1 Introduction

As computer scientists we are familiar with the notion of *abstraction* at various levels. In particular, programming languages provide abstractions both of the physical processes of computation at the hardware level, and of the actual data and algorithms that they represent. In the theory of computing we want to be able to discuss and reason about computing itself without reference to any specific language so yet another level of abstraction is required.

Fortunately, mathematics provides a ready built set of abstract concepts that are ideal for this sort of reasoning. In the first few lectures of COSC 341 we will (re)introduce these concepts. We *are not* going to need a large body of mathematical knowledge – but we do need its notation, and the basic concepts of logic that allow us to write proofs.

2 Sets

A *set*, X, is just a collection of objects. It divides the universe into two parts: those things that belong to, or *are elements of* X, and those which do not. This is all it does so the elements of a set can also be thought of as an unordered collection without duplication. The easiest way to specify a set is by listing its elements explicitly:

$$\begin{aligned} X &= \{1, 2, 5\} \\ Y &= \{a, b, c, d, e\} \end{aligned}$$

The belonging relationship is denoted by the symbol $\in$. So:

$$1 \in X,\ 2 \in X,\ 3 \notin X,\ 4 \notin X,\ 5 \in X,\ \ldots$$

Two sets are equal if (and only if) they have exactly the same collection of elements. That is:

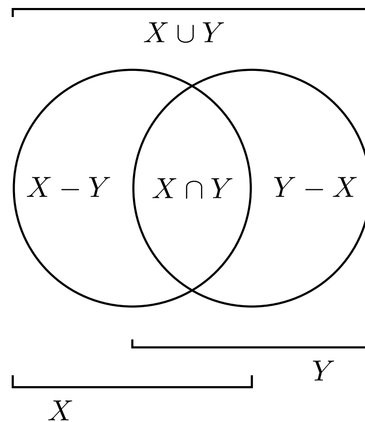
$$X = Y \iff \begin{array}{l} \text{for all } x \in X, x \in Y \\ \text{and, for all } y \in Y, y \in X \end{array}$$

If the first of these conditions holds then we say that X *is a subset of* Y and write $X \subseteq Y$.

The *empty set* is the unique set that has no elements and is denoted $\emptyset$.

3 Operations on sets

We can combine sets in various ways, summarized in the following diagram:



- $\cup$ is called *union*, $a \in X \cup Y$ if $a \in X$ or $a \in Y$;
- $\cap$ is called *intersection*, $a \in X \cap Y$ if $a \in X$ and $a \in Y$
- $-$ is called *difference* or *relative complement*, $a \in X - Y$ if $a \in X$ and $a \notin Y$;
- if our sets exist in some "universe", U, (e.g. the integers) then we also have a notion of *complement*, where $X^c = U - X$.

4 Partitions

A collection of sets X_1, X_2, $\ldots$, X_k *partition* a set X if

- $X = X_1 \cup X_2 \cup \cdots \cup X_k$, and
- for $i \neq j$, X_i and X_j are *disjoint*, that is $X_i \cap X_j = \emptyset$.

5 Set builder notation

A second, and more common, way to represent a set is by defining explicitly the rule or formula that its elements must satisfy in order to belong to the set. We write:

$$X = \{x \mid \text{some property of } x \text{ holds}\}$$

which is to be read as "X is the set of those elements x such that some property of x holds". For instance, the set of perfect squares $\{0, 1, 4, 9, 16, \dots\}$ can be defined as

$$S = \{m \mid \text{for some } n \in \mathbb{N},\ m = n^2\}.$$

Here and henceforth we use $\mathbb{N}$ as a symbol for the *natural numbers*, or non negative integers

$$\mathbb{N} = \{0, 1, 2, 3, \dots\}.$$

6 Ordered pairs and tuples

Given two sets A and B we can form *ordered pairs*, (a, b) whose first coordinate comes from A and second from B. The rule here is that:

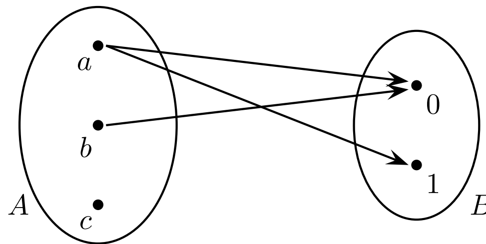
$$(a, b) = (a', b') \iff a = a' \text{ and } b = b'$$

i.e. two ordered pairs are equal if and only if their coordinates are equal and in the same order. The set of all ordered pairs whose first coordinate comes from A and second from B is called the *Cartesian product* of A and B and denoted $A \times B$.

More generally we can form *ordered tuples* of any fixed length and corresponding Cartesian products of sets A_1, A_2, ..., A_k.

7 Relations

A *relation* from A to B (or between A and B) is just a subset of $A \times B$. Relations are often illustrated schematically by "potato and arrow" diagrams. For instance the relation from $\{a, b, c\}$ to $\{0, 1\}$ which is the set $\{(a, 0), (a, 1), (b, 0)\}$ could be drawn as follows:



8 Functions and partial functions

A *function* is a special sort of relation. In the potato and arrow diagram there must be exactly one arrow from each element of A to some element of B. In other words a function is like a procedure that takes elements of A as input and produces elements of B as output. A *partial function* is like a function except that some elements of A may not have arrows that start from them (the "procedure" is not fully defined, or crashes on some inputs). If we want to be completely clear we sometimes call functions which are defined for every element of A *total functions*.

If we have a function, f, and an element $a \in A$, then the element of B that is paired with it is generally denoted $f(a)$. That is:

$$f = \{(a, f(a)) \mid a \in A\}.$$

The set of all functions from A to B is denoted B^A.

A function is *one to one* or *injective* if no two arrows point to the same element (that is, different inputs give different outputs). A function is *onto* or *surjective* if every element of B has at least one arrow pointing to it. A function is a *one to one correspondence* or *bijection* if it is both injective and surjective.

9 Tutorial problems

1. Let $X = \{1, 2, 5\}$ and $Y = \{0, 2, 4, 6\}$. List the elements of:
   (a) $X \cup Y$ (the union of X and Y)
   (b) $X \cap Y$ (the intersection of X and Y)
   (c) $X - Y$ (the complement of Y relative to X)
   (d) $Y - X$ (the complement of X relative to Y)
   (e) $\mathcal{P}(X)$ (the power set of X, i.e. the set of all subsets of X)
2. Let $X = \{0, 1, 2\}$ and $Y = \{f, t\}$.
   (a) List all the members of $X \times Y$.
   (b) List all total functions from Y to X.
   (c) List all partial (but not total) functions from Y to X.
3. Let X be a set with 3 elements, and Y a set with 4 elements.
   (a) How many elements are there in $X \times Y$?
   (b) How many total functions are there from X to Y?
   (c) How many total functions are there from Y to X?
   (d) How many partial (possibly total) functions are there from X to Y?
4. Give examples of functions $f : \mathbb{N} \to \mathbb{N}$ that satisfy:
   (a) f is total and injective but not surjective,
   (b) f is total and surjective but not injective,
   (c) f is total, injective and surjective, but is not the identity,
   (d) f is not total, but is surjective.

Today we're going to talk about partitions of sets, equivalence relations and how they are equivalent. Then we are going to talk about the size of a set and will see our first example of a diagonalisation argument (and show that some sets are countable, and some are not).

1 Partitions and Equivalence Relations

1.1 Partitions

A collection of sets $X_1, X_2, ...X_k$ partition a set if:

$$X = X_1 \cup X_2 \cup \cdots \cup X_k$$
$$X_i \cap X_j = \phi \text{ for } i \neq j.$$

Example: Suppose

$$X_0 = \{2n | n \in \mathbb{N}\}$$
$$X_1 = \{2n + 1 | n \in \mathbb{N}\}$$

Then X_0, X_1 partition $\mathbb{N}$ since:

$$\mathbb{N} = X_0 \cup X_1$$
$$X_0 \cap X_1 = \phi$$

1.2 Equivalence Relations

An equivalence relation is a relation from a set to itself. It is typically denoted by $\sim$. E.g. $a \sim b$ reads "a is equivalent to b". An equivalence relation satisfies:

Reflexivity $a \sim a$ for all $a \in A$

Symmetry if $a \sim b$ then $b \sim a$

Transitivity if $a \sim b$ and $b \sim c$ then $a \sim c$.

Example (modular arithmetic):

$n \sim m$ iff $n \bmod 2 = m \bmod 2$, is an equivalence relation on $\mathbb{N}$.

means $n/2$ and $m/2$ have the same remainder. $(n - m)/2$ has remainder 0. $n - m = 2k$ for some $k \in \mathbb{Z}$.

Let's check if it is an equivalence relation:

Reflexivity $n \bmod 2 = n \bmod 2$ for all $n \in N$.

Symmetry If $n \sim m$ then

$$\begin{aligned} (n-m) &= 2k \\ (m-n) &= 2(-k) \\ (m-n) \bmod 2 &= 0 \\ m &\sim n \end{aligned}$$

Transitivity If $n \sim m$ and $m \sim o$ then:

$$\begin{aligned} (n-m) &= 2k \\ (m-o) &= 2j \\ (n-m)+(m-o) &= 2(k+j) \\ (n-o) &= 2(k+j) \\ n &\sim o \end{aligned}$$

1.3 Equivalence Relations Partition Sets

Given an equivalence relation $\sim$ on A, we define the equivalence class $[a]$ of a by:

$$[a] = \{b | b \sim a\},$$

i.e. $[a]$ is the subset of everything in A equivalent to a.

Example (modular arithmetic):

For the equivalence relation $\mathrm{mod} 2$:

$$\begin{aligned} [0] &= \{n \in \mathbb{N} | n = 0 \bmod 2\} \\ &= \{n \in \mathbb{N} | n - 0 = 2k \text{ for some } k \in \mathbb{Z}\} \\ &= \{2k | k \in \mathbb{N}\} \\ &= X_0 \\ [1] &= \{n \in \mathbb{N} | n = 1 \bmod 2\} \\ &= \{n \in \mathbb{N} | n - 1 = 2k \text{ for some } k \in \mathbb{Z}\} \\ &= \{n \in \mathbb{N} | n = 2k+1\} \\ &= \{2k+1 | k \in \mathbb{N}\} \\ &= X_1 \end{aligned}$$

And we have already seen that X_0 and X_1 partitions $\mathbb{N}$.

Theorem 1.1. *Let $\sim$ be an equivalence relation on A. The equivalence classes of $\sim$ form a partition of A. Specifically:*

- *for every $a \in A$, $a \in [a]$*
- *for every $a, b \in A$ either $[a] = [b]$ (if $a \sim b$) or $[a] \cap [b] = \emptyset$ (if $a \not\sim b$).*

2 Cardinality

Cardinality is the size of a set, written $card(X)$, or sometimes $|X|$.

Examples:

$$card(\{1,2,3,4\}) = 4$$
$$card(\{a,b,c\}) = 3$$
$$card(\mathbb{N}) = \infty$$

What does it mean to say $card(\mathbb{N})$ is infinite? Does $card(\mathbb{N}) = card(\mathbb{Z})$? What about $card(\mathbb{Q})$, $card(\mathbb{R})$?

Two sets, X and Y have the same cardinality if there exists a bijection $f : X \to Y$.

If there exists an injection $f : X \to Y$, then $card(X) \leq card(Y)$.

Theorem 2.1 (Schroder-Bernstein)**.** *If $card(X) \leq card(Y)$ and $card(Y) \leq card(X)$, then $card(X) = card(Y)$.*

Examples: Suppose

$$X = \{1,2,3\}$$
$$Y = \{a,b,c\}$$

Let $f : X \to Y$ be defined by $f(1) = a; f(2) = b; f(3) = c$. Since f is $1-1$ and onto, $card(X) = card(Y)$.

Let $f : \mathbb{N} \to \mathbb{Z}$ be defined by:

$$f(n) = \begin{cases} n/2 & \text{for } n \text{ even} \\ \dfrac{-n-1}{2} & \text{for } n \text{ odd.} \end{cases}$$

Since f is bijective, $card(\mathbb{N}) = card(\mathbb{Z})$.

It is also known that $card(\mathbb{N}) = card(\mathbb{Q})$, but $card(\mathbb{N}) \neq card(\mathbb{R})$. In other words, there are more reals than natural numbers (or rather, the reals are more infinite than the natural numbers).

An infinite set with $card(A) = card(\mathbb{N})$ is countably infinite or denumerable[1].

An infinite set with $card(A) \neq card(\mathbb{N})$ is uncountably infinite.

2.1 Cantor's Diagonal Argument

Cantor's diagonal argument is used to prove that a set is uncountable.

For example:

[1] Finite sets are also denumerable.

Theorem 2.2. *The set denoted $\mathbb{N}^{\mathbb{N}}$ defined to be the set of all functions from $\mathbb{N} \to \mathbb{N}$ is uncountable.*

Proof. Suppose to the contrary that $\mathbb{N}^{\mathbb{N}}$ is countable. List each function as $f_0, f_1, \ldots$.

	0	1	2	3	$\cdots$
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	$\cdots$
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$\cdots$
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Now define the function $f : \mathbb{N} \to \mathbb{N}$ by:

$$f(n) = f_n(n) + 1$$

By the definition of f it is different to every function in our list. Therefore our list does not include all possible functions which contradicts our assumption that $\mathbb{N}^{\mathbb{N}}$ is countable. Therefore $\mathbb{N}^{\mathbb{N}}$ must be uncountable. □

3 Tutorial problems

1. Define a binary relation $\sim \mathbb{N}$ by: $n \sim m$ if and only if $n - m$ is a multiple of 12. Show that $\sim$ is an equivalence relation. How many equivalence classes does it have?

2. Define a binary relation $\sim \mathbb{N}$ by: $n \sim m$ if and only if n and m have the same first digit (written in base 10). Show that $\sim$ is an equivalence relation. How many equivalence classes does it have?

3. Show that the set of even natural numbers is denumerable.

4. Show that the set of even integers is denumerable.

5. Show that the set of total functions from $\mathbb{N}$ to $\{0, 1\}$ is uncountable.

6. A total function $f : \mathbb{N} \to \mathbb{N}$ is called *monotone increasing* if $f(n) < f(n+1)$ for all $n \in \mathbb{N}$. Prove that the set of monotone increasing functions from $\mathbb{N}$ to $\mathbb{N}$ is uncountable.

7. (Harder) Show that, for any set A, $card(A) < card(\mathcal{P}(A))$.

8. (Harder) A function $f : \mathbb{N} \to \{0, 1\}$ is said to have *finite support* if it is non-zero only finitely many times (or, put another way, there is some n such that for all $m \geq n$, $f(m) = 0$). Show that the set of functions from $\mathbb{N}$ to $\{0, 1\}$ that have finite support is denumerable.

1 Motivation

It's worthwhile just briefly recapping why we are covering this current material. What is the purpose of learning this stuff? Where are the applications of this set stuff? There are a couple of ways to answer such questions:

1. This is theory, we don't need no stinking applications. Sometimes, stuff is intrinsically interesting.
2. Set theory though is the foundation of theoretical computer science. We're going over this stuff now as: introductory material that we may use later; to give you a chance to get familiar with the notation used later; and to give you an introduction to proof techniques on relatively simple problems.
3. But I also understand that many CS students are driven by practical considerations (me too). So here are a few real questions we are going to answer in this paper:
   (a) Can we build an html parser using regular expressions?
   (b) Can we build a program that will tell us if there is a bug in another program? Simpler: can we build a program that will tell us if another program will finish?
   (c) Can we efficiently solve this particular problem?

   The answer to the first two questions is no (we deal with these in the first half of the course). The answer to the third question is: not as far as we know, but maybe it's possible and maybe it isn't.

In today's lecture, we're going to look at defining infinite sets recursively and the method of proof-by-induction. Both of these techniques are foundational for much of theoretical CS, and you have probably already come across them in 242.

2 Recursion

Recursion helps us describe infinite sets.

A recursive definition of a set, X includes:

- A basis, or set of atomic cases, which is a list of some objects that must belong to X;
- A recursive step, which is a rule or collection of rules for producing new objects in X from existing ones;
- A closure rule, stating that everything in X must be generated by a finite number of applications of the recursive step. The closure rule is usually left unstated.

2.1 Example: Natural Numbers

The set $\mathbb{N}$ can be defined as:

1. Basis: $0 \in \mathbb{N}$
2. Recursive step: If $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$
3. Closure: $n \in N$ only if it can be obtained from 0 by a finite number of applications of the operation $+$.

Now we can "construct" $\mathbb{N}$:

$$\begin{aligned} 0 \in \mathbb{N} &\implies 0 + 1 = 1 \in \mathbb{N} \\ &\implies 1 + 1 = 2 \in \mathbb{N} \\ &\implies 2 + 1 = 3 \in \mathbb{N} \end{aligned}$$

or $\mathbb{N} = \{0, 1, 2, ...\}$.

Actually, this definition is imprecise. Could an alien unfamiliar with our digits understand it? Probably not. Here is a more precise definition:

1. Basis: $0 \in \mathbb{N}$
2. Recursive step: if $n \in \mathbb{N}$, then $s(n) \in \mathbb{N}$
3. Closure: as before.

In this case, we "construct" $\mathbb{N}$ as:

$$\begin{aligned} 0 \in \mathbb{N} &\implies s(0) \in \mathbb{N} \\ &\implies s(s(0)) \in \mathbb{N} \\ &\implies s(s(s(0))) \in \mathbb{N} \end{aligned}$$

Then we can define $1 \equiv s(0)$, $2 \equiv s(s(0))$, etc.

2.2 Example: Unlabeled Complete Binary Tree

The set of unlabeled complete binary trees, B, can be defined as:

1. Basis: $() \in B$.
2. Recursive step: if $b \in B$, then $(bb) \in B$.
3. Closure: as before.

Which we can construct as:

$$\begin{aligned} () \in B &\implies (()()) \in B \\ &\implies ((()())(()())) \in B \\ &\implies (((()())(()()))((()())(()()))) \in B \end{aligned}$$

2.3 Example: Sum

Let's define the function $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, taking arguments m and n (ie $m + n$):

1. Basis: if $n = 0$ then $m + n = m$.
2. Recursive step: $m + s(n) = s(m + n)$

Let's try $3 + 2$:

$$\begin{aligned} &s(s(s(0))) + s(s(0)) \\ =&s(s(s(s(0))) + s(0)) \\ =&s(s(s(s(s(0))) + 0)) \\ =&s(s(s(s(s(0))))) \end{aligned}$$

3 Induction

Now that we can recursively define sets, we probably will want to prove things about them. The fall back proof technique for recursively defined sets is called induction.

If we want to prove some statement S about a recursively defined set X, then it is sufficient to:

1. Basis: show S is true for the base case X_0
2. Assume that S is true for element X_{n-1}, show that S is true for element X_n.

3.1 Example: Binary Tree

Consider the set of binary trees with natural numbers as nodes:

- A natural number is a binary tree
- If T_L and T_R are binary trees, and n is a natural number, then the triple $T = (n, T_L, T_R)$ is a binary tree.

Define the number of leaves of a binary tree to be 1 if it is a leaf, and the sum of the number of leaves of T_L and T_R otherwise. Define the number of internal nodes of a binary tree to be 0 if a leaf, and 1 plus the sum of the number of internal nodes of T_L and T_R otherwise.

Theorem 3.1. *In any binary tree, the number of internal nodes is one less than the number of leaves.*

Proof. Base case: One leaf and no internal nodes, therefore it is obviously true.

Inductive step: assume it's true for T_L and T_R and show it's true for T:

$$\begin{aligned} internal(T) &= 1 + internal(T_L) + internal(T_R) \\ &= 1 + leaves(T_L) - 1 + leaves(T_R) - 1 \\ &= leaves(T_L) + leaves(T_R) - 1 \\ &= leaves(T) - 1 \end{aligned}$$

□

3.2 Example: Sum of numbers

Theorem 3.2.

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}.$$

Proof. Base: $1 = 1 * 2/2$

Inductive step: Suppose $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Show $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$ □

3.3 Example: Number of nodes in a complete binary tree

Theorem 3.3. *The number of nodes in a complete binary tree of height n is $2^{n+1} - 1$. Or $1 + 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 1$.*

Proof. Base case: $1 = 2^1 - 1$.

Inductive step: assume $1+2+2^2+\ldots+2^n = 2^{n+1}-1$, show that $1+2+2^2+\ldots+2^n+2^{n+1} = 2^{n+2} - 1$. □

4 Tutorial problems

Recursive definitions

1. Give a recursive definition of the set $P = \{1, 2, 4, 8, 16, \ldots\}$ of powers of two within $\mathbb{N}$. (You may assume the operation $+$ on $\mathbb{N}$ has already been defined.)
2. Give a recursive definition of the subset, Eq, of $\mathbb{N} \times \mathbb{N}$ representing the relation *is equal to* using the successor function $s(n) = n + 1$.
3. Give a recursive definition of "linked list of natural numbers" (hint: the base case should be equivalent to `nil`, and the construction should use a pair).
4. (Harder) Give a recursive definition of the set of *finite* subsets of $\mathbb{N}$, using the successor function s and union as the operators.

Inductive proofs

1. Prove that $2 + 5 + 8 + \cdots + (3n - 1) = n(3n + 1)/2$ for all $n > 0$.
2. Prove that $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ for all $n \geq 0$.
3. Suppose that two algorithms do the same thing, but the first requires $f(n) = 4n + 1$ steps, while the second requires $g(n) = n^2$ steps. For small values of n, the second algorithm is better, but it seems clear that when n is "big enough" we should prefer the first. Make this precise by proving that $f(n) < g(n)$ for all $n \geq 5$.
4. Consider the following recursively defined function $f : \mathbb{N} \to \mathbb{N}$.

$$\begin{aligned} f(0) &= 0 \\ f(n+1) &= f(n) + 2n + 1 \end{aligned}$$

   Compute the first few values of f, form a conjecture for a more "natural" description of f, and prove that your conjecture is correct.

5. Let a and b be two symbols. Recursively define a set of strings (sequences), D as follows:

   Basis The empty string is in D.

   Recursive step If a string s is in D, then so is the string asb. If strings s and t are in D then so is the string st.

   (a) List all the strings in D consisting of 6 or fewer symbols.
   (b) Prove by induction that every string in D has even length.
   (c) Prove by induction that in any prefix of a string in D there are at least as many a's as b's.

6. Consider the following piece of pseudocode defining a function $\mathsf{foo}(x, y)$, where we assume $x, y \in \mathbb{N}$ (the division operator is the usual e.g. Java integer division, and % is the remainder operation). To maintain some mathematical integrity we use $=$ in the mathematical sense (i.e. of an equality test) and use $\leftarrow$ to denote assignment.

   if $x = 0$ **then**
     `return` 0
   end if
   $s \leftarrow \mathsf{foo}(x/2, 2y)$
   if $x\%2 \neq 0$ **then**
     $s \leftarrow s + y$
   end if
   `return` s

   Compute a table of values for foo, form a conjecture about its more natural definition, and prove that conjecture by induction.

1 Introduction

Let's consider what an actual computer does. What is it's base language? 0's and 1's. What does it do with those 0's and 1's? It manipulates them to produce other 0's and 1's. Essentially, then, a computer is a string manipulator. It takes as input strings of 0's and 1's and produces as output other strings of 0's and 1's. So it makes sense to turn to the problem of representing sets of strings, or *languages* and characterising languages of particularly simple or useful form. In particular, we are going to build up a hierarchy of increasingly more powerful languages. First we're going to start with regular languages.

Since languages are made up of strings, we're going to start by defining what a string is.

2 Strings

An *alphabet* is any finite set, and is usually denoted Σ. The most common alphabet we work with will be $\{a, b\}$, though others will certainly appear.

We will give two formal definitions of string or word over Σ. Intuitively, a string is a sequence of symbols from Σ of some finite length. So, one definition represents a string of length n as a function:

$$f : \{0, 1, 2, ..., n-1\} \to \Sigma$$

You can think of this exactly as an array of characters if you like. For convenience, we could also write this as: $f : n \to \Sigma$.

The length of a string $f : n \to \Sigma$ is n, denoted $|f| = n$.

The fundamental operation on strings is concatenation. If $f : n \to \Sigma$ and $g : m \to \Sigma$ are two strings, then $f \cdot g : m + n \to \Sigma$ is the concatenation of f and g and is given by:

$$(f \cdot g)(k) = \begin{cases} f(k) & \text{if } k < n \\ g(k-n) & \text{if } n \leq k \end{cases}$$

Give an example.

It should be clear that $|f \cdot g| = |f| + |g|$.

The empty string $f : \phi \to \Sigma$ is denoted λ, and $f = \lambda \cdot f = f \cdot \lambda$ for any f. Often we omit the dot: $f = \lambda f = f\lambda$.

The set of all strings over an alphabet Σ is denoted Σ^*.

We can also define strings recursively, in a similar way to how we'd define a linked list. In this case we define the set Σ^*:

Base: $\lambda \in \Sigma^*$

Recursive step: if $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$.

And for concatenation:

$$u \cdot \lambda = u$$
$$u \cdot wx = uwx$$

From now on we'll generally ignore the formality and assume "string", "concatenation", and "length" as primitive concepts.

3 Languages

A language over Σ is a subset of Σ^*.

Examples:

$L_a = \{w \in \Sigma^* | w \text{ starts with an } a\}$ is a language over $\Sigma = \{a, b\}$. It can be defined recursively as:

- Base: $a \in L_a$
- RS: if $w \in L_a$ and $x \in \Sigma$ then $wx \in L_a$.

$L_e = \{w \in \Sigma^* : |w| \text{ is even}\}$ is a language over $\Sigma = \{a, b\}$. Defined recursively as:

- Base: $\lambda \in L_e$
- RS: if $w \in L_e$ and $x \in \{aa, ab, ba, bb\} = T$ then $wx \in L_e$.

If there's time, let's prove L_e is the set of strings of even length. There are two parts: that every string in L_e has even length; and that all strings of even length are in L_e.

First part (by induction):

- Base: $\lambda \in L_e$ and $|\lambda| = 0$ which is even.
- RS: Assume $u \in L_e$. ux also in L_e where $x \in T$. $|ux| = |u| + |x| = \text{even} + 2 = \text{even}$. By induction, all members of L_e have even length.

Second part (also by induction). Choose some string of even length, w. Either:

- Base: $|w| = 0$, $w = \lambda \in L_e$; or
- RS: $w = uv$ for some u of even length and $|v| = 2$. By induction, assume $u \in L_e$ and since $v \in T$, we get $w \in L_e$.

4 Regular Languages

A common way to specify a language is to use a "regular expression". These languages are called regular languages. To construct a regular language over $\Sigma = \{a, b\}$:

1. Start with the languages:

$$\phi, \underline{\lambda} = \{\lambda\}, \underline{a} = \{a\}, \underline{b} = \{b\}$$

2. Form a new language using finite combinations of three operations applied to two languages L and K:

   - $L \cup K$ (union)
   - $LK = \{xy | x \in L, y \in K\}$ (concatenation)
   - L^* concatenation of 0 or more elements of L. Or: $\lambda \in L^*$, and if $w \in L^*$ and $u \in L$, then $wu \in L^*$.

For convenience we also use $L^+ = LL^*$ to mean "one or more occurrences".

Examples:

$$\begin{aligned} L_a &= \underline{a}(\underline{a} \cup \underline{b})^* \\ &= a(a \cup b)^* \end{aligned}$$

$$L_e = ((a \cup b)(a \cup b))^*$$

$$L_{bb} = (a \cup b)^* bb (a \cup b)^*$$

$$L_{2b} = a^* b a^* b a^*$$

The grep utility also uses regular regular expressions, and many programming languages have regular expression libraries. E.g. $t[a-z]n$, or $t[a-z]*n$.

5 Tutorial problems

Recursive definition of languages

1. Give a simple recursive definition of the language Eq consisting of strings over $\{a, b\}$ which have an equal number of a's and b's.
2. Give a recursive definition of the language NoRep over the alphabet $\{a, b, c\}$ consisting of all strings in which no consecutive pair of symbols are the same (i.e. aa, bb, and cc must not occur as substrings of words in NoRep.

Defining languages by regular expressions

1. Let $X = \{ab, aa\}$, $Y = \{\lambda, b, ba\}$.
   (a) List the strings in XY.
   (b) How many strings of length 6 are there in X^*?
   (c) List the strings in Y^* of length three or less.
   (d) List the strings in X^*Y^* of length four or less.
2. Find a regular expression for the language over $\{a, b, c\}$ consisting of all those strings in which all the a's (if any) precede all the b's (ditto) which in turn precede all the c's (likewise).
3. The same as the previous exercise but with the condition that the string not be empty.
4. Find a regular expression for the strings over $\{a, b\}$ that contain both aa and bb as substrings.
5. Find a regular expression for the strings over $\{a, b\}$ that contain both aba and bab as substrings.
6. Find a regular expression for the strings over $\{a, b, c\}$ that begin with a, contain exactly two b's, and end with cc.
7. Find a regular expression for the strings over $\{a, b\}$ with an even number of a's or an odd number of b's.
8. Do you think it is possible to describe the language Square over the alphabet $\{a\}$ consisting of all strings whose length is a perfect square using a regular expression?

1 Introduction

The collection of regular languages, while interesting, is not rich enough to be useful. For instance, even a simple language such as $\{a^n b^n \mid n \geq 0\}$ is not regular (though, to be frank, we have no way of proving this – yet). The notion of a *grammar* and in particular of a *context free grammar* provides a more general way of recursively defining languages which is still “mechanical” in some sense. In this lecture we explore this concept.

2 Rules

A *rule* or *production rule* is an expression of the form:

$$A \to w.$$

The left hand side, A must be an element of a finite set V of *variables* or *nonterminals* (generally written as upper case letters). The right hand side w must be a string in $(V \cup \Sigma)^*$ where Σ is some fixed finite alphabet disjoint from V (whose elements are generally written as lower case letters). The elements of Σ are also called *terminals*. For instance, all of the following are rules (over appropriate alphabets):

$$S \to aSb,\ S \to \lambda,\ T \to aTbT,\ T \to aPbQcR.$$

A rule can be *applied* to any string in $(V \cup \Sigma)^*$ which contains the symbol occurring on the left hand side of the rule. The result of such an application is to replace *any one* instance of the left hand side by the string on the right hand side. That is, if the rule $A \to w$ is applied to the string uAv, the result is the string uwv. We write:

$$uAv \stackrel{A \to w}{\Longrightarrow} uwv$$

or just $uAv \Rightarrow uwv$ if we don’t care to mention the rule that was applied.

3 Grammars

A *context free grammar*, G, is just some set of rules over fixed sets of nonterminal and terminal symbols, together with a single distinguished nonterminal symbol (universally denoted S) called the *start* symbol.

A *derivation* in (or of) G is a sequence of rule applications:

$$v = v_1 \Rightarrow v_2 \Rightarrow v_3 \Rightarrow \cdots \Rightarrow v_n = w$$

where each rule is one of the rules of G which we write in shorthand as $v \stackrel{*}{\Rightarrow} w$, and in this case we say that w is *derivable* from v (in G).

Formally we could define derivability recursively: v is derivable from itself (base), and if $u = xAy$ is derivable from v and $A \to w$ is a rule of G then xwy is derivable from v.

The *language of* G, $L(G)$ is:

$$\{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\}$$

i.e. the set of strings over the terminal symbols that are derivable from the start symbol.

4 Examples

The language of:

$$S \to aS,\; S \to bS,\; S \to \lambda$$

is $(\boldsymbol{a} \cup \boldsymbol{b})^*$.

When a grammar has a number of rules with the same LHS they are frequently collected together, e.g. the preceding example could be written:

$$S \to aS \mid bS \mid \lambda$$

The language of

$$S \to aS \mid bT,\; T \to bT \mid \lambda$$

is $\boldsymbol{a}^* \boldsymbol{b}^+$.

The language of

$$S \to aSb \mid \lambda$$

is $\{a^n b^n \mid n \geq 0\}$.

5 Derivation trees

Any derivation $S \stackrel{*}{\Rightarrow}$ has a tree associated with it. The root of this tree is labelled S. At each individual production $A \to x$, make the symbols of x the children of the node corresponding to A in left to right order.

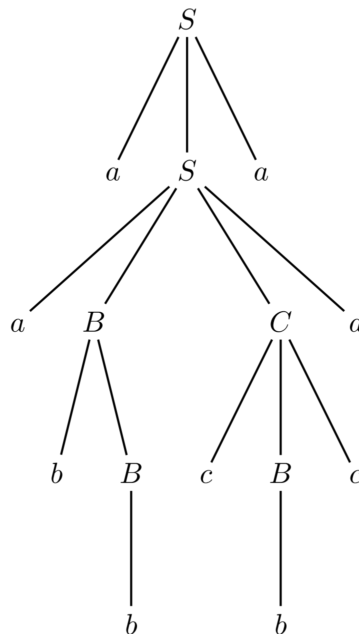
For example, consider the grammar:

$$S \to aSa \mid aBCa,\; B \to bB \mid b,\; C \to cCc \mid cBc$$

and the derivation:

$$S \Rightarrow aSa \Rightarrow aaBCaa \Rightarrow aabBCaa \Rightarrow aabBcBcaa \Rightarrow aabbcBcaa \Rightarrow aabbcbcaa$$

then the corresponding derivation tree is:



6 Regular grammars

A context free grammar is called a *regular grammar* if every rule has one of the three forms:

$$A \to a,\ A \to aB,\ A \to \lambda$$

where in the second form $a \in \Sigma$, $B \in V$ (and $B = A$ is allowed).

Clearly in any derivation from S in a regular grammar there will only ever be at most one variable symbol produced, and it will always be rightmost in the current word.

From the name it's clear that the intention is that regular languages should be produced from regular grammars (and presumably vice versa) but we don't yet have the mechanics which will allow us to prove that conveniently (it *could* be done from the definitions, but we will shortly be introducing another way of generating regular languages and it will be most convenient to show that all three are equivalent to one another).

7 Tutorial problems

"The faster you derive, the bigger the mess"

1. Let G be the grammar:

$$\begin{aligned} S &\rightarrow abSc \mid A \\ A &\rightarrow cAd \mid cd \end{aligned}$$

   (a) Give a derivation of $ababccddcc$ and build its derivation tree.
   (b) Use set notation to describe $L(G)$.

2. Design context free grammars for the following languages (the alphabet is $\{a, b\}$ throughout).

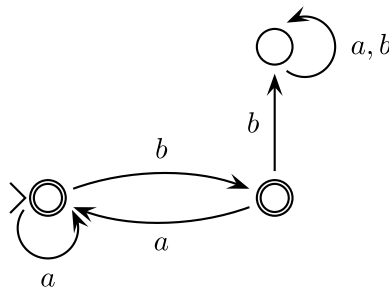
   (a) $L_1 = \{a^{4n} \mid n > 0\}$.
   (b) The language, PALINDROME, consisting of all strings that read the same forwards as backwards.
   (c) The language of strings that contain at least one occurrence of aa as a substring.
   (d) The language, EQUAL, consisting of all strings that contain the same number of a's as b's.
   (e) The language EVEN-EVEN consisting of all strings that contain both an even number of a's and an even number of b's

3. Find regular expressions and design regular grammars for the following languages.

   (a) The language of all strings of the form $a^k b^2$ where $k \geq 0$.
   (b) The language of all strings in which every a is either immediately preceded by, or immediately followed by a b (or both).
   (c) The language of all strings with an even number of a's.
   (d) The language of all strings with an odd number of b's.
   (e) The language of all strings of even length.
   (f) The language EVEN-EVEN (see above).

1 Introduction

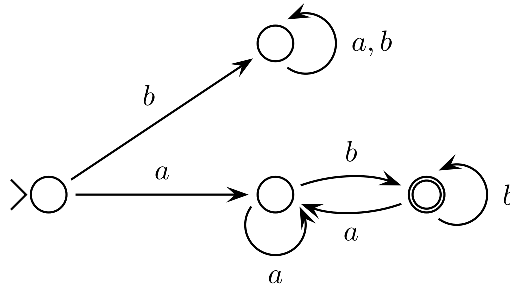
Finally we arrive at a point where we're going to talk about some "computing machines". But, these machines are very limited compared to the general purpose computers we are used to. In fact, they are much more similar to various hardware controlled units (old fashioned adding machines, vending machines, typewriters). The basic idea is that the machine is a black box with a key board attached, and a light. Certain combinations of key strokes cause the light to shine – what's actually going on is that there are finitely many internal states possible, and the various key strokes cause transitions between those states. Some of the states are marked as "accepting" and if we are in one of those states, then the light shines. Associated to the machine is a language that it accepts – those sequences of key strokes which, from the initial state, cause the light to be on. The challenge is to model all this and to determine what sorts of languages are accepted by these finite state automata.



In the diagram above, the little $>$ symbol points at the initial state, and the arrows indicate the state transitions caused by various inputs. The double circles represent accepting or "light on" states, while the single circle represents a rejecting state. It's easy to see that once we get to the rejecting state we stay there and we get there exactly if we see two b's in a row in the input. So, this machine recognizes the language HAS-NO-*bb*.

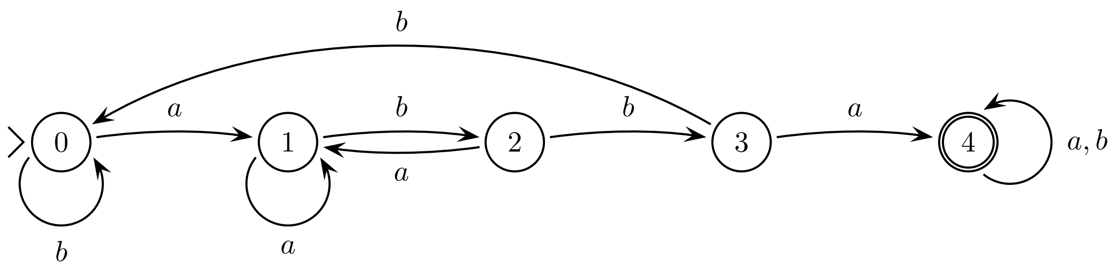
2 More examples

The following machine recognizes the language "starts with a and ends with b"



Note again that we enter a "cannot succeed" state if we begin with b. The light is on whenever we started with a and finished with b – it stays on if we weren't actually finished typing but type some more b's. It turns off again if we add another a.

A somewhat more complicated example, the language HAS-$abba$ of strings that contain $abba$ as a consecutive substring. This time to be helpful we label the states. The labels indicate how long a prefix of $abba$ we have seen in the last few characters of the input. Once we reach 4 of course the light goes on and stays on. This time we don't need a "cannot succeed" state because there's always hope of seeing $abba$ in the future.



3 Deterministic finite state automata

A *deterministic finite state automaton*, M, consists of:

- A finite set, Q, of *states*;
- A finite alphabet Σ of *actions*;

- A total function $\delta : Q \times \Sigma \to Q$ called the *transition function*
- A distinguished state $q_0 \in Q$ called the *initial state*; and
- A subset F of Q called the *final* or *accepting* states.

The letters chosen don't mean anything but are traditional.

Given a word $w = w_0 w_1 \dots w_{n-1} \in \Sigma^*$ the *computation* carried out by M on input w is a sequence of states $q_0, q_1, q_2, \dots, q_n$ defined as follows:

$$q_1 = \delta(q_0, w_0),\ q_2 = \delta(q_1, w_1),\ \dots,\ q_n = \delta(q_{n-1}, w_{n-1})$$

We say that M *accepts* or *recognises* w if $q_n \in F$ and otherwise it *rejects* w.

The *language of* M, $L(M)$ is just the set of strings in Σ^* that M accepts.

4 Non determinism

We can loosen the definition above to allow several forms of non-determinism in finite state automata.

The first of these is akin to keyboard jam – instead of demanding that δ be a total function from $Q \times \Sigma$ to Q we allow it to be partial. This allows us to eliminate "can't accept" states, by simply not including any transitions that would lead to such states.

The other forms of non-determinism extend rather than restrict the transition function. In "multiple worlds" (not an official name) non-determinism, we simply allow δ to be a relation on $Q \times \Sigma \times Q$. For each triple $(q, x, q') \in \delta$ we might transition from q to q' on symbol x. But equally there might be some other $(q, x, q'') \in \delta$ and we could go to q'' instead. Now, instead of a single possible computation on input w we have a whole range of possibilities – basically we just require that $(q_i, w_i, q_{i+1}) \in \delta$. We define $L(M)$ in this case to be the set of words for which *some* computation winds up in F.

Finally, we could allow λ transitions, a.k.a. "bumping the machine" (also not official). Here we allow some transitions $q \to q'$ which do not consume any input (i.e. occur on a symbol λ). We accept w if there is some way of interspersing λ's among its elements and then following an accepting computation.

5 Tutorial problems

"If you build it, you will have built it."

1. In each part of this question, design a finite automaton that accepts the given language. As usual, take the alphabet to be $\{a, b\}$.
   (a) The language $\mathbf{a}^*\mathbf{b}^*$.
   (b) The language of all strings that contain exactly two a's.
   (c) The language EvenString consisting of all strings of even length.
   (d) The language consisting of all strings not containing the substring bbb.
   (e) The language $\mathsf{Even-even}$ consisting of all strings containing both an even number of a's and an even number of b's.
2. For each of the following languages, try to design a finite automaton that accepts it – but don't try for too long, because you can't. Try to get a feeling for what the problem is.
   (a) The language of all strings of the form $a^n b^n$ ($n \geq 0$).
   (b) The language of all strings over the alphabet $\{\langle, \rangle\}$ described by the grammar:
   $$S \to \langle\, S\, \rangle\, S \mid \lambda$$
   (these are called *balanced bracket sequences*).
3. In each part of this question, design an NFA that accepts the given language. Try to make it as small as possible, and compare to a DFA for the same language.
   (a) The language of all strings ending with b.
   (b) The language of all strings containing the substring bb.
   (c) The language of all strings containing either the substring bb or the substring $baab$.

1 Introduction

Today we'll explore some more the relationship between deterministic and non-deterministic finite state automata, with and without λ-transitions. Remember that in the non-deterministic case, M accepts w if *some* computation of M on w leads to an accepting state.

2 NFA-λ machines as modules

By introducing λ transitions we can "encapsulate" finite state automata in such a way that:

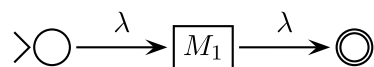
- there are no incoming transitions to the initial state
- there is a unique accepting state with no outgoing arrows

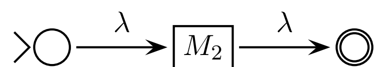
This is accomplished simply by adding a new initial state with a λ-transition to the previous initial state (and no other associated transitions), and adding a new accepting state, taking each original accepting state adding a λ-transition to it, and then making all of them no longer be accepting states. Clearly this new machine accepts exactly the same language as the original one. The advantage of enforcing the two conditions is that it makes it easy to combine NFAs.

Specifically we can prove:

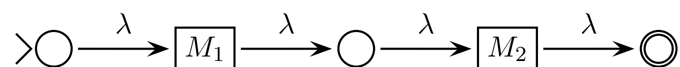
Theorem 2.1. *Suppose that M_1 and M_2 are NFAs accepting languages L_1 and L_2 respectively. Then there are NFAs that accept L_1L_2, $L_1 \cup L_2$ and L_1^*.*

Proof. Represent M_1 and M_2 as suggested above:

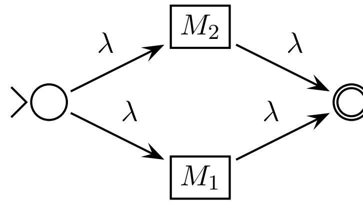




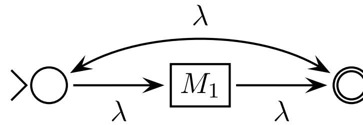
Now it is easy to combine these to recognize L_1L_2:



And $L_1 \cup L_2$:



And L_1^*:



□

Corollary 2.2. *Every regular language is accepted by some NFA.*

Proof. This is obvious since it's clear that there are NFAs which accept the "basic" regular languages $\emptyset$, $\boldsymbol{\lambda}$, and $\boldsymbol{x}$ for $x \in \Sigma$, and the previous result shows that the construction step in the recursive definition of regular languages can also be applied to NFAs. □

3 Eliminating non-determinism

Non-determinism is a bit sneaky – and it's hard to imagine how we could have "real" machines which implement it. Fortunately it turns out that it's not actually required, as anything that can be recognized by an NFA-λ can actually be recognized by a DFA. So, we can freely make use of NFAs in proofs (because they are more flexible) secure in the knowledge that if necessary they can be implemented via DFAs.

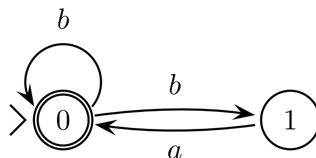
First let's see how to get rid of λ-transitions. Suppose that M is an NFA-λ and r is one of its states. Define the λ-closure of r to be the set of states that can be reached from r using only λ-transitions. Recursively: r is in the λ-closure of r, and if s is in the λ-closure of r, and $s \stackrel{\lambda}{\to} t$ then t is in the λ-closure of r.

Now define a new automaton M' that has the same set of states as M (and the same initial and accepting states) but without λ-transitions as follows. For each state q of M and each letter a in Σ define:

$$\delta'(q, a) = \{t \mid \text{for some } r \text{ in the } \lambda\text{-closure of } q,\ t \text{ is in the } \lambda\text{-closure of some } s \in \delta(r, a)\}$$

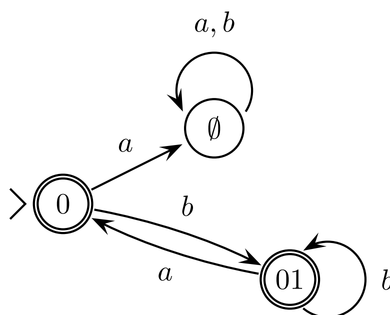
i.e. $\delta'(q, a)$ is just the set of all states we could get to from q by following some λ-transitions, then an a-transition, and then some more λ-transitions. It's clear that M and M' accept the same language, and M' has no λ-transitions.

How can we get from an NFA to a DFA? The idea is to trace all possible paths of a computation simultaneously, using states of the DFA to represent sets of states in the NFA – all the "places we might be" at this point. Before doing this formally here's a simple example.



It's easy to see that this automaton accepts $(\boldsymbol{b} \cup \boldsymbol{ba})^*$ and it's non-deterministic both because there are two outgoing b-transitions from state 0 and no a-transition there.

Now let's examine what sets of states we might be in after (partially) processing a word. Certainly we could be in state 0 alone (e.g. when we begin). From that state if we get an a we're nowhere, so we could be in the empty set of states (denote this by $\emptyset$). If we get a b we could be in state 0 or state 1 (denote by 01). We don't need to worry about what happens from the empty set (if we get either a or b we stay there – this is a "cannot accept" state). From the state 01, a b could leave us in 0 (if that's where we were) or take us to 1 (again if we were in 0), and a could take us to 0 (if we were in 1). So, we get to 01 on b and to 0 on a. Now there are no remaining states to worry about so let's try to draw the new automaton:



There was one remaining subtlety – each state of this automaton that includes an accepting state must be made into an accepting state.

The idea behind this construction gives us:

Theorem 3.1. *Let M be an NFA. Then there is a DFA DM that accepts the same language.*

Proof. Rather than explicitly working out "where we might be", in proving this result it's easiest to assume "we might be anywhere". That is, if Q is the set of states of M we take

the set of states of DM to be $\mathcal{P}(Q)$, i.e. the set of all subsets of Q. Then, for each $c \in \Sigma$ and $X \subseteq Q$ we define:

$$\delta_{DM}(X, c) = \{y \in Q \mid \text{for some } x \in X, y \in \delta_M(x, c)\}$$

Taking the initial state of DM to be $\{q_0\}$ where q_0 is the initial state of M, and the set of final states of DM to be the set of all $X \subseteq Q$ such that X contains some final state of M we have completed the construction.

By construction, each transition from a state X of DM identifies "where we might be" if we started in one of the states of M that belongs to X and processed the given character. So, an accepting computation in DM represents some accepting computation in M and vice versa. □

It's worth noting that the number of states we use in DM is 2^m where m is the number of states in M. It can be shown that this exponential blow up is *necessary* in general.

For a fixed positive integer N, consider the following language, L, over $\{a, b\}$. A word $w \in L$ if there is some pair of b's in the word so that the total number of a's between them is a multiple of N (there may be other b's between them as well - and there need not be any a's, i.e. *bb* is in the language). It's not immediately obvious that this language is regular, but it's easy to design an NFA that accepts it. This NFA has $N + 2$ states called S, 0, 1, 2, $\ldots N - 1$, and F.

The start state S, has a, and b loops, as well as a b arrow to state 0. The states 0, 1, 2, through $N - 1$ form a cycle with a arrows from each to the next (and from $N - 1$ to 0), and b loops. Finally there is a b arrow from 0 to F, and a and b loops on F. State F is the only accepting state.

To accept a word by this automaton we need to hang around S for a while, kick over to the cycle using a b, run around the cycle some number of times (thereby consuming a multiple of N a's, while ignoring b's) and finally kick off the cycle using another b. In other words, we accept L.

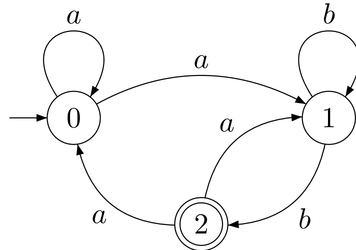
Now consider a DFA that accepts L. Since it is not allowed to "guess", it must know at any point what the possible lengths of "a's following a prior b" are modulo N. Given any subset $X \subseteq \{0, 1, 2, \ldots, N - 1\}$ it's easy to construct a word w_X where the set of such lengths is X. Just for example, if $N = 12$ and $X = \{1, 4, 6, 7\}$ we can take the $w_X = babaabaaaba$, which has 7 a's after the first b, 6 after the second etc.

Now, I claim that if $X \neq Y$ then the state we are in when we have processed w_X must be different from that we are in when we process w_Y. The reason is that there is some word v such that $w_X v \in L$ and $w_Y v \notin L$ or vice versa (if we were in the same state we would have to accept or reject both of these words). For instance if k belongs to X but not to Y (or vice versa) we can take $v = a^{N-k}b$.

So, any DFA to accept L must have at least 2^N states, and an exponential blow up cannot be ruled out.

4 Tutorial problems

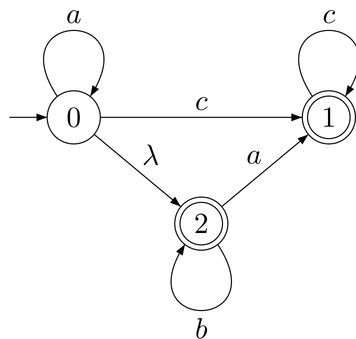
1. Let M be the nondeterministic finite automaton:



   - Trace all computations of the string $aaabb$ in M. Is it in $L(M)$?
   - Give a regular expression for $L(M)$.

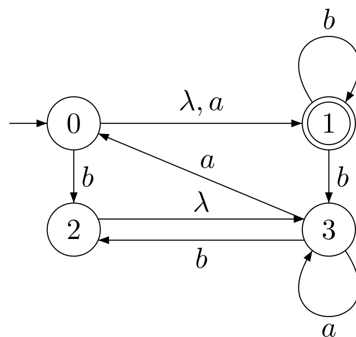
2. Design an NFA that accepts the language of strings over $\{a, b, c\}$ whose length is a multiple of three, and which can be divided into blocks of length three, each of which contains each symbol exactly once.

3. Let M be the NFA



   - Compute the λ-closure of each state.
   - Construct a DFA that is equivalent to M
   - Give a regular expression for $L(M)$.

4. Repeat the preceding question with the following automaton:



5. Build an NFA that accepts $(ab)^*$ and one that accepts $(ba)^*$. Use λ-transitions to combine them into an NFA accepting $(ab)^*(ba)^*$. Convert that NFA to an equivalent DFA.

6. It's very easy to build an NFA that accepts the language HAS-$abba$ (just loop on a or b in the initial state, then a chain of transitions to accept specifically $abba$, and then loop on a or b in the final state). Convert this NFA to a DFA and compare the result to the example constructed in Lecture 6.

1 Introduction

Today we wrap up our treatment of finite state automata by showing that they accept (only and all) regular languages, that these languages have some unexpected closure properties, and with a theorem that allows us to prove that certain languages are not regular.

2 Regular grammars and NFAs

Recall that the rules of a regular grammar are of the form:

$$A \to cB, \ A \to c, \ A \to \lambda.$$

We can actually eliminate rules of the second type by introducing a special state Z, replacing them by $A \to cZ$, and having the only rule for Z be $Z \to \lambda$. Having done that, we can immediately construct an NFA that accepts the same words that a regular grammar generates. Namely, we take its states to be the non-terminals, its initial state to be S, its final states to be all those non-terminals for which there is a rule $X \to \lambda$ and its transitions to be $A \xrightarrow{c} B$ whenever there is a rule $A \to cB$ of the grammar.

We can equally well produce a regular grammar from an NFA and so we obtain:

Theorem 2.1. *The collection of languages generated by regular grammars, and the collection of languages accepted by NFAs (or DFAs) are the same.*

3 Regular languages and NFAs

We know that every regular language is accepted by some NFA. We would like to show that the language accepted by an NFA is always regular. This is accomplished by a reduction technique which generalizes the transitions allowed in an NFA – instead of having transitions labelled by single letters, we'll allow them to be labelled by regular languages.

So, begin with an NFA, M, and as usual assume that it has a unique start state and a unique distinct accepting state. For convenience we'll also add λ-transitions from each

state, other than the initial and accepting states, of M to itself. We will construct a sequence of smaller and smaller NFAs with "extended transitions" that accept the same language as M does. Suppose that we've done this up to some point and there are still states other than the start and finish states remaining. Choose any such state q. Now consider every other pair of states r and s. If there are transitions:

$$r \stackrel{\boldsymbol{u}}{\to} q \quad \text{and} \quad q \stackrel{\boldsymbol{v}}{\to} s$$

then we will add a new transition $r \to s$ as follows, let $\boldsymbol{w}$ be the language on the loop on q and add:

$$r \xrightarrow{\boldsymbol{u}\boldsymbol{w}^*\boldsymbol{v}} s.$$

Having done this (for every pair), delete q. If we wind up with multiple transitions between any two states replace them by a single transition whose label is the union of their labels.

If we carry on like this we will eventually wind up with a single transition from the initial state to the accepting state. Its label is the language accepted by M. Since the steps we used in building labels are steps that are allowed in building regular languages, and since the initial labels are all basic regular languages, $L(M)$ must be regular. Thus we have proved:

Theorem 3.1. *The collections:*

- *regular languages,*
- *languages accepted by NFAs*
- *languages accepted by DFAs*
- *languages generated by a regular grammar*

are all the same.

4 Closure properties of regular languages

From the definition of regular language we know that regular languages are closed under union, concatenation, and Kleene star. However, we can use the preceding theorem to give us a much richer class of closure operations (i.e. many more ways of generating regular languages from simpler ones).

- The complement of a regular language is regular. Because, we can take a DFA accepting the language and then interchange all accepting and non-accepting states. The new automaton accepts the complement.

- The intersection of two regular languages is regular. Because:

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

  and we already have closure under complement and union.

- If L is a regular language, then the collection of all suffixes (or prefixes) of words in L is also a regular language. Because, we can take a DFA for L which we can assume has the property that every state can be reached somehow from the initial state (if there were inaccessible states we could just throw them all away without changing the language). Now add a λ-transition from the initial state to every other state. The resulting automaton accepts suffixes of words in L. A similar construction works for prefixes.

- If L is a regular language, then so is L^r the language of all reversals of words in L. This is probably most easily proven inductively from the recursive definition. It's certainly true for the basic languages and thereafter we have $(L_1 L_2)^r = L_2^r L_1^r$, $(L_1 \cup L_2)^r = L_1^r \cup L_2^r$ and $(L^*)^r = (L^r)^*$.

We could extend this list even further, but the properties above are among the most useful. This shows that the regular languages form a very robust set – we can build lots and lots of them. And yet we believe that some simple languages like $\{a^n b^n \mid n \geq 0\}$ are not regular. It's time to introduce some techniques for proving such results.

5 The pumping lemma

Suppose that L is a regular language. So, we can find some DFA, M that accepts it. This DFA has a certain number of states, call it k. Now consider any word $z \in L$ with $|z| \geq k$. In processing w we will visit a sequence of states:

$$q_0 \to\ q_1 \to\ q_2 \to\ \ldots \to\ q_{|z|-1} \to\ q_{|z|}$$

since this sequence contains at least $k+1$ states, and there are only k states in the automaton, there must be a repetition somewhere – in other words we travelled around a loop in the automaton in processing w. In fact, we can be sure we've found a loop by the time we reach q_k. Now, there's nothing to stop us from taking a short cut and not passing around that loop at all, or indeed from taking the scenic tour and passing around that loop two or more times. In other words, we have proven:

Theorem 5.1 (The pumping lemma)**.** *For any regular language L there is a positive integer k such that if $z \in L$, $|z| \geq k$ then for some u, v and w:*

$$\begin{aligned} z &= u \cdot v \cdot w \\ |u| + |v| &\leq k \\ |v| &> 0 \\ uv^i w &\in L \quad \textit{for all } i \geq 0. \end{aligned}$$

We don't use the pumping lemma to prove that languages are regular: it only tells us certain properties that a regular language must possess. We use it to show that languages *aren't* regular by demonstrating that they don't possess these properties.

6 Applying the pumping lemma

Consider the language $L = \{a^n b^n \mid n \geq 0\}$. We can now show it is not regular. Suppose that it were regular. Then the pumping lemma applies, and we can choose a positive integer k with the properties that it specifies. But now consider $z = a^k b^k \in L$. According to the pumping lemma we can write

$$z = uvw$$

with $|u| + |v| \leq k$ (and various things ...). But, this implies that v consists entirely of a's, say $v = a^t$ for some $t > 0$. Then

$$uv^2w = a^{k+t}b^k \notin L$$

while according to the pumping lemma it should be in L. From this contradiction we conclude that L is not regular.

Consider now the language $L = \{a^p \mid p \text{ is prime}\}$. Again we will use the pumping lemma to show it is not regular. Suppose it were, and let k be as specified. There is a prime $p > k$ so we can write $a^p = uvw$. Suppose again that $v = a^t$. Now

$$uv^iw = a^{p+(i-1)t}$$

in particular if we pick $i = p + 1$:

$$uv^{p+1}w = a^{p+pt} = a^{p(t+1)}.$$

According to the pumping lemma, this word is in L, but $p(t + 1)$ is not prime since $t > 0$, so it is not. Again the contradiction shows that L is not regular.

7 Tutorial problems

1. Use the *pumping lemma* to show that none of the following languages are regular:
   (a) $\mathsf{Even}-\mathsf{Palindrome}$, the set of strings over $\{a, b\}$ of even length that are the same spelled forward or backward.
   (b) $L = \{a^n b^{n+1} \mid n \geq 0\}$.
   (c) $L = \{a^n b^{2n} \mid n \geq 0\}$.
   (d) $L = \{a^n b^m \mid n \leq m\}$.
   (e) $\mathsf{Sum} = \{a^n b a^m b a^{n+m} \mid n, m \geq 0\}$.
   (f) Square, the set of strings a^n whose length is a perfect square.

2. In each of the following cases, give examples of languages L_1 and L_2 over $\{a, b\}$ such that:
   (a) L_1 is regular, L_2 is not, and $L_1 \cup L_2$ is regular.
   (b) L_1 is regular, L_2 is not, and $L_1 \cup L_2$ is not regular.
   (c) L_1 is regular, L_2 is not, and $L_1 \cap L_2$ is regular.
   (d) L_1 is not regular, L_2 is not regular, and $L_1 \cup L_2$ is regular.
   (e) L_1 is not regular and L_1^* is regular.

   (Hint for question 2: keep things simple!)

1 Introduction

One of the things that limits the power of finite state automata is that they have no dynamic memory. All the information they contain must be encoded in the current state, and this is a static resource. Our next class of machines, the *push down automata* remedy this by adding a single stack to the control structure. It turns out that this is still quite limiting – the languages accepted by push down automata are all context free. A *pumping lemma* for context free languages, somewhat more complex than that for regular languages establishes that while we can now finally recognize the $a^n b^n$ language, we're still stuck with $a^n b^n c^n$. In a nutshell, the problem is that the stack memory is "use once" and can't be refreshed.

2 Definitions

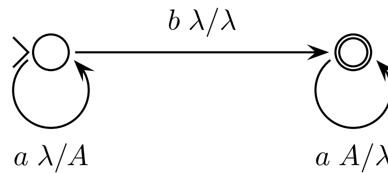
A *push down automaton* is a nondeterministic finite state automaton augmented by a stack. So its parts are:

Q	a finite set of states,
Σ	the input alphabet (lower case letters)
Γ	the stack alphabet (upper case letters)
δ	a transition function
q_0	the initial state
F	the set of accepting states

Some more details about the transition function – it takes as input the current state, an input letter (or λ) and the stack top letter (or λ). It produces as output a set of possibilities (non determinism) each of which consists of a new state and a new stack top (or λ). The interpretation is: consume the input letter (if any), pop the original stack top (if any), move to the new state, and push the new stack top (if any).

A computation proceeds in the normal way following various transitions as allowed by the input (and choosing them arbitrarily if more than one is available). The exact conditions for acceptance are that at the end of the computation no more input should remain, the stack should be empty, and the state should be an accepting one.

Despite their complexity transitions can be represented relatively compactly in diagrams.



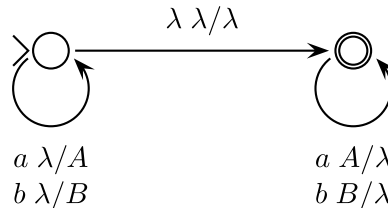
In this machine the three transitions behave as follows:

$a\ \lambda/A$	read a, ignore the stack top, push A
$b\ \lambda/\lambda$	read b, leave the stack alone
$a\ A/\lambda$	read a, pop A from stack top, don't push

To accept, all the pushes that occur in the left state must match pops in the right state, so the language accepted by this push down automaton is $\{a^n b a^n \mid n \geq 0\}$ – certainly a non-regular language.

3 Facts about push down automata

The use of non-determinism is essential for push down automata to gain power. For instance consider the language of even length palindromes over $\{a, b\}$. This is easily accepted by a push down automaton:



But, it's essential that the machine can "guess" when to stop pushing and start popping.

The following result is important, but the proof is technical and uninspiring.

Theorem 3.1. *The languages accepted by push down automata are precisely the languages that have context free grammars.*

4 Pumping lemma for context free languages

For a change, theorem first, then proof as is more traditional.

Theorem 4.1. *Let L be a context free language. There is a positive integer k such that for all $z \in L$ with $|z| \geq k$ we can write $z = uvwxy$ such that:*

$$|vwx| \leq k$$
$$|v| + |x| > 0$$
$$uv^iwx^iy \in L \quad \text{for all } i \geq 0.$$

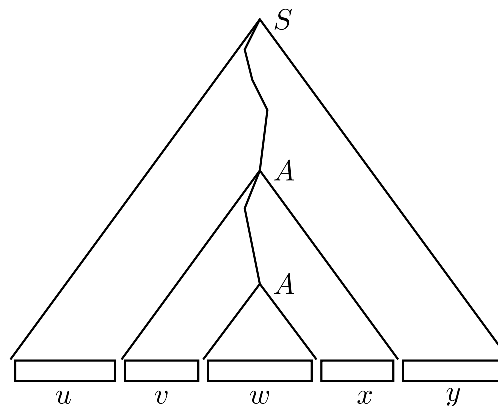
This looks rather like the pumping lemma for regular languages except we need to split the part that is to be pumped into two pieces. The proof will also make use of the "repeated configuration" idea, except in the derivation tree of a word.

Proof. Choose a grammar for L. Each rule is of the form:

$$A \to B_1B_2 \dots B_t$$

where the B_i are either terminals or non-terminals. Let m be the maximum length of the right hand side of a rule, and let j be the total number of non-terminal letters. Choose $k = m^{j+1}$.

Let $z \in L$ be given with $|z| > k$. Since the maximum possible branching factor in its derivation tree is m, the depth of the tree must be greater than $j + 1$. So, some branch of the tree has at least $j + 1$ internal nodes. Since these are labelled with non-terminals, there is a repeated label on this branch, in fact there is a repeated label among the last $j + 1$ internal nodes of this branch. This gives us the situation pictured below:



Now the theorem follows because we can either replace the top A with the bottom one, eliminating v and x, or the bottom one with a duplicate of the top one which gives us an extra repetition of v and x – and this latter construction can be repeated as often as we like. □

As with the pumping lemma for regular languages, this pumping lemma is used to prove that certain languages are not context free. For example, consider the language:

$$L = \{a^n b^n c^n \mid n \geq 0\}.$$

We will show that this is not context free by contradicting the pumping lemma. So, suppose it were context free and choose k as given by the pumping lemma. Let $w = a^k b^k c^k$ and factor $w = uvwxy$ as guaranteed by the pumping lemma. Since $|vwx| \leq k$ the substring vwx contains at most two out of the three characters a, b, and c. But in that case in uv^2wx^2y at least one character occurs exactly k times (one that doesn't belong to vwx) while some other character occurs more than k times (one that does belong to vwx). Hence, $uv^2wx^2y \notin L$ and we have our desired contradiction. So, L cannot be context free.

5 Closure properties for context free languages

The following result is relatively easy to prove either by the manipulation of grammars or of push down automata.

Theorem 5.1. *Let L_1 and L_2 be context free languages, and let R be a regular language. The following languages are all context free: $L_1 \cup L_2$, $L_1 L_2$, L_1^*, and $L_1 \cap R$.*

However, the intersection of two context free languages is not necessarily context free:

$$\{a^n b^n c^k \mid n \geq 0,\ k \geq 0\} \cap \{a^n b^k c^k \mid n \geq 0,\ k \geq 0\} = \{a^n b^n c^n \mid n \geq 0\}.$$

Therefore, the complement of a context free language need not be context free either (if it were, then because unions are, intersections would be too).

6 Tutorial problems

1. Build a PDA to accept each of the following languages:
   (a) Equal, the set of strings having the same number of a's as b's in any order.
   (b) $\{a^n b^n c^m \mid n, m \geq 0\}$.
   (c) BalancedParentheses, the set of strings over $\{(,), a, b\}$ in which the parentheses are properly balanced (and the other symbols can occur arbitrarily).
   (d) (*) PostFix, the set of strings over $\{a, b, +, -\}$ that represent legitimate expressions written in postfix notation, where $+$ is a binary operator, and $-$ a unary operator.

      Briefly, a and b represent values and $+$ and $-$ represent operators. There is a stack to hold values – any input which is a value is pushed onto the stack. Any input which is an operator, causes either one (in the case of $-$) or two (for $+$) values to be popped off the stack – the operator is applied, and the result is pushed back on to the stack. Since we aren't really computing anything for this exercise, this can be simulated by just modifying the stack size appropriately. An expression is legitimate if there are always enough symbols in the stack for any operator that arrives, and after processing it completely, there is exactly one symbol in the stack.

2. Apply the pumping lemma and write out detailed arguments showing that the following languages are not context-free:
   (a) $\{a^n b^m a^n b^m \mid n, m \geq 0\}$.
   (b) $\{a^p \mid p \text{ is prime}\}$.
   (c) $\{a^n b^n a^n \mid n \geq 0\}$.

3. Show, by intersection with a suitable regular language and deriving a contradiction, that Repeat, the set of all words of the form ww, where $w \in \{a, b\}^*$ is not context-free. (NB – look up the page ...)

1 Introduction

Our not entirely unambitious goal in the next few lectures is to try to come to grips with the question:

> What do we mean by the phrase "mechanical computation"? In particular, what are some examples of problems that can, or more interestingly cannot be solved by mechanical computation?

In attacking this problem we will introduce an abstract version of a mechanical computer, the *Turing machine* (TM). The key insight is that, as far as anyone knows, this model and every other "sufficiently powerful" model of mechanical computing solve exactly the same set of problems. This idea is called the *Church-Turing thesis*:

> Any two models of mechanical computation solve the same set of problems.

A reason to believe in the truth of the Church-Turing thesis (aside from the fact that no one has ever found a counterexample) is that one of the things that mechanical computation devices can do is to *simulate* other such devices. If A can simulate B, and B can also simulate A, then their computational powers are equivalent.

2 Definition of Turing Machine

The main thing that will ensure that the model we are working with remains 'mechanical' is that we will insist there only be finitely many possible internal 'states'. However, as in the PDA models we will conceive of a (potentially) infinite memory.

Furthermore, the other main change we will make is to remove the separation between the input (words to be recognised) and the contents of the memory. A simple version of this involves a preprocessing phase of simply copying the entire input to memory and then manipulating it there.

The mental model of the memory that a TM uses is of a tape divided into cells where each cell can hold one symbol. Sitting above the tape is a read-write head. A single atomic step of a TM is:

- Read the current symbol on the tape
- Based on it and the current state,
  - Write a symbol in the current position,
  - Move the read-write head one cell to the right or left,
  - Change state.

So, to specify a TM we simply need to define the set of states and the transition function implicit in the second item above. Inputs to the transition function are pairs consisting of the current state and current symbol, and outputs are triples consisting of new symbol, motion direction, and new state. The transition function will almost always be partial since we want to allow the machine to halt, which happens when there is no transition defined for the current state and symbol pair. It may be deterministic or not – initially we will assume that it is.

The alphabet, Γ of a TM consists of an *input alphabet*, Σ, usually denoted using lower case letters, a special *blank* symbol representing an empty cell, denoted B, and other working or marker symbols denoted by other upper case letters or punctuation marks. The only restriction is that the complete alphabet be finite.

To process a word $w \in \Sigma^*$ we assume that the initial contents of the tape are a blank cell, followed by w (one character per cell), followed by an infinite sequence of blanks. We do not worry about what the machine might do to input of any other form. The TM begins in some initial state START with the read-write head over the leftmost (blank) cell.

We can be fairly flexible about the acceptance criteria, but they must involve the machine halting (i.e. there being no transition defined from the current state when reading the current symbol) and may involve being in an accepting state. One new thing that can happen is that we might crash the machine by trying to move the read-write head off the end of the tape. Such crashes are always considered to be rejecting computations.

3 An example

Consider the language $L = \{wcw \mid w \in \{a,b\}^*\}$ over $\Sigma = \{a,b,c\}$. We know that this cannot be recognised by a PDA. Can it be recognised by a TM? Of course.

Before designing the TM in detail let's consider how it could operate. Basically on an acceptable input (any word over Σ preceded by a blank) we want to read the first character, move forward past the first c and check that the next character matches. Then we need to mark those characters as 'processed' somehow and proceed in much the same fashion with the first unprocessessed character. If we ever find a mismatch we halt and reject. After processing all the characters before the first c, if there are any unprocessed characters remaining after it, we should also reject. Otherwise, we accept.

We can now assemble our machine in tabular form:

State	Read	Write	Move	New state
START	B	B	R	READ
READ	a	X	R	SEEK-a
	b	X	R	SEEK-b
	c	c	R	CHECK
	X	X	R	READ
SEEK-a	a	a	R	SEEK-a
	b	b	R	SEEK-a
	c	c	R	LOOK-a
SEEK-b	a	a	R	SEEK-b
	b	b	R	SEEK-b
	c	c	R	LOOK-b
LOOK-a	X	X	R	LOOK-a
	a	X	L	MOVE-BACK
LOOK-b	X	X	R	LOOK-b
	b	X	L	MOVE-BACK
MOVE-BACK	a	a	L	MOVE-BACK
	b	b	L	MOVE-BACK
	c	c	L	MOVE-BACK
	X	X	L	MOVE-BACK
	B	B	R	READ
CHECK	X	X	R	CHECK
	B	B	R	ACCEPT

Note that there are no transitions defined for ACCEPT so as soon as we enter this state we will halt (and of course, accept). If we halt in any other situation (e.g. seeing a b when in state LOOK-a, or seeing anything other than X or B in CHECK) then we reject.

Of course you should trace the computation of this machine on various valid (though not necessarily acceptable) inputs such as:

$$BBB\cdots,\ BcB\cdots,\ BababB\cdots,\ BacaB\cdots,\ldots$$

4 Tutorial problems

Designing some Turing machines – it's important that you understand these, so if you're stuck – ask! For the simpler cases try to get every detail right – for the more complex one think first about the "big picture" of how the machine works before sweating out the details (but mainly, try to convince yourself that it's possible!)

Design Turing machines to accept the following languages (over $\{a, b\}$ unless obviously otherwise), either by final state or by halting.

1. $(\mathbf{a} \cup \mathbf{b})^*\mathbf{ab}(\mathbf{a} \cup \mathbf{b})^*$
2. $\mathbf{a}^*\mathbf{b}^*$
3. The set of all strings containing an even number of a's and an even number of b's.
4. The set of all palindromes (strings that read the same forwards as backwards).
5. $\{a^n b^n c^n \mid n \geq 0\}$.
6. $\{a^{2^n} \mid n \geq 0\}$.

As a more general problem think of how you could generalise a PDA to a machine with a queue instead of a stack. Is it possible to simulate a PDA with a queue machine? What about a machine with two stacks instead of one? Are these machines equivalent or are some more powerful than others?

1 Introduction

Today's lecture consists mainly of further examples of TMs along with a short discussion concerning halting and acceptance conditions.

2 TMs as function computers

Instead of using TMs as language acceptors we can use them to compute functions. That is, we could say that a TM computes the function f (with domain some subset of Σ^*) if, on input w from its domain it halts with the contents of the tape being $f(w)$ (or possibly $w\#f(w)$). For input not in the domain of f we would generally require that the machine should crash or run forever (signifying the fact that f is not defined there).

As an example consider the problem "compute 2^k". In other words we want to design a TM that on input a^k produces an output tape of $a^k\#b^{2^k}$. In designing this machine it's useful (as usual) to think in slightly higher level terms first:

- Find the end of input and replace it with $\#$, followed by b
- For each a in the input, double the number of b's past the $\#$

How does a "double the b's" module work? Take each b, replace it by X, and add an X at the end. When no more b's are left, change all the X's to b's.

Let's see if that is enough to put together a transition table:

State	Read	Write	Move	New State
START	B	B	R	FIND-END
FIND-END	a	a	R	FIND-END
	B	$\#$	R	ADD-b
ADD-b	B	b	L	FIND-a
FIND-a	B	B	R	HALT-ACCEPT
	a	A	R	START-DOUBLE
	b	b	L	FIND-a
	A	A	L	FIND-a
	$\#$	$\#$	L	FIND-a
START-DOUBLE	A	A	R	START-DOUBLE
	$\#$	$\#$	R	FIND-b
FIND-b	b	X	R	ADD-X
	X	X	R	FIND-b
	B	B	L	CHANGE-X-TO-b
ADD-X	b	b	R	ADD-X
	X	X	R	ADD-X
	B	X	L	FIND-$\#$
CHANGE-X-TO-b	X	b	L	CHANGE-X-TO-b
	$\#$	$\#$	L	FIND-a
FIND-$\#$	b	b	L	FIND-$\#$
	X	X	L	FIND-$\#$
	$\#$	$\#$	R	FIND-b

This simple version of the machine halts in some abnormal way on 'bad' input, but we can't really recognize that (unless we agree that only HALT-ACCEPT is an accepting state). However, we could fix this by sending it into a loop or crash whenever some unexpected input was received. For instance, if in FIND-END we see something other than an a or B we could add a transition to CRASH. The transitions from CRASH are 'whatever you see leave it alone and move left, staying in CRASH'. This will eventually drive the head off the left hand end of the tape resulting in a crash. If we prefer to leave the computer looping we could move right instead (forever).

3 Accepting variations

The machines we've looked at so far accept by halting in some designated accepting state(s). Is this really necessary? Can we dispense with the need for specifying accepting states? Yes we can.

Theorem 3.1. *The collections of languages accepted by Turing machines with designated final states and those accepted by Turing machines by halting are the same.*

One direction is clear since if we have a language accepted by halting we can simply designate all the states as accepting states. The problem is to show how to transform a

machine in which "abnormal" i.e. non-accepting halts might occur into one which crashes or loops instead. But, we already saw how to do this in the example above. Simply add a new state CRASH which causes leftward movement (and remains in CRASH) on any input. Then, take any non-defined transition which represents an abnormal halt, and replace it by one which enters the CRASH state.

4 Testing primality

To begin to establish the idea that perhaps TMs are rather powerful computing device, let's ask if we can recognise the language PRIME consisting of all strings a^p where p is a prime number. The purpose of this example is not so much to show that it can be done, but rather to illustrate that the time has definitely come to extend the allowed operation of our 'mechanical computing devices' to make this sort of thing easier!

The basic idea of testing a number n for primality is: 'for each possible factor, k, determine whether or not k is a divisor of n'. If no such test produces a factor then the number is prime.

We can see how to do this using some basic ideas that we've already explored. Start with an input tape containing a^n. First, transform that to $a^n\#X^2$. The idea in general now is starting from a tape in the form $a^n\#X^k$ we're going to check whether or not k is a factor of n. We do this by 'pairing off a's (changing them to A's) and X's (changing them to Y's). So, while there are still a's and X's on the tape change an X to Y and an a to A. When we can't do this any more:

- If there are no a's left, but still some X's, then n was not a multiple of k. Change all the A's back to a's, all the Y's to X's and add one more X. Repeat.
- If there are no X's left, but still some a's, then change all the Y's back to X's and start pairing X's with a's again.
- If there are neither X's nor a's left, then we found a factor. Check whether this factor is n or something smaller. In the former case, accept, and in the latter reject.

5 Tutorial problems

Design Turing machines to accept the following languages either by final state or by halting. We will generally use the symbol # as a divider between parts of the input. Remember that you may use (a finite set of) additional symbols to write on the tape if that is convenient.

1. The language consisting of all strings of the form $w\#c$ where $w \in \{a, b\}^*$ is non-empty, and c is the first character of w.
2. The language consisting of all strings of the form $w\#v$ where the length of v is greater than the length of w.
3. Modify the design of the 'compute 2^k' machine into a 'compute k^2' machine.
4. Like ordinary computers, Turing machines can also be thought of as machines that transform input into output. Design a Turing machine which, on input $w \in \{a, b\}^*$ halts with the tape showing $a^k b^l$, where k is the number of a's in w, and l is the number of b's (i.e. it sorts w.)

1 Introduction

In this lecture we consider various additions to the capacity of TMs. Perhaps surprisingly, we will be able to show that none of them actually change the languages we can recognise, or the functions that we can compute.

However, this is actually helpful in at least two ways:

- To show that a problem is mechanically solvable we can use a more powerful and convenient extended machine; and
- To show that a problem is not mechanically solvable we can restrict ourselves to ordinary TMs.

2 Multi track machines

In a multi-track machine, instead of one tape we have several. There is still a single read-write head which simultaneously reads aligned symbols on the various tapes (and can write on them all, before moving in the same direction on all of them).

But, thinking of the column of letters that the read-write head sees as a single letter we see that this is really just an ordinary TM with an expanded alphabet (Γ^k where k is the number of tracks instead of Γ). So, adding multiple tracks does not change the power of a TM.

3 Two way tapes

In a two way tape machine we assume that the tape is infinite in both directions (i.e. the tape cells can be indexed by integers rather than by natural numbers).

Given a two way tape machine, imagine wrapping the tape around (duplicating the 0 cell) so that cell -1 lies above cell 1, cell -2 above cell 2 and so on. The extra copy of the 0 cell (in the upper tape) has a special symbol # written on it. Now simulate the original machine by a new two track machine here by duplicating each state q of the original into a pair q^{upper} and q^{lower}. Then each transition of the original machine can be transformed into transitions of the new one. For instance if we have a transition between q and s that reads a, writes b and moves left, it becomes:

State	Read	Write	Move	New state
q^{lower}	ax	bx	L	s^{lower}
q^{lower}	$a\#$	$b\#$	R	s^{upper}
q^{upper}	xa	xb	R	s^{upper}

For each possible symbol x on the other tape.

This transformation shows the general strategy we need to pursue – take an instance of

the more powerful machine and show how to build one of a simpler type whose computations simulate it.

4 Multi tape machines

In a multi tape machine we again have a number of separate tapes. Each tape has its own read-write head and these can move independently of one another (or indeed stay in place). Transitions are based on the complete sequence of symbols read at any one time, as well as the current state – that is, there is one central control unit which takes charge of all the different heads simultaneously.

Simulating this using a multi track machine is a little bit complicated. Suppose that there are k tapes. We will use $2k + 1$ tracks. One of these is pretty much for decoration. It has a # in the leftmost cell and nothing else. Its only purpose is to allow us to reliably wind the head back to the leftmost cell.

For each of the k tapes of the multi tape machine we dedicate two tracks. One of these contains the actual contents of the tape. The other contains a single symbol M which marks where the 'virtual' version of the read-write head for that tape is currently located.

Now the simulation of a single transition in the multi tape machine requires a long sequence of operations in the multi track machine. We always assume that the read write head begins at the leftmost cell. One by one we advance to the various marked cells and determine the symbols on each 'virtual' tape, storing these by means of state (this is possible because there are only finitely many possibilities for the symbols we see). Then we rewind back to the left again. Now we determine what should be written on each tape and which way its head should be moved (by means of the original transition table in the multi tape machine). Again, we advance to each of the marks, change the symbol on the virtual copy, and move the mark as necessary. When all this is done, we go back to the left (remembering the new state), and start again.

5 Non determinism

Non determinism in TMs comes in two apparently different flavours.

The first, non determinism by transition (NDT for local reference) is just like non determinism in finite state or push down automata. That is, given a current state, and a current symbol, there may be more than one possibility for the read write action, movement direction, or resulting state. Basically our machine is 'loose' and might take one of several actions in certain configurations. An input word w is accepted by such a machine if *some* choice of those actions results in an accepting computation.

The second, non determinism by oracle tape (NDO) introduces a new idea. We imagine a standard two tape deterministic TM. One tape is called the input tape and on it we write the input word w that we wish to process. Just before setting the computation in motion

we summon a genie and say 'Oh most clever and puissant genie, it would be our dearest wish if you would be so kind as to write upon this second tape some helpful guidance for our computation'. Since we were so polite[1], the genie then writes some information on the oracle tape. We then run the machine and accept w if it halts in an accepting state. Since we don't *entirely* trust the genie it is our responsibility to have designed the machine in such a way that the set of words we want to accept is precisely the set of words that will be accepted for *some* string on the oracle tape. That is, while the genie can ensure that we accept the words we want, we can never be tricked into accepting a word that we don't want.

Despite their apparent differences, these two models accept the same languages – which is useful as the second one is *much* easier to work with in a theoretical context (as well as allowing for all sorts of clever variations where we might restrict access to the oracle tape in some way, or the types of sequences that can be written on it, . . .)

Theorem 5.1. *If a language, L, is accepted by an NDT Turing machine, T, then it is accepted by an NDO Turing machine O and vice versa.*

Proof. Suppose first that L is accepted by an NDT machine T. Create a two tape machine O where on the oracle tape the head always moves one place to the right with each transition. Let m be the maximum degree of non-determinism in T (i.e. the maximum number of transitions associated with a single state-symbol pair) and take the alphabet of the oracle tape to be $\{1, 2, \ldots, m\}$. Now simply make each of the non deterministic transitions deterministic by indexing them from 1 to (at most) m, using the contents of the oracle tape to justify the choice. Given an acceptable word w, the genie simply chooses an accepting computation in T and writes the appropriate indices on the oracle tape meaning that O accepts w. However, whatever the genie writes on the oracle tape leads to some computation path in T, so we can never be forced to accept a word that T does not accept.

Now suppose that L is accepted by an NDO machine O. We may assume that O works as in the previous paragraph (i.e. each step advances the read-write head on the oracle tape one step to the right). This is because the contents of the oracle tape are completely under the genie's control. While it might be more convenient to allow any sort of movement on that tape, it can't hurt to simply write out in order all the symbols that will ever be seen there instead (and for a powerful genie, questions of convenience don't really enter into it). Now we produce our NDT T simply by the reverse of the preceding construction – just erase all references to the second tape from the transitions. If T accepts a word w then we can choose an accepting computation, 'remember' which transitions involving the oracle tape were used, and construct an input sequence for the oracle tape that shows that O accepts w. On the other hand if O accepts w then we can just ignore the oracle tape and 'see' an accepting computation of T on w. So, T and O accept the same language. □

[1]For reasons that should be obvious, it is always advisable to be polite to powerful supernatural entities.

6 Tutorial problems

Use multiple tracks, tapes, or other variations on the Turing theme to find machines that accept the following languages (or compute certain functions). Again, worry more about the high level description, and understanding how, while these variations may improve efficiency, their computations could all be accomplished by a standard TM

1. Convert binary numbers to unary. That is, on input $w \in \{0, 1\}^*$ arrange output on a working tape of a^n where n is the value of w interpreted as a binary number.
2. Convert unary numbers to binary ones.
3. Accept the language $L = \{a^p \mid p \text{ is prime}\}$.
   - Think about the standard loop version "for each $2 \leq i < p$ check whether the remainder when p divided by i is 0. If so, reject. If all these tests succeed, accept.
   - There is an improvement to this method where you only look at $i \leq \sqrt{p}$ since if a number is composite it has a proper factor less than or equal to its square root – how might that be implemented?
   - Does using non-determinism seem to help for this problem? What about for the complement of L, i.e. the set of strings a^n where n is composite?

1 Introduction

In this lecture we introduce two classes of languages associated with acceptance by Turing machines. We also consider a *universal Turing machine*, the closest thing to a 'real' computer that we have seen so far.

2 Recursive and recursively enumerable languages

A language, L is *recursively enumerable* if it is the language accepted by some Turing machine M. That is, $w \in L$ if and only if M halts on input w in an accepting state. Note that if we process some word $v \notin L$ with M then M might halt in a non-accepting state, might crash, or might simply run forever. This situation is rather unsatisfactory – we know that if a 'yes' answer is coming it will arrive eventually, but if the answer is 'no' we may never be sure.

That makes it natural to define a (possibly) stronger notion: a language, L, is *recursive* if it is the language accepted by some Turing machine M that *halts on all inputs*. Now we are guaranteed a yes or no answer.

Theorem 2.1. *If both L and $\Sigma^* \setminus L$ are recursively enumerable, then L is recursive.*

Proof. Let M_{in} accept L and M_{out} accept $\Sigma^* \setminus L$. We use these two machines to put together a multi tape machine. Given an input w we first copy w onto another tape. Then we run M_{in} on one tape in parallel with M_{out} on the other (the states of this new machine correspond to pairs of states of the two given machines, so there are still only finitely many of them). As soon as one of the two simulations halts, we can halt and announce the status of w. Since one of the two simulations is guaranteed to halt (M_{in} if $w \in L$ and M_{out} otherwise) this is sufficient. □

3 Language enumerators

There are certainly circumstances in which we might be more interested in making a list of the elements of a language L rather than recognizing membership in L (for instance, a list of primes, a list of possible solutions to some optimization problem, ...). A machine M is said to *enumerate* L if it produces on some output tape (which always moves to the right, though not necessarily at every computation step) a sequence:

$$u_1 \# u_2 \# u_3 \# \cdots$$

where # is an extra symbol (not from the alphabet of L) and $w \in L$ if and only if $w = u_i$ for some i. In other words, the machine just lists the elements of L in some order, and possibly with repetitions, but every element gets listed eventually.

Proposition 3.1. *If there is a TM that enumerates L, then there is a TM that enumerates L without repetitions.*

Proof. Take a TM enumerating L and augment it with another output tape – the 'real' output tape. Each time M writes a word u_k on its output tape, pause M and use a separate control module to check whether $u_k = u_j$ for some $j < k$. If so, just restart M. If not, write u_k to the 'real' output tape (followed by a #) and then restart M. □

Of course the main result (and the reason for the name 'recursively enumerable') is:

Theorem 3.2. *A language L is enumerated by some TM if and only if it is recursively enumerable.*

Proof. Suppose that L is enumerated by M. To build a machine T that accepts M simply add an extra input tape on which an input word w is written. Then start M. Each time M produces a new word u_i check whether $u_i = w$. If so, halt and accept. If not, carry on. Thus, if $w \in L$ we halt and accept, and otherwise we do not.

Now suppose that L is recursively enumerable, accepted by some machine T. To build an enumerator M we use the 'simulate for a fixed number of steps on each possible input of a fixed length' idea. That is, we build a machine M that successively runs T for k steps on all input sequences of length at most k for k from 1 to $\cdots$. Each time we spot an acceptance we add it to the output tape. Thus, we eventually add all the words that T accepts to the output tape and no others. □

The problem with language enumerators (as with recursively enumerable languages) is that if we haven't seen a word yet we have no indication whether we ever will. If we could enumerate the strings of a language in order of their length, then that would be a different story – once we saw a string longer than the one we were interested in we would know that waiting any longer would be a forlorn hope.

Say that L *can be enumerated by length* if there is an enumerator for L which first lists all the strings of length 0 in L, then all the strings of length 1, then of length 2, and so on. Not surprisingly:

Theorem 3.3. *A language L can be enumerated by length if and only if it is recursive.*

Proof. If L is recursive, take a machine M that accepts it and halts on all inputs. Run it on each string of length 0, then each string of length 1 and so on. Each time a string is accepted, add it to an output tape. The resulting machine enumerates L by length.

Conversely, suppose that L can be enumerated by length. If L is finite then it is regular and so certainly recursive, and there is nothing to prove. Otherwise, take the enumerator by length and a desired input word w. Wait until either w or some longer word appears, and accept or reject accordingly. □

4 Representing machines as strings

In several of the preceding arguments we have been piecing together new machines from old in an *ad hoc* fashion as required. There's nothing wrong with that, but effectively we're treating machines as modules here – and it would seem sensible to perhaps represent them in some uniform fashion (then we might feel more comfortable about adding certain operations 'erase the working tape and write the first string of length 6 on it, then run M for 6 steps')

A first step in that process is to come up with a representation of (basic) TMs as strings – and there's nothing particularly special about the representation we might choose – in practice, anything vaguely sensible will do. Since we're worried more about theoretical rather than practical arguments we'll just take a very simple representation.

We'll use a set alphabet of $\{0, 1\}$. The 0 symbol will be used only as punctuation, so we'll effectively encode everything else in unary. What do we need to encode?

- An actual alphabet Γ. Call the elements of Γ, a_1 through a_n and encode a_i as 1^i (a string of i 1's).
- A set of states. Call the states q_1 through q_m and encode q_j as 1^j
- The directions. Call L, 1 and R, 11.
- The transitions. Transitions can be thought of as 5-tuples $(q_{\text{start}}, a_{\text{read}}, a_{\text{write}}, d_{\text{move}}, q_{\text{finish}})$. So, simply encode these as the encodings of the individual parts separated from one another by single 0's.
- The complete set of transitions. Take each transition, encode it, and separate it from the next by 00.
- The start/finish of the encoding. Add a 000 at the beginning and at the end.

For convenience, having done all this to a Turing machine M we'll call the resulting string the *representation* of M and denote it by $R(M)$.

5 A universal Turing machine

Now that we can represent a Turing machine by a string (think 'program') we can design a so called *universal Turing machine*, U. It's easiest to think of U as a multi tape machine. Its input or program tape is initially assumed to be of the form $R(M)w$ where M is a TM and w an input word. The job that U is meant to do is to simulate the operation of M on w. This is easy enough. First, w is copied to some working tape. Then a third tape, the state register, is initialized to '1', representing the 'current state' (assumed to be q_1 of M). Now operation proper can begin – the program tape is scanned for a transition from the

current state using the current symbol of w. Once that is found, an appropriate update is made to the working tape and the state register. Then the next step is simulated ...

If M halts on w then U will halt on $R(M)w$. In particular, if we assume that U accepts by halting, the language accepted by U is:

$$\text{HALT} = \{R(M)w \mid M \text{ halts on } w\}.$$

So, the language HALT is recursively enumerable.

This makes sense – we can recognise *positive* instances of the problem "does M halt on w" (where both M and w are parameters) simply by simulating M and answering 'yes' if it halts. Of greater interest is whether we can also somehow recognise *negative* instances of the same problem.

6 Tutorial problems

1. Consider the language $\{a^n \mid n \text{ is composite}\}$.

   - Describe a simple mechanism for accepting this language on a normal two tape Turing machine, with an input tape and an oracle tape.
   - Do the same, but with a "non deterministic transitions" two tape machine.

2. Consider the following problem: given two input tapes, one containing a string b^n, the other containing input of the form $a^{m_1}\#a^{m_2}\#\cdots\#a^{m_k}$ (i.e. a sequence of blocks of a's), determine whether some subset of the blocks of a's is of total length n (this is called the SUBSET-SUM problem). Show that a three tape TM with oracle tape makes this problem almost trivial. Likewise model it in a TM with non deterministic transitions. Finally, consider the difficulties in dealing with it deterministically.

3. (Review) Remember that regular languages form a subset of the context free languages, which in turn form a subset of recursive languages. To prove that a language is regular, we can either design a finite state automata that accepts it, or show that it is generated by a regular grammar or a regular expression. To prove that a language is not regular, we usually use the pumping lemma for regular languages. Similarly, to prove that a language is context free, we either show that it is accepted by a push down automata, or give a context free grammar for it. Again, to prove a language is not context free, we use the pumping lemma for a context free languages. To prove that a language is recursive, we design a TM which halts on all inputs and accepts it. So far, we have no mechanisms for proving that a language is not recursive. For each of the following languages determine exactly which type it is, and prove it (so, for instance if you are claiming it is context-free, you should both give a grammar/PDA and an argument that it is not regular.)

   (a) $L = \{a^{3k}ba^{2l} \mid 0 \leq k, l\}$

   (b) $L = \{a^{3k}ba^{2k} \mid 0 \leq k\}$

   (c) $L = \{a^{k^2}ba^{k} \mid 0 \leq k\}$