on the stack and then starts the matching process. Since n is a variable there is a value of n that works.

Example 10.2. *The language of all palindromes, $L = \{w = w^R : w \in \{0,1\}^*\}$ is accepted by the PDA of example 10.1 with the addition of the transitions*

$$\delta(q_0, 1, X) = (q_1, X)$$
$$\delta(q_0, 0, X) = (q_1, X)$$

The graphical representation of the PDA of example 10.2 is shown in Figure 10.1.

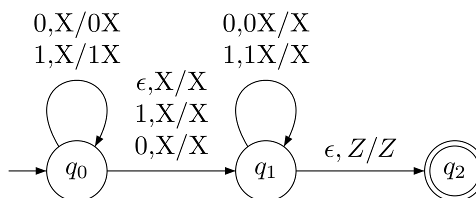

Figure 10.1: PDA for palindromes

We will argue that the PDA accepts all palindromes but we need to show that all palindromes of even length are of the form xx^R and the ones with odd length are of the form xax^R where $a = 0$ or $a = 1$.
even. We prove by induction that all even palindromes are of the form xx^R.
Basis. $\epsilon = \epsilon\epsilon^R$.
Hypothesis. Assume that all palindromes of length n even are of the form xx^R and consider a palindrome w of length $n + 2$. Since n starts and ends with the same

symbol we can write $w = axa$ where $a = 0$ or $a = 1$. On the other hand since $w = w^R$ we have $w^R = (axa)^R = ax^Ra = axa$. Therefore x is a palindrome of length n and by the induction hypothesis we can write $x = yy^R$. Thus $w = ayy^Ra = (ay)(ay)^R$.

odd. We also prove by induction that all odd palindromes are of the form xax^R where $a = 0$ or $a = 1$.

Basis. Odd palindrome of length one is $a = \epsilon a \epsilon = \epsilon a \epsilon^R$.

Hypothesis. Assume that all palindromes of length $n + 1$, n even, are of the form xax^R. Consider a palindrome w with $|w| = n + 3$. Again, w starts and ends with the same symbol so we can write: $w = axa$ and as before this implies that x is a palindrome of length $n + 1$. By the induction hypothesis $x = yay^R$.Therefore $w = ayay^Ra = (ay)a(ay)^R$.

From the above we can see that any palindrome can be written as $x\alpha x^R$ where α could be 0,1 or ϵ. Then given an input palindrome $x\alpha y$, with $y = x^R$, we will use the following strategy to recognize palindromes by starting in state q_0 (see PDA in figure 10.1.

1. If the input symbol is in x push it on the stack.
2. If α is reached, consume it and move to state q_1.
3. For every symbol in y pop a matching symbol from the stack.
4. When all symbols are matched, all the input string is consumed and the stack is empty. Move to state q_2 and accept.

10.3 Instantaneous Descriptions of a PDA

When a PDA reads an input symbol it goes from given configuration to another configuration which unlike the case for finite automata includes more than the state. A PDA configuration is specified by a triple (q, w, γ) with

1. q is the state of the PDA.
2. w is the remaining input string.
3. γ is the content of the stack.

Given a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ and suppose that $\delta(q, a, X)$ contains (p, α) then we define the configuration transition $\vdash$ as follows: for all $w \in \Sigma^*$ and $\beta \in \Gamma^*$

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

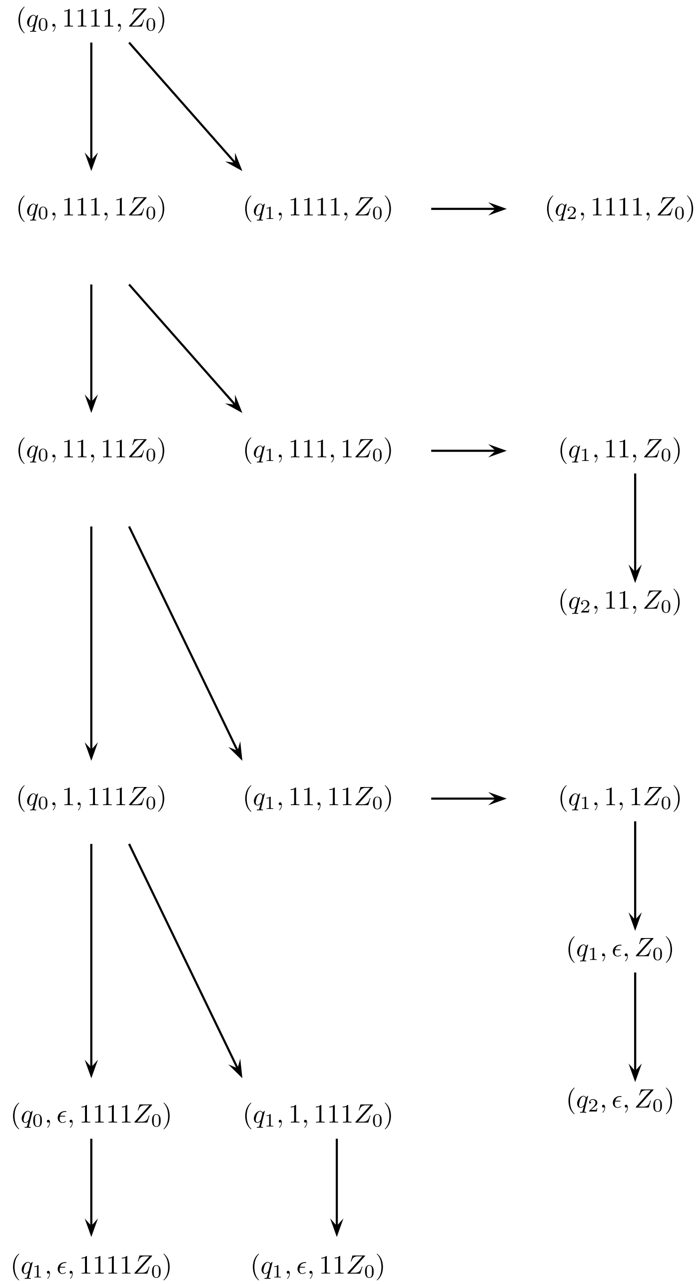

Figure 10.2: ID's of PDA on input 1111

Lecture 11

CYK Algorithm

11.1 Introduction

Given a grammar $G =< V, T, P, S >$ in Chomsky Normal Form (CNF) and a string $w \in T^*$ we would like to answer the question: is $w \in L(G)$. This can be answered by using the CYK algorithm. Suppose that we can write $w = a_1 a_2 \dots a_n$. and let

$$X_{ij} = \{X \in V | X \stackrel{*}{\Rightarrow} a_i \dots a_j\}$$

It is clear that $w \in L(G) \Leftrightarrow S \in X_{1n}$. Once the set X_{1n} is computed we check if the start symbol S is in X_{1n}. If it is then the string w is in the language of G. But since G is in CNF then $\exists Y, Z \in V$ such that $X \to YZ$. Combining both results we get

$$YZ \stackrel{*}{\Rightarrow} a_i \dots a_j$$

The above derivation has many possibilities for Y and Z. It could be that $Y \stackrel{*}{\Rightarrow} a_1$ and $Z \stackrel{*}{\Rightarrow} a_2 \dots a_j$ or $Y \stackrel{*}{\Rightarrow} a_1 a_2$ and $Z \stackrel{*}{\Rightarrow} a_3 \dots a_j$... etc. In general we can write $Y \stackrel{*}{\Rightarrow} a_1 \dots a_k$ and $Z \stackrel{*}{\Rightarrow} a_{k+1} \dots a_j$ for $i \le k \le j - 1$. Using the definition of the $X's$ we have

$$Y \in X_{ik}$$
$$Z \in X_{k+1j}$$

Collecting all the above results we get a recursive way of computing the X_{ij}

$$X_{ij} = \{X \in V | X \to YZ, Y \in X_{ik}, Z \in X_{k+1j}, i \le k \le j - 1\}$$

A simple top-down approach to compute the X_{ij} would take exponential time. Instead we use a bottom-up calculation, i.e. dynamic programming.

11.2 Examples

Example 11.1. *Consider the grammar (in CNF)*

$$
\begin{aligned}
S &\to AB|BC \\
A &\to BA|a \\
B &\to CC|b \\
C &\to AB|a
\end{aligned}
$$

Does the string $w = baaba$ belong to the language?We know that it does if $S \in X_{15}$ where

$$
\begin{aligned}
X_{15} &= \{X|X \to YZ, Y \in X_{1k}, Z \in X_{(k+1)5}, 1 \le k \le 4\} \\
&= \{X|X \to YZ\} \\
&\textit{where} \\
&Y \in X_{11}, Z \in X_{25} \\
&\textit{or} \\
&Y \in X_{12}, Z \in X_{35} \\
&\textit{or} \\
&Y \in X_{13}, Z \in X_{45} \\
&\textit{or} \\
&Y \in X_{14}, Z \in X_{55}
\end{aligned}
$$

And each of the above $X's$ needs to be computed from smaller components. For example to compute X_{12} we need to compute X_{11} and X_{22}...etc. As mentioned before we do it in a bottom-up manner. The entries in the table above were obtained as

{S,A,C}				
∅	{S,A,C}			
∅	{B}	{B}		
{S,A}	{B}	{S,C}	{S,A}	
{B}	{A,C}	{A,C}	{B}	{A,C}
b	a	a	b	a

Table 11.1: Solution for example 11.1

follows.
Row one: $X_{11} = \{X|X \to b\} = \{B\}$. $X_{22} = \{X|X \to a\} = \{A, C\}$. $X_{33} = X_{22}$. $X_{44} = X_{11}$ *and* $X_{55} = X_{22}$.
Row two: $X_{12} = \{X|X \to YZ, Y \in X_{11}, Z \in X_{22}\} = \{X|X \to BA \textit{ or } X \to BC\} = \{A, S\}$.
$X_{23} = \{X|X \to YZ, Y \in X_{22}, Z \in X_{33}\} = \{X|X \to AA \textit{ or } X \to AC \textit{ or } X \to CA \textit{ or } X \to CC\} = \{B\}$
$X_{34} = \{X|X \to YZ, Y \in X_{33}, Z \in X_{44}\} = \{X|X \to AB \textit{ or } X \to CB\} = \{S, C\}$
$X_{45} = \{X|X \to YZ, Y \in X_{44}, Z \in X_{55}\} = \{X|X \to BA \textit{ or } X \to BC\} = \{A, S\}$

Row three: $X_{13} = \{X | X \to YZ, Y \in X_{11}, Z \in X_{23} \textit{ or } Y \in X_{12}, Z \in X_{33}\} = \{X | X \to BB \textit{ or } X \to SA \textit{ or } X \to SC \textit{ or } X \to AA \textit{ or } X \to AC\} = \emptyset$
$X_{24} = \{X | X \to YZ, Y \in X_{22}, Z \in X_{34} \textit{ or } Y \in X_{23}, Z \in X_{44}\} = \{X | X \to AS \textit{ or } X \to AC \textit{ or } X \to CS \textit{ or} X \to CC \textit{ or } X \to BB\} = \{B\}$
$X_{35} = \{X | X \to YZ, Y \in X_{33}, Z \in X_{45} \textit{ or } Y \in X_{34}, Z \in X_{55}\} = \{X | X \to AS \textit{ or } X \to AA \textit{ or } X \to CS \textit{ or } X \to CA \textit{ or } X \to SA \textit{ or } X \to SC \textit{ or } X \to CA \textit{ or } X \to CC\} = \{B\}$
Row four: $X_{14} = \{X | X \to YZ, Y \in X_{11}, Z \in X_{24} \textit{ or } Y \in X_{12}, Z \in X_{34} \textit{or } Y \in X_{13}, Z \in X_{44}\} = \{X | X \to BB \textit{ or } X \to SS \textit{ or } S \to SC \textit{ or } X \to AS \textit{ or } X \to AC \textit{ or } \emptyset\} = \emptyset$
$X_{25} = \{X | X \to YZ, Y \in X_{22}, Z \in X_{35} \textit{ or } Y \in X_{23}, Z \in X_{45} \textit{ or } Y \in X_{24}, Z \in X_{55}\} = \{X | X \to AB \textit{ or } X \to CB \textit{ or } X \to BS \textit{ or } X \to BA \textit{ or } X \to BA \textit{ or } X \to BC\} = \{S, C, A\}$
Row five: $X_{15} = \{X | X \to YZ, Y \in X_{11}, Z \in X_{25} \textit{ or } Y \in X_{12}, Z \in X_{35} \textit{ or } Y \in X_{13}, Z \in X_{45} \textit{ or} Y \in X_{14}, Z \in X_{55}\} = \{X | X \to BS \textit{ or } X \to BA \textit{ or } X \to BC \textit{ or } X \to SB \textit{ or } X \to AB \textit{ or } X \to \emptyset \textit{ or } X \to \emptyset\} = \{A, S, C\}$

Example 11.2. *In this example we first convert into the grammar $G =< V = \{S\}, T = \{a, b\}, P, S >$ with P, the production rules given by*

$$S \to aSb$$
$$S \to ab$$

We introduce two new variables A and B with $A \to a$ and $B \to b$ to get

$$S \to ASB$$
$$S \to AB$$

Then we introduce a third variable C with $C \to SB$. The final result is

$$S \to AC$$
$$S \to AB$$
$$C \to SB$$
$$A \to a$$
$$B \to b$$

Now having the grammar in CNF we can check if the string $aabb$ is in $L(G)$. By running the CYK algorithm as we did in the previous example we get the results shown in table 14.2 below.

{S}			
∅	{C}		
∅	{S}	∅	
{A}	{A}	{B}	{B}
a	a	b	b

Table 11.2: Solution for example 11.2

11.3 Complexity

Given an input string w with $|w| = n$, the complexity of the CYK algorithm can be obtained as follows. Recall that

$$X_{ij} = \{X|X \to YZ, Y \in X_{ik}, Z \in X_{k+1j}, i \leq k \leq j-1\}$$

Now for each $YZ \in X_{ik}X_{k+1j}$ we need to find another variable X such that $X \to YZ$. Since the number of variables in the grammar is independent of the size of the size of the input string, then the size of $X_{ik}X_{k+1j}$ is independent of the input size therefore the cost of finding X such that $X \to YZ$ for each $X_{ik}X_{k+1j}$ is independent of the size of the input. For each X_{ij} there are at most $O(n)$ pairs $X_{ik}X_{k+1j}$ thus the cost of computing X_{ij} is $O(n)$. On the other hand the number of X_{ij} to be computed is $O(n^2)$ therefore the total cost of the algorithm is $O(n^3)$.

Lecture 12

Turing Machines

12.1 Introduction

So far we have looked at two models of computations: the equivalent models of DFA, NFA and REGEXP and the equivalent model of PDA and CFG. We saw that some languages are not regular and therefore cannot be computed using any of the regular language tools and some languages are not context-free and therefore cannot be computed using PDA's. In this lecture we introduce a new model: Turing Machines. Intuitively, a TM is a DFA or NFA with an infinite tape that can store symbols. A TM has a head position on one of the tape cells. The TM makes a decision based on the current state and the symbol "under" the head. The action of the TM can involve moving to a new state (which could be the same as the old) replacing the symbol under the head with another symbol(could be the same) and moving the head one cell to the right or left. We will see that such a model allows us to solve problems that could not be solved with the models that we have considered so far.

12.2 Formal Definition

Formally a Turing Machine is a 9-tuple

$$M = (Q, \Sigma, \Gamma, \delta, \rhd, \sqcup, s, t, r)$$

where

1. Q is a finite set of states.
2. Σ is a finite *input* alphabet that does not contain $\sqcup$ and $\rhd$.
3. Γ is a finite *tape* alphabet. $\Sigma \subset \Gamma$.
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L\}$ is the transition function.
5. $\rhd$ is the left endmarker.

6. $\sqcup$ is the blank symbol.

7. $s \in Q$ is the starting state.

8. $t \in Q$ is the accepting state.

9. $r \in Q$ is the rejecting state.

Intuitively, $\delta(p, a) = (q, b, R)$ means that if the TM is in state p and the head reads the symbol a then it moves to state q, replaces a by b and the head moves to the right. There are a few restrictions on the above definition. For every $p \in Q$ there exist $q \in Q$ such that $\delta(p, \triangleright) = (q, \triangleright, R)$. This means that the head "does not fall" of the end of the tape. Also the accepting and rejecting states are "halting" states. When the TM reaches them it never "leaves" them. More precisely, for every $b \in \Gamma$ there exists $c, c' \in \Gamma$ and $d, d' \in \{L, R\}$ such that

$$\delta(t, b) = (t, c, d)$$
$$\delta(r, b) = (r, c', d')$$

12.3 Configurations

Define the infinite string $\sqcup^w = \sqcup \sqcup \ldots$. The tape of a TM contains a infinite string of the form $x\sqcup^w$ where $x \in \Gamma^*$ is a finite string. Let $\Gamma^w = \{x \sqcup^w | x \in \Gamma^*\}$. A configuration of a TM is an element from $Q \times \Gamma^w \times N$: it specifies the state the TM is in, the content of the tape and the position of the head. For example the configuration

$$(p, \triangleright abc\sqcup^w, 1)$$

means that the TM is in state p, the tape contains the string $\triangleright abc\sqcup^w$ and the head is positioned (reading) on "a". Alternatively, since for any finite number of steps the tape will always contain a infinite number of blank symbols to right it is not necessary to write them explicitly. On the other hand, since we are using symbols, $p, r, s, t \ldots$ for states different than the alphabet,$a, b, c \ldots$ we can use a simple notation for a configuration as : $\triangleright pabc$. This means that the tape contains $abc\sqcup^w$ and the head is positioned over the a and the TM is in state p.
Let $\alpha = \triangleright a_1 \ldots a_{k-1} p a_k \ldots a_n$ be a configuration. We say that α yields in one step the configuration β, denoted by $\alpha \vdash \beta$ or $\alpha \vdash^1_\beta$ if

$$\beta = \triangleright a_1 \ldots a_{k-1} b q a_{k+1} \ldots a_n \qquad \textbf{and } \delta(p, a_k) = (q, b, R)$$
OR
$$\beta = \triangleright a_1 \ldots q a_{k-1} b a_{k+1} \ldots a_n \qquad \textbf{and } \delta(p, a_k) = (q, b, L)$$

As usual, we define $\alpha \vdash_\beta$ in a recursive manner

1. $\alpha \vdash^0_\alpha$.

2. $\alpha \vdash^{n+1}_\beta$ if $\alpha \vdash^n_\gamma \vdash^1_\beta$ for some γ.

3. $\alpha \vdash_{\beta}$ if $\alpha \vdash_{\beta}^{n}$ for some n.

A Turing Machine, M is said to *accept* an input string $x \in \Sigma^*$ if

$$\triangleright sx \vdash_{\triangleright} ytz \text{ for some } y, z \in \Gamma^*$$

for some $y \in \Gamma^*$ and $n \geq 0$. A Turing Machine, M is said to *reject* an input string $x \in \Sigma^*$ if

$$\triangleright sx \vdash_{y} rz \text{ for some } y, z \in \Gamma^*$$

for some $y \in \Gamma^*$ and $n \geq 0$. We say that a Turing Machine *halts* on x if it *accepts* or *rejects* x. An **important** point should be noted: for a Turing Machine "not accepting" is **NOT** the same as rejecting because a Turing Machine can fail to *halt* (loops).

Definition 12.1. *The set of strings that a Turing Machine M accepts is called the language of M or the language* ***recognized by*** *M, denoted by $L(M)$.*

Definition 12.2. *A language L is called recursively enumerable if there exists a Turing Machine M such that $L = L(M)$.*

Definition 12.3. *A Turing Machine that always halt is called a decider or total.*

Definition 12.4. *A language L is called recursive or decidable if there exists a total Turing Machine M such that $L = L(M)$.*

Figure 12.1: Solution for example 1

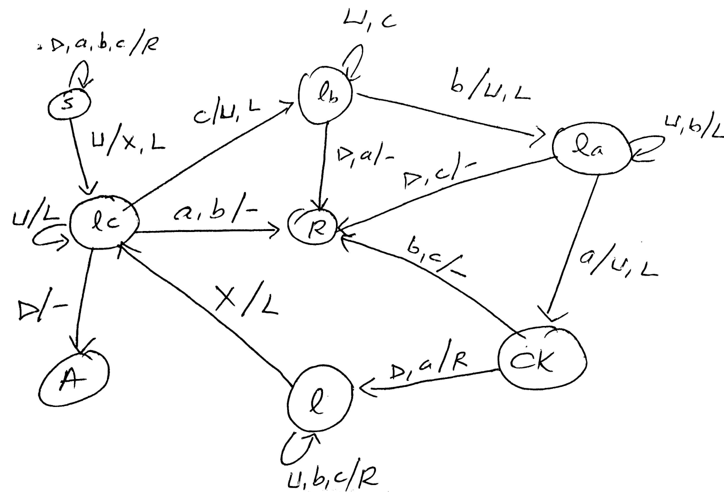

Figure 12.2: Corrected solution for example 1

Example 12.1. *A Turing Machine that accepts* $L = \{a^n b^n c^n \mid n \geq 0\}$

The basic idea is to do multiple "passes" and in each pass the machine "removes", by replacing the symbol with a blank, one **c**, one **b** and one **a**. This is done as follows. For a given "pass" the machine is looking for a **c**, if it finds one it replaces it with a blank and starts looking for a **b**, when it finds a **b** it replaces it with a blank and starts looking for an **a**. When it finds an **a** it replaces it with a blank and rewinds by going right until if finds an "X" which means the current "pass" is over. If at a certain there are no more a's or b's or c's then it accepts. If at any point the machine does not find the symbol it is looking for it rejects.
Actually the previous solution is not totally correct because it also accepts strings of the form $abcabc \ldots abc$. Once an 'a' is matched we need to check that it is not followed by a 'c', or a 'b' as shown in the modified TM in Figure 12.2

Example 12.2. *A Turing machine that accepts the language*
$L = \{0^p \mid p = 2^n, n \geq 0\}$
The basic idea is to divide the number of 0's by two in every pass. If initially the number of zeros is a power of 2 then after every pass the result of the "division" of the number of zeros is even except at the last step when only one zero is left. The "division" operation is done by marking every other 0 by an "X". The implementation is shown in Figure 15.2.

Example 12.3. *A Turing machine that accepts the language*
$L = \{w\#w \mid w \in \{0,1\}^*\}$
The basic idea is to match every symbol read before the "#" mark with the same symbol after the "#" mark. Again the TM does multiple passes, in each pass it reads the first symbol, say ***a****, replaces it with an "X" then starts looking for the first non "X"*

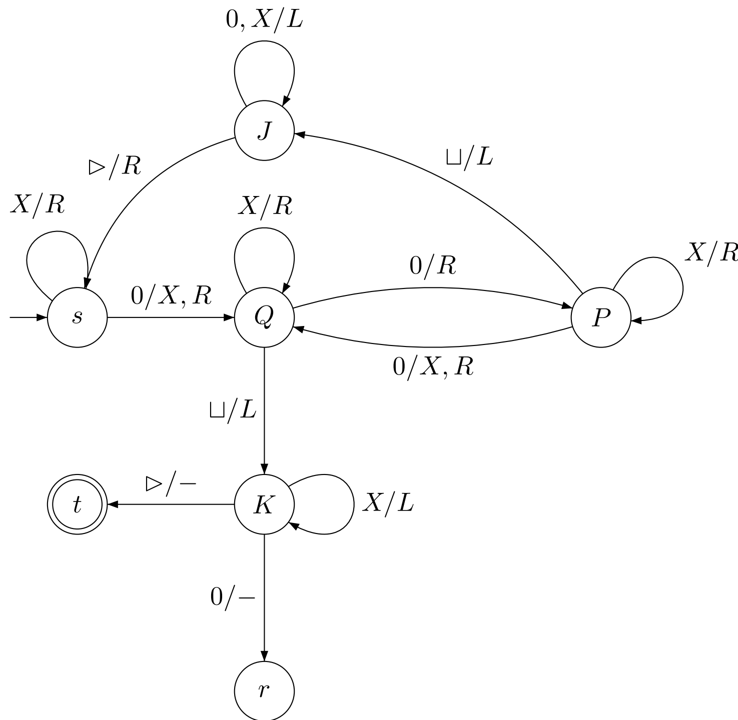

Figure 12.3: Solution for example 2

*symbol after the "#" mark. If that symbol is not **b** reject otherwise replace it by an "X". This constitutes a single pass. Then do as many passes as required to complete the input string.*

12.4 Multitape Turing Machine

A multitape machine is a direct extension of a single tape Turing Machine. A multitape Turing machine has k tapes and k heads. When the machine is in state p it reads the k symbols, $a_1 \dots a_k$, pointed to by the k heads and makes the transition to a state q (possibly the same as q), overwrites the symbols $a_1 \dots a_k$ by the symbols $b_1 \dots b_k$ (with the possibility of some of them being the same), and moves each head independently to the right or to the left. Initially, the input string is placed in the first tape and all other tapes are blank. As the computation progresses the configuration of a k tape machine can be describe by $(p, z_1, \dots, z_k, n_1, \dots, n_k)$ where $z_i \in \Gamma^w$ and n_i is the position of head i. In one step the machine moves to configuration $(q, z'_1, \dots, z'_k, n'_1, \dots, n'_k)$ with z_i and z'_i differ in one symbol at position n_i and $n'_i = n_i \pm 1$.

Example 12.4. *A two-tape Turing machine that will be useful later is one that moves*

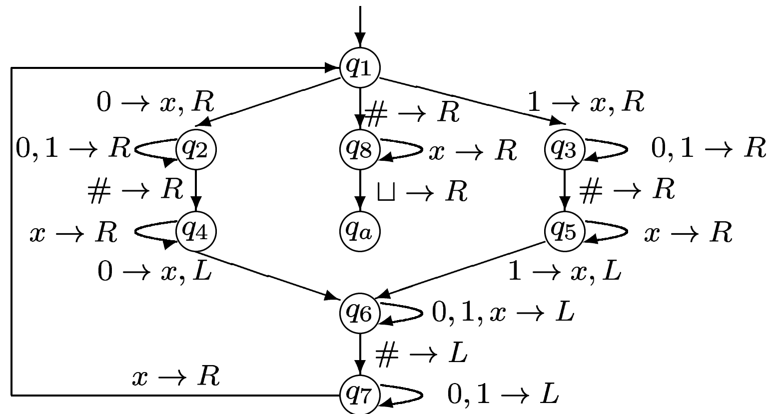

Figure 12.4: Solution for example 3

its second tape k times where k is a value stored on its first tape. We still use the same notation as before but since there are two tapes we label the values of each tape to distinguish between them. The two-tape Turing machine that performs this operation is shown below in figure 12.5. The basic idea is is to do k passes where in each pass the head of tape 2 moves one position to the right and the value in tape 1 is decremented by 1.
The two-tape machine stops when the value in tape 1 reaches zero. The machine starts in state q_0 and stays there while moving the head to the right as long as the head reads zero. If no 1 is read and the blank symbol is reached the machine stops. If the value in tape 1 is non-zero, i.e. contains at least one 1, it decrements the value by one as follows: the machine "rewinds" to the end of the input and starts scanning left turning a 0 into a 1 until it reads a 1 which it turns into a zero and goes back to state q_0.

Example 12.5. *Another two-tape Turing machine that we will use is later is one that copies the content of its second tape to it first tape. The solution is shown below in figure 12.6. Both tapes start at the beginning of the input and the value read in tape 1 is copied to tape 2 until it encounters a comma or a star symbol then the copying stops and tape 1 is rewinded to the beginning.*

12.4.1 Storing State Information

A number of states can be added to a Turing machine to allow it to save information for later use or to "remember" what it is doing. As an example suppose that we want to design an TM that recognizes the language of all strings in which the first symbol does not appear again in the string, using regular expressions we can write: $01^* \cup 10^*$. A Turing machine that recognize that language might work as follows: initially the machine is waiting to read its first symbol, once it does it moves to a different state

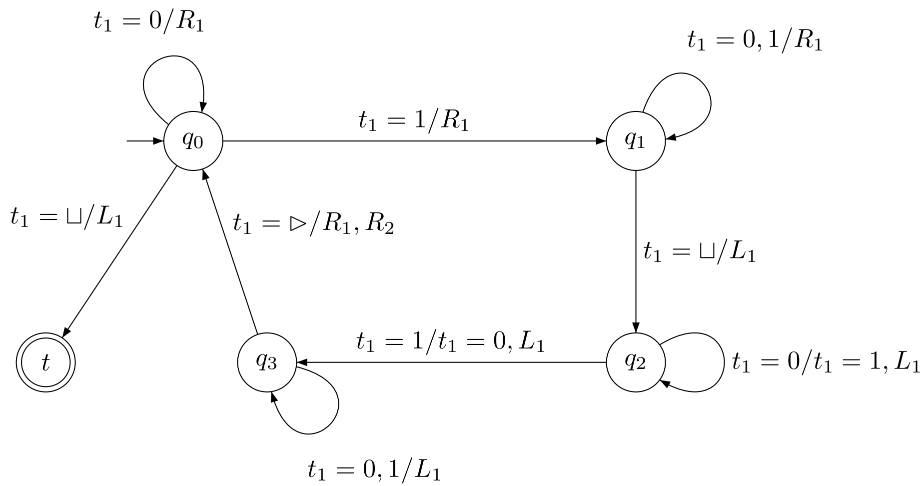

Figure 12.5: A two-tape machine that moves its second head k number of times, k is stored in tape 1

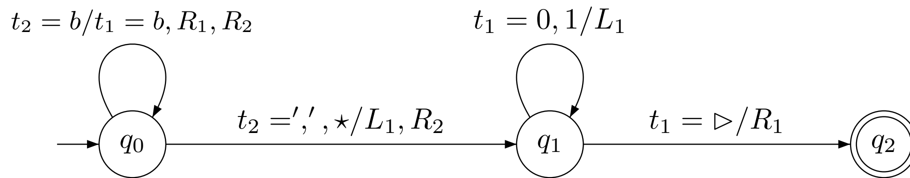

Figure 12.6: A two-tape machine that copies the content of its second tape to its first tape

that symbolizes that the TM has now read its first symbol and reads the others. At any time the TM needs to be able to compare the read symbol with the first symbol which means it has to store the first symbol somewhere. This is done by using tuples to denote the state of a TM. For example, initially the TM is in state (q_0, U) then if it reads a 1 it moves to state $(q_1, 1)$ and moves right otherwise it goes to state $(q_1, 0)$ and moves right. These moves can be described by the transition function $\delta((q_0, U), 0) = ((q_1, 0), 0, R)$ and $\delta((q_0, U), 1) = ((q_1, 1), 1, R)$. When the TM is in state $(q_1, 0)$ it means it has read the first symbol and it is a 0 and when it is in state $(q_1, 1)$ it means it has read the first symbol and it is a 1. The TM continues until it finds a blank where it halts and accepts. If at any moment it reads a symbol equal to the first one it rejects. These moves are described by

$$\delta((q_0,U),0) = ((q_1,0),0,R) \qquad \text{first symbol is 0}$$
$$\delta((q_0,U),1) = ((q_1,1),1,R) \qquad \text{first symbol is 1}$$
$$\delta((q_1,0),1) = ((q_1,0),1,R) \qquad \text{continue}$$
$$\delta((q_1,0),0) = ((q_1,U),0,R) \qquad \text{reject}$$
$$\delta((q_1,1),0) = ((q_1,1),1,R) \qquad \text{continue}$$
$$\delta((q_1,1),1) = ((q_1,U),1,R) \qquad \text{reject}$$
$$\delta((q_1,0),\sqcup) = ((q_1,A),\sqcup,R) \qquad \text{accept}$$
$$\delta((q_1,1),\sqcup) = ((q_1,A),\sqcup,R) \qquad \text{accept}$$

12.4.2 Simulating a Multitape Machine

Given a multitape machine N with a finite set of states Q_N and a tape alphabet Γ and a transition function $\delta(p,a_1,\ldots,a_k) = (q,b_1,\ldots,b_k,H_1,\ldots,H_k)$ where each of the H_i's can be either R or L, the head moves right or left. N can be simulated by a single tape Turing machine M as follows. The states of M are tuples from $Q_N \times \Gamma^k \times K \times H^K$ where $K = \{0,1,\ldots,k\}$. The tape alphabet for M are form the set $(A \cup \Gamma)^k$ where $A = \{\sqcup, X\}$. In other words, a single tape symbol in M contains a tuple of the form $(A,b_1,A,b_2,\ldots,A,b_k)$ with $b_i \in \Gamma$. The role of the X's is to denote the position of the head for each tape. For example, if tape i contains an X in cell m then the symbol to be read in tape $i+1$ is in the position m. A convenient graphical representation would be for the tape of M to contain $2k$ tracks as shown in the figure below. Using the storage in the state, introduced in the previous section M scans its input from left to right looking for an X in the tracks. For each X in track i it copies the content of the cell in track $i+1$ and stores it until it reads all of them then it uses the transition function of N. If N starts with state p then M starts in state $(p,\underbrace{U,\ldots,U}_{k\ times},0,\underbrace{U,\ldots,U}_{k\ times})$ where U symbolizes a don't care symbol and the 0 means that M has not read any symbol yet. It scans from left to right and when it encounters an X in track i and the symbol in track $i+1$ is a_i then it moves to state $(p,\underbrace{U,\ldots,a_i,\ldots,U}_{k\ times},1,\underbrace{U,\ldots,U}_{k\ times})$ and it keeps doing this until the it reaches the state $(p,a_1,\ldots,a_k,k,\underbrace{U,\ldots,U}_{k\ times})$ which means that it has read all the symbols. At this point M consults the transition function of N and if it is $\delta(p,a_1,\ldots,a_k) = (q,b_1,\ldots,b_k,H_1,\ldots,H_k)$ then M moves into state $(q,b_1,\ldots,b_k,k,H_1,\ldots,H_k)$. M starts scanning again from left to right and when it encounters an X in tape i it replaces the content of tape $i+1$ with b_i and moves the X in tape i one cell to the left or right depending on the value of H_i and moves to state $(q,b_1,\ldots,\underbrace{U}_{i},\ldots,b_k,k-1,H_1,\ldots,\underbrace{U}_{i},\ldots,H_k)$ it keeps doing this until $k=0$

and it enters the state $(q, \underbrace{U, \ldots, U}_{k\ times}, 0, \underbrace{U, \ldots, U}_{k\ times})$ and it has simulated one step of N.

12.4.3 Running Time

Definition 12.5. *We say that a Turing machine M has a running time $T(n)$ if for every input w of length n, M halts after making at most $T(n)$ moves. Note that the running time is $T(n)$ regardless of whether M accepts w or it does not accept it. If M does not halt then the running time is infinite.*

Since any multi-tape machine can be simulated by a single tape machine can we find a relationship between the running time of the two? This question is answered by the following theorem.

Theorem 12.1. *Simulating n moves of a multiple tape machine requires $O(n^2)$ steps on a single tape machine.*

Proof. From the previous section we know that a single tape has to make two passes over the input: one pass, from the left end marker to the last "X" to read all the markers and another pass, from right to the left end marker, to make the changes. Since we assume that initially all heads start at the left end marker then after n steps the heads cannot be more than n places away from the left end marker. This means that at step n the single tape machine needs to do at most n scans to the right and n scans to the left. Also for each tape if an X is encountered in the "writing" phase the machine might need to do 2 extra moves if the head is to be moved to the right to bring it back to its position. Thus we need to add $2k$ extra operations for a k-head machine so the total cost is $2 * n + 2 * k$ at step n. To simulate n steps therefore the single tape machine needs $O(n^2)$ operations because k is independent of n.

■

12.5 Representing TM with Strings

Any Turing machine with input in $\{0, 1\}^*$ can be represented as a binary string. Suppose $M = (Q, \Sigma, \Gamma, \delta, \rhd, \sqcup, r, s, t)$ is a Turing machine with $Q = \{q_1, \ldots, q_k\}$ the states of the TM where always q_1 is the starting state,q_2 is the accepting state and q_3 is the rejecting state. Also, $\Gamma = \{X_1, \ldots, X_l\}$ with $X_1 = \rhd$,$X_2 = \sqcup$,$X_3 = 0$ and $X_4 = 1$. Finally, $R = D_1$ and $L = D_2$ then a given transition $\delta(q_i, X_a) = (q_j, X_b, D_u)$ can be encoded as $0^i 10^a 10^j 10^b 10^u$

Example 12.6. *Let $M = (\{q_1, q_2, q_3, q_4, q_5\}, \{0, 1\}, \{\rhd, \sqcup, 0, 1\}, \delta, \rhd, \sqcup, q_1, q_2, q_3)$*

with

$\delta(q_1, 0) = (q_5, 0, R)$	$010^3 10^5 10^3 10$
$\delta(q_1, 1) = (q_4, 1, R)$	$010^4 10^4 10^4 10$
$\delta(q_4, 0) = (q_4, 0, R)$	$0^4 10^3 10^4 10^3 10$
$\delta(q_4, 1) = (q_3, 1, R)$	$0^4 10^4 10^3 10^4 10$ *reject*
$\delta(q_4, \sqcup) = (q_2, \sqcup, R)$	$0^4 10^2 10^2 10^2 10$ *accept*
$\delta(q_5, 0) = (q_3, 0, R)$	$0^5 10^3 10^3 10^3 10$ *reject*
$\delta(q_5, 1) = (q_5, 1, R)$	$0^5 10^4 10^5 10^4 10$
$\delta(q_5, \sqcup) = (q_2, \sqcup, R)$	$0^5 10^2 10^2 10^2 10$ *accept*

Since nowhere a sequence of 11 occurs the transition function can be encoded as

$$\underbrace{010^3 10^5 10^3 10}_{\delta(q_1,0)=(q_5,0,R)}\,\underline{11}\,\underbrace{010^4 10^4 10^4 10}_{\delta(q_1,1)=(q_4,1,R)}\,\underline{11}\,\underbrace{0^4 10^3 10^4 10^3 10}_{\delta(q_4,0)=(q_4,0,R)}\,\underline{11}\,\underbrace{0^4 10^4 10^3 10^4 10}_{\delta(q_4,1)=(q_3,1,R)}$$
$$\underline{11}\,\underbrace{0^4 10^2 10^2 10^2 10}_{\delta(q_4,\sqcup)=(q_2,\sqcup,R)}\,\underline{11}\,\underbrace{0^5 10^3 10^3 10^3 10}_{\delta(q_5,0)=(q_3,0,R)}\,\underline{11}\,\underbrace{0^5 10^4 10^5 10^4 10}_{\delta(q_5,1)=(q_5,1,R)}\,\underline{11}\,\underbrace{0^5 10^2 10^2 10^2 10}_{\delta(q_5,\sqcup)=(q_2,\sqcup,R)}$$

12.6 Universal Turing Machine

The universal Turing Machine U can simulate any machine M with input w. To do so we need an encoding for a TM configuration. Recall that a configuration consists of three values: the state, the content of the tape and the position of the head. All of these can be represented by a string of the form $0^p 10^r 10^{a_1} \ldots 10^{a_k}$ where 0^p is an encoding of state p, 0^r is encoding of the head position at r and the $0^{a_1} \ldots 0^{a_k}$ are the symbols on the tape and finally the symbol 1 is used as a separator. Since the patter 11 does not occur we can use it as a separator between different configurations. The UTM has 2 tapes

1. Tape 1 holds the code for M using the encoding of the previous sections.
2. Tape 2 holds configurations of M, again using the encoding of the previous sections.

Initially tape 2 has the configuration defined by the starting state and the input string.To simulate a move of M,

1. U reads the configuration of M from tape 2 which will be of the form $0^p 10^s 10^{k_1} 1 \ldots 10^{k_s} \ldots 10^{k_n}$.
2. U will scan tape 1 for a transition of the form $0^p 10^{k_s} 10^q 10^{k_t} 10^m$. This means move to a state represented by 0^q, replace the symbol 0^{k_s} by the symbol 0^{k_t} and move the head left or right depending on the value of m.

3. U will append the configuration $0^q10^{p\pm1}10^{k_1}1\ldots10^{k_t}\ldots10^{k_n}$ where $p\pm1$ depends on whether the head moved left or right. If this new configuration is accepting/rejecting then the UM accepts/rejects. Otherwise it becomes the current configuration and go to step (1).

12.7 Nondeterministic Turing Machine

Similar to the case of automata a Nondeterministic Turing machine can have, for a given input, more than one possible transition. Given a configuration $\triangleright a_1\ldots a_{k-1}pa_k\ldots a_n$ the result of the transition function is a set of m triplets: $\delta(p,a_k)=\{(q_1,b_1,D_1),\ldots,(q_m,b_m,D_m)\}$ where $D_i\in\{L,R\}$. An example is shown in Figure 12.7 the NTM that accepts the language $L=\{w\in\{a,b\}^*\ \mid\ w$ contains a c preceded or followed by $ab\}$

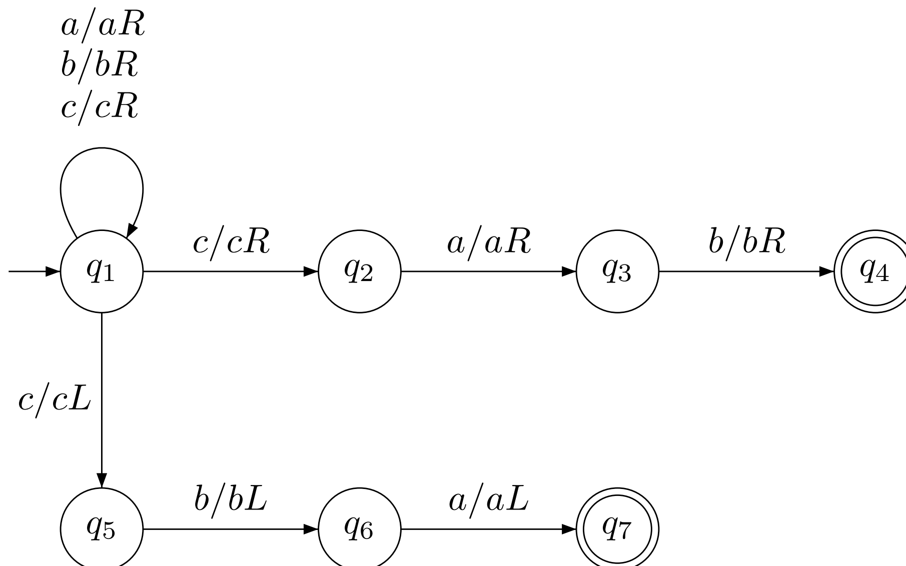

Figure 12.7: Non deterministic TM example

A more interesting example of a non-deterministic Turing machine (NTM) is given in the next example

Example 12.7. *Give a non-deterministic Turing machine (NTM) that decides the following language*

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

The above language is decided by the Non-deterministic Turing machine shown in Figure 12.8. The working of the solution NTM can be gleamed by observing how it functions on an example input, say $w=10101010$, which should belong to L. To

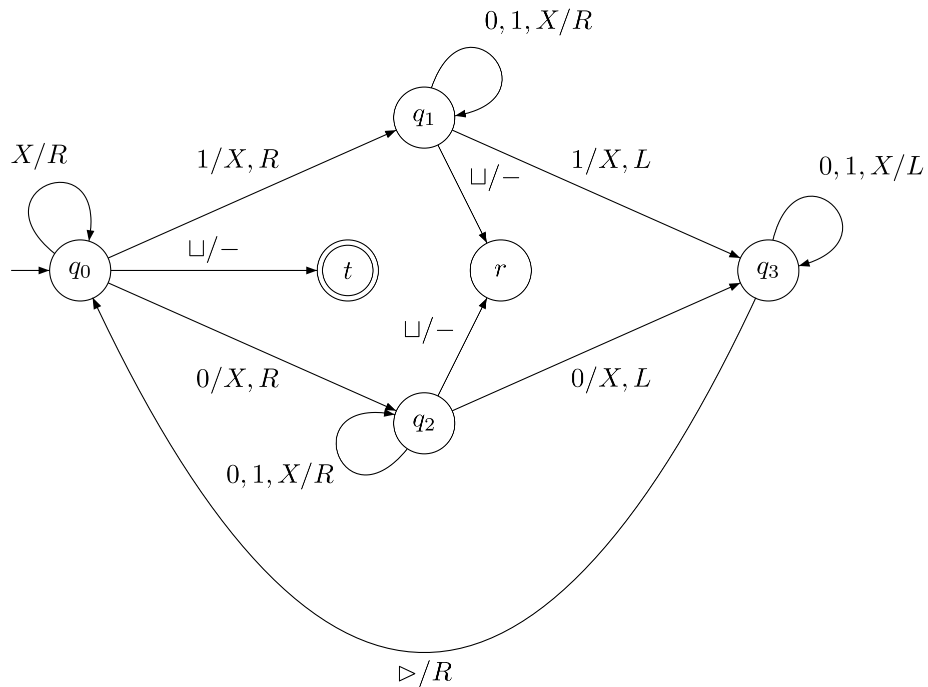

Figure 12.8: Non deterministic TM solution for example 12.7

illustrate the example we will be using the notation for TM configurations that we have established earlier. The initial configuration is

$$\triangleright q_0 10101010$$

As explained before, the above (initial) configuration means that the TM is initially in the starting state (q_0), the input to the TM is 101010 and the position of q_0 denotes the position of the head, in this case it is reading the first symbol, 1. Recall that the symbol $\vdash$ is used to denote the transition from one configuration to another. The first such transition in our example is

$$\triangleright q_0 10101010 \vdash \triangleright X q_1 0101010$$

The above transition reflects the $q_0 \overset{1/X,R}{\longrightarrow} q_1$ transition, because when in state q_0 and reading the symbol 1 the TM replaces the 1 with an X and moves to the right. Continuing with our computation we get the following

$$\triangleright X q_1 0101010 \vdash \triangleright X 0 q_1 101010$$

So far both steps were deterministic in the sense that the TM had only one possible transition. The next step illustrates the idea of a non-deterministic machine that always makes the **correct** guess. The TM is in configuration $\triangleright X 0 q_1 101010$ and by

inspecting Figure 12.8 we can see that the TM has two options: either it moves right or replaces the 1 by X and moves left. Since the machine can always makes the correct guess, it guesses that it should move right **without** replacing the 1 by X :

$$\triangleright X0q_1101010 \vdash X01q_101010$$

The remainder of the computation, with the TM always guessing right, is as follows

$$\begin{aligned}
&\triangleright X0q_1101010 \vdash X01q_101010 \vdash \triangleright X010q_11010 \triangleright X010Xq_3010 \\
&\vdash_{q\ 3} \triangleright X010X010 \vdash \triangleright q_0X010X010 \vdash Xq_0010X010 \vdash XXq_210X010 \\
&\vdash_X X10Xq_2010 \vdash \triangleright XX10XXq_310 \vdash_{q\ 3} \triangleright XX10XX10 \\
&\vdash \triangleright q_0XX10XX10 \vdash \triangleright XXq_010XX10 \vdash \triangleright XXXq_10XX10 \\
&\vdash_\triangleright XXX0XXq_110 \vdash \triangleright XXX0XXXq_30 \vdash_{q\ 3} \triangleright XXX0XXX0 \\
&\vdash \triangleright q_0XXX0XXX0 \vdash_\triangleright XXXXq_00XXX0 \vdash \triangleright XXXXq_2XXX0 \\
&\vdash_\triangleright XXXXXXXq_20 \vdash \triangleright XXXXXXXXq_3 \sqcup \vdash_{q\ 3} \triangleright XXXXXXXX \\
&\vdash \triangleright q_0XXXXXXXX \vdash_\triangleright XXXXXXXXq_0 \sqcup \vdash \triangleright XXXXXXXXtX
\end{aligned}$$

Therefore, the NTM by guessing right at every step accepts the input 10101010 which indeed belongs to L.

12.8 Simulating a non-deterministic Turing machine

Next we will show that the NTM is not more powerful than the DTM by showing that any NTM N can be simulated by a DTM D. Given a NTM N we can construct a DTM M with two tapes as we did for the universal Turing Machine. The only difference in this case is that a given configuration can have a transition to multiple configurations and an absence of a transition means a rejecting configuration. The first tape contains the description of N. The second tape contains a sequence of configurations of N. Initially the second tape contains the initial configuration ID_0. The DTM M does the following steps

1. If the current configuration is an accepting state for N then accept and halt.

2. If the current configuration is not accepting then the first tape is scanned for all matching transitions from the current configuration. If non exist stop and reject. If there are k possible transitions to configurations, say $ID_1 \ldots ID_k$ from the second tape and the possible transitions of N are read from the first tape.

3. The configurations $ID_1 \ldots ID_k$ are appended to the second tape and ID_1 becomes the current configuration.

4. Goto the first step.

AS an example we simulate the input 1010 on the NTM given in example ???

$\rhd q_0 1010 \star$
$\rhd q_0 1010 \bigstar \rhd X q_1 010$
$\rhd q_0 1010 \star \rhd X q_1 010 \bigstar \rhd X 0 q_1 10$
$\rhd q_0 1010 \star \rhd X q_1 010 \star \rhd X 0 q_1 10 \bigstar \rhd X 01 q_1 0 \star \rhd X q_3 0 X 0$
$\rhd q_0 1010 \star X q_1 010 \star \rhd X 0 q_1 10 \star \rhd X 01 q_1 0 \bigstar \rhd X q_3 0 X 0 \star \rhd X 010 q_1$
$\rhd q_0 1010 \star X q_1 010 \star \rhd X 0 q_1 10 \star \rhd X 01 q_1 0 \star \rhd X q_3 0 X 0 \bigstar \rhd X 010 q_1 \star \rhd q_3 X 0 X 0$

For the remainder of the computation we list only the "active" configurations

$$
\begin{array}{l}
\bigstar \rhd X010q_1 \star \rhd q_3 X0X0 \\
\bigstar \rhd q_3 X0X0 \star X010r \\
\bigstar X010r \star q_3 \rhd X0X0 \\
\bigstar q_3 \rhd X0X0 \\
\bigstar \rhd q_0 X0X0 \\
\bigstar \rhd X q_0 0X0 \\
\bigstar \rhd XX q_2 X0 \\
\bigstar \rhd XXX q_2 0 \\
\bigstar \rhd XXX0 q_2 \star \rhd XX q_3 XX \\
\bigstar \rhd XX q_3 XX \star \rhd XXX0r \\
\bigstar \rhd XXX0r \star \rhd X q_3 XXX \\
\bigstar \rhd X q_3 XXX \\
\bigstar \rhd q_3 XXXX \\
\bigstar q_3 \rhd XXXX \\
\bigstar \rhd q_0 XXXX \\
\bigstar \rhd X q_0 XXX \\
\bigstar \rhd XX q_0 XX \\
\bigstar \rhd XXX q_0 X \\
\bigstar \rhd XXXX q_0 \\
\bigstar \rhd XXXXt
\end{array}
$$

After showing the procedure of simulating a non-deterministic machine with a deterministic one we need to compute the number of steps needed to do the simulation.

Theorem 12.2. *If a non-deterministic machine N takes d steps to halt then a deterministic machine M that simulates N takes at most m^d where m is a constant independent of n.*

Proof. Let m be the maximum branching number, i.e. the largest number of possible transitions that a non-deterministic machine can have in any given state. Suppose that the non-deterministic machine accepts after d steps in some path of its computation. For the deterministic machine to be able to reach the accepting state of the non-deterministic one, it has to do, in the worst-case,
$m + m^2 + m^3 + \ldots + m^d = O(m^d)$ steps.

■

Lecture 13

Undecidability

13.1 Introduction

In this lecture we study what can and cannot be computed by Turing machines. It turns out that many (infinitely many) languages cannot be computed by a TM. We will consider three sets of languages: languages that can be decided by TM, languages that can be recognized by a TM and the languages that cannot be even recognized by a TM.

13.2 Decidable Languages

A language L is decidable if there exists a TM, M that halts on all input and answers yes(by accepting the input) or no (by rejecting the input).

Example 13.1. *Consider the language $A_{DFA} = \{Dw | w \in L(D)\}$ where D is a DFA and w is a string. The language A_{DFA} can be decided by the following 3-tape Turing Machine M. On input D and w place the starting state of D,q_0, on Tape 1, the string $w = w_1 \ldots w_n$ on tape 2 and the transitions of D on tape 3. Initially tape 1 contains the initial state of D: $\triangleright q_0$. Tape 2 contains the input w : $\triangleright w_1 w_2 \ldots w_n$. The transitions of D are placed on tape 3 as follows $\triangleright p_1, a, q_1 \star p_2, a, q_2 \star \ldots p_k, a, q_k$ where each triplet p_i, a, q_i denotes a transition $p_i \xrightarrow{a} q_i$ and $a \in \Sigma$. The TM works as follows*

1. *Suppose the current state stored on tape 1 is p and the position of tape two is on w_i then scan tape 3 for a triplet p, w_i, q, overwrite p with q and move tape 2 to the right (to consider w_{i+1} next).*

2. *When a blank is encountered in tape 2 halt. If the current state stored on tape 1 is accepting, accept. Otherwise reject.*

Since at every step the TM moves the head of tape 2 to the right then the TM will do at most n (the length of w) iterations and therefore it always halts after a finite number of steps. Also M accepts iff D accepts w.

Example 13.2. *Consider the language $E_{DFA} = \{D \text{ DFA} | L(D) = \emptyset\}$. A two-tape TM, M can decide E_{DFA} as follows. Suppose D has n states labeled $0 \ldots (n-1)$. Tape 1 of M contains an n bit number where bit at position i refers to DFA state i. It is set to 1 if state i is marked and 0 otherwise. Initially only state 0 (starting state) is marked therefore tape 1 contains $\triangleright 1 \underbrace{0 \ldots \ldots 0}_{n-1\ times}$. Tape 2 contains the transitions of D as done in the previous example. Machine M works as follows*

1. *Scan tape 2 for transitions of the for i, j. If bit i in tape 1 is set to 1 then mark state j by setting bit j to 1.*
2. *Repeat the above until no new state is marked.*
3. *If no accepting state is marked reject otherwise accept.*

The decider TM, M always halts in a finite number of steps. This is true because M repeats its operation if a new state is marked but there are at most $n-1$ states to mark therefore M will repeat at most $n-1$ times.

In the previous two examples we saw languages that can be decided by Turing machines. To study languages that cannot be decided by a Turing machine we use the diagonalization method discussed below as our starting point.

13.3 Diagonalization Method

We will use a technique called the Diagonalization Method first used by G. Cantor to prove important results about Turing Machines and languages. First we describe the method and show a few classic results.

Theorem 13.1. *There does not exist an onto function $f : N \to 2^N$.*

Proof. By contradiction. Assume that an onto function $f : N \to 2^N$ exists. Then for every $i \in N$, $f(i)$ is a subset of N. We can build a matrix where each element at index (i, j) is set to 1 if $j \in f(i)$ and set to 0 otherwise. The matrix below is an example:

	0	1	2	3	4	...
$f(0)$	1	0	1	0	1	...
$f(1)$	0	1	0	0	1	...
$f(2)$	1	1	0	1	0	...
$f(3)$	0	0	0	1	1	...
...	...	...	...	...	...	...

in the table above $f(0) = \{0, 2, 4, \ldots\}$ and $f(1) = \{1, 4, \ldots\}$. We can build a new set (an element of 2^N) that does not appear in any row in the above matrix. Consider the *diagonal* set of elements, in this example $\{1, 1, 0, 1, \ldots\}$, and construct a set which is the complement of this set, $D = \{0, 0, 1, 0, \ldots\}$. D does not appear in any row because it differs from *all* rows: it differs from row i by the element (i, i). To be more

specific we can give few examples to illustrate the method. In this example D does not contain the element 0, $f(0)$ contains 0 thus $D \neq f(0)$. D does not contain 1 and $f(1)$ contains 1 thus $D \neq f(1)$. D contains the element 2 and $f(2)$ does not contain 2 thus $D \neq f(2)$.

■

We can generalize the previous theorem to the following.

Theorem 13.2. *For any set A no onto function $f : A \to 2^A$ exists.*

Proof. Suppose that such a function exists and consider the set
$D = \{x \in A | x \notin f(x)\}$. Clearly $D \subseteq A$ or $D \in 2^A$. Since f is onto then there exist a y such that $f(y) = D$. is $y \in f(y)$?

$$\begin{aligned} y \in f(y) &\Leftrightarrow y \in D && \text{because } D = f(y) \\ &\Leftrightarrow y \notin f(y) && \text{definition of } D \end{aligned}$$

■

13.4 Universal and Diagonal Languages

Definition 13.1. *The universal language, L_{RE}, is defined as*
$L_{RE} = \{Mx | M \text{ accepts } x\}$ where M is a Turing machine and x is an arbitrary input.

Using the binary encoding of Turing machines discussed previously, we can enumerate them and we have an equivalent definition

Definition 13.2. *The universal language can be specified as*
$L_{RE} = \{(u, w) | u \text{ accepts } w\}$ or equivalently $L_{RE} = \{uw | w \in L(u)\}$. Where u is a binary string representing the encoding of M_u. Note that u and w in the definition of L_{RE} can take all possible values.

Note that if a string is not a valid representation of a TM we can assume that it describes a TM M such that $L(M) = \emptyset$. Central to our discussion is the *diagonal* language defined below.

Definition 13.3. *The diagonal language is defined as*
$L_D = \{ww | w \text{ does not accept } w\}$ or equivalently $L_D = \{ww | w \notin L(w)\}$.

Now we are ready for our first result in computability,namely that there exists a language that is not recognized by any Turing machine.

Theorem 13.3. *The language L_D is not recursively enumerable, no Turing machine M exists such that $L_D = L(M)$.*

Proof. Consider the language recognized by a given TM, u (i.e. u is fixed) $L_u = \{uv | v \in L(u)\}$. Then for *every possible* u we have $L_u \neq L_D$.

$$uu \in L_D \Rightarrow u \notin L(u) \Rightarrow uu \notin L_u$$
$$uu \notin L_D \Rightarrow u \in L(u) \Rightarrow uu \in L_u$$

It follows that no Turing machine u exists such that $L_D = L(u)$. Therefore L_D is not Turing recognizable (recursively enumerable). ■

Lemma 13.1. *A language is recursive if and only if its complement is recursive.*

Proof. Suppose that a language L is recursive. Then there exists a *total* TM, M that recognizes L. For any $x \in \{0,1\}^*$ M accepts and halts if $x \in L$ and M rejects and halts if $x \notin L$. Construct a *total* TM, N as follows

- On input x, N accepts and halts if M rejects x and halts. This implies that N accepts x and halts if $x \notin L$, i.e. $x \in \bar{L}$.
- On input x, N rejects and halts if M accepts x and halts. This implies that N rejects x and halts if $x \in L$, i.e $x \notin \bar{L}$.

The above two conditions lead to N decides $\bar{L}$ and therefore $\bar{L}$ is recursive. ■

Another important lemma is the following:

Lemma 13.2. *A language, L, is decidable iff it is RE and its complement, $\bar{R}$, is also RE.*

Proof. One direction is easy: if L is decidable then obviously it is RE and since by lemma 13.1 $\bar{L}$ is decidable thus $\bar{L}$ is also RE.
For the other direction: assume that both L and $\bar{L}$ are RE and they are recognized by M and N then construct TM R than has two tapes and works as follows:

- One tape is used to simulate M
- Second tape is used to simulate N
- R alternates between M and N.

Since any input x is either in L or $\bar{L}$ then for any input either M or N will accept at some point. If M accepts then accept and if N accepts then reject. ■

Theorem 13.4. *The language L_{RE} is undecidable.*

Proof. We assume, by way of contradiction, that L_{RE} is decidable. Then there exists a TM, R that decides L_{RE}. Construct the following TM, D: On input M, D runs R on MM

1. If R accepts MM then D rejects, i.e. $M \notin L(D)$

2. If R rejects MM then D accepts, i.e. $M \in L(D)$

What happens if the input to D is D itself? It runs R on DD

1. If R accepts then $D \in L_{RE}$ which means D accepts D. But by construction D rejects D, a contradiction.

2. If R rejects then $D \notin L_{RE}$ which means D does not accepts D. But by construction D accepts D, a contradiction.

Therefore our assumption that L_{RE} is decidable is false. ■

We have shown that L_{RE} is not decidable but is it Turing recognizable?. It is recognizable by a TM U that works as follows: On input M, w the TM U runs M on w

1. If M accepts U accepts

2. If M rejects U rejects

The third possibility, M loops on input w then U will not terminate.

13.5 Halting Problem

Theorem 13.5. *The language $L_{HP} = \{Mx | M$ halts on input $x\}$ is undecidable.*

Remark 1. *Before proving formally the undecidability of the halting problem it is helpful to get an intuitive feel why it is undecidable. One way to build an algorithm to decide if a machine M halts on input x is to simulate (i.e. run) M on x and wait for the result. If M halts on x our algorithm returns true. The problem is how can our algorithm decide that M does* ***not*** *halt? If our algorithm waits for, say , 10 minutes and M did not halt it* ***cannot*** *conclude that M does not halt. Maybe if we have waited a little longer it would've halted. This reasoning gives us also an intuitive understanding of semi-decidability, our algorithm works in "positive" case, when M halts on x but it does not work in the "negative" case, we cannot tell if a given M does not halt.*

We will show that the halting problem is undecidable by reducing it to L_{RE}. Assume that L_{HP} is decidable then there exists a total TM, R such that $L(R) = L_{HP}$. Now construct a TM, S as follows

1. Given an input Mx, run R on Mx. Since R is total then it always halts:

2. If R rejects then S rejects because this means M does not halt on x.

3. If R accepts then we are sure that M halts on x then run M on x:

    (a) If M accepts x then S accepts.

    (b) If M rejects x then S rejects.

Therefore we have build a total TM, S that decides L_{RE} which is a contradiction since we know that L_{RE} is not decidable.

13.6 Reduction

In the proof of theorem 13.4 we used a technique of reducing one problem to another. There is a general method called *reduction* or *mapping reduction* which allow us to prove the decidability/undecidability of a language by reducing the problem to another. Given two sets $L_1, L_2 \in \Sigma^*$ a map σ is called a reduction from L_1 to L_2, and we write $L_1 \leq_m L_2$ if

1. σ is computable by a *total* Turing machine.
2. $x \in L_1 \Leftrightarrow \sigma(x) \in L_2$.

Sometimes it is helpful to visualize the reduction using figure 13.1. Note that σ maps instance of L_1 into instances of L_2. The map does not have to be onto nor one-to-one. The notation $L_1 \leq_m L_2$ gives us a sense of the direction when we need to prove a language undecidable. Intuitively one can read $L_1 \leq_m L_2$ as L_2 is "harder" than L_1 so if L_1 is undecidable then L_2, being "harder" than L_1, has to be undecidable.

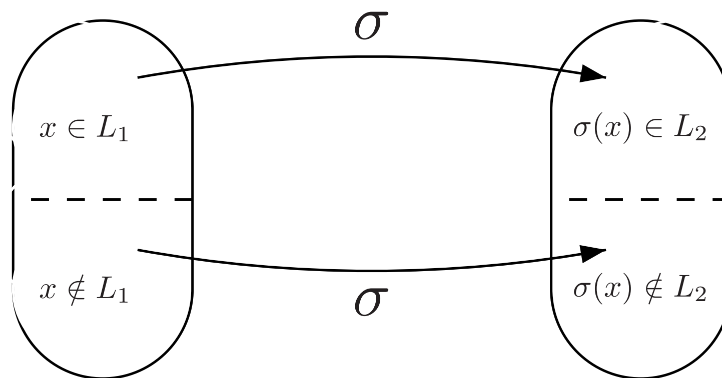

Figure 13.1: Reduction is a computable map σ

Theorem 13.6. *Suppose that there is a reduction from L_1 to L_2 then*

1. *If L_2 is recursively enumerable then L_1 is.*
2. *If L_2 is recursive then L_1 is.*

Proof. The proof is similar for both cases. Assume that L_2 is recursively enumerable then $\exists M$ (if it is recursive then M is total) such that $L_2 = L(M)$ which means if $x \in L_2$ then on input x M halts and accepts. Construct N such that it halts and accepts x if M halts and accepts $\sigma(x)$.

$$\begin{aligned} x \in L_1 &\Leftrightarrow \sigma(x) \in L_2 \\ &\Leftrightarrow \sigma(x) \in L(M) \\ &\Leftrightarrow x \in L(N) \end{aligned}$$

Example 13.3. *We will show that the halting problem is undecidable by using reduction.*

Our goal is to describe a computable map such that:

$$w \in L_{RE} \Leftrightarrow \sigma(w) \in L_{HP}$$

Therefore σ on input Mw should

1. If M accepts w convert it into $M'w' \in L_{HP}$
2. If M rejects w convert it into $M'w' \notin L_{HP}$
3. If M loops on w convert it into $M'w' \notin L_{HP}$

To accomplish the above, given Mw convert it to $M'w$, i.e. $w = w'$ and M' works as follows: run M on w

1. If M accepts then accept.
2. If M rejects then loop.

The third case is if M loops on w then M' will also loop on w. It is important to note that σ is computable (always terminates) even if M loops on w. What σ is doing is to build M' with M as a "subroutine" and this procedure terminates. The above is a reduction from L_{RE} to L_H but since L_{RE} is not decidable then L_{HP} is undecidable.

Example 13.4. *We will use reduction to prove that the language $L_{FIN} = \{M \mid L(M) \text{ is finite}\}$ is undecidable. We reduce the complement of L_{RE} to it: $\neg L_{RE} = \{Mw \mid M \text{ does not accept } w\}$ (actually $\neg L_{RE}$ is not RE)*

The computable function σ converts Mw into a $M'w'$ such that on input w', M' works as follows: M' runs M on w (disregarding w'):

- If M accepts w then M' accepts.
- If M rejects w then M' rejects.

This means that if $Mw \in \neg L_{RE}$, i.e. M rejects w or loops on w then $L(M') = \emptyset$ (finite) because regardless of w', M does not accept w(rejects or loops) and therefore w' is not accepted by M'. And if $Mw \notin \neg L_{RE}$ then M accepts w and M' accepts ALL input, i.e. $L(M') = \Sigma^*$ (infinite). The map σ is illustrated in Figure 13.2.

Example 13.5. *Let $L_\emptyset = \{M \mid L(M) = \emptyset\}$. We use reduction to show that the complement of $L_\emptyset$ is undecidable. Then by lemma 13.1 we conclude that $L_\emptyset$ is also undecidable.*

We reduce L_{RE} to $\neg L_\emptyset$. Given Mw then construct M' that on input w' works as follows:

- Run M on w

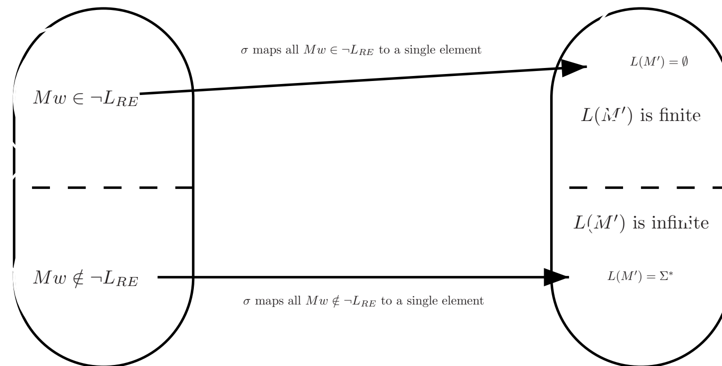

Figure 13.2: Reduction from $\neg L_{RE}$ to L_{FIN}

  - If M accepts accept regardless of w'
  - If M rejects reject regardless of w'

Obviously there is the case where M does not halt on w, then M' will not halt and does not accept any w'. The above means that if M accepts w then M' accepts ALL input, i.e. $L(M') = \Sigma^*$ and therefore $M' \in \neg L_\emptyset$. And if M does not accept w then M' doesn't accept anything, i.e. $L(M') = \emptyset$ and therefore $M' \notin \neg L_\emptyset$. We have shown that $\neg L_\emptyset$ is not decidable and therefore by lemma 13.1 $L_\emptyset$ is not decidable.

Example 13.6. *Let $L_R = \{M \mid L(M) \text{ is regular}\}$. L_R is undecidable.*

It is important to recall that the reduction map does not need to be one-to-one nor onto. We only require from a reduction σ from A to be B that $w \in A \Leftrightarrow \sigma(w) \in B$. In this example we provide a reduction σ that maps "M accepts w" to $M'w'$ where M' accepts a regular language, namely Σ^*, and maps "M does not accept w" to a non-regular language 0^n1^n. It works as follows:
Given M and w construct M' such that on input w'

- If w' is of the form 0^n1^n accept (remember that we already build a TM that can decided if the input is of that form).
- Otherwise run M on w
  - If M accepts w then accept all w'
  - IF M rejects w then reject all w'

Note that in the above reduction M' accepts input of the form 0^n1^n only, i.e $L(M')$ is non-regular, if M does not accept w. And M' accepts all input, i.e. $L(M') = \Sigma^*$ regular, if M accepts w.

Example 13.7. *Let $L_{EQ} = \{M_1M_2 \mid L(M_1) = L(M_2)\}$. L_{EQ} is undecidable.*

We reduce $L_\emptyset$ to L_{EQ} as follows: given M construct $M' = MM_\emptyset$ where $M_\emptyset$ is a TM that rejects all input. To show that it is a correct reduction:

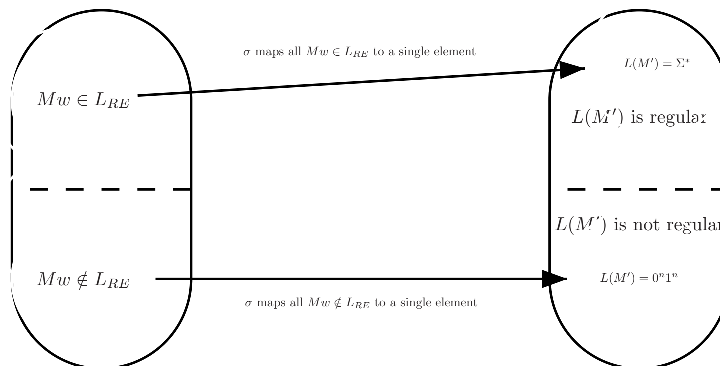

Figure 13.3: Reduction from L_{RE} to L_{REG}

- If $M \in L_\emptyset \Rightarrow L(M) = \emptyset = L(M_\emptyset)$ thus $M' = MM_\emptyset \in L_{EQ}$.
- Conversely if $M \notin L_\emptyset \Rightarrow L(M) \neq \emptyset \neq L(M_\emptyset)$ thus $M' = MM_\emptyset \notin L_{EQ}$.

Example 13.8. $\neg HP = \{Mx \mid M$ *does not halt on* $x\}$ *is not RE*

We reduce $L_D = \{MM \mid M$ does not accept $M\}$, which we know is not RE to it.
Given M construct M' on input x

- Run M on x and accept if M accepts.
- loop if M rejects.

Our reduction map on input M produces $M'M$. If $M \in L_D$ then M does not accept M then M' loops on M and therefore $M'M \in \neg HP$. Conversely, if $M \notin L_D$ then M accepts M and M' accepts (and halts) on M therefore $M'M \notin \neg HP$.
We will use the above result to show that

Example 13.9. $L_{INF} = \{M \mid L(M)$ *is infinite* $\}$ *is not RE.*
It is sufficient to reduce $\neg HP$ *to* L_{INF}*. We use the fact that* $L_{INF} = \neg L_{FIN}$ *and if* $HP \leq_m L_{FIN}$ *then* $\neg HP \leq_m \neg L_{FIN}$ *. Therefore it is sufficient to show that* HP *reduces to* L_{FIN}*. Given* Mx *construct* M' *which in input* y *works as follows:*

- *Run* M *on* x *for* $|y|$ *steps*
  - *If* M *halts reject*
  - *if* M *does not halt in* $|y|$ *steps accepts.*

If M *does not halt on* x *then it will not halt for* $|y|$ *steps for any* y *thus* $L(M') = \Sigma^*$ *which is infinite. If* M *halts on* x *say in* n *steps then* M' *accepts all strings* y *such that* $|y| < n$ *which is finite.*

Example 13.10. *Consider the language* $L_{TOTAL} = \{M|M$ *halts on all input* $\}$*. We show that* L_{TOTAL} *is not RE by showing that* $\neg HP \leq L_{TOTAL}$*.*
Given Mw *construct* R *such that on input* x *works as follows:*

- *Run M on w for $|x|$ steps*

1. *If M halts on w within $|x|$ steps then R loops*
2. *If M does not halt on w within $|x|$ steps then R halts*

The result of the reduction

1. *If M does not halt on w i.e. $Mw \in \neg HP$ then R halts on all input x.*
2. *If M halts on w after s steps then*
   (a) *For all x such $|x| > s$ then R loops*
   (b) *For all x such that $|x| \leq s$ then R halts*

The consequence is that if M does not halt on w then R halts on all input x and if M halts on w then R does not halt on some input. Therefore $\neg HP \leq L_{TOTAL}$.

Remark 2. *If L_{TOTAL} was decidable (we know it is not even RE) then we could solve some interesting problems, for example, the Collatz conjecture. Let M_c be a Turing machine that reads its input $w \in \{0,1\}^*$, considered as integer, computes $w/2$ if w is even, computes $3w+1$ if w is odd. M_c keeps doing the computation until the value on its tape is 1 then it stops. The Collatz conjecture claims that for any input w, M_c will stop (i.e. halts on all input). If L_{TOTAL} were decidable then $\exists M$ that decides L_{TOTAL} and therefore decides if $M_c \in L_{TOTAL}$ and the conjecture can be solved. Note that L_{TOTAL} being undecidable does NOT mean the conjecture is true or false.*

13.7 Rice's Theorem

Theorem 13.7 (Rice)**.** *Any non-trivial property of RE sets is undecidable.*

First recall that a set (language) is RE if it is recognized by some Turing Machine. But the property is for languages not Turing Machines, for example the set of all TM such that *their language* is finite or infinite. Non trivial means there are at least one TM whose language has the property and at least one that does not have the property.

Proof. We provide a reduction from the halting problem. Assume that $L_\emptyset = \emptyset$ does not have the property. Since it is non-trivial then $\exists R$ such that $L(R)$ has the property. Given Mx construct M_x on input y

- If M halts on x run R on y and accept if R accepts

From the above reduction we see that

- If M does not halt on x then $L(M_x) = \emptyset$, i.e. does not have the property
- If M halts on x then $L(M_x) = L(R)$, i.e. has the property.

As a final note Rice's theorem is about all sets defined as

$$L = \{M | L(M) \text{ has property } P\}$$

■

13.8 RE Completeness

Let B be a language such that for all RE languages A we have $A \leq_m B$ then B is said to be re-hard. If B is also RE then B is said to be RE complete.

Theorem 13.8. *L_{RE} is RE complete.*

Proof. We have already shown that L_{RE} is RE so we still need to show that L_{RE} is re-hard, for every RE language A, $A \leq_m L_{RE}$. Since A is RE then $\exists R$ such that $L(R) = A$. The reduction is as follows:

- Given $x \in \Sigma^*$, $\sigma(x) = Rx$.
- If $x \in A$ then R accepts x and thus $Rx \in L_{RE}$.
- IF $x \notin A$ then R does not accept x and thus $Rx \notin L_{RE}$.

∎

 ©Hikmat Farhat

Lecture 14

Post Correspondence Problem

14.1 Introduction

In this section we introduce a word matching problem called the Post Correspondence Problem (PCP) formulate by Emile Post in 1946. An instance of the PCP is a finite set of pairs of strings over some alphabet. Each pair can be visualized as a ”domino” with a string on top and another string at the bottom. For example if the alphabet is $\Sigma = \{a, b\}$ one possible pair (domino) is shown below

$$\left[\frac{abb}{ba}\right]$$

An example of a set of such pairs looks like:

$$\left\{\left[\frac{ab}{a}\right], \left[\frac{aba}{ab}\right], \left[\frac{a}{baba}\right]\right\}$$

The problem can be described as follows: given a set of pairs, can we list them (repetition allowed) in some order such that the resulting string on top is the same as the string at the bottom? Below is a instance of the PCP problem.

i	t_i	b_i
1	ab	a
2	aba	ab
3	a	baba

Table 14.1: An instance of PCP

One possible solution is:

$$\left[\frac{ab}{a}\right] \cdot \left[\frac{aba}{baba}\right]$$

In the above the resulting string on the top, $ababa$, is the same as the bottom string. Another possible solution is:

$$\left[\frac{aba}{ab}\right] \cdot \left[\frac{ab}{a}\right] \cdot \left[\frac{ab}{a}\right] \cdot \left[\frac{a}{baba}\right]$$

Where the resulting top string is $abaababa$ which matches the bottom string. Not all PCP instances have a solution. For example, we will show that the instance below has no solution.

i	t_i	b_i
1	ab	aba
2	baa	aa
3	aba	baa

Table 14.2: An instance of PCP that has no solution

If there were a solution it has to start with with the first pairs since all the others have a mismatch in first symbol. Now we look for the second item. The only possibility is pair 3 since the other choices lead to

$$\left[\frac{ab}{aba}\right] \cdot \left[\frac{ab}{aba}\right] \ldots \text{ if we choose item 1}$$
$$\left[\frac{ab}{aba}\right] \cdot \left[\frac{baa}{aa}\right] \ldots \text{ if we choose item 2}$$

Having decided the second item the partial solution looks like

$$\left[\frac{ab}{aba}\right] \cdot \left[\frac{aba}{baa}\right] = \left[\frac{ababa}{ababaa}\right]$$

There is a partial match but the string in the bottom is longer by one symbol. Again our only choice is item 3 which means the bottom string will have an extra a and we go back to where we started.

14.2 Modified PCP is Undecidable

If we force the solution to the PCP to always start from a particular ”domino” then we obtain the modified PCP (MPCP) problem. In this section we show how to construct from a Turing Machine M and input w a instance of the MPCP problem. Given a TM $M =< Q, \Sigma, \Gamma, \delta, \triangleright, \sqcup, r, s, t >$ and an input string $w = w_1 \ldots w_n$, we can construct an instance of MPCP, by adding dominos, as follows. First we add the ”separator” domino:

$$\left[\frac{\#}{\#}\right]$$

Then we add the domino representing the initial input:

$$\left[\frac{\#}{\#w_1\dots w_n\#}\right]$$

Also for every $a \in \Gamma$ we add to the set of dominos

$$\left[\frac{a}{a}\right]$$

Next we handle the transitions :

$$\forall a,b \in \Gamma \;\; p,q \in Q \text{ if } \delta(p,a) = (q,b,R) \text{ then add } \left[\frac{pa}{bq}\right]$$

$$\forall a,b,c \in \Gamma,\; p,q \in Q \text{ if } \delta(p,a) = (q,b,L) \text{ then add } \left[\frac{cpa}{qcb}\right] \text{ for every } c \in \Gamma$$

For the accepting state t we add the following :

$$\forall a \in \Gamma \text{ add } \left[\frac{at}{t}\right], \left[\frac{ta}{t}\right]$$

and

$$\left[\frac{t\#\#}{\#}\right]$$

Note that in the construction above all the dominos, except the first domino and the ones containing the accepting state, the number of symbols at the top is equal to the one at the bottom.

Example 14.1. *Consider the following TM*
$M =< Q = \{p,q\}, \Sigma = \{0,1\}, \Gamma = \{0,1,\sqcup\}, \delta, r, s, t >$ *with the input* $w = 001$ *and the following transitions:*

$$\delta(p,0) = (p,1,R)$$
$$\delta(p,1) = (q,0,L)$$
$$\delta(q,1) = (t,1,R)$$

Following the procedure described above, the first transition results in the addition of

$$\left[\frac{p0}{1p}\right]$$

The second transition results in the addition of

$$\left[\frac{0p1}{q00}\right], \left[\frac{1p1}{q10}\right], \left[\frac{\sqcup p1}{q \sqcup 0}\right]$$

Finally the third transition results in the addition of

$$\left[\frac{q1}{1t}\right]$$

Since the TM does not write blanks we will omit all blank symbols in what follows. The MPCP instance becomes

$$\left\{\left[\frac{\#}{\#}\right],\left[\frac{\#}{\#p001\#}\right],\left[\frac{0}{0}\right],\left[\frac{1}{1}\right],\left[\frac{p0}{1p}\right],\left[\frac{0p1}{q00}\right],\left[\frac{1p1}{q10}\right],\left[\frac{q1}{1t}\right],\left[\frac{t\#\#}{\#}\right],\left[\frac{t0}{t}\right],\left[\frac{0t}{t}\right],\left[\frac{t1}{t}\right],\left[\frac{1t}{t}\right]\right\}$$

The above MPCP instance has the following solution:

$$\left[\frac{\#}{\#p001\#}\right]\cdot\left[\frac{p0}{1p}\right]\cdot\left[\frac{0}{0}\right]\left[\frac{1}{1}\right]\cdot\left[\frac{\#}{\#}\right]\cdot\left[\frac{1}{1}\right]\left[\frac{p0}{1p}\right]\left[\frac{1}{1}\right]\left[\frac{\#}{\#}\right]\left[\frac{1}{1}\right]\left[\frac{1p1}{q10}\right]\left[\frac{\#}{\#}\right]\left[\frac{1}{1}\right]\left[\frac{q1}{1t}\right]\left[\frac{0}{0}\right]\left[\frac{\#}{\#}\right]$$

we continue with

$$\left[\frac{1}{1}\right]\left[\frac{1t}{t}\right]\left[\frac{0}{0}\right]\left[\frac{\#}{\#}\right]\left[\frac{1t}{t}\right]\left[\frac{0}{0}\right]\left[\frac{\#}{\#}\right]\left[\frac{t0}{t}\right]\left[\frac{\#}{\#}\right]\left[\frac{t\#\#}{\#}\right]$$

Not only is the above a solution for the MPCP but if one looks at the bottom string we can see that it is a sequence of legal moves of the TM leading to an accepting state. Furthermore, since in all dominos, except the ones involving the symbol t, the upper string is equal to the bottom string, therefore when we start with the firs domino that has the bottom string longer that the top there is no way a solution to the MPCP can be obtained unless we involve the ones containing the symbol t.

The conclusion we have reached in the above example is not particular to that example. In fact we have shown how to convert any Mw instance of a TM, M and input w into an instance of the MPCP problem where the MPCP has a solution iff M accepts w.

Theorem 14.1. *The MPCP is undecidable*

Proof. We have shown that any pair Mw can be converted into MPCP instance where the MPCP instance has a solution iff M accepts w. Suppose that MPCP is decided by a TM S then for for any Mw we convert it into an instance I of the MPCP and run S on I:

1. If S accepts then we accept
2. If S rejects then we reject

Since S is a decider then we have constructed a decider for L_{RE} which is a contradiction because we know L_{RE} is undecidable. ■

Lecture 15

Complexity

15.1 Class P

As an example we give a multitape Turing machine that, given a graph G and two nodes k and l, accepts if there is a path from s to t and rejects otherwise. The nodes of the graph are labeled by numbers, the first node is labeled 0, the second 1,etc. If the graph has N nodes that we need $\lceil \log N \rceil$ bits to represent them. The TM has five tapes T_1, T_2, T_3, T_4, T_5. Initially tape T_2 contains an N bits code representing node k (NOT the value k). This code is all zeroes except the kth bit is set to one. Similarly, tape T_4 contains the code for l.Tape T_3 contains all the edges of G coded as e, f where e is the eth node connected to the fth edge, separated by a comma. The edge representations are separated by a $\star$. Finally, tapes T_1 and T_5 are scratch tapes that will be used during the computation. To make things clear the graph below in Figure 15.1 where the test is the connectivity of node 1 and node 4 will be represented as:
T_2 : $\triangleright$010000
T_4 : $\triangleright$000010
T_3 : $\triangleright 000, 001 \star 001, 010 \star 001, 101 \star 101, 011 \star 101, 100$

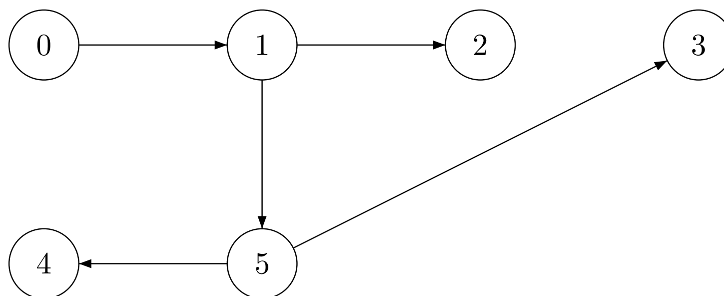

Figure 15.1: Example graph connectivity problem

The algorithm works by doing multiple passes over the edges in tape T_3. When we

encounter a link (i, j) then the head on T_2 is positioned on bit i, if it is 0 then j is skipped and we consider the next edge. If i is 1 then the j bit in T_3 is set to 1, the value in T_5 is set to one meaning that a change has occurred. If $j = k$ the machine stops and accepts, if not then the machine considers the next edge. If there are no more edges and $T_5 = 0$ then there was no change since the last pass and the machine stops and rejects, if $T = 1$ then it is set to 0 and the head of tape T_3 repositioned on the beginning of T_3 and the computation continues.
To implement the above algorithm we make use of two TMs we have introduced in last chapter. The first is the copy "routine" that copies values of i and j from tape T_3 to tape T_1. The copy routine is shown below in Figure 15.2. The question marks on some states means that those states are placeholders to be connected to a larger machine as a subroutine.

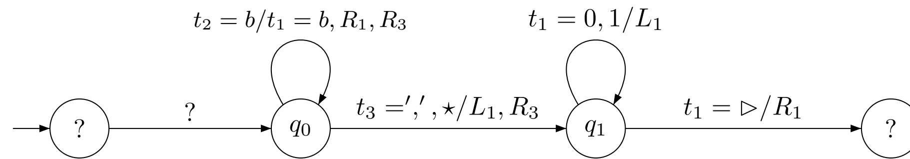

Figure 15.2: Copies a value from tape T_3 to tape T_1

and the second routine from last chapter is the one that moves the head of tape T_2, to the kth position where k is the value stored in T_1. This operation is done step by step as follows: the value in T_1 is decremented by one and simultaneously the head on T_2 is moved one position to the to the right. The TM keeps doing this operation until the value in T_1 reaches zero. This subroutine is shown in Figure 15.3. The meaning of the question marks is the same as above.
Making use of the above two algorithms we obtain the following TM that tests whether two nodes are connected in a graph.

15.2 Class NP

Class NP is defined as all the languages that can be decided by a NTM in polynomial time. Example
HAMPATH={$(G, s, t)|$ a graph G has a Hamiltonian path from s to t }.
An algorithm that decides HAMPATH works as follows. Let n be the number of nodes in the graph. Each node is encoded with n bits. Node i has all bits zeros except the ith bit is set to one. One could use $\lceil \log n \rceil$ to represent each node but it is easier to use n bits and the polynomial time will be preserved.The NTM has three tapes.

1. Non-deterministically write n values from 1 to n on tape 1. The values are separated by commas ','.

2. Check that there are no duplicates in the n values in tape 1. If there is stop and reject because in a Hamiltonian path each of the n nodes occurs exactly once. Operation in this step is described by the routine check below.

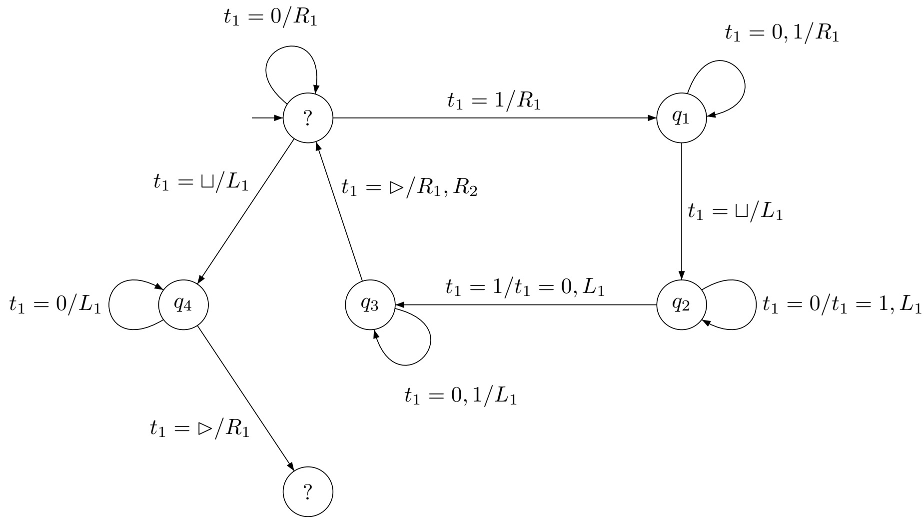

Figure 15.3: Position the head of T_2 at the kth position, where k is the value stored in tape T_1

3. In this step we are sure that each node appears on tape 1 exactly once. This is because there are n values with no duplicates. Next test that every pair of consecutive nodes i, j appears in the list of edges in tape 3. This is done in routine IsEdge described below.

Routine check. Tape two has initially n zeros. Scan a value in tape 1 while scanning tape 2. When a 1 is encountered in tape 1 check the corresponding bit in tape 2, if that bit is 1 it means the node has been visited before, then stop and reject. If the bit on tape 2 is a zero, set it to 1. Rewind tape 2 and scan the next value in tape 1. Keep doing this until there are no more values in tape 1. If no duplicates are detected go to the next step.
Routine IsEdge. Scan, simultaneously i on tape 1 and i' on tape 3. There are two cases

Case 1 $i \neq i'$. Then rewind i and skip j'. Note that this means rewind tape 1 n bits to the left and skip n bits on tape 3 to the right. Then scan again.

Case 2 $i = i'$.

> **Case 2a** $j = j'$. Go to the next value in tape 1, i.e. j this means rewind tape 1 n bits to the left. Also rewind tape 3 to the beginning this means $2n \times m$ steps.
>
> **Case 2b** $j \neq j'$. Rewind tape to the left $2n$ bits to reread the pair i, j.

Definition 15.1. *A* ***verifier*** *for a language L is an algorithm V such that for $w \in \Sigma^*$, $\exists c$ such that on input (w, c) the verifier V accepts iff $w \in L$.*

If $|c| = O(|w|^n)$ for some n, then V is said to be a **polynomial time verifier**.

Lecture 16

Lambda Calculus

Lambda calculus was invented by Alonzo Church to study the concept of computability. The formalism is Turing Complete in a sense that any function that is computable by a Turing machine is also computable by λ-calculus. the syntax of λ-calculus is simple and at the heart of it is the "term", notation M. Let x denote variables then a λ-calculus term can be

$$M := x|\lambda x.M|MM$$

To avoid ambiguity we follow two rules

1. application associate left so $xyz = (xy)z$
2. the body of a function extends all the way to the right so $\lambda x.xyz = \lambda x.(xyz)$

Few examples are in order. suppose that the multiplication operation is defined.

$$F \stackrel{\text{def}}{=} \lambda x. * 2x$$

what is the results if we apply it to $F3$ gives the output 6. How about $F(F3)$? this will give 12. Conventions

16.1 Free and Bound Variables

given $\lambda x.M$, if x occurs in M it is called a *bound* variable otherwise it is called *free*. For example $M \equiv (\lambda x.xy)(\lambda y.yz)$ x is a bound variable, z is a free variable and y has a free (the first) occurrence and (the second) a bound one. Bound variables are like dummy variables in other programming languages therefore $\lambda x.x \equiv \lambda y.y$ and the two expressions are called α-equivalent.

16.2 Substitution

an important operation on λ-calculus is substitution where a variable is substituted by a λ-term. Given two terms M and N we write $M[N/x]$ where N replaces x in M. On

many occasions we would like to use an α-equivalent terms in order not to confuse things. for example consider the expression

$$M \equiv \lambda x.yx$$
$$N \equiv \lambda z.xz$$

doing the substitution naively will not work $M[N/y] = \lambda x.(\lambda z.xz)x$ makes the x that was free in N bound. In this case we rename the bound variable before the substitution $M[N/y] = \lambda u.(\lambda z.xz)u$
Also substitution affects only free variables for example $(\lambda y.yx(\lambda x.xz))[N/x]$ will reduce to $\lambda y.yN(\lambda x.xz)$ and not to $(\lambda y.yN(\lambda x.Nz))$

16.3 Reductions

computation in λ-calculus is done through β-reductions. A term of the form $(\lambda x.M)N$ is called a β-redex and it reduces to $M[N/x]$. A λ term without a redex is said to be in normal form. example

$$\begin{aligned}(\lambda x.xy)((\lambda z.zz)(\lambda u.u)) &\rightarrow_\beta (\lambda x.xy)((\lambda u.u)(\lambda u.u)) \\ &\rightarrow_\beta (\lambda x.xy)(\lambda u.u) \\ &\rightarrow_\beta (\lambda u.u)y \\ &\rightarrow_\beta y\end{aligned}$$

16.4 Programming

16.4.1 Booleans

The truth values True and False are encoded as λ-terms.

$$\mathbf{T} = \lambda xy.x$$
$$\mathbf{F} = \lambda xy.y$$

to see that the above have the correct representation define the **AND** function as $AND \stackrel{\text{def}}{=} \lambda uv.uv(\lambda ab.b)$ then

$$\begin{aligned}\mathbf{AND\ TT} &= (\lambda uv.uv(\lambda ab.b))(\lambda xy.x)(\lambda zw.z) \\ &\rightarrow_\beta (\lambda v.(\lambda xy.x)v(\lambda ab.b))(\lambda zw.z) \\ &\rightarrow_\beta (\lambda xy.x)(\lambda zw.z)(\lambda ab.b) \\ &\rightarrow_\beta (\lambda y.(\lambda zw.z))(\lambda ab.b) \\ &\rightarrow_\beta (\lambda zw.z) = \mathbf{T}\end{aligned}$$

one might be tempted that the last term (i.e. $\lambda ab.b$) in the definition has not effect but check this

$$\begin{aligned}\textbf{AND FT} &= (\lambda uv.uv(\lambda ab.b))(\lambda xy.y)(\lambda zw.z)\\ &\to_\beta (\lambda v.(\lambda xy.y)v(\lambda ab.b))(\lambda zw.z)\\ &\to_\beta (\lambda xy.y)(\lambda zw.z)(\lambda ab.b)\\ &\to_\beta (\lambda y.y)(\lambda ab.b)\\ &\to_\beta (\lambda ab.b) = \mathbf{F}\end{aligned}$$

We can use the above to define the usual if-then-else construct, $\mathbf{IFE} = (\lambda x.x)$. then we have

$$\begin{aligned}\textbf{IFE T M N} &= (\lambda x.x)(\lambda uv.u)\mathbf{M}\,\mathbf{N}\\ &\to_\beta (\lambda uv.u)\mathbf{M}\,\mathbf{N}\\ &\to_\beta (\lambda v.\mathbf{M})\mathbf{N}\\ &\to_\beta \mathbf{M}\end{aligned}$$

16.4.2 Natural Numbers

Natural numbers are represented by Church's numerals

$$\begin{aligned}\bar{0} &= \lambda fx.x\\ \bar{1} &= \lambda fx.fx\\ \bar{2} &= \lambda fx.f(fx)\\ \bar{3} &= \lambda fx.f(f(fx))\end{aligned}$$

we define the successor function as $\mathbf{succ} = \lambda nfx.f(nfx)$ then

$$\begin{aligned}\mathbf{succ}\,\overline{n} &= (\lambda nfx.f(nfx))(\lambda gy.g^n y)\\ &\to_\beta \lambda fx.f((\lambda gy.g^n y)fx)\\ &\to_\beta \lambda fx.f((\lambda y.f^n y)x)\\ &\to_\beta \lambda fx.f(f^n x)\\ &\to_\beta \lambda fx.f^{n+1}x\\ &= \overline{n+1}\end{aligned}$$

how about addition? define $\mathbf{add} = \lambda mnfx.mf(nfx)$ then use it to add 2 and 3

$$
\begin{aligned}
\textbf{add 2 3} &= (\lambda mnfx.mf(nfx))(\lambda gy.g(gy))(\lambda hz.h(h(hz))) \\
&\rightarrow_\beta (\lambda nfx.(\lambda gy.g(gy))f(nfx))(\lambda hz.h(h(hz))) \\
&\rightarrow_\beta (\lambda nfx.(\lambda y.f(fy))(nfx))(\lambda hz.h(h(hz))) \\
&\rightarrow_\beta (\lambda nfx.f(f(nfx)))(\lambda hz.h(h(hz))) \\
&\rightarrow_\beta (\lambda fx.f(f((\lambda hz.h(h(hz)))fx))) \\
&\rightarrow_\beta (\lambda fx.f(f((\lambda z.f(f(fz)))x))) \\
&\rightarrow_\beta \lambda fx.f(f((f(f(fx))))) \\
&\rightarrow_\beta \lambda fx.f^5x = \overline{5}
\end{aligned}
$$

multiplication can be defined as $mult = \lambda nmf.n(mf)$.

16.4.3 Recursion

Let M and N be λ-terms. We say that N is a fixpoint of M if $MN = N$. In untyped lambda calculus *every* term has a fixpoint

untyped. Let $Y = \lambda xy.y(xxy)$. For any term M, YYM is a fixpoint of M. in other words $M(YYM) = (YYM)$. This follows from

$$
\begin{aligned}
YYM &= (\lambda xy.y(xxy))YM \\
&\rightarrow_\beta (\lambda y.y(YYy))M \\
&\rightarrow_\beta M(YYM)
\end{aligned}
$$

The term YY is sometimes called the Turing fixpoint combinator. ■

This will help us define recursive functions like factorial.

$$
\textbf{fact}\ n = \textbf{IFE}(\textbf{iszero}\ n)(\overline{1})(\textbf{mult}\ n(\textbf{fact}(\textbf{pred}\ n))
$$

the definition of **fact** contains itself on the right hand side.