Lecture 2: Strings, Languages, DFAs

24 January 2010

This lecture covers material on strings and languages from Sipser chapter 0. Also, this lecture covers an account of countable and uncountable sets, and shows that C-programs cannot decide all languages.

1 Alphabets, strings, and languages

1.1 Alphabets

An ***alphabet*** is any *finite* set of characters.

Here are some examples for such alphabets:

(i) $\{\mathtt{0}, \mathtt{1}\}$.

(ii) $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

(iii) $\{\mathtt{0}, \mathtt{1}, \#\}$.

(iv) $\{\mathtt{a}, ...\mathtt{z}, \mathtt{A}, ...\mathtt{Z}\}$: all the letters in the English language.

(v) ASCII - this is the standard encoding schemes used by computers mappings bytes (i.e., integers in the range 0..255) to characters. As such, `a` is 65, and the space character ␣ is 32.

(vi) $\{\mathtt{moveforward}, \mathtt{moveback}, \mathtt{rotate90}, \mathtt{reset}\}$.

1.2 Strings

This section should be recapping stuff already seen in discussion section 1.

A ***string*** over an alphabet Σ is a *finite* sequence of characters from Σ.

Some sample strings with alphabet (say) $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ are `abc`, `baba`, and `aaaabbbbccc`.

The ***length*** of a string x is the number of characters in x, and it is denoted by $|x|$. Thus, the length of the string $w = \mathtt{abcdef}$ is $|w| = 6$.

The ***empty string*** is denoted by ϵ, and it (of course) has length 0. The empty string is the string containing zero characters in it.

The ***concatenation*** of two strings x and w is denoted by xw, and it is the string formed by the string x followed by the string w. As a concrete example, consider $x = \mathtt{cat}$, $w = \mathtt{nip}$ and the concatenated strings $xw = \mathtt{catnip}$ and $wx = \mathtt{nipcat}$.

Naturally, concatenating with the empty string results in no change in the string. Formally, for any string x, we have that $x\epsilon = x$. As such $\epsilon\epsilon = \epsilon$.

For a string w, the string x is a ***substring*** of w if the string x appears contiguously in w.

As such, for $w =$abcdef
we have that bcd is a substring of w,
but ace is not a substring of w.

A string x is a ***suffix*** of w if its a substring of w appearing in the end of w. Similarly, y is a ***prefix*** of w if y is a substring of w appearing in the beginning of w.

As such, for $w =$abcdef
we have that abc is a prefix of w,
and def is a suffix of w.

Here is a formal definition of prefix and substring.

Definition 1.1 The string x is a ***prefix*** of a string w, if there exists a string z, such that $w = xz$.

Similarly, x is a substring of w if there exist strings y and z such that $w = yxz$.

1.3 Languages

A ***language*** is a set of strings. One special language is Σ^*, which is the set of all possible strings generated over the alphabet Σ^*. For example, if

$$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \qquad \text{then} \qquad \Sigma^* = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{aa}, \mathtt{ab}, \mathtt{ac}, \mathtt{ba}, \ldots, \mathtt{aaaaaabbbaababa}, \ldots\}.$$

Namely, Σ^* is the "full" language made of characters of Σ. Naturally, any language over Σ is going to be a subset of Σ^*.

Example 1.2 The following is a language

$$L = \{\mathtt{b}, \mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, ...\}.$$

Now, is the following a language?

$$\{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \epsilon\}.$$

Sure – it is not a very "interesting" language because its finite, but its definitely a language.

How about $\{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \emptyset\}$. Is this a language? No! Because $\emptyset$ is no a valid string (which comes to demonstrate that the empty word ϵ and $\emptyset$ are not the same creature, and they should be treated differently.

Lexicographic ordering of a set of strings is an ordering of strings that have shorter strings first, and sort the strings alphabetically within each length. Naturally, we assume that we have an order on the given alphabet.

Thus, for $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, the Lexicographic ordering of Σ^* is

$$\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \mathtt{aab}, \ldots.$$

1.3.1 Languages and set notation

Most of the time it would be more useful to use set notations to define a language; that is, define a language by the property the strings in this language posses.

For example, consider the following set of strings

$$L_1 = \left\{ x \,\middle|\, x \in \{\texttt{a}, \texttt{b}\}^* \text{ and } |x| \text{ is even } \right\}.$$

In words, L_1 is the language of all strings made out of $\texttt{a}, \texttt{b}$ that have even length.

Next, consider the following set

$$L_2 = \left\{ x \,\middle|\, \text{ there is a } w \text{ such that } xw = \texttt{illinois} \right\}.$$

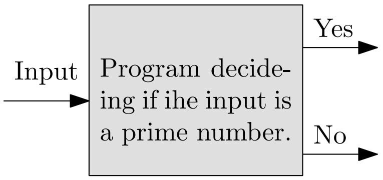
So L_2 is the language made out of all prefixes of L_2. We can write L_2 explicitly, but its tedious. Indeed,

$$L_2 = \{\epsilon, \texttt{i}, \texttt{il}, \texttt{ill}, \texttt{illi}, \texttt{illin}, \texttt{illino}, \texttt{illinoi}, \texttt{illinois}\}.$$

1.3.2 Why should we care about languages?

Consider the language L_{primes} that contains all strings over $\Sigma = \{0, 1, \ldots, 9\}$ which are prime numbers. If we can build a fast computer program (or an automata) that can tell us whether a string s (i.e., a number) is in L_{primes}, then we decide if a number is prime or not. And this is a very useful program to have, since most encryption schemes currently used by computers (i.e., RSA) rely on the ability to find very large prime numbers.

Let us state it explicitly: The ability to decide if a word is in a specific language (like L_{primes}) is equivalent to performing a computational task (which might be extremely non-trivial). You can think about this schematically, as a program that gets as input a number (i.e., string made out of digits), and decides if it is prime or not. If the input is a prime number, it outputs `Yes` and otherwise it outputs `No`. See figure on the right.



1.4 Strings and programs

An text file (i.e., source code of a program) is a long one dimensional string with special ⟨`NL`⟩ (i.e., newline) characters that instruct the computer how to display the file on the screen. That is, the special ⟨`NL`⟩ characters instruct the computer to start a new line. Thus, the text file

```
if x=y then
  jump up and down and scream.
```

Is in fact encoded on the computer as the string

`if␣x=y␣then⟨NL⟩␣␣jump␣up␣and␣down␣and␣scream.`

Here, ␣ denote the special space character and ⟨`NL`⟩ is the new-line character.

It would be sometime useful to use similar "complicated" encoding schemes, with subparts separated by # or \$ rather than by ⟨`NL`⟩.

Program input and output can be consider to be files. So a standard program can be taught of as a function that maps strings to strings.[1] That is $P : \Sigma^* \to \Sigma^*$. Most machines in this class map input strings to two outputs: "yes" and "no". A few automatas and most real-world machines produce more complex output.

2 Countable and uncountable sets

The notion of cardinality of finite sets is known to you. For example, most sensible people will agree that the set $\{a, b, c\}$ is of the same *cardinality* (or size) as the set $\{x, y, z\}$. Why? Because, you would say, both elements have 3 elements.

Now, suppose I told you that I don't like/know numbers. Can you explain why the two sets above are of the same cardinality, without using numbers?

Aside: In fact, I have noticed that teaching numbers to little children is hard to motivate. Why should they learn to count? Here is a simple motivation. If you give the kid 4 pieces of candy, and asked her to distribute among 5 friends, you'll see perplexion (unless, of course, she decides there are too few, and she will have it all herself). But you could argue that one way to figure out whether you have enough, is to *count* (using numbers) the number of pieces of candy and people.

So, how do we argue that $\{a, b, c\}$ and $\{x, y, z\}$ have the same cardinality, without using numbers? A simple way is through a 1-1 correspondence: there is a 1-1 correspondence between the two sets, for example f that associates a to y, b to x, and c to z. So we could say two sets have the same cardinality if there is a 1-1 correspondence between them.

Aside: Notice that in motivating the child to learn numbers, above, the real problem was to see whether there is a 1-1 correspondence between friends and pieces of candy— one candy for each friend.

The remarkable property of the above definition is that it extends to *infinite* sets, and gives an interesting way to see that two infinite sets may have *different* cardinality. This study was set forth by Georg Cantor (1845-1918).

An infinite set A is said to have the same cardinality as that of B, denoted $|A| = |B|$, if there is a function $f : A \to B$ that is a 1-1 correspondence (i.e. injective and surjective) between A and B.

For example, consider $\mathbb{N} = \{1, 2, 3, \ldots\}$ and the set of all even numbers $Even = \{2, 4, 6, \ldots\}$. Then $\mathbb{N}$ and $Even$ have the same cardinality, i.e. $|\mathbb{N}| = |Even|$, since the function $f : \mathbb{N} \to Even$, defined as $f(n) = 2n$, for every $n \in \mathbb{N}$, is a 1-1 correspondence.

An infinite set A is said to be ***countable*** if there is a 1-1 correspondence between $\mathbb{N}$ and A. (Eg. $Even$ is countable.)

Intuitively, a set A is countable, if you can lay out the elements of A as $a_1, a_2, a_3, \ldots,$ and this list will cover all of A. In other words, you can say "a_1 is the first element, a_2 is the second element, a_3 is the third, ..." and lay out the entire set A. It's tempting to think that all infinite sets are countable— but this is not true, as we will show below.

[1] Here, we are considering simple programs that just read some input, and print out output, without fancy windows and stuff like that.

Before we show that, here are a few easy things to show:

Theorem 2.1 *If an infinite set A is countable, and $B \subseteq A$ and B is infinite, then B is countable as well.*

Theorem 2.2 *If A and B are countable infinite sets, then $A \times B$ is also countable.*

The above can be shown as follows. If A and B are countable, then we can lay out A as $\{a_1, a_2, \ldots\}$ and $B = \{b_1, b_2, \ldots\}$. Now $A \times B$ can be laid out as

$$(a_1, b_1), (a_2, b_1), (a_1, b_2), (a_1, b_3), (a_2, b_2), (a_3, b_1), (a_1, b_4), \ldots$$

Intuitively, we lay out all (a_i, a_j) such that $i + j = n$, for increasing values of n. If you draw this on a table, you'll see this as exploring larger and larger diagonals. Clearly this will cover all elements of $A \times B$— every element of $A \times B$ will occur at some point in this ordering.

We can also show that the set of all *finite strings* over a finite alphabet Σ is countable. For example, let $\Sigma = \{0, 1\}$. We can show that Σ^* is countable, using the lexicographic ordering over Σ^*. Fixing an ordering on Σ (say $0 < 1$), we can lay down the elements of Σ^* as $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Intuitively, we lay down the words in increasing order of length, and for each length, we define the ordering on words of that length in lex ordering (dictionary order). We won't define the ordering formally, but it is clear that it can be defined, and will cover the whole of Σ^*.

Uncountability: A set is *uncountable* if it is not countable.

Let us now consider *infinite strings* over a finite alphabet Σ. An infinite string is just an infinite sequence of letters in Σ: e.g. if $\Sigma = \{a, b\}$, then

$$abbaabaabbababababbbababababbabbbabab\ldots\ldots\ldots$$

is an infinite string. Let us now show that the set of all infinite sequences over Σ (let's denote this as Σ^∞) is uncountable, i.e. there is no way to lay down all the infinite sequences as "this is the first sequence, this is the second, etc.".

The proof works by contradiction, and is due to a technique called *diagonalization* by Cantor. Let us assume $\Sigma = \{a, b\}$ (the proof is similar for larger alphabets; note that if there is only one letter in Σ, then there is only one infinite string). Assume that the set of all infinite strings was countable, by way of contradiction, and let $f : \mathbb{N} \to \Sigma^\infty$ be a 1-1 correspondence.

We can view this function f as the following table, where each row denotes an infinite string $f(i)$ for a particular i, and each column represents a particular position in the sequence:

	1	2	3	4	...
$f(1)$	**a**	b	b	a	...
$f(2)$	a	**b**	a	a	...
$f(3)$	b	a	**b**	b	...
$f(4)$	a	b	a	**a**	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Now we are going to consider the *diagonal word* ($abba\ldots$) in the above table, and flip it (to get $baab\ldots$ for the above table). More formally, we consider the infinite sequence $s = x_1, x_2, x_3, \ldots$, where $x_i = a$ if $f(i)[i] = b$ and $x_i = b$ if $f(i)[i] = a$. Here $f(i)[i]$ refers to "the i'th letter in the i'th string". Intuitively, we are taking the *diagonal* infinite word and flipping it, changing a's to b's and b's to a's. Then we claim that this infinite word s does not occur in the range of f. This is easy to show. For any $j \in \mathbb{N}$, note that $f(j) \neq s$ as $f(j)[j] \neq s[j]$. In other words, $f(j)$ and s differ from each other at least at the j'th letter (as the j'th letter of s was the obtained by flipping the j'th letter of $f(j)$). Hence f is *not* a 1-1 correspondence between $\mathbb{N}$ and Σ^∞, and hence Σ^{infty} is uncountable.

The set of infinite sequences being uncountable has many consequences. For example, we can use a similar proof as above to show that the set of all real numbers is uncountable.

The set of all languages is uncountable: Note that a language over Σ, $L \subseteq \Sigma^*$, can be seen as an infinite sequence over $\{0, 1\}$. First, we know that Σ^* is *countable*— let's say we order it as $w_1, w_2, w_3, \ldots$. Now, let's form an infinite sequence for a language L as follows: $\alpha_L = b_1 b_2 b_3 \ldots$ where $b_i = 1$ if $w_i \in L$, and $b_i = 0$ if $w_i \notin L$, for every $i \in \mathbb{N}$. It is easy to see that every language corresponds to a unique infinite sequence and every infinite sequence corresponds to a unique language. Hence there is a 1-1 correspondence between the class of all languages and the class of all infinite strings over $\{0, 1\}$. Hence the class of all languages over any alphabet Σ (even a singleton alphabet) is also *uncountable*.

Programs cannot decide all languages: Let us now consider the class of all C-programs that take in an input string and output "YES" or "NO". A C-program hence defines a *language* over an alphaber (ASCII alphabet).

Also, note that a C-program is, after all, a finite string (written in ASCII), and since the set of all finite strings over an alphabet is countable, the class of all C programs is countable.

Since every C program accepts some language, and since the class of C-programs is countable, it is obvious that the class of languags accepted by C-programs is also countable. Since the class of all languages is *uncountable*, the class of C-programs cannot capture every language! In other words, there is a language (in fact, uncountably many) that cannot be decided by *any* C-program!

Note that this remarkable fact follows simply by counting arguments— no matter how programs are written, as long as they are of finite length over a finite fixed alphabet, they cannot capture all languages.

Later in the course, we will look at *particular* problems that no C-program can solve. This will be more interesting, as it will show that concrete and interesting problems, which *we would like to solve,* are unsolvable.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 3: More on DFAs

26 January 2010

This lecture continues with material from section 1.1 of Sipser.

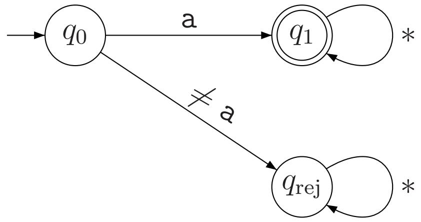
1 JFLAP demo

Go to `http://www.jflap.org`. Run the applet ("Try applet" near the bottom of the menu on the lefthand side). Construct some small DFA and run a few concrete examples through it.

2 State machines

2.1 A simple automata

Here is a simple ***state machine*** (i.e., finite automaton) M that accepts all strings starting with `a`.



Here $*$ represents any possible character.

Notice key pieces of this machine: three states, q_0 is the start state (arrow coming in), q_1 is the final state (double circle), transition arcs.

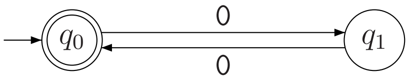
To run the machine, we start at the start state. On each input character, we follow the corresponding arc. When we run out of input characters, we answer "yes" or "no", depending on whether we are in the final state.

The language of a machine M is the set of strings it accepts, written $L(M)$. In this case $L(M) = \{\texttt{a}, \texttt{aa}, \texttt{ab}, \texttt{aaa}, \ldots\}$.

2.2 Another automata

(This section is optional and can be skipped in the lecture.)

Here is a simple ***state machine*** (i.e., finite automaton) M that accepts all ASCII strings ending with `ing`.



Notice key pieces of this machine: four states, q_0 is the start state (arrow coming in), q_3 is the final state (double circle), transition arcs.

To run the machine, we start at the start state. On each input character, we follow the corresponding arc. When we run out of input characters, we answer "yes" or "no", depending on whether we are in the final state.

The language of a machine M is the set of strings it accepts, written $L(M)$. In this case $L(M) = \{\texttt{walking}, \texttt{flying}, \texttt{ing}, \ldots\}$.

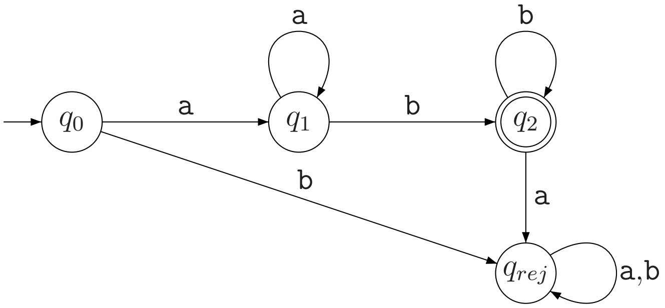
2.3 What automatas are good for?

People use the technology of automatas in real-world applications:

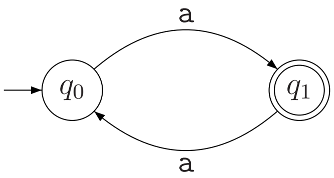
- Find all files containing `-ing` (grep)
- Translate each `-ing` into `-iG` (finite-state transducer)
- How often do words in Chomsky's latest book end in `-ing`?

2.4 DFA - deterministic finite automata

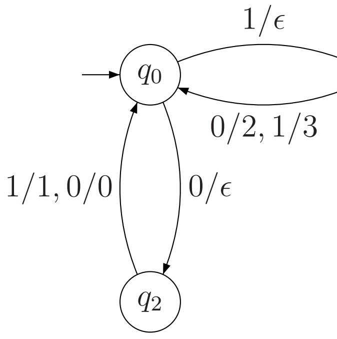
We will start by studying ***deterministic finite automata*** (DFA). Each node in a deterministic machine has exactly one outgoing transition for each character in the alphabet. That is, if the alphabet is $\{\texttt{a}, \texttt{b}\}$, then all nodes need to look like



Both of the following are bad, where $q_1 \neq q_2$ and the right hand machine has no outgoing transition for the input character `b`.



So our `-ing` detector would be redrawn as:

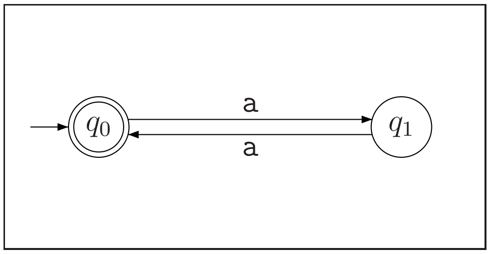


3 More examples of DFAs

3.1 Number of characters is even

Input: $\Sigma = \{0\}$.

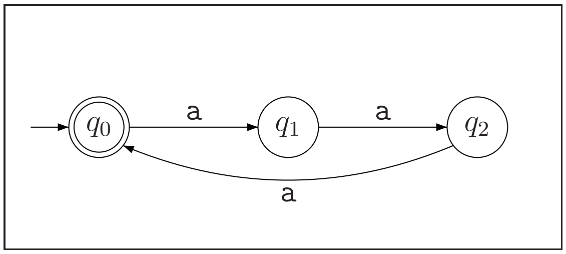
Accept: all strings in which the number of characters is even.



3.2 Number of characters is divisible by 3

Input: $\Sigma = \{0\}$.

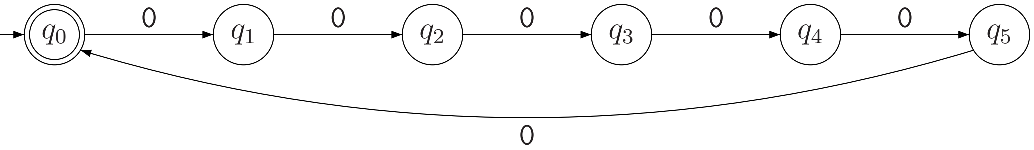
Accept: all strings in which the number of characters is divisible by 3.



3.3 Number of characters is divisible by 6

Input: $\Sigma = \{0\}$.

Accept: all strings in which the number of characters is divisible by 6.

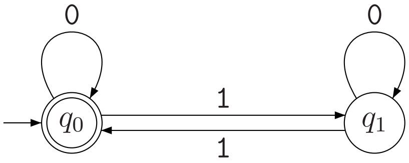


This example is especially interesting, because we can achieve the same purpose, by observing that $n \bmod 6 = 0$ if and only if $n \bmod 2 = 0$ and $n \bmod 3 = 0$ (i.e., to be divisible by 6, a number has to be divisible by 2 and divisible by 3 [a generalization of this idea is known as the Chinese remainder theorem]). So, we could run the two automatas of Section 3.1 and Section 3.2 in parallel (replicating each input character to each one of the two automatas), and accept only if both automatas are in an accept state. This idea would become more useful later in the course, as it provide a building operation to construct complicated automatas from simple automatas.

3.4 Number of ones is even

Input is a string over $\Sigma = \{0, 1\}$.

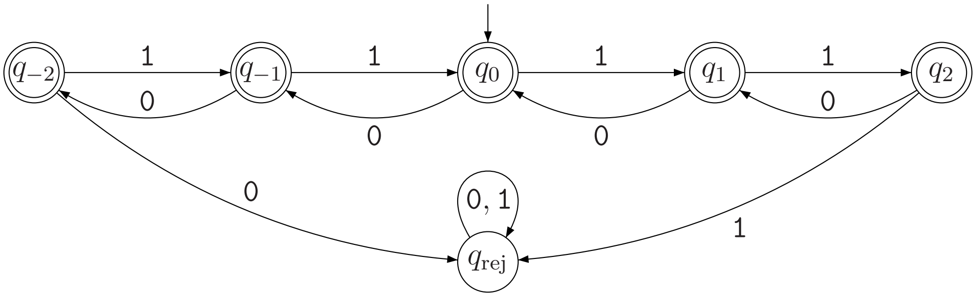
Accept: all strings in which the number of ones is even.



3.5 Number of zero and ones is always within two of each other

Input is a string over $\Sigma = \{0, 1\}$.

Accept: all strings in which the difference between the number of ones and zeros in any prefix of the string is in the range $-2, \dots, 2$. For example, the language contains ϵ, 0, 001, and 1101. You even have an extended sequence of one character e.g. 001111, but it depends what preceded it. So 111100 isn't in the language.



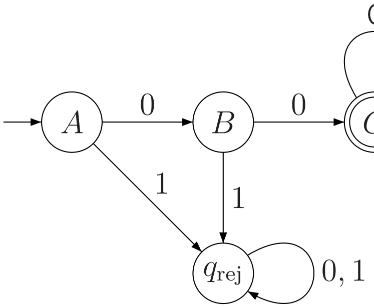
Notice that the names of the states reflect their role in the computation. When you come to analyze these machines formally, good names for states often makes your life much easier. BTW, the language of this DFA is

$$L(M) = \left\{ w \;\middle|\; w \in \{\texttt{0}, \texttt{1}\}^* \text{ and for every } x \text{ that is a prefix of } w, |\#\texttt{1}(x) - \#\texttt{0}(x)| \leq 2 \right\}.$$

3.6 More complex language

The input is strings over $\Sigma = \{\texttt{0}, \texttt{1}\}$.

Accept: all strings of the form $00w$, where w contains an even number of ones.



You can name states anything you want. Names of the form q_X are often convenient, because they remind you of what's a state. And people often make the initial state q_0. But this isn't obligatory.

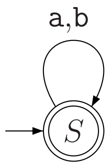
4 The pieces of a DFA

To specify a DFA (***deterministic finite automata***), we need to describe

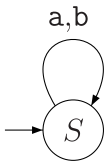
- a (finite) alphabet
- a (finite) set of states
- which state is the start state?
- which states are the final states?
- what is the transition from each state, on each input character?

5 Some special DFAs

For $\Sigma = \{\texttt{a}, \texttt{b}\}$, consider the following DFA that accepts Σ^*:

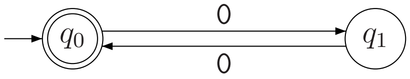


The DFA that accepts nothing, is just



6 Formal definition of a DFA

Consider the following automata, that we saw in the previous lecture:



We saw last class that the following components are needed to specify a DFA:

(i) a (finite) alphabet

(ii) a (finite) set of states

(iii) which state is the start state?

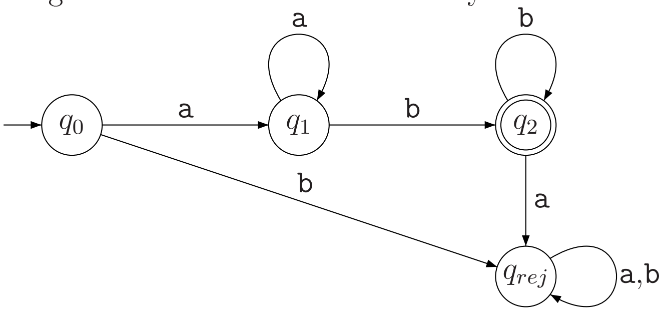
(iv) which states are the final states?

(v) what is the transition from each state, on each input character?

Formally, a ***deterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q: A finite set (the set of ***states***).
- Σ: A finite set (the ***alphabet***)
- $\delta : Q \times \Sigma \to Q$ is the ***transition function***.
- q_0: The ***start*** state (belongs to Q).
- F: The set of ***accepting*** (or ***final***) states, where $F \subseteq Q$.

For example, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and consider the following DFA M, whose language $L(M)$ contains strings consisting of one or more a's followed by one or more b's.



Then $M = (Q, \Sigma, \delta, q_0, F)$, $Q = \{q_0, q_1, q_2, q_{rej}\}$, and $F = \{q_2\}$. The transition function δ is defined by

δ	a	b
q_0	q_1	q_{rej}
q_1	q_1	q_2
q_2	q_{rej}	q_2
q_{rej}	q_{rej}	q_{rej}

We can also define δ using a formula

$$\delta(q_0, a) = q_1$$

$$\delta(q_1, a) = q_1$$

$$\delta(q_1, b) = q_2$$

$$\delta(q_2, b) = q_2$$

$$\delta(q, t) = q_{rej} \text{ for all other values of } q \text{ and } t.$$

Tables and state diagrams are most useful for small automata. Formulas are helpful for summarizing a group of transitions that fit a common pattern. They are also helpful for describing algorithms that modify automatas.

7 Formal definition of acceptance

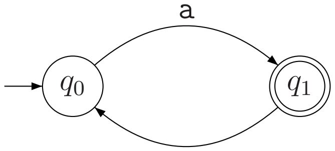
We've also seen informally how to run a DFA. Let us turn that into a formal definition. Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a given DFA and $w = w_1 w_2 \ldots w_k \in \Sigma^*$ is the input string. Then M ***accepts*** w iff there exists a sequence of states $r_0, r_1, \ldots r_k$ in Q, such that

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, \ldots, k-1$.
3. $r_k \in F$.

The ***language recognized*** by M, denoted by $L(M)$, is the set $\left\{ w \;\middle|\; M \text{ accepts } w \right\}$.

For example, when our automaton above accepts the string aabb, it uses the state sequence $q_0 q_1 q_1 q_2 q_2$. (Draw a picture of the transitions.) That is $r_0 = q_0$, $r_1 = q_1$, $r_2 = q_1$, $r_3 = q_2$, and $r_4 = q_2$.

Note that the states do not have to occur in numerical order in this sequence, e.g. the following DFA accepts aaa using the state sequence $q_0 q_1 q_0 q_1$.



A language (i.e. set of strings) is ***regular*** if it is recognized by some DFA.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 4: The product construction: Closure under intersection and union

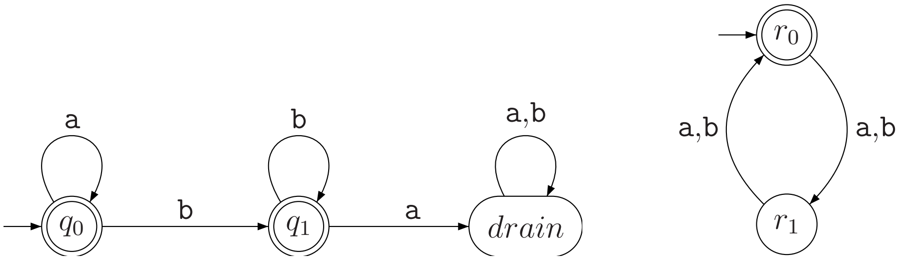
28 January 2010

This lecture finishes section 1.1 of Sipser and also covers the start of 1.3.

1 Product Construction

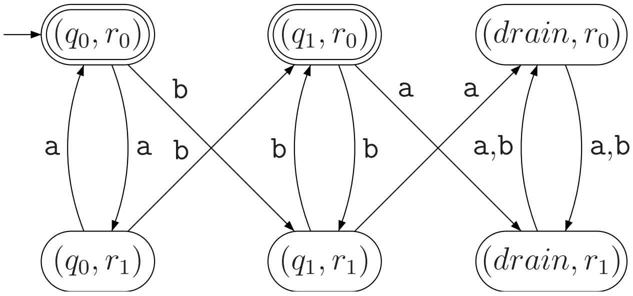
1.1 Product Construction: Example

Let $\Sigma = \{\texttt{a}, \texttt{b}\}$ and L is the set of strings in Σ^* that have the form $\texttt{a}^*\texttt{b}^*$ and have even length. L is the intersection of two regular languages $L_1 = \texttt{a}^*\texttt{b}^*$ and $L_2 = (\Sigma\Sigma)^*$. We can show they are regular by exhibiting DFAs that recognize them.



L_1 L_2

We can run these two DFAs together, by creating states that remember the states of both machines.



Notice that the final states of the new DFA are the states (q, r) where q is final in the first DFA ***and*** r is final in the second DFA. To recognize the union of the two languages, rather than the intersection, we mark all the states (q, r) such that either q or r are accepting states in the their respective DFAs.

State of a DFA after reading a word w. In the following, given a DFA $M = (Q, \Sigma, \delta, q_0, F)$, we will be interested in the state the DFA M is in, after reading the a string w. Let us denote by $\delta^* : Q \times \Sigma^* \to Q$ the function such that $\delta^*(q, w)$ is the state the DFA will land in, if started from state q and fed the word w.

Formally, we define $\delta^* : Q \times \Sigma^* \to Q$ using the following *inductive definition*:

- $\delta^*(q, \epsilon) = q$ for every $q \in Q$,
- $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ for each $q \in Q, w \in \Sigma^*, a \in \Sigma$.

Note that by the above definition of δ^*, we have that $w \in L(M)$ iff $\delta^*(q_0, w) \in F$.

2 Product Construction: Formal construction

We are given two DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q_0', F')$ both working over the same alphabet Σ. A ***product automaton*** of M and M' is an automaton

$$N = \Big(\mathcal{Q},\ \Sigma,\ \delta_N,\ (q_0, q_0'),\ F_N\Big),$$

where $\mathcal{Q} = Q \times Q'$, and $\delta_N : \mathcal{Q} \times \Sigma \to \mathcal{Q}$. Also, for any $q \in Q$, $q' \in Q'$ and $c \in \Sigma$, we require

$$\delta_N\big(\ \underbrace{(q, q')}_{\text{state of } N}\ , c\ \big) = \Big(\delta(q, c),\ \delta'(q', c)\Big). \qquad (1)$$

The set $F_N \subseteq \mathcal{Q}$ of accepting states is free to be whatever we need it to be, depending on what we want N to recognize. For example, if we would like N to accept the intersection $L(M) \cap L(M')$ then we will set $F_N = F \times F'$. If we want N to recognize the union language $L(M) \cup L(M')$ then $F_N = (F \times Q')\ \cup \cup (Q \times F')$.

Lemma 2.1 *For any input word $w \in \Sigma^*$, the product automata N of the DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q_0', F')$, is in state (q, q') after reading w, if and only if (i) M in the state q after reading w, and (ii) M' is in the state q' after reading w. In other words, $\delta_N^*((q_0, q_0'), w) = (\delta^*(q_0, w), \delta'^*(q_0', w))$.*

Proof: The proof is by induction on the length of the word w.

If $w = \epsilon$ is the empty word, then N is initially in the state (q_0, q_0') by construction, where q_0 (resp. q_0') is the initial state of M (resp. M'). As such, the claim holds in this case.

Formally, $\delta_N^*((q_0, q_0'), \epsilon) = (q_0, q_0') = (\delta^*(q_0, \epsilon), \delta'^*(q_0', \epsilon))$ (by definition of δ_N^*, δ^* and δ'^*).

Otherwise, $|w| > 0$, and let us hence assume $w = w_1 a$ ($w_1 \in \Sigma^*, a \in \Sigma$), and assume the induction hypothesis that the claim is true for all input words of length strictly smaller than $|w|$ (in particular $|w_1|$).

Let (q_{k-1}, q_{k-1}') be the state that N is in after reading the string w_1. By the induction hypothesis, as $|w_1| = k - 1$, we know that M is in the state q_{k-1} after reading $\widehat{w}$, and M' is in the state q_{k-1}' after reading $\widehat{w}$.

Let $q_k = \delta(q_{k-1}, a) = \delta(\delta^*(q_0, w_1)\ ,\ a) = \delta(q_0, w)$ and

$$q_k' = \delta'(q_{k-1}', a) = \delta'(\delta'(q_0', w_1)\ ,\ a) = \delta'(q_0', w).$$

As such, by definition, M (resp. M') would in the state q_k (resp. q'_k) after reading w.

Also, by the definition of its transition function, after reading w the DFA N would be in the state

$$\begin{aligned}\delta_N((q_0, q'_0), w) &= \delta_N(\delta^*_N((q_0, q'_0), w_1)\,, a) = \delta_N\big((q_{k-1}, q'_{k-1}),\ a\big)\\ &= \big(\delta(q_{k-1}, a),\ \delta(q'_{k-1}, a)\big) = (q_k,\ q'_k)\,,\end{aligned}$$

(see Eq. (1)). This establishes the claim. ■

Lemma 2.2 *Let $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q'_0, F')$ be two given DFAs. Let N be a product automaton with set of accepting states is $F \times F'$. Then $L(N) = L(M) \cap L(M')$.*

Proof: $w \in L(N)$ iff $\delta^*_N((q_0, q'_0), w) \in F \times F'$
iff $(\delta^*(q_0, w), \delta'^*(q'_0, w)) \in F \times F'$ (by Lemma 2.1)
iff $\delta^*(q_0, w) \in F$ and $\delta'^*(q'_0, w) \in F'$
iff $w \in L(M)$ and $w \in L(M')$.

More verbosely:

If $w \in L(M) \cap L(M')$, then let $q_w = \delta^*(q_0, w) \in F$ and $q'_w = \delta'^*(q'_0, w) \in F'$. By Lemma 2.1, this implies that $\delta^*_N((q_0, q'_0), w) = (q_w, q'_w) \in F \times F'$. Namely, N accepts the word w, implying that $w \in L(N)$, and as such $L(M) \cap L(M') \subseteq L(N)$.

Similarly, if $w \in L(N)$, then $(p_w, p'_w) = \delta^*_N((q_0, q'_0), w)$ must be an accepting state of N. But the set of accepting states of N is $F \times F'$. That is $(p_w, p'_w) \in F \times F'$, implying that $p_w \in F$ and $p'_w \in F'$. Now, by Lemma 2.1, we know that $\delta^*(q_0, w) = p_w \in F$ and $\delta'^*(q'_0, w) = p'_w \in F'$. Thus, M and M' both accept w, which implies that $w \in L(M)$ and $w \in L(M')$. Namely, $w \in L(M) \cap L(M')$, implying that $L(N) \subseteq L(M) \cap L(M')$.

Putting the above together proves the lemma. ■

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 5: Closure under complement; Nondeterministic Automata

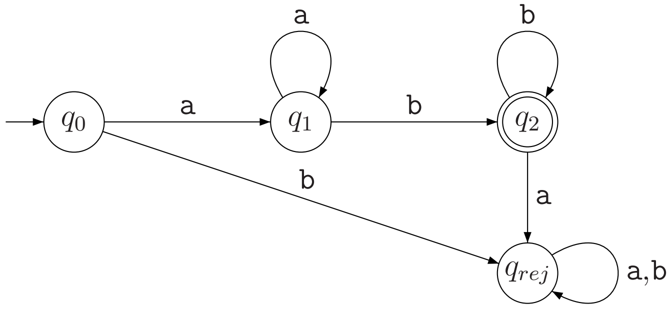
February 3, 2009

This lecture covers mainly the first part of section 1.2 of Sipser, through p 54.

1 Closure under complement of regular languages

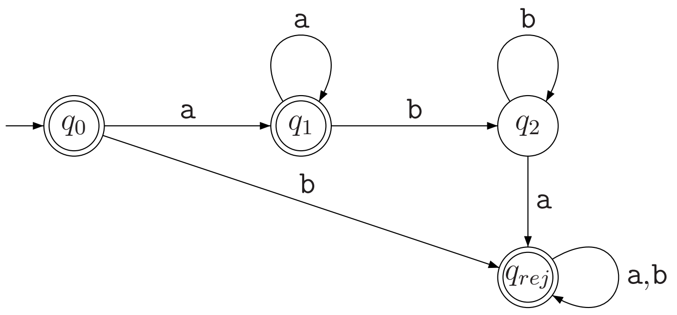
Here we are interested in the question of whether the regular languages are closed under set complement. (The complement language keeps the same alphabet.) That is, if we have a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepting some language L, can we construct a new DFA M' accepting $\overline{L} = \Sigma^* \setminus L$?

Consider the automata M from above, where L is the set of all strings of at least one a followed by at least one b.



The complement language $\overline{L}$ contains the empty string, strings in which some b's precede some a's, and strings that contain only a's or only b's.

Our new DFA M' should accept exactly those strings that M rejects. So we can make M' by swapping final/non-final markings on the states:



Formally, $M' = (Q, \Sigma, \delta, q_0, Q \setminus F)$.

Theorem 1.1 *Let $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Then $L(M') = \Sigma^* \setminus L(M)$.*

Proof: Note that since Q and δ are the same in M and M', M and M' have the same δ'. Also, note that they have the same initial state q_0.

Let $w \in \Sigma^*$.

Then $w \in L(M')$ iff $\delta^*(q_0, w) \in (Q \setminus F)$
iff $\neg(\delta^*(q_0, w) \in F))$
iff $\neg(w \in L(M))$
iff $w \in \Sigma^* \setminus L(M)$. ■

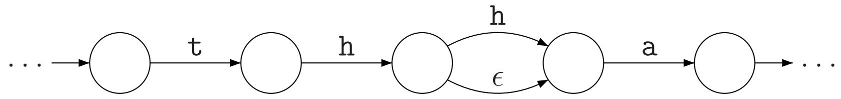
2 Non-deterministic finite automata (NFA)

A ***non-deterministic finite automata*** (***NFA***) is like a DFA but with three extra features. These features make them easier to construct, especially because they can be composed in a modular fashion. Furthermore, they are easier to read, and they tend to be much smaller and as such easier to describe. Computationally, they are equivalent to DFAs, in the sense that they recognize the same languages.

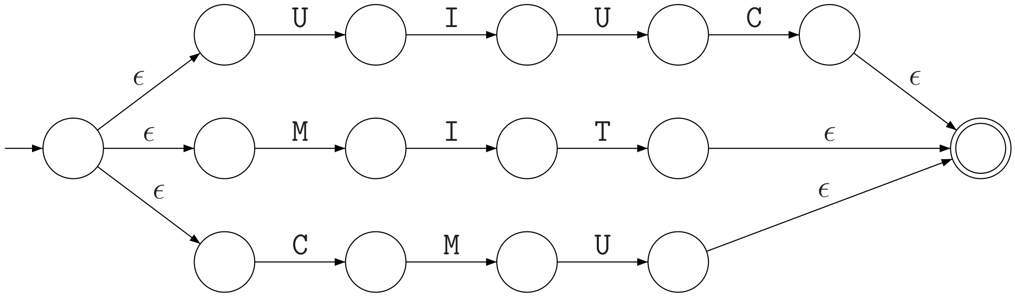
For practical applications of any complexity, users can write NFAs or regular expressions (trivial to convert to NFAs). A computer algorithm might compile these to DFAs, which can be executed/simulated quickly.

2.1 NFA feature #1: Epsilon transitions

An NFA can do a state transition without reading input. This makes it easy to represent optional characters. For example, "`Northampton`" is commonly misspelled as "`Northhampton`". A web search type application can recognize both variants using the pattern `North(h)ampton`.



Epsilon transitions also allow multiple alternatives (set union) to be spliced together in a nice way. E.g. we can recognize the set $\{\texttt{UIUC}, \texttt{MIT}, \texttt{CMU}\}$ with the following automaton. This allows modular construction of large state machines.



2.1.1 How do we execute an NFA?

Assume a NFA N is in state q, and the next input character is c. The NFA N may have multiple transitions it could take. That is, multiple possible next states. An NFA accepts if there is *some* path through its state diagram that consumes the whole input string and ends in an accept state.

Here are two possible ways to think about this:

(i) the NFA magically guesses the right path which will lead to an accept state.

(ii) the NFA searches all paths through the state diagram to find such a path.

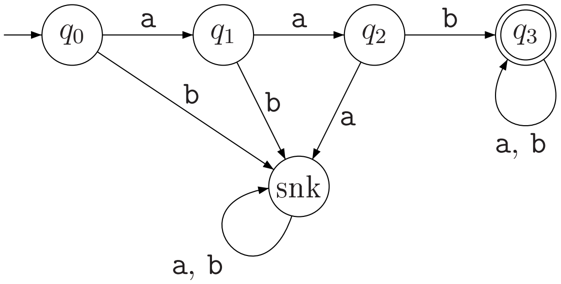
The first view is often the best for mathematical analysis. The second view is one reasonable approach to implementing NFAs.

2.2 NFA Feature #2: Missing transitions

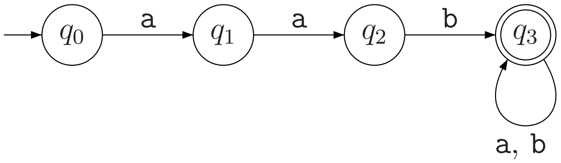
Assume a NFA N is in state q, and the next input character is c. The NFA may have no outgoing transition from q that corresponds to the input character c.

This means that you can not get to an accepting state from this point on. So the NFA will reject the input string unless there is some other alternative path through the state diagram. You can think of the missing transitions as going to an implicit sink state. Visually, diagrams of NFAs are much simpler by not having to put in the sink state explicitly.

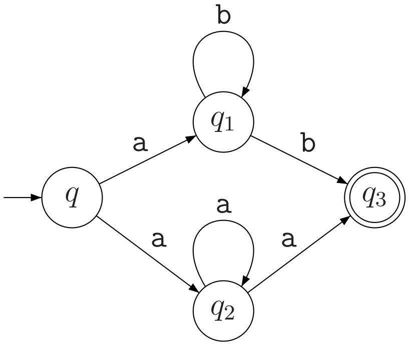
Example. Consider the DFA that accepts all the strings over $\{\texttt{a}, \texttt{b}\}$ that starts with aab. Here is the resulting DFA.



The NFA for the same language is even simpler if we omit transitions, and the sink state. In particular, the NFA for the above language is the following.



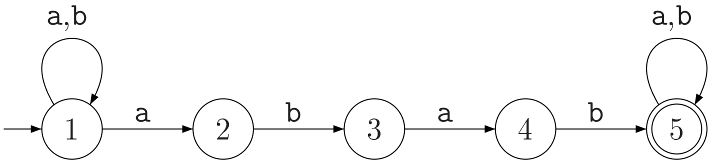
2.3 NFA Feature #3: Multiple transitions



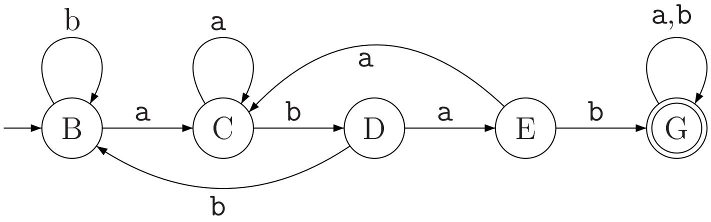
A state q in a NFA may have more than one outgoing transition for some character t. This means that the NFA needs to "guess" which path will accept the input string. Or, alternatively, search all possible paths. This complicates deciding if a string is accepted by a NFA, but it greatly simplifies the resulting machines. Thus, the automata on the right accepts all strings of the form $\texttt{a}b^i$ or $\texttt{aa}^{\texttt{i}}$ (for any $i \in \mathbb{N}$). Of course, its not too hard to build a DFA for this language, but even here the description of the NFA is simpler.

As another example, the automata below accepts strings containing the substring abab.

(N1)



The respective DFA, shown below, needs a lot more transitions and is somewhat harder to read.



3 More Examples

3.1 Running an NFA via search

Let us run an explicit search for the above NFA (N1) on the input string ababa. Initially, at time $t = 0$, the only possible state is the start state 1. The search is depicted in table on the right. When the input is exhausted, one of the possible states (E) is an accept state, and as such the NFA (N1) accepts the string ababa.

Time	Possible states	Remaining input
$t = 0$	$\{1\}$	ababa
$t = 1$	$\{1, 2\}$	baba
$t = 2$	$\{1, 3\}$	aba
$t = 3$	$\{1, 2, 4\}$	ba
$t = 4$	$\{1, 3, 5\}$	a
$t = 5$	$\{1, 2, 4, 5\}$	ϵ

3.2 Interesting guessing example

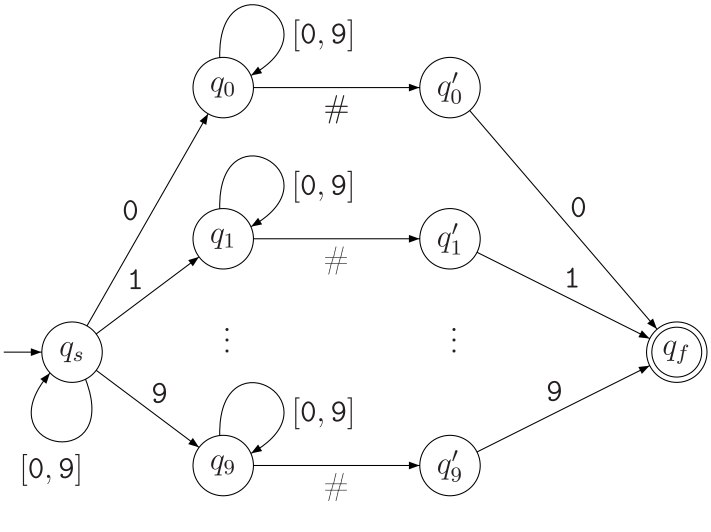
Some NFAs are easier to construct and analyze if you take the "guessing" view on how they work.

Let $\Sigma = \{0, 1, \ldots, 9\}$, denote this as $[0, 9]$ in short form. Let

$$L = \left\{ w\#c \;\middle|\; c \in \Sigma, w \in \Sigma^*, \text{ and } c \text{ occurs in } w \right\}.$$

For example, the word 314159#5 is in L, and so is 314159#3. But the word 314159#7 is not in L.

Here is the NFA M that recognizes this language.



The NFA M scans the input string until it "guesses" that it is at the character c in w that will be at the end of the input string. When it makes this guess, M transitions into a state q_c that "remembers" the value c. The rest of the transitions then confirm that the rest of the input string matches this guess.

A DFA for this problem is considerably more taxing. We will need a state to remember each digit encountered in the string read so far. Since there are 2^{10} different subsets, we will require an automata with at least 1024 states! The NFA above requires only 22 states, and is much easier to draw and understand.

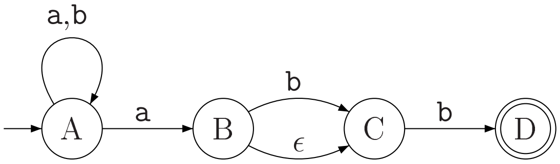
4 Formal definition of an NFA

An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Similar to a DFA except that the type signature for δ is

$$\delta : Q \times \Sigma_\epsilon \rightarrow \mathbb{P}(Q),$$

where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\mathbb{P}(Q)$ is the power set of Q (i.e., all possible subsets of Q). As such, the input character for $\delta(\cdot)$ can be either a real input character or ϵ (in this case the NFA does not eat [or drink] any input character when using this transition). The output value of δ is a *set* of states (unlike a DFA).

Example 4.1 Consider the following NFA:



Here

$\delta(A, \texttt{a}) = \{A, B\}$

$\delta(B, \texttt{a}) = \emptyset$ (NB: not $\{\emptyset\}$)

$\delta(B, \epsilon) = \{C\}$ (NB: not just C)

$\delta(B, \texttt{b}) = \{C\}$ (NB: just follows one transition arc).

The trace for recognizing the input `abab`:

$t = 0$: state $= A$, remaining input `abab`.

$t = 1$: state $= A$, remaining input `bab`.

$t = 2$: state $= A$, remaining input `ab`.

$t = 3$: state $= B$, remaining input `b`.

$t = 4$: state $= C$, remaining input `b` (ϵ transition used, and no input eaten).

$t = 5$: state $= D$, remaining input ϵ.

Is every DFA an NFA? Technically, no (why?[1]). However, it is easy to convert any DFA into an NFA. If δ is the transition function of the DFA, then the corresponding transition of the NFA is going to be $\delta'(q, t) = \{\delta(q, t)\}$.

4.1 Formal definition of acceptance

Let $M = \big(Q, \Sigma, \delta, q_0, F\big)$ be an NFA. Let w be a string in Σ^*.

The NFA M accepts w if and only if there is a sequence of states $r_0, r_1, \ldots, r_n$ and a sequence of inputs $x_1, x_2, \ldots, x_n$, where each x_i is either a character from Σ or ϵ, such that

(i) $w = x_1 x_2 \ldots x_n$.

(The input string "eaten" by the NFA is the input string w.)

(ii) $r_0 = q_0$.

(The NFA starts from the start state.)

(iii) $r_n \in F$.

(The final state in the trance in an accepting state.)

(iv) $r_{i+1} \in \delta(r_i, x_{i+1})$ for every i in $[0, n-1]$.

(The transitions in the trace are all valid. That is, the state r_{i+1} is one of the possible states one can go from r_i, if the NFA consumes the character x_{i+1}.

So, in the above example, $n = 6$, our state sequence is $AAABCD$, and our sequence of inputs is abaϵb.

Key differences the notation of acceptance from DFA are

(i) Inserting/allowing ϵ into input character sequence.

(ii) Output of δ is a set, so in condition (iv) above, r_{i+1} is a member of δ's output. (For a DFA, in this case, we just had to check that the new state is equal to δ's output.)

[1] Because, the transition function is defined differently.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 6: Closure properties

February 5, 2009

This lecture covers the last part of section 1.2 of Sipser (pp. 58–63), beginning of 1.3 (pp. 63–66), and also closure under string reversal. We also include a proof of the string-reversal construction and the union construction for NFAs, which Sipser unfortunately does not.

1 Operations on languages

Regular operations on languages (sets of strings). Suppose L and K are languages.

- ***Union***: $L \cup K = \left\{ x \,\middle|\, x \in L \quad \text{or} \quad x \in K \right\}$.
- ***Concatenation***: $L \circ K = LK = \left\{ xy \,\middle|\, x \in L \quad \text{and} \quad y \in K \right\}$.
- ***Star*** (***Kleene star***):

$$L^* = \left\{ w_1 w_2 \ldots w_n \,\middle|\, w_1, \ldots, w_n \in L \text{ and } n \geq 0 \right\}.$$

We (hopefully) all understand what union does. The other two have some subtleties. Let

$$L = \{\texttt{under}, \texttt{over}\}, \quad \text{and} \quad K = \{\texttt{ground}, \texttt{water}, \texttt{work}\}.$$

Then

$$LK = \{\texttt{underground}, \texttt{underwater}, \texttt{underwork}, \texttt{overground}, \texttt{overwater}, \texttt{overwork}\}.$$

Similarly,

$$K^* = \left\{ \begin{array}{l} \epsilon, \texttt{ground}, \texttt{water}, \texttt{work}, \texttt{groundground}, \\ \texttt{groundwater}, \texttt{groundwork}, \texttt{workground}, \\ \texttt{waterworkwork}, \ldots \end{array} \right\}.$$

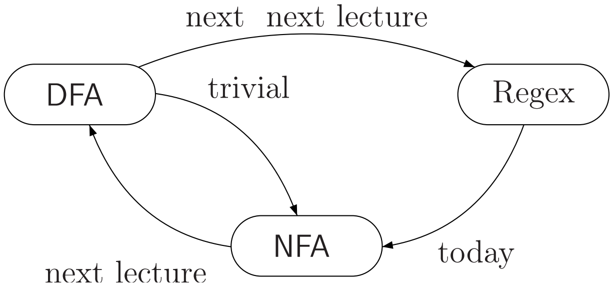
For star operator, note that the resulting set *always* contains the empty string ϵ (because n can be zero).

Also, each of the substrings is chosen independently from the base set and you can repeat. E.g. `waterworkwork` is in K^*.

Regular languages are closed under many operations, including the three "regular operations" listed above, set intersection, set complement, string reversal, "homomorphism" (formal version of shifting alphabets). We have seen (last class) why regular languages are closed under set complement. We will prove the rest of these bit by bit over the next few lectures.

2 Overview of closure properties

We defined a language to be ***regular*** if it is recognized by some DFA. The agenda for the new few lectures is to show that three different ways of defining languages, that is NFAs, DFAs, and regexes, and in fact all equivalent; that is, they all define regular languages. We will show this equivalence, as follows.



One of the main properties of languages we are interested in are closure properties, and the fact that regular languages are closed under union, intersection, complement, concatenation, and star (and also under homomorphism).

However, closure operations are easier to show in one model than the other. For example, for DFAs showing that they are closed under union, intersection, complement are easy. But showing closure of DFA under concatenation and $*$ is hard.

Here is a table that lists the closure property and how hard it is to show it in the various models of regular languages.

Model					
Property	$\cap$ intersection	$\cup$ union	$\overline{L}$ complement	$\circ$ concatenation	$*$ star
DFA	Easy (done)	Easy (done)	Easy (done)	Hard	Hard
NFA	Doable (hw?)	Easy: Lemma 4.1	Hard	Easy: Lemma 4.2	Easy: Lemma 4.3
regex	Hard	Easy	Hard	Easy	Easy

Recall what it means for regular languages to be closed under an operation `op`. If L_1 and L_2 are regular, then L_1 `op` L_2 is regular. That is, if we have an NFA recognizing L_1 and an NFA recognizing L_2, we can construct an NFA recognizing L_1 `op` L_2.

The extra power of NFAs makes it easy to prove closure properties for NFAs. When we know all DFAs, NFAs, and regexes are equivalent, these closure results then apply to all three representations. Namely, they would imply that regular languages have these closure properties.

3 Closure under string reversal for NFAs

Consider a word w, we denote by w^R the ***reversed*** word. It is just w in the characters in reverse order. For example, for $w =$ `barbados`, we have $w^R =$ `sodabrab`. For a language L, the ***reverse*** language is

$$L^R = \left\{ w^R \;\middle|\; w \in L \right\}.$$

We would like to claim that if L is regular, then so is L^R. Formally, we need to be a little bit more careful, since we still did not show that a language being regular implies that it is recognized by an NFA.

Claim 3.1 *If L is recognized by an NFA, then there is an NFA that recognizes L^R.*

Proof: Let M be an NFA recognizing L. We need to construct an NFA N recognizing L^R.

The idea is to reverse the arrows in the NFA $M = (Q, \Sigma, \delta, q_0, F)$, and swap final and initial states. There is a bug in applying this idea in a naive fashion. Indeed, there is only one initial state but multiple final states.

To overcome this, let us modify M to have a single final state q_S, connected to old ones with epsilon transitions. Thus, the modified NFA accepting L, is

$$M' = \big(Q \cup \{q_S\}, \Sigma, \delta', q_0, \{q_S\}\big),$$

where q_S is the only accepting state for M. Note, that δ' is identical to δ, except that

$$\forall q \in F \quad \delta'(q, \epsilon) = q_S. \tag{1}$$

Note, that $L(M) = L(M') = L$.

As such, q_S will become the start state of the "reversed" NFA.

Now, the new "reversed" NFA N, for the language L^R, is

$$N = \big(Q \cup \{q_S\}, \Sigma, \delta'', q_S, \{q_0\}\big).$$

Here, the transition function δ'' is defined as

(i) $\delta''(q_0, t) = \emptyset$ for every $t \in \Sigma$.

(ii) $\delta''(q, t) = \left\{ r \in Q \;\middle|\; q \in \delta'(r, t) \right\}$, for every $q \in Q \cup \{q_S\}$, $t \in \Sigma_\epsilon$.

(iii) $\delta''(q_S, \epsilon) = F$ (the reversal of Eq. (1)). [1]

Now, we need to prove formally that if $w \in L(M)$ then $w^R \in L(N)$. To this end, let us prove the following claim.

Lemma: For every word w, the automaton M' can reach a state q from its initial state q_0 reading w iff the automaton N, when started from state q, can reach the state q_0 reading w^R.

Proof of lemma:

Let $w \in \Sigma^*$ and let M' reach a state q from q_0 on reading w, say using a sequence of states $r_0, r_1, \ldots, r_k$ (where $r_0 = q_0$ and $r_k = q$). Then, since N contains the reversal of the edges in M', the automaton N, when started from state q, can reach q_0 on reading w^R using the sequence $r_k, r_{k-1}, \ldots, r_1, r_0$.

[1] This can be omitted, since it is implied by the (ii) rule.

Similarly, assume that N, when started from state q, can read w and end in state q_0, say using a sequence of states $r_0, r_1, \ldots, r_k$, where $r_0 = q$ and $r_k = q$. Then, since M' contains the reversal of the edges in N, M' when started at q_0 can reach q on reading w^R, using the sequence of states $r_k, r_{k-1}, \ldots, r_1, r_0$.
End of proof of lemma.

Let us now use the claim to prove that $w \in L(M')$ iff $w^R \in L(N)$.

$w \in L(M')$ iff M can reach q_s reading w when started from q_0
iff N can reach q_0 reading w^R when started from q_s
iff $w^R \in L(N)$.

■

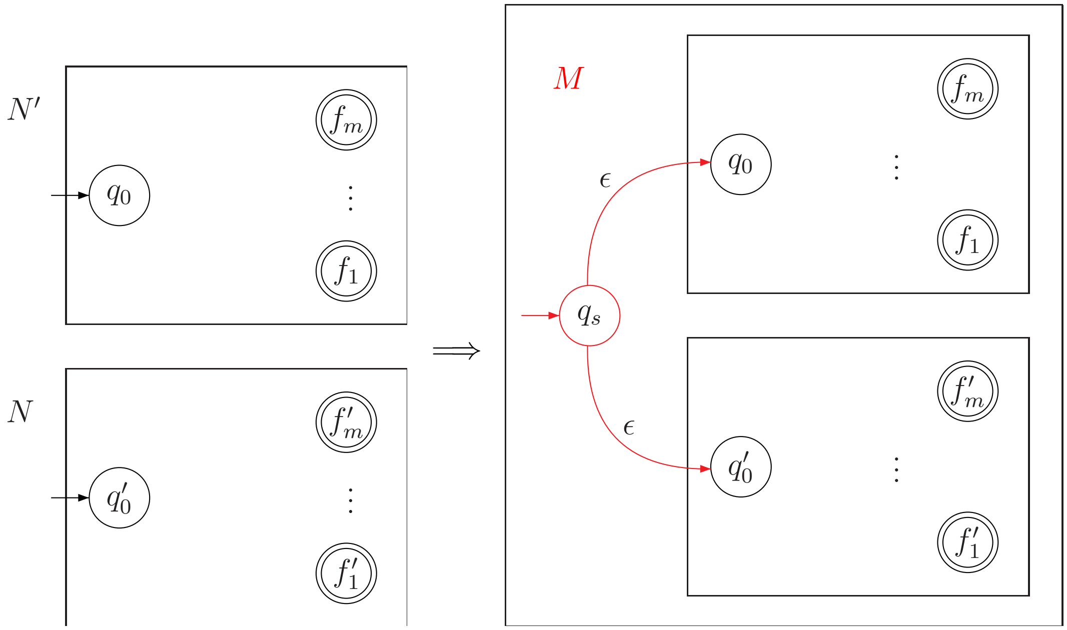
Note, that this will not work for a DFA. First, we can not force a DFA to have a single final state. Second, a state may have two incoming transitions on the same character, resulting in non-determinism when reversed.

4 Closure of NFAs under regular operations

We consider the ***regular operations*** to be union, concatenation, and the star operator.

4.1 NFA closure under union

Given two NFAs, say N and N, we would like to build an NFA for the language $L(N) \cup L(N)$. The idea is to create a new initial state q_s and connect it with an ϵ-transition to the two initial states of N and N. Visually, the resulting NFA M looks as follows.



Formally, we are given two NFAs $N = (Q, \Sigma, \delta, q_0, F)$ and $N' = (Q', \Sigma, \delta', q'_0, F')$, where $Q \cap Q' = \emptyset$ and the new state q_s is not in Q or Q'. The new NFA M is

$$M = (Q \cup Q' \cup \{q_s\}, \Sigma, \delta_M, s_q, F \cup F'),$$

where

$$\delta_M(q,c) = \begin{cases} \delta(q,c) & q \in Q, c \in \Sigma_\epsilon \\ \delta'(q,c) & q \in Q', c \in \Sigma_\epsilon \\ \{q_0, q_0'\} & q = q_s, c = \epsilon \\ \emptyset & q = q_s, c \neq \epsilon. \end{cases}$$

Let us now formally prove that $L(M) = L(N) \cup L(N')$.

First, let us show that $L(N) \cup L(N') \subseteq L(M)$. Let $w \in L(N) \cup L(N')$. So $w \in L(N)$ or $w \in L(N')$. Let us consider the case when $w \in L(N)$ (the other case is similarly argued). Since $w \in L(N)$, by definition, there exists a sequence of states $r_0, r_1, \ldots, r_k$ that is a run of N on w, where $r_0 = q_0$ and $r_k \in F$. Now consider the run $q_s, r_0, \ldots, r_k$ in M. Since $q_0 \in \delta(q_s, \epsilon)$, and the transitions of N are all present in M, this sequence is a run of M on w. Since the final states of M include the final states of N, r_k is a final state of M as well, and hence this run is accepting. Hence $w \in L(M)$. The case when $w \in L(N')$ can be argued similarly. Hence $L(N) \cup L(N') \subseteq L(M)$.

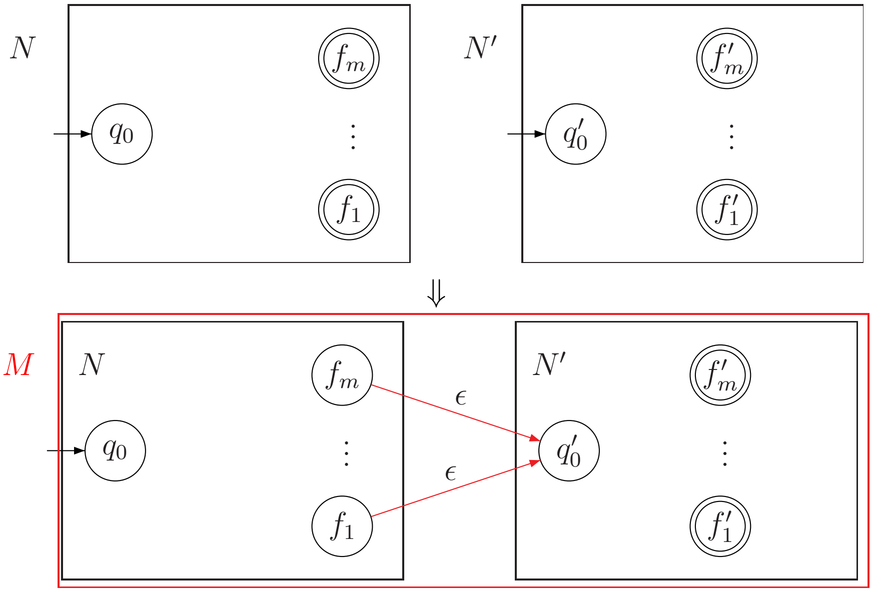
Now let us show that $L(M) \subseteq L(N) \cup L(N')$. Let $w \in L(M)$. Then, by definition, there exists an accepting run $r_0, r_1, \ldots, r_k$ of M on w with $r_0 = q_s$ and $r_k \in F \cup F'$. First, $k \neq 0$, as if $k = 0$, then $r_k = r_0 = q_s \notin F \cup F'$, which is a contradiction. Notice that since there is only one transition from q_s, we must have that $r_1 = q_0$ or $r_1 = q_0'$. Let us assume that $r_1 = q_0$ (the other case is similarly argued). Notice that taking transitions starting from q_0 will keep us within the states of N, and hence $r_k \in F$. Further, since all these transitions belong to N, and since the transition from q_s to q_0 was on ϵ, the run $r_1, \ldots, r_k$ must be a run in N on the word w. Since $r_k \in F$, $w \in L(N)$. In the other case, when $r_1 = q_0'$, we can similarly argue that $w \in L(N')$. Hence $w \in L(N) \cup L(N')$.

We have thus showed the following.

Lemma 4.1 *Given two NFAs N and N', one can construct an NFA M (as given above), such that $L(M) = L(N) \cup L(N')$.*

4.2 NFA closure under concatenation

Given two NFAs N and N', we would like to construct an NFA for the concatenated language $L(N) \circ L(N') = \left\{ xy \,\middle|\, x \in L(N) \text{ and } y \in L(N') \right\}$. The idea is to concatenate the two automata, by connecting the final states of the first automata, by ϵ-transitions, into the start state of the second NFA. We also make the accepting states of N not-accepting. The idea is that in the resulting NFA M, given input w, it "guesses" how to break it into two strings $x \in L(N)$ and $y \in L(N')$, so that $w = xy$. Now, there exists an execution trace for N accepting x, then we can jump into the starting state of N' and then use the execution trace accepting y, to reach an accepting state of the new NFA M. Here is how visually the resulting automata looks like.



Formally, we are given two NFAs $N = (Q, \Sigma, \delta, q_0, F)$ and $N' = (Q', \Sigma, \delta', q'_0, F')$, where $Q \cap Q' = \emptyset$. The new automata is

$$M = (Q \cup Q', \Sigma, \delta_M, q_0, F'),$$

where

$$\delta_M(q, c) = \begin{cases} \delta(q, \epsilon) \cup \{q'_0\} & q \in F, c = \epsilon \\ \delta(q, c) & q \in F, c \neq \epsilon \\ \delta(q, c) & q \in Q \setminus F, c \in \Sigma_\epsilon \\ \delta'(q, c) & q \in Q', c \in \Sigma_\epsilon. \end{cases}$$

Lemma 4.2 *Given two NFAs N and N' one can construct an NFA M, such that $L(M) = L(N) \circ L(N') = L(N)L(N')$.*

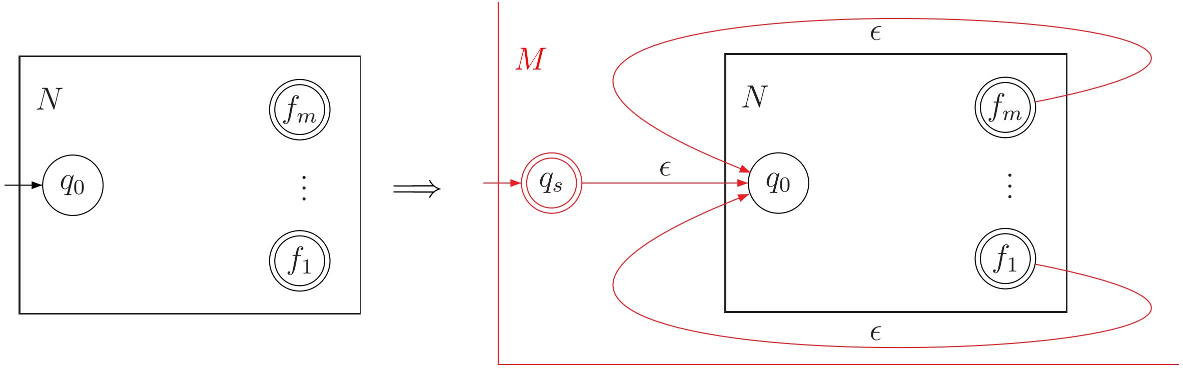
Proof: The construction is described above, and the proof of the correctness (of the construction) is easy and sketched above, so we skip it. You might want to verify that you know how to fill in the details for this proof (wink, wink). ■

4.3 NFA closure under the (Kleene) star

We are given a NFA N, and we would like to build an NFA for the Kleene star language

$$(L(N))^* = \left\{ w_1 w_2 \ldots w_k \;\middle|\; w_1, \ldots, w_k \in L(N), k \geq 0 \right\}.$$

The idea is to connect the final states of N back to the initial state using ϵ-transitions, so that it can loop back after recognizing a word of $L(N)$. As such, in the ith loop, during the execution, the new NFA M recognized the word w_i. Naturally, the NFA needs to guess when to jump back to the start state of N. One minor technicality, is that $\epsilon \in (L(N))^*$, but it might not be in $L(N)$. To overcome this, we introduce a new start state q_s (which is accepting), and its connected by (you guessed it) an ϵ-transition to the initial state of N. This way, $\epsilon \in L(M)$, and as such it recognized the required language. Visually, the transformation looks as follows.



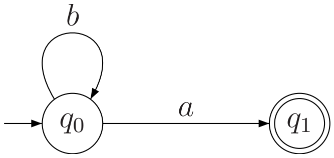
Formally, we are given the NFA $N = (Q, \Sigma, \delta, q_0, F)$, where $q_s \notin Q$. The new NFA is

$$M = \Big(\, Q \cup \{q_s\} \, , \; \Sigma, \; \delta_M, \; q_s, \; F \cup \{q_s\} \, \Big) \, ,$$

where

$$\delta_M(q, c) = \begin{cases} \delta(q, \epsilon) \cup \{q_0\} & q \in F, c = \epsilon \\ \delta(q, \epsilon) & q \in F, c \neq \epsilon \\ \delta(q, c) & q \in Q \setminus F \\ \{q_0\} & q = q_0, c = \epsilon \\ \emptyset & q = q_0, c \neq \epsilon. \end{cases}$$

Why the extra state? The construction for star needs some explanation. We add arcs from final states back to initial state to do the loop. But then we need to ensure that ϵ is accepted. It's tempting to just make the initial state final, but this doesn't work for examples like the following. So we need to add a new initial state to handle ϵ.



Notice that it also works to send the loopback arcs to the new initial state rather than to the old initial state.

Lemma 4.3 *Given an NFA N, one can construct an NFA M that accepts the language $(L(N))^*$.*

We don't give a proof; but you are encouraged to write a formal proof (as done for the union construction).

5 Regular Expressions

Regular expressions are a convenient notation to specify regular languages. We will prove in a few lectures that regular expressions can represent ***exactly*** the same languages that

DFAs can accept.

Let us fix an alphabet Σ. Here are the basic regular expressions:

regex	conditions	set represented
$\mathtt{a}$	$\mathtt{a} \in \Sigma$	$\{\mathtt{a}\}$
ϵ		$\{\epsilon\}$
$\emptyset$		$\{\}$

Thus, $\emptyset$ represents the empty language. But ϵ represents that language which has the empty word as its only word in the language.

In particular, for a regular expression $\langle \mathtt{exp} \rangle$, we will use the notation $L(\langle \mathtt{exp} \rangle)$ to denote the language associated with this regular expression. Thus,

$$L(\epsilon) = \{\epsilon\} \quad \text{and} \quad L(\emptyset) = \{\},$$

which are two *different* languages.

We will slightly abuse notations, and write a regular expression $\langle \mathtt{exp} \rangle$ when in reality what we refer to is the language $L(\langle \mathtt{exp} \rangle)$. (Abusing notations should be done with care, in cases where it reduces clutter, but it is well defined. Naturally, as Humpty Dumpty does, you need to define your "abused" notations explicitly.[2])

Suppose that $L(R)$ is the language represented by the regular expression R. Here are recursive rules that make complex regular expressions out of simpler ones. (Lecture will add some randomly-chosen small concrete examples.)

regex	conditions	set represented
$R \cup S$ or $R + S$	R, S regexes	$L(R) \cup L(S)$
$R \circ S$ or RS	R, S regexes	$L(R)L(S)$
R^*	R a regex	$L(R)^*$

And some handy shorthand notation:

regex	conditions	set represented
R^+	R a regex	$L(R)L(R)^*$
Σ		Σ

Exponentiation binds most tightly, then multiplication, then addition. Just like you probably thought. Use parentheses when you want to force a different interpretation.

Some specific boundary case examples:

1. $R\epsilon = R = \epsilon R$.

2. $R\emptyset = \emptyset = \emptyset R$.

[2] From *Through the Looking Glass*, by Lewis Carroll:

> 'And only one for birthday presents, you know. There's glory for you!'
>
> 'I don't know what you mean by "glory",' Alice said.
>
> Humpty Dumpty smiled contemptuously. 'Of course you don't – till I tell you. I meant "there's a nice knock-down argument for you!"'
>
> 'But "glory" doesn't mean "a nice knock-down argument",' Alice objected.
>
> 'When I use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean – neither more nor less.'
>
> 'The question is,' said Alice, 'whether you can make words mean so many different things.'
>
> 'The question is,' said Humpty Dumpty, 'which is to be master – that's all.'

This is a bit confusing, so let us see why this is true, recall that

$$R\emptyset = \left\{ xy \,\middle|\, x \in R \text{ and } y \in \emptyset \right\}.$$

But the empty set ($\emptyset$) does not contain any element, and as such, no concatenated string can be created. Namely, its the empty language.

3. $R \cup \emptyset = R$ (just like with any set).

4. $R \cup \epsilon = \epsilon \cup R$.

   This expression can not always be simplified, since ϵ might not be in the language $L(R)$.

5. $\emptyset^* = \{\epsilon\}$, since the empty word is always contain in the language generated by the star operator.

6. $\epsilon^* = \{\epsilon\}$.

5.1 More interesting examples

Suppose $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

1. $(\Sigma\Sigma)^*$ is the language of all even-length strings.

   (That is, the language associated with the regular expression $(\Sigma\Sigma)^*$ is made out of all the even-length strings over Σ.)

2. $\Sigma(\Sigma\Sigma)^*$ is all odd-length strings.

3. $\mathtt{a}\Sigma^*\mathtt{a} + \mathtt{b}\Sigma^*\mathtt{b} + \mathtt{c}\Sigma^*\mathtt{c}$ is all strings that start and end with the same character.

5.1.1 Regular expression for decimal numbers

Let $D = \{\mathtt{0}, \mathtt{1}, ..., \mathtt{9}\}$, and consider the alphabet $E = D \cup \{-, .\}$. Then decimal numbers have the form

$$(- \cup \epsilon)\, D^*\, (\epsilon \cup .)D^*.$$

But this does not force the number to contain any digits, which is probably wrong. As such, the correct expression is

$$(- \cup \epsilon)(D^+(\epsilon \cup .)D^* \cup D^*(\epsilon \cup .)D^+).$$

Notice that $\mathtt{a}^n$ is **not** a regular expression. Some things written with non-star exponents are regular and some are not. It depends on what conditions you put on n. E.g. $\left\{ \mathtt{a}^{2n} \,\middle|\, n \geq 0 \right\}$ is regular (even length strings of a's). But $\left\{ \mathtt{a}^n\mathtt{b}^n \,\middle|\, n \geq 0 \right\}$ is not regular.

However, $\mathtt{a}^3$ (or any other fixed power) is regular, as it just a shorthand for aaa. Similarly, if R is a regular expression, then R^3 is regular since its a shorthand for RRR.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 7: NFAs are equivalent to DFAs

9 February 2010

1 From NFAs to DFAs

1.1 NFA handling an input word

For the NFA $N = (Q, \Sigma, \delta, q_0, F)$ that has no ϵ-transitions, let us define $\Delta_N(X, c)$ to be the set of states that N might be in, if it was in a state of $X \subseteq Q$, and it handled the input c. Formally, we have that

$$\Delta_N(X, c) = \bigcup_{x \in X} \delta(x, c).$$

We also define $\Delta_N(X, \epsilon) = X$. Given a word $w = w_1, w_2, \ldots, w_n$, we define

$$\Delta_N(X, w) = \Delta_N\Big(\Delta_N(X, w_1 \ldots w_{n-1}), w_n\Big) = \Delta_N(\Delta_N(\ldots \Delta_N(\Delta_N(X, w_1), w_2) \ldots), w_n).$$

That is, $\Delta_N(X, w)$ is the set of all the states N might be in, if it starts from a state of X, and it handles the input w.

The proof of the following lemma is by an easy induction on the length of w.

Lemma 1.1 *Let $N = (Q, \Sigma, \delta, q_0, F)$ be a given NFA with no ϵ-transitions. For any word $w \in \Sigma^*$, we have that $q \in \Delta_N(\{q_0\}, w)$, if and only if, there is a way for N to be in q after reading w (when starting from the start state q_0).*

More details. We include the proof for the sake of completeness, but the reader should by now be able to fill in such a proof on their own.

Proof: The proof is by induction on the length of $w = w_1 w_2 \ldots w_k$.

If $k = 0$ then w is the empty word, and then N stays in q_0. Also, by definition, we have $\Delta_N(\{q_0\}, w) = \{q_0\}$, and the claim holds in this case.

Assume that the claim holds for all word of length at most n, and let $k = n + 1$ be the length of w. Consider a state q_{n+1} that N reaches after reading $w_1 w_2 \ldots w_n w_{n+1}$, and let q_n be the state N was before handling the character w_{n+1} and reaching q_{n+1}. By induction, we know that $q_n \in \Delta_N(\{q_0\}, w_1 w_2 \ldots w_n)$. Furthermore, we know that $q_{n+1} \in \delta(q_n, w_{n+1})$. As such, we have that

$$\begin{aligned} q_{n+1} \in \delta(q_n, w_{n+1}) \subseteq & \bigcup_{q \in \Delta_N(\{q_0\}, w_1 w_2 \ldots w_n)} \delta(q, w_{n+1}) \\ &= \Delta_N(\Delta_N(\{q_0\}, w_1 w_2 \ldots w_n), w_{n+1}) = \Delta_N(\{q_0\}, w_1 w_@ \ldots w_{n+1}) \\ &= \Delta_N(\{q_0\}, w). \end{aligned}$$

Thus, $q_{n+1} \in \Delta_N(\{q_0\}, w)$.

As for the other direction, if $p_{n+1} \in \Delta_N(\{q_0\}, w)$, then there must exist a state $p_n \in \Delta_N(\{q_0\}, w_1 \ldots w_n)$, such that $p_{n+1} \in \delta(p_n, w_{n+1})$. By induction, this implies that there is execution trace for N starting at q_0 and ending at p_n, such that N reads $w_1 \ldots w_n$ to reach p_n. As such, appending the transition from p_n to p_{n+1} (that read the character w_{n+1} to this trace, results in a trace for N that starts at q_0, reads w, and end up in the state p_{n+1}.

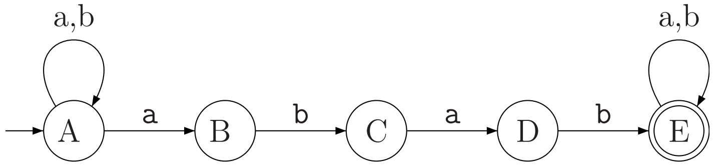
Putting these two arguments together, imply the claim. ■

1.2 Simulating NFAs with DFAs

One possible way of thinking about simulating NFAs is to consider each state to be a "light" that can be either on or off. In the beginning, only the initial state is on. At any point in time, all the states that the NFA might be in are turned on. As a new input character arrives, we need to update the states that are on.

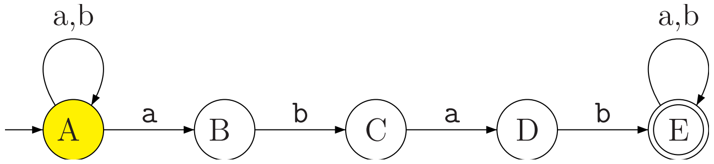
As a concrete examples, consider the automata below (which you had seen before), that accepts strings containing the substring `abab`.

(N1)



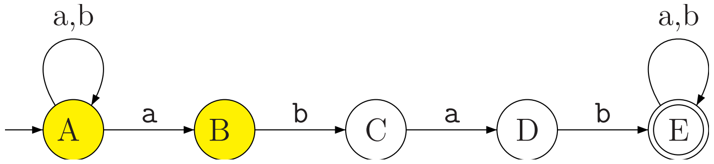
Let us run an explicit search for the above NFA (N1) on the input string `ababa`.

$t = 0$:



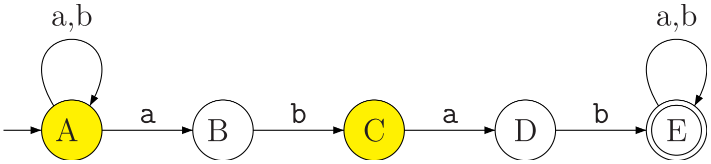
Remaining input: `ababa`.

$t = 1$:



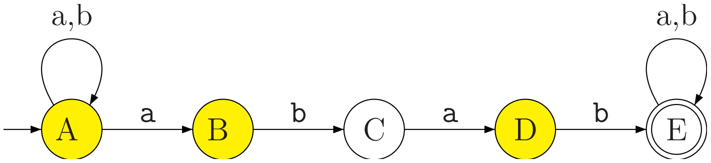
Remaining input: `baba`.

$t = 2$:



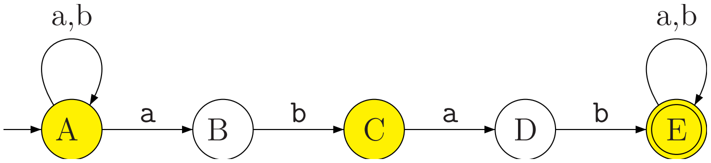
Remaining input: aba.

$t = 3$:



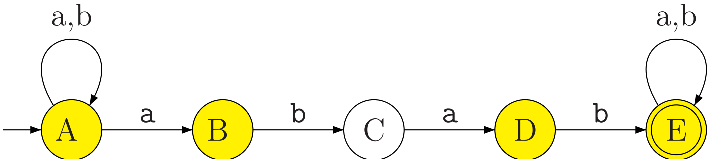
Remaining input: ba.

$t = 4$:



Remaining input: a.

$t = 5$:



Remaining input: ϵ.

Note, that (N1) accepted ababa because when its done reading the input, the accepting state is on.

This provide us with a scheme to simulate this NFA with a DFA: (i) Generate all possible configurations of states that might be turned on, and (ii) decide for each configuration what is the next configuration, what is the next configuration. In our case, in all configurations the first state is turned on. The initial configuration is when only state A is turned on. If this sounds familiar, it should, because what you get is just a big nasty, hairy DFA, as shown on the last page of this class notes. The same DFA with the unreachable states removed is shown in Figure 1.

Every state in the DFA of Figure 1 can be identified by the subset of the original states that is turned on (namely, the original automata might be any of these states).

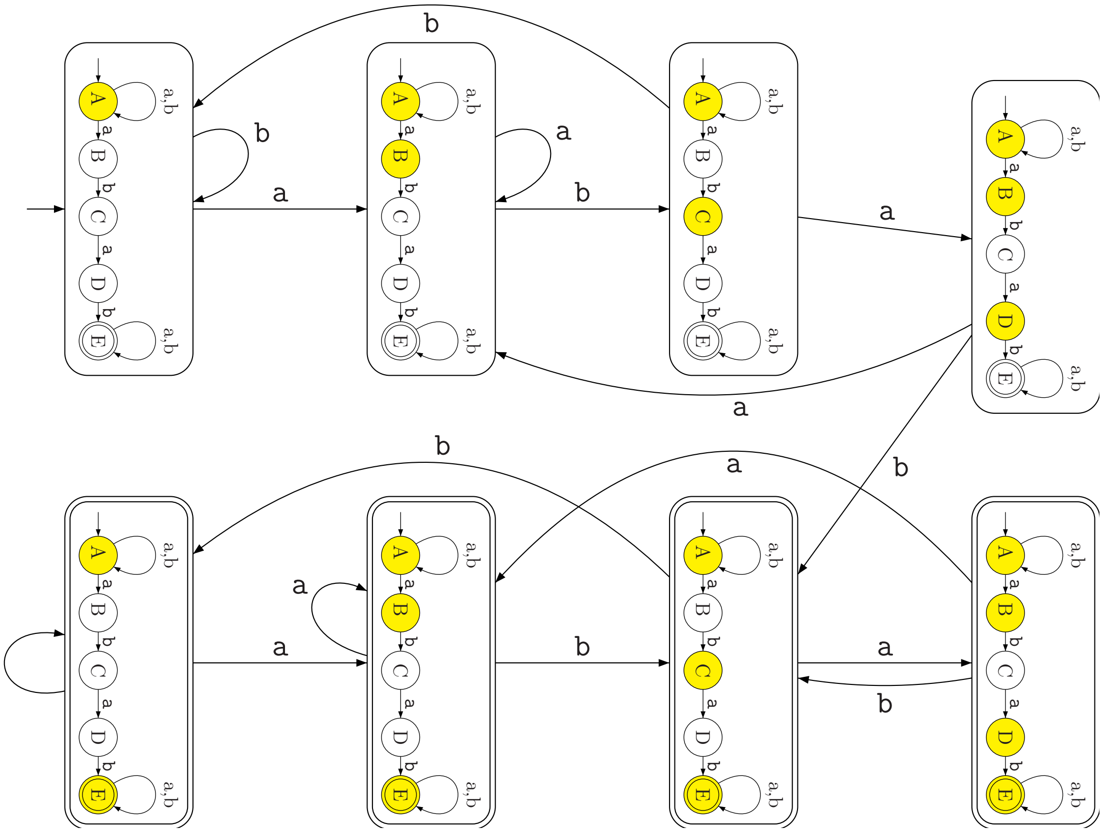
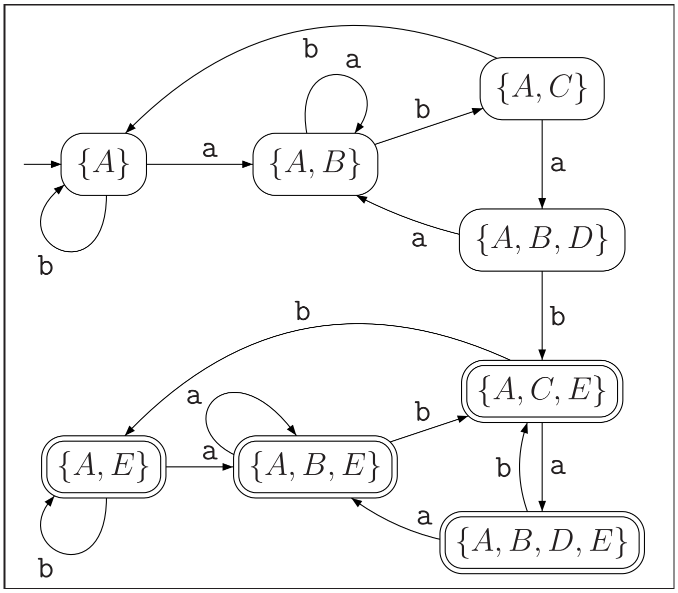


Figure 1: The resulting DFA



Thus, a more conventional drawing of this automata is shown on the right.

Thus, to convert an NFA N with a set of states Q into a DFA, we consider all the subsets of Q that N might be realized as. Namely, every subset of Q (i.e., a member of $\mathbb{P}(Q)$ – the power set of Q) is going to be a state in the new automata. Now, consider a subset $X \subseteq Q$, and for every input character $c \in \Sigma$, let us figure out in what states the original NFA N might be in if it is in one of the states of X, and it handles the characters c. Let Y be the resulting set of such states.

Clearly, we had just computed the transition function of the new (equivalent) DFA, showing that if the NFA is in one of the states of X, and we receive c, then the NFA now might be in one of the states of Y.

Now, if the initial state of the NFA N is q_0, then the new DFA M_{DFA} would start with the state (i.e., configuration) $\{q_0\}$ (since the original NFA might be only in q_0 at this point in time).

Its important that our simulation is *faithful*: At any point in time, if we are in state X in M_{DFA} then there is a path in the original NFA N, with the given input, to reach each state of Q that is in X (and similarly, X includes all the states that are reachable with such an input).

When does M_{DFA} accepts? Well, if it is in state X (here $X \subseteq Q$), then it accepts only if X includes one of the accepting states of the original NFA N.

Clearly, the resulting DFA M_{DFA} is equivalent to the original NFA.

1.3 The construction of a DFA from an NFA

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the given NFA that does not have any ϵ-transitions. The new DFA is going to be

$$M_{\mathsf{DFA}} = \left(\mathbb{P}(Q)\,, \Sigma, \widehat{\delta}, \widehat{q_0}, \widehat{F}\right),$$

where $\mathbb{P}(Q)$ is the power set of Q, and $\widehat{\delta}$ (the transition function), $\widehat{q_0}$ the initial state, and the set of accepting states $\widehat{F}$ are to be specified shortly. Note that the states of M_{DFA} are subsets of Q (which is slightly confusing), and as such the starting state of M_{DFA}, is $\widehat{q_0} = \{q_0\}$ (and not just q_0).

We need to specify the transition function, so consider $X \in \mathbb{P}(Q)$ (i.e., $X \subseteq Q$), and a character c. For a state $s \in X$, the NFA might go into any state in $\delta(s, c)$ after reading q. As such, the set of all possible states the NFA might be in, if it started from a state in X, and received c, is the set

$$Y = \bigcup_{s \in X} \delta(s, c).$$

As such, the transition of M_{DFA} from X receiving c is the state of M_{DFA} defined by Y. Formally,

$$\widehat{\delta}(X, c) = Y = \bigcup_{s \in X} \delta(s, c). \tag{1}$$

As for the accepting states, consider a state $X \in \mathbb{P}(Q)$ of M_{DFA}. Clearly, if there is a state of F in X, then X is an accepting state; namely, $F \cap X \neq \emptyset$. Thus,

$$\widehat{F} = \left\{ X \;\middle|\; X \in \mathbb{P}(Q)\,, X \cap F \neq \emptyset \right\}.$$

1.3.1 Proof of correctness

Claim 1.2 *For any $w \in \Sigma^*$, the set of states reached by the NFA N on w is precisely the state reached by M_{DFA} on w. That is $\Delta_N(\{q_0\}\,, w) = \widehat{\delta}(\{q_0\}\,, w)$.*

Proof: The proof is by induction on the length of w.

If w is the empty word, then N is at q_0 after reading ϵ (i.e., $\Delta_N(\{q_0\}\,, \epsilon) = \{q_0\}$), and the M_{DFA} is still in its initial state which is $\{q_0\}$.

So assume that the claim holds for all words of length at most k.

Let $w = w_1 w_2 \ldots w_{k+1}$. Let X be the set of states that N might reach from q_0 after reading $w' = w_1 \ldots w_n$; that is $X = \Delta_N(\{q_0\}, w')$. By the induction hypothesis, we have that M_{DFA} is in the state X after reading w' (formally, we have that $\widehat{\delta}(\{q_0\}, w') = X$).

Now, the NFA N, when reading the last character w_{k+1}, can start from any state of X, and use any transition from such a state that reads the character w_{k+1}. Formally, the NFA N is in one of the states of

$$Z = \Delta_N(X, w_{k+1}) = \bigcup_{s \in X} \delta(s, w_{k+1}).$$

Similarly, by the definition of M_{DFA}, we have that from the state X, after reading w_{k+1}, the DFA M_{DFA} is in the state

$$Y = \widehat{\delta}(X, w_{k+1}) = \bigcup_{s \in X} \delta(s, w_{k+1}),$$

see Eq. (1). But clearly, $Z = Y$, which establishes the claim. ■

Lemma 1.3 *Any NFA N, without ϵ-transitions, can be converted into a DFA M_{DFA}, such that M_{DFA} accepts the same language as N.*

Proof: The construction is described above.

So consider a word $w \in \Sigma^*$, and observe that $w \in L(N)$ if and only if, the set of states N might be in after reading w (that is $\Delta_N(\{q_0\}, w)$), contains an accepting state of N. Formally, $w \in L(N)$ if and only if

$$\Delta_N\Big(\{q_0\}, w\Big) \cap F \neq \emptyset.$$

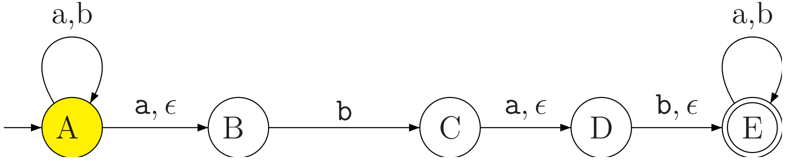
The DFA M_{DFA} is in the state $\widehat{\delta}(\{q_0\}, w)$ after reading w. Claim 1.2, implies that $Y = \widehat{\delta}(\{q_0\}, w) = \Delta_N(\{q_0\}, w)$. By construction, the M_{DFA} accepts at this state, if and only if, $Y \in \widehat{F}$, which equivalent to that Y contains a final state of N. That is $Y \cap F \neq \emptyset$. Namely, M_{DFA} accepts w if

$$\widehat{\delta}\Big(\{q_0\}, w\Big) \cap F \neq \emptyset \iff \Delta_N\Big(\{q_0\}, w\Big) \cap F \neq \emptyset.$$

Implying that M_{DFA} accepts w if and only if N accepts w. ■

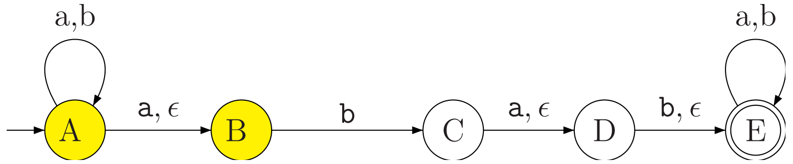
1.3.2 Handling ϵ-transitions

Now, we would like to handle a general NFA that might have ϵ-transitions. The problem is demonstrated in the following NFA in its initial configuration:

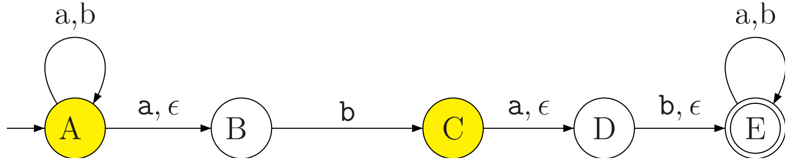


Clearly, the initial configuration here is $\{A, B\}$ (and not the one drawn above), since the automata can immediately jump to B if the NFA is already in A. So, the configuration $\{A\}$ should not be considered at all. As such, the true initial configuration for this automata is

(N2)

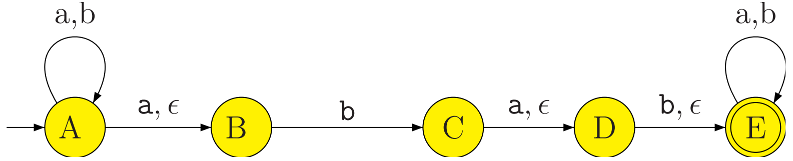


Next, consider the following more interesting configuration.



But here, not only we can jump from A to B, but we can also jump from C to D, and from D to E. As such, this configuration is in fact the following configuration

(N3)



In fact, this automata can only be in these two configurations because of the ϵ-transitions.

So, let us formalize the above idea: Whenever the NFA N might be in a state s, we need to extend the configuration to all the states of the NFA reachable by ϵ-transitions from s. Let $R_\epsilon(s)$ denote the set of all states of N that are reachable by a sequence of ϵ-transitions from s (s is also in $R_\epsilon(s)$ naturally, since we can reach s without moving anywhere).

Thus, if N might be any state of $X \subseteq Q$, then it might be in any state of

$$\mathcal{E}(X) = \bigcup_{s \in X} R_\epsilon(s)\,.$$

As such, whenever we consider the set of states X for Q, in fact, we need to consider the *extended set of states* $\mathcal{E}(X)$. As such, for the above automata, we have

$$\mathcal{E}(\{A\}) = \{A, B\} \quad \text{and} \quad \mathcal{E}(\{A, C\}) = \{A, B, C, D, E\}\,.$$

Now, we can essentially repeat the above proof.

Theorem 1.4 *Any NFA N (with or without ϵ-transitions) can be converted into a DFA M_{DFA}, such that M_{DFA} accepts the same language as N.*

Proof: Let $N = (Q, \Sigma, \delta, q_0, F)$. The new DFA is going to be

$$M_{\mathsf{DFA}} = \left(\mathbb{P}(Q)\,, \Sigma, \delta_M, q_S, \widehat{F}\right).$$

Here, $\mathbb{P}(Q)$, Σ and $\widehat{F}$ are the same as above.

Now, for $X \in \mathbb{P}(Q)$ and $c \in \Sigma$, let

$$\delta_M(X) = \mathcal{E}\left(\widehat{\delta}(X, c)\right),$$

where $\widehat{\delta}$ is the old transition function from the proof of Lemma 1.3; namely, we always extend the new set of states to include all the states we can reach by ϵ-transitions. Similarly, the initial state is now

$$q_S = \mathcal{E}(\{q_0\})\,.$$

It is now straightforward to verify that the new DFA is indeed equivalent to the original NFA, using the argumentation of Lemma 1.3. ■

CS 373: Theory of Computation
Madhusudan Parthasarathy

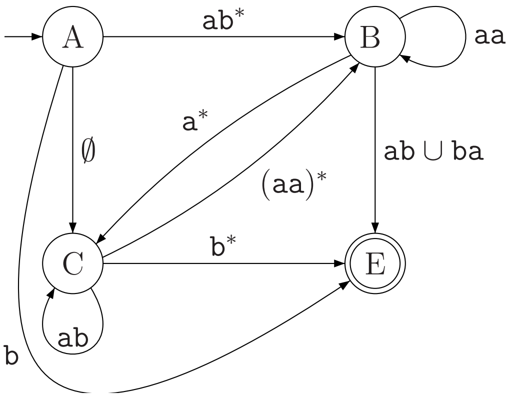
Lecture 8: From DFAs/NFAs to Regular Expressions

11 February 2010

In this lecture, we will show that any DFA can be converted into a regular expression. Our construction would work by allowing regular expressions to be written on the edges of the DFA, and then showing how one can remove states from this generalized automata (getting a new equivalent automata with the fewer states). In the end of this state removal process, we will remain with a generalized automata with a single initial state and a single accepting state, and it would be then easy to convert it into a single regular expression.

1 From NFA to regular expression

1.1 GNFA— A Generalized NFA



Consider an NFA N where we allowed to write any regular expression on the edges, and not only just symbols. The automata is allowed to travel on an edge, if it can matches a prefix of the unread input, to the regular expression written on the edge. We will refer to such an automata as a ***GNFA*** (***generalized non-deterministic finite automata*** [Don't you just love all these shortcuts?]).

Thus, the GNFA on the right, accepts the string abbbbaaba, since

$$A \xrightarrow{\texttt{abbbb}} B \xrightarrow{\texttt{aa}} B \xrightarrow{\texttt{ba}} E.$$

To simplify the discussion, we would enforce the following conditions:

(C1) There are transitions going from the initial state to all other states, and there are no transitions into the initial state.

(C2) There is a single accept state that has only transitions coming into it (and no outgoing transitions).

(C3) The accept state is distinct from the initial state.

(C4) Except for the initial and accepting states, all other states are connected to all other states via a transition. In particular, each state has a transition to itself.

When you can not actually go between two states, a GNFA has a transitions labelled with ∅, which will not match any string of input characters. We do not have to draw these transitions explicitly in the state diagrams.

1.2 Top-level outline of conversion

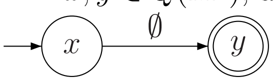
We will convert a DFA to a regular expression as follows:

(A) Convert DFA to a GNFA, adding new initial and final states.

(B) Remove all states one-by-one, until we have only the initial and final states.

(C) Output regex is the label on the (single) transition left in the GNFA. (The word ***regex*** is just a shortcut for regular expression.)

Lemma 1.1 *A DFA M can be converted into an equivalent GNFA G.*

Proof: We can consider M to be an NFA. Next, we add a special initial state q_{init} that is connected to the old initial state via ϵ-transition. Next, we add a special final state q_{final}, such that all the final states of M are connected to q_{final} via an ϵ-transition. The modified NFA M' has an initial state and a single final state, such that no transition enters the initial state, and no transition leaves the final state, thus M' comply with conditions (C1–C3) above. Next, we consider all pair of states $x, y \in Q(M')$, and if there is no transition between them, we introduce the transition 

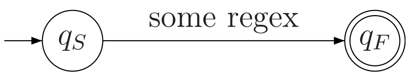
. The resulting GNFA G from M' is now compliant also with condition (C4).

It is easy now to verify that G is equivalent to the original DFA M. ■

We will remove all the intermediate states from the GNFA, leaving a GNFA with only initial and final states, connected by one transition with a (typically complex) label on it. The equivalent regular expression is obvious: the label on the transition.

Lemma 1.2 *Given a GNFA N with $k = 2$ states, one can generate an equivalent regular expression.*

Proof: A GNFA with only two states (that comply with conditions (C1)-(C4)) have the following form.



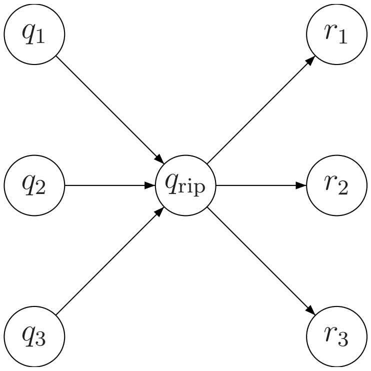
The GNFA has a single transition from the initial state to the accepting state, and this transition has the regular expression R associated with it. Since the initial state and the accepting state do not have self loops, we conclude that N accepts all words that matches the regular expression R. Namely, $L(N) = L(R)$. ■

1.3 Details of ripping out a state

We first describe the construction. Since $k > 2$, there is at least one state in N which is not initial or accepting, and let q_{rip} denote this state. We will "rip" this state out of N and fix the GNFA, so that we get a GNFA with one less state.

Transition paths going through q_{rip} might come from any of a variety of states q_1, q_2, etc. They might go from q_{rip} to any of another set of states r_1, r_2, etc.

For each pair of states q_i and r_i, we need to convert the transition through q_{rip} into a direct transition from q_i to r_i.

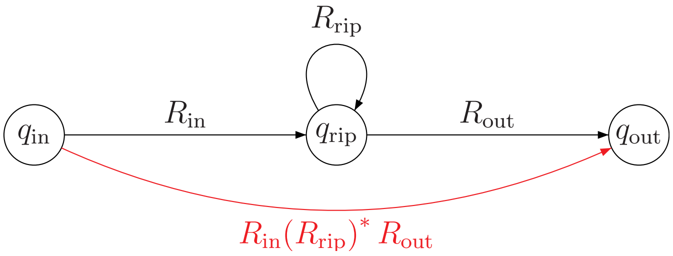


1.3.1 Reworking connections for specific triple of states

To understand how this works, let us focus on the connections between q_{rip} and two other specific states q_{in} and q_{out}. Notice that q_{in} and q_{out} might be the same state, but they both have to be different from q_{rip}.

The state q_{rip} has a self loop with regular expression R_{rip} associated with it. So, consider a fragment of an accepting trace that goes through q_{rip}. It transition into q_{rip} from a state q_{in} with a regular expression R_{in} and travels out of q_{rip} into state q_{out} on an edge with the associated regular expression being R_{out}. This trace, corresponds to the regular expression R_{in} followed by 0 or more times of traveling on the self loop (R_{rip} is used each time we traverse the loop), and then a transition out to q_{out} using the regular expression R_{out}. As such, we can introduce a direct transition from q_{in} to q_{out} with the regular expression

$$R = R_{\text{in}}(R_{\text{rip}})^* R_{\text{out}}.$$



Clearly, any fragment of a trace traveling $q_{\text{in}} \to q_{\text{rip}} \to q_{\text{out}}$ can be replaced by the direct transition $q_{\text{in}} \xrightarrow{\texttt{R}} q_{\text{out}}$. So, let us do this replacement for any two such stages, we connect them directly via a new transition, so that they no longer need to travel through q_{rip}.

Clearly, if we do that for all such pairs, the new automata accepts the same language, but no longer need to use q_{rip}. As such, we can just remove q_{rip} from the resulting automata. And let M' denote the resulting automata.

The automata M' is not quite legal, yet. Indeed, we will have now parallel transitions because of the above process (we might even have parallel self loops). But this is easy to fix: We replace two such parallel transitions $q_i \xrightarrow{\texttt{R}_1} q_j$ and $q_i \xrightarrow{\texttt{R}_2} q_j$, by a single transition

$$q_i \xrightarrow{\texttt{R}_1+\texttt{R}_2} q_j.$$

As such, for the triple $q_{\text{in}}, q_{\text{rip}}, q_{\text{out}}$, if the original label on the direct transition from q_{in} to q_{out} was originally R_{dir}, then the output label for the new transition (that skips q_{rip}) will be

$$R_{dir} \; + \; R_{\text{in}}(R_{\text{rip}})^* R_{\text{out}}. \tag{1}$$

Clearly the new transition, is equivalent to the two transitions it replaces. If we repeat this process for all the parallel transitions, we get a new GNFA M which has $k-1$ states, and furthermore it accepts exactly the same language as N.

1.4 Proof of correctness of the ripping process

Lemma 1.3 *Given a GNFA N with $k > 2$ states, one can generate an equivalent GNFA M with $k-1$ states.*

Proof: Since $k > 2$, N contains least one state in N which is not accepting, and let q_{rip} denote this state. We will "rip" this state out of N and fix the GNFA, so that we get a GNFA with one less state.

For every pair of states q_{in} and q_{out}, both distinct from q_{rip}, we replace the transitions that go through q_{rip} with direct transitions from q_{in} to q_{out}, as described in the previous section.

Correctness. Consider an accepting trace T for N for a word w. If T does not use the state q_{rip} than the same trace exactly is an accepting trace for M. So, assume that it uses q_{rip}, in particular, the trace looks like

$$T = \ldots q_i \xrightarrow{\mathtt{S_i}} q_{\text{rip}} \overbrace{\xrightarrow{\mathtt{S_{i+1}}} q_{\text{rip}} \ldots \xrightarrow{\mathtt{S_{j-1}}} q_{\text{rip}}}^{\text{0 or more times}} \xrightarrow{\mathtt{S_{j-1}}} q_j \ldots .$$

Where $S_i S_{i+1} \ldots, S_j$ is a substring of w. Clearly, $S_i \in R_{\text{in}}$, where R_{in} is the regular expression associated with the transition $q_i \rightarrow q_{\text{rip}}$. Similarly, $S_{j-1} \in R_{\text{out}}$, where R_{out} is the regular expression associated with the transition $q_{\text{rip}} \rightarrow q_j$. Finally, $S_{i+1}S_{i+2} \cdots S_{j-1} \in (R_{\text{rip}})^*$, where R_{rip} is the regular expression associated with the self loop of q_{rip}.

Now, clearly, the string $S_i S_{i+1} \ldots S_j$ matches the regular expression $R_{\text{in}}(R_{\text{out}})^* \, R_{\text{out}}$. in particular, we can replace this portion of the trace of T by

$$T = \ldots q_i \xrightarrow{\mathtt{S_i S_{i+1} \ldots S_{j-1} S_j}} q_j \ldots .$$

This transition is using the new transition between q_i and q_j introduced by our construction. Repeating this replacement process in T till all the appearances of q_{rip} are removed, results in an accepting trace $\widehat{T}$ of M. Namely, we proved that any string accepted by N is also accepted by M.

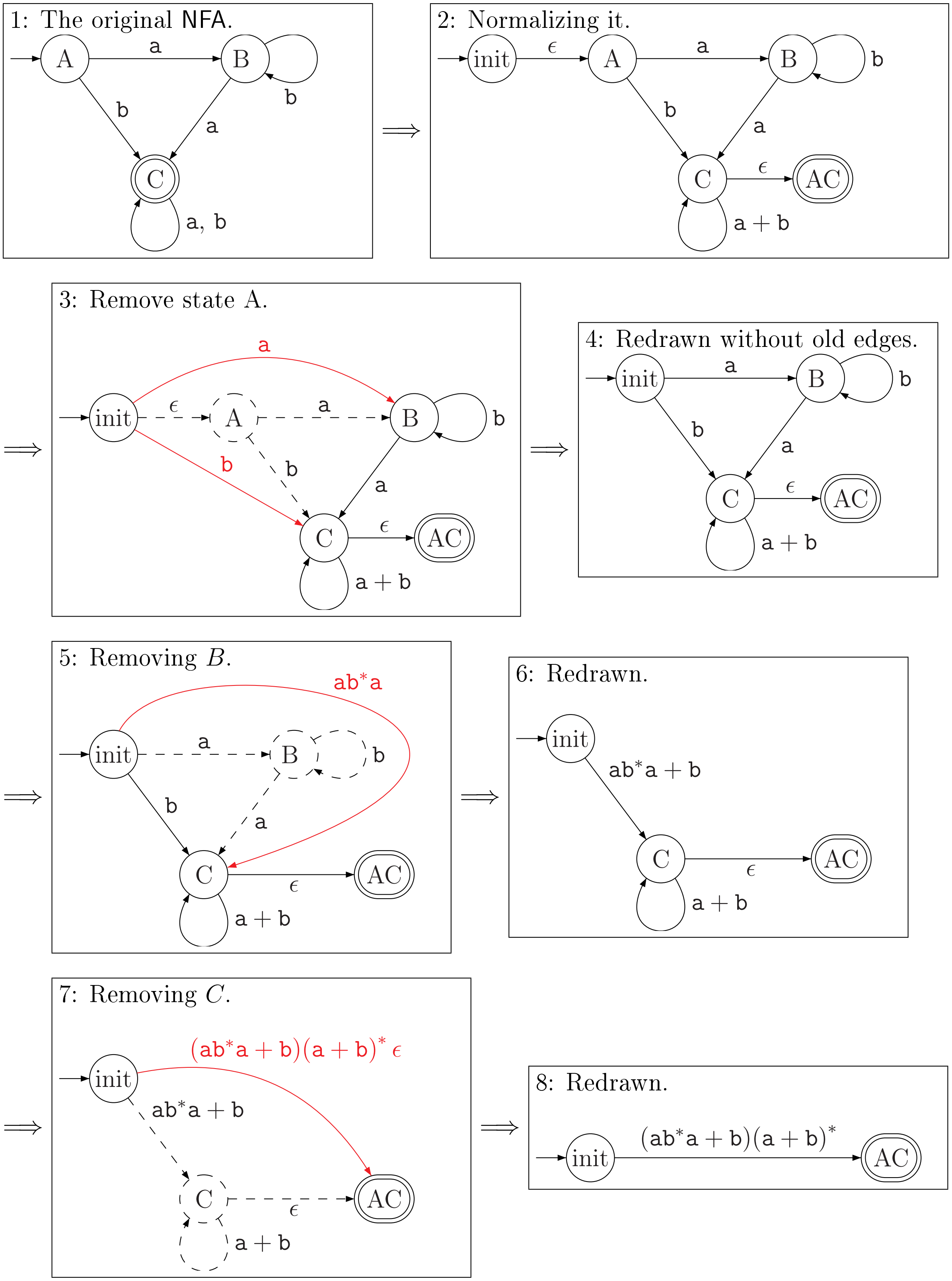
We need also to prove the other direction. Namely, given an accepting trace for M, we can rewrite it into an equivalent trace of N which is accepting. This is easy, and done in a similar way to what we did above. Indeed, if a portion of the trace uses a new transition of M (that does not appear in N), we can place it by a fragment of transitions going through q_{rip}. In light of the above proof, it is easy and we omit the straightforward but tedious details. ■

Theorem 1.4 *Any DFA can be translated into an equivalent regular expression.*

Proof: Indeed, convert the DFA into a GNFA N. As long as N has more than two states, reduce its number of states by removing one of its states using Lemma 1.3. Repeat this process till N has only two states. Now, we convert this GNFA into an equivalent regular expression using Lemma 1.2. ■

2 Examples

2.1 Example: From GNFA to regex in 8 easy figures



1: The original NFA.

2: Normalizing it.

3: Remove state A.

4: Redrawn without old edges.

5: Removing B.

6: Redrawn.

7: Removing C.

8: Redrawn.

Thus, this automata is equivalent to the regular expression $(\mathtt{ab}^*\mathtt{a} + \mathtt{b})(\mathtt{a} + \mathtt{b})^*$.

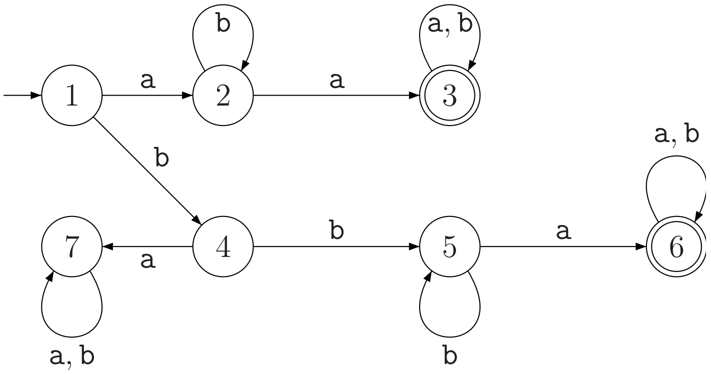
Lecture 09: Myhill-Nerode Theorem

16 February 2010

In this lecture, we will see that every language has a unique minimal DFA. We will see this fact from two perspectives. First, we will see a practical algorithm for minimizing a DFA, and provide a theoretical analysis of the situation.

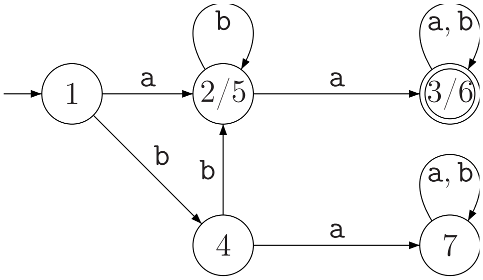
1 On the number of states of DFA

1.1 Starting a DFA from different states



Consider the DFA on the right. It has a particular defined start state. However, we could start it from any of its states. If the original DFA was named M, define M_q to be the DFA with its start state changed to state q. Then the language L_q, is the one accepted if you start at q.

For example, in this picture, L_3 is $(\mathtt{a}+\mathtt{b})^*$, and L_6 is the same. Also, L_2 and L_5 are both $\mathtt{b}^*\mathtt{a}(\mathtt{a}+\mathtt{b})^*$. Finally, L_7 is $\emptyset$.



Suppose that $L_q = L_r$, for two states q and r. Then once we get to q or r, the DFA is going to do the same thing from then on (i.e., its going to accept or reject *exactly* the same strings).

So these two states can be merged. In particular, in the above automata, we can merge 2 and 5 and the states 3 and 6. We can the new automata, depicted on the right.

1.2 Suffix Languages

Let Σ be some alphabet.

Definition 1.1 Let $L \subseteq \Sigma^*$ be any language.

The ***suffix language*** of L with respect to a word $x \in \Sigma^*$ is defined as

$$[\![L/x]\!] = \left\{ y \,\middle|\, x\, y \in L \right\}.$$

In words, $[\![L/x]\!]$ is the language made out of all the words, such that if we append x to them as a prefix, we get a word in L.

The ***class of suffix languages*** of L is

$$\mathcal{C}(L) = \left\{ [\![L/x]\!] \;\middle|\; x \in \Sigma^* \right\}.$$

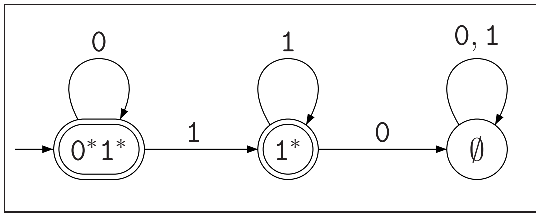
Example 1.2 For example, if $L = \mathtt{0}^*\mathtt{1}^*$, then:

- $[\![L/\epsilon]\!] = \mathtt{0}^*\mathtt{1}^* = L$
- $[\![L/\mathtt{0}]\!] = \mathtt{0}^*\mathtt{1}^* = L$
- $[\![L/\mathtt{0}^i]\!] = \mathtt{0}^*\mathtt{1}^* = L$, for any $i \in \mathbb{N}$
- $[\![L/\mathtt{1}]\!] = \mathtt{1}^*$
- $[\![L/\mathtt{1}^i]\!] = \mathtt{1}^*$, for any $i \geq 1$
- $[\![L/\mathtt{10}]\!] = \left\{ y \;\middle|\; \mathtt{10}y \in L \right\} = \emptyset$.

Hence there are only three suffix languages for L: $\mathtt{0}^*\mathtt{1}^*$, $\mathtt{1}^*$, $\emptyset$. So $\mathcal{C}(L) = \left\{ \mathtt{0}^*\mathtt{1}^*,\ \mathtt{1}^*,\ \emptyset \right\}$.

As the above example demonstrates, if there is a word x, such that any word w that have x as a prefix is not in L, then $[\![L/x]\!] = \emptyset$, which implies that $\emptyset$ is one of the suffix languages of L.

Example 1.3 The above suggests the following automata for the language of Example 1.2: $L = \mathtt{0}^*\mathtt{1}^*$.



And clearly, this is the automata with the smallest number of states that accepts this language.

1.2.1 Regular languages have few suffix languages

Now, consider a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepting some language L. Let $x \in \Sigma^*$, and let M reach the state q on reading x. The suffix language $[\![L/x]\!]$ is precisely the set of strings w, such that xw is in L. But this is exactly the same as L_q. That is, $[\![L/x]\!] = L_q$, where q is the state reached by M on reading x. Hence the suffix languages of a regular language accepted by a DFA are precisely those languages L_q, where $q \in Q$.

Notice that the definition of suffix languages is more general, because it can also be applied to non-regular languages.

Lemma 1.4 *For a regular language L, the number of different suffix languages it has is bounded; that is $\mathcal{C}(L)$ is bounded by a constant (that depends on L).*

Proof: Consider the DFA $M = (Q, \Sigma, \delta, q_0, F)$ that accepts L. For any string x, the suffix language $[\![L/x]\!]$ is just the languages associated with L_q, where q is the state M is in after reading x.

Indeed, the suffix language $[\![L/x]\!]$ is the set of strings w such that $xw \in L$. Since the DFA reaches q on x, it is clear that the suffix language of x is precisely the language accepted by M starting from the state q, which is L_q. Hence, for every $x \in \Sigma^*$, $[\![L/x]\!] = L_{\delta(q_0,x)}$, where q is the state the automaton reaches on x.

As such, any suffix language of L is realizable as the language of a state of M. Since the number of states of M is some constant k, it follows that the number of suffix languages of L is bounded by k. ■

An immediate implication of the above lemma is the following.

Lemma 1.5 *If a language L has infinite number of suffix languages, then L is not regular.*

1.2.2 The suffix languages of a non-regular language

Consider the language $L = \left\{ \mathtt{a}^n \mathtt{b}^n \;\middle|\; n \in \mathbb{N} \right\}$. The suffix language of L for $\mathtt{a}^i$ is

$$[\![L/\mathtt{a}^i]\!] = \left\{ \mathtt{a}^{n-i} \mathtt{b}^n \;\middle|\; n \in \mathbb{N} \right\}.$$

Note, that $\mathtt{b}^i \in [\![L/\mathtt{a}^i]\!]$, but this is the only string made out of only $\mathtt{b}$s that is in this language. As such, for any i, j, where i and j are different, the suffix language of L with respect to $\mathtt{a}^i$ is different from that of L with respect to $\mathtt{a}^j$ (i.e. $[\![L/\mathtt{a}^i]\!] \neq [\![L/\mathtt{a}^j]\!]$). Hence L has infinitely many suffix languages, and hence is not regular, by Lemma 1.5.

Let us summarize what we had seen so far:

- Any state of a DFA of a language L is associated with a suffix language of L.
- If two states are associated with the same suffix language, that we can merge them into a single state.
- At least one non-regular language $\left\{ \mathtt{a}^n \mathtt{b}^n \;\middle|\; n \in \mathbb{N} \right\}$ has an infinite number of suffix languages.

It is thus natural to conjecture that the number of suffix languages of a language, is a good indicator of how many states an automata for this language would require. And this is indeed true, as the following section testifies.

2 Regular Languages and Suffix Languages

2.1 A few easy observations

Lemma 2.1 *If $\epsilon \in \llbracket L/x \rrbracket$ if and only if $x \in L$.*

Proof: By definition, if $\epsilon \in \llbracket L/x \rrbracket$ then $x = x\epsilon \in L$. Similarly, if $x \in L$, then $x\epsilon \in L$, which implies that $\epsilon \in \llbracket L/x \rrbracket$. ■

Lemma 2.2 *Let L be a language over alphabet Σ. For all $x, y \in \Sigma^*$ we have that if $\llbracket L/x \rrbracket = \llbracket L/y \rrbracket$ then for all $\mathtt{a} \in \Sigma$ we have $\llbracket L/x\mathtt{a} \rrbracket = \llbracket L/y\mathtt{a} \rrbracket$.*

Proof: If $w \in \llbracket L/x\mathtt{a} \rrbracket$, then (by definition) $x\mathtt{a}w \in L$. But then, $\mathtt{a}w \in \llbracket L/x \rrbracket$. Since $\llbracket L/x \rrbracket = \llbracket L/y \rrbracket$, this implies that $\mathtt{a}w \in \llbracket L/y \rrbracket$, which implies that $y\mathtt{a}w \in L$, which implies that $w \in \llbracket L/y\mathtt{a} \rrbracket$. This implies that $\llbracket L/x\mathtt{a} \rrbracket \subseteq \llbracket L/y\mathtt{a} \rrbracket$, a symmetric argument implies that $\llbracket L/y\mathtt{a} \rrbracket \subseteq \llbracket L/x\mathtt{a} \rrbracket$. We conclude that $\llbracket L/x\mathtt{a} \rrbracket = \llbracket L/y\mathtt{a} \rrbracket$. ■

2.2 Regular languages and suffix languages

We can now state a characterization of regular languages in term of suffix languages.

Theorem 2.3 (Myhill-Nerode theorem.) *A language $L \subseteq \Sigma^*$ is regular if and only if the number of suffix languages of L is finite (i.e. $\mathcal{C}(L)$ is finite).*

Moreover, if $\mathcal{C}(L)$ contains exactly k languages, we can build a DFA for L that has k states; also, any *DFA accepting L must have k states.*

Proof: If L is regular, then $\mathcal{C}(L)$ is a finite set by Lemma 1.4.

Second, let us show that if $\mathcal{C}(L)$ is finite, then L is regular. Let the suffix languages of L be

$$\mathcal{C}(L) = \left\{ \llbracket L/x_1 \rrbracket, \llbracket L/x_2 \rrbracket, \ldots, \llbracket L/x_k \rrbracket \right\}. \tag{1}$$

Note that for any $y \in \Sigma^*$, $\llbracket L/y \rrbracket = \llbracket L/x_j \rrbracket$, for some $j \in \{1, \ldots, k\}$.

We will construct a DFA whose states are the various suffix languages of L; hence we will have k states in the DFA. Moreover, the DFA will be designed such that after reading y, the DFA will end up in the state $\llbracket L/y \rrbracket$.

The DFA is $M = (Q, \Sigma, q_0, \delta, F)$ where

- $Q = \left\{ \llbracket L/x_1 \rrbracket, \llbracket L/x_2 \rrbracket, \ldots, \llbracket L/x_k \rrbracket \right\}$
- $q_0 = \llbracket L/\epsilon \rrbracket$,
- $F = \left\{ \llbracket L/x \rrbracket \;\middle|\; \epsilon \in \llbracket L/x \rrbracket \right\}$. Note, that by Lemma 2.1, if $\epsilon \in \llbracket L/x \rrbracket$ then $x \in L$.
- $\delta\big(\llbracket L/x \rrbracket, \mathtt{a}\big) = \llbracket L/x\mathtt{a} \rrbracket$ for every $\mathtt{a} \in \Sigma$.

The transition function δ is well-defined because of Lemma 2.2.

We can now prove, by induction on the length of x, that after reading x, the DFA reaches the state $\llbracket L/x \rrbracket$. If $x \in L$, then $\epsilon \in \llbracket L/x \rrbracket$, which implies that $\delta(q_0, x) = \llbracket L/x \rrbracket \in F$. Thus,

$x \in L(M)$. Similarly, if $x \in L(M)$, then $[\![L/x]\!] \in F$, which implies that $\epsilon \in [\![L/x]\!]$, and by Lemma 2.1 this implies that $x \in L$. As such, $L(M) = L$.

We had shown that the DFA M accepts L, which implies that L is regular, furthermore M has k states.

We next prove that *any* DFA for L must have at least k states. So, let $N = (Q', \Sigma, \delta_N q_{\text{init}}, F)$ any DFA accepting L. The language L has k suffix languages, generated by the strings $x_1, x_2, \ldots, x_k$, see Eq. (1).

For any $i \neq j$, we have that $[\![L/x_i]\!] \neq [\![L/x_j]\!]$. As such, there must exist a word w such that $w \in [\![L/x_j]\!]$ and $w \notin [\![L/x_j]\!]$ (the symmetric case where $w \in [\![L/x_j]\!] \setminus [\![L/x_i]\!]$ is handled in a similar fashion. But then, $x_i w \in L$ and $x_j w \notin L$. Namely, $N(q_{\text{init}}, x) \neq N(q_{\text{init}}, y)$, and the two states that N reaches for x_i and x_j respectively, are distinguishable. Formally, let $q_i = \delta(q_{\text{init}}, x_i)$, for $i = 1, \ldots, k$. All these states are pairwise distinguishable, which implies that N must have at least k states. ■

Remark 2.4 The full Myhill-Nerode theorem also shows that all minimal DFAs for L are isomorphic, i.e. have identical transitions as well as the same number of states, but we will not show that part.

This is done by arguing that any DFA for L that has k states must be *identical* to the DFA we created above. This is a bit more involved notationally, and is proved by showing a $1-1$ correspondence between the two DFAs and arguing they must be connected the same way. We omit this part of the theorem and proof.

2.3 Examples

Let us explain the theorem we just proved using an example.

Consider the language $L \subseteq \{\mathtt{a}, \mathtt{b}\}^*$:

$$L = \left\{ w \;\middle|\; w \text{ has an odd number of } \mathtt{a}\text{'s} \right\}.$$

The suffix language of $x \in \Sigma^*$, where x has an even number of a's is:

$$[\![L/x]\!] = \left\{ w \;\middle|\; w \text{ has an odd number of } \mathtt{a}\text{'s} \right\} = L.$$

The suffix language of $x \in \Sigma^*$, where x has an odd number of a's is:

$$[\![L/x]\!] = \left\{ w \;\middle|\; w \text{ has an even number of } \mathtt{a}\text{'s} \right\}.$$

Hence there are only two distinct suffix languages for L. By the theorem, we know L must be regular and the minimal DFA for L has two states. Going with the construction of the DFA mentioned in the proof of the theorem, we see that we have two states, $q_0 = [\![L/\epsilon]\!]$ and $q_1 = [\![L/\mathtt{a}]\!]$. The transitions are as follows:

- From $q_0 = [\![L/\epsilon]\!]$, on a we go to $[\![L/\mathtt{a}]\!]$, which is the state q_1.
- From $q_0 = [\![L/\epsilon]\!]$, on b we go to $[\![L/b]\!]$, which is same as $[\![L/\epsilon]\!]$, i.e. the state q_0.
- From $q_1 = [\![L/\mathtt{a}]\!]$, on a we go to $[\![L/aa]\!]$, which is same as $[\![L/\epsilon]\!]$, i.e. the state q_0.

- From $q_1 = [\![L/a]\!]$, on b we go to $[\![L/ab]\!]$, which is same as $[\![L/a]\!]$, i.e. the state q_1.

The initial state is $[\![L/\epsilon]\!]$ which is the state q_0, and the final states are those states $[\![L/x]\!]$ that have ϵ in them, which is the set $\{q_1\}$.

We hence have a DFA for L, and in fact this is the minimal automaton accepting L.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 10: Proving non-regularity using Myhill-Nerode Thm and Pumping Lemma

18 February 2010

In this lecture, we will see how to prove that a language is **not** regular.

We will see two methods for showing that a language is not regular. The Myhill-Nerode theorem and the "pumping lemma" show that certain key "seed" languages are not regular. From these seed languages, we can show that many similar languages are also not regular, using closure properties.

1 Proving non-regularity via the Myhill-Nerode Theorem

Recall that the Myhill-Nerode theorem says, among other things, that if L is a language that has an infinite number of suffix languages, then L is not regular. Intuitively, we need a state of a DFA for every suffix language of L, and hence need an infinite number of states that no DFA can accommodate.

This gives a way of proving a language is not regular. Let $L \subseteq \Sigma^*$. Then L is not regular if there are an infinite number of suffix languages, say $\{\llbracket L/x_1 \rrbracket, \llbracket L/x_2 \rrbracket, \ldots\}$ that are all distinct from each other (i.e. $\llbracket L/x_i \rrbracket \neq \llbracket L/x_j \rrbracket$, for any $i \neq j$).

In other words, in order to prove L is not regular, we need to exhibit an infinite set of strings $S = \{x_1, x_2, \ldots\}$ such that for each $x, y \in S$, if $x \neq y$, then $\llbracket L/x \rrbracket \neq \llbracket L/y \rrbracket$. But when is $\llbracket L/x \rrbracket \neq \llbracket L/y \rrbracket$ hold? Clearly, this holds if and only if there exists a $z \in \Sigma^*$ that is in one suffix language but not in the other, i.e. $\exists z \in \Sigma^*$ such that $z \in \llbracket L/x \rrbracket$ *and* $z \notin \llbracket L/y \rrbracket$, or $z \notin \llbracket L/x \rrbracket$ *and* $z \in \llbracket L/y \rrbracket$. Restating this, we must show that $\exists z \in \Sigma^*$ such that ($xz \in L$ *and* $yz \notin L$), or ($xz \notin L$ *and* $yz \in L$).

This leads us to the following definition of when two strings x and y are *distiguishable* with respect to L: they are distinguishable if we can find a z such that xz and yz have a different membership status in L.

Definition 1.1 Two strings $x, y \in \Sigma^*$ are ***distinguishable*** with respect to $L \subseteq \Sigma^*$, if there exists a word $z \in \Sigma^*$, such that *precisely one* of the strings xw and yw is in L (and the other is not).

Hence, continuing our above discussion, in order to prove L is non-regular, it is sufficient to show that there is an infinite set $S \subseteq \Sigma^*$ such that for every $x, y \in S$ with $x \neq y$, x and y are distinguishable with respect to L, i.e. $\exists z \in \Sigma^*$ such that ($xz \in L$ *and* $yz \notin L$), or ($xz \notin L$ *and* $yz \in L$).

We hence have

Theorem 1.2 *Let $L \subseteq \Sigma^*$. If there is an infinite set $S \subseteq \Sigma^*$ such that for every $x, y \in S$ with $x \neq y$, x and y are distinguishable with respect to L, then L is not a regular language.*

The above theorem gives us the first technique for proving non-regularity, where to establish that a language is not regular, we exhibit an infinite set S and show that elements of S are pairwise distinguishable.

1.1 Examples of Proving Non-regularity using MNT

1.1.1 Example

Lemma 1.3 *The language*

$$L = \left\{ \mathtt{1}^k y \;\middle|\; y \in \{\mathtt{0}, \mathtt{1}\}^*, \textit{ and } y \textit{ contains at most } k \textit{ ones} \right\}$$

is not regular.

Proof: Let $S = \{\mathtt{1}^i \mid i \in \mathbb{N}\}$. Clearly S is infinite. Let $x = \mathtt{1}^i$ and $y = \mathtt{1}^j$ be two different elements in S. Then $i \neq j$. Assume, without loss of generality, that $j > i$. Now choose $z = \mathtt{01}^j$. Then $xz = \mathtt{1}^i\mathtt{01}^j \notin L$ but $y\mathtt{01}^j = \mathtt{1}^j\mathtt{01}^j \in L$. Hence x and y are distinguishable with respect to L, for any $x, y \in S$ with $x \neq y$. We conclude, by Theorem 1.2, that L is not regular. ■

1.1.2 Example: ww is not regular

Claim 1.4 *For $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, the language $L = \left\{ ww \;\middle|\; w \in \Sigma^* \right\}$ is not regular.*

Proof: Let $S = \{\mathtt{0}^i \mid i \in \mathbb{N}\}$. Of course, S is infinite. Let $x, y \in S$ with $x \neq y$. Let $x = \mathtt{0}^i$ and $y = \mathtt{0}^j$, with $i \neq j$. Now choose $z = 10^i1$. Then

$$xz = \mathtt{0}^i \underbrace{\mathtt{10}^i\mathtt{1}}_{z} \in L \quad \text{but} \quad yz = \mathtt{0}^j \underbrace{\mathtt{10}^i\mathtt{1}}_{z} \notin L$$

Which means that x and y are distinguishable with respect to L. Hence we have that L is not regular (by Theorem 1.2). ■

2 The Pumping Lemma

2.1 Proof by repetition of states

We next prove a language non-regular by a slightly different argument.

Claim. *The language $L = \left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\}$ is not regular.*

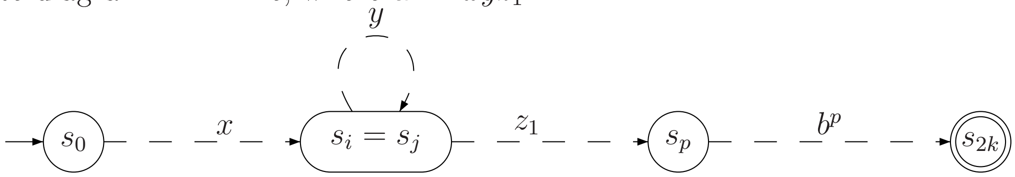
Proof: Suppose that L were regular. Then L is accepted by some DFA

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

Suppose that M has p states.

Consider the string $\mathtt{a}^p\mathtt{b}^p$. It is accepted using a sequence of states $s_0 s_1 \ldots s_{2p}$. Right after we read the last $\mathtt{a}$, the machine is in state s_p.

In the sub-sequence $s_0 s_1 \ldots s_p$, there are $p+1$ states. Since L has only p distinct states, this means that two states in the sequence are the same (by the pigeonhole principle). Let us call the pair of repeated states q_i and q_j, $i < j$. This means that the path through M's state diagram looks like, where $\mathtt{a}^p = xyz_1$.



But this DFA will accept all strings of the form $xy^j z_1 \mathtt{b}^p$, for $j \geq 0$. Indeed, for $j = 0$, this is just the string $xz_1\mathtt{b}^p$, which this DFA accepts, but it is not in the language, since it has less $\mathtt{a}$s than $\mathtt{b}$s. That is, if $|y| = m$, the DFA accepts all strings of the form $\mathtt{a}^{p-m+jm}\mathtt{b}^m$, for any $j \geq 0$. For any value of j other than 1, such strings are not in L.

So our DFA M accepts some strings that are not in L. This is a contradiction, because L was supposed to accept L. Therefore, we must have been wrong in our assumption that L was regular. ■

2.2 The pumping lemma

The pumping lemma generalizes the above argument into a standard template, which we can prove once and then quickly apply to many languages.

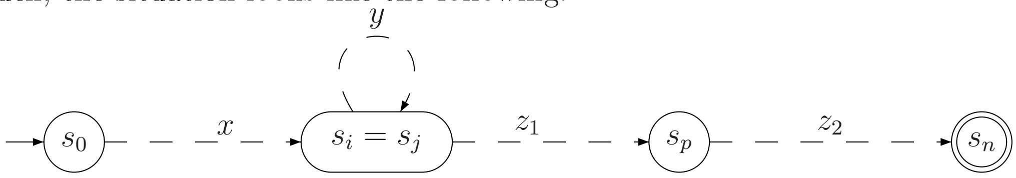
Theorem 2.1 (Pumping Lemma.) *Let L be a regular language. Then there exists an integer p (the "pumping length") such that for any string $w \in L$ with $|w| \geq p$, w can be written as xyz with the following properties:*

- $|xy| \leq p$.
- $|y| \geq 1$ *(i.e. y is not the empty string).*
- $xy^k z \in L$ *for every $k \geq 0$.*

Proof: The proof is written out in full detail in Sipser, here we just outline it.

Let M be a DFA accepting L, and let p be the number of states of M. Let $w = c_1 c_2 \ldots c_n$ be a string of length $n \geq p$, and let the accepting state sequence (i.e., trace) for w be $s_0 s_1 \ldots s_n$.

There must be a repeat within the sequence from s_0 to s_p, since M has only p states, and as such, the situation looks like the following.



So if we set $z = z_1 z_2$, we now have x, y, and z satisfying the conditions of the lemma.

- $|xy| \leq p$ because repeat is within first $p+1$ states

- $|y| \geq 1$ because i and j are distinct
- $xy^k z \in L$ for every $k \geq 0$ because a loop in the state diagram can be repeated as many or as few times as you want.

  Formally, for any k, the word $xy^i z$ goes through the following sequence of states:

$$s_0 \xrightarrow{\texttt{x}} \overbrace{s_i \xrightarrow{\texttt{y}} s_i \xrightarrow{\texttt{y}} \cdots \xrightarrow{\texttt{y}}}^{k \text{ times}} s_i = s_j \xrightarrow{\texttt{z}} s_n,$$

  and s_n is an accepting state. Namely, M accepts $xy^k z$, and as such $xy^k z \in L$.

This completes the proof of the theorem. ■

Notice that we do not know exactly where the repeat occurs, so we have very little control over the length of x and z_1.

2.3 Using the PL to show non-regularity

If L is regular, then it satisfies the pumping lemma (PL). Therefore, intuitively, if L does not satisfy the pumping lemma, L cannot be regular.

2.3.1 Restating the Pumping Lemma via the contrapositive

We want to restate the pumping lemma in the contrapositive. Now, it is **not** true that if L satisfies the conditions of the PM, then L must be regular. Reminder from CS 173: contrapositive of if-then statement is equivalent, converse is not.

What does it mean to **not** satisfy the Pumping Lemma? Write out PL compactly:

$$\begin{array}{c}\text{L is}\\ \text{regular.}\end{array} \implies \left(\exists p \, \forall w \in L \quad |w| \geq p \Rightarrow \left(\exists x, y, z \text{ s.t. } \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \text{ and } \quad \forall i \quad xy^i z \in L. \right) \right).$$

Now, we know that if A implies B, then $\overline{B}$ implies $\overline{A}$ (contraposition), as such the Pumping Lemma, can be restated as

$$\overline{\left(\exists p \, \forall w \in L \quad |w| \geq p \Rightarrow \left(\exists x, y, z \; \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \text{ and } \quad \forall i \quad xy^i z \in L. \right) \right)} \implies \overline{L \text{ is regular}}.$$

Now, the logical statement $A \Rightarrow B$ is equivalent to $\overline{A} \vee B = \overline{A \wedge \overline{B}}$. As such $\overline{A \Rightarrow B} = A \wedge \overline{B}$. In addition, negation flips quantifies, as such, the above is equivalent to

$$\left(\forall p \, \exists w \in L \quad |w| \geq p \text{ and } \overline{\left(\exists x, y, z \; \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \text{ and } \quad \forall i \quad xy^i z \in L. \right)} \right) \implies \begin{array}{c}\text{L is}\\ \text{not regular.}\end{array}$$

Since, $\overline{A \wedge \overline{B}} = A \Rightarrow B$ we have that $\overline{A \wedge B} = \bigl(A \Rightarrow \overline{B}\bigr)$. Thus, we have

$$\left(\forall p \ \exists w \in L \quad |w| \geq p \text{ and } \left(\forall x, y, z \ \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \implies \overline{\forall i \quad xy^i z \in L.} \right) \right) \implies \begin{array}{c} \text{L is} \\ \text{not regular.} \end{array}$$

Which is equivalent to

$$\left(\forall p \ \exists w \in L \quad |w| \geq p \text{ and } \left(\forall x, y, z \ \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \implies \exists i \quad xy^i z \notin L. \right) \right) \implies \begin{array}{c} \text{L is} \\ \text{not regular.} \end{array}$$

The translation into words is the contrapositive of the Pumping Lemma (stated in Theorem 2.2 below).

2.3.2 The contrapositive of the Pumping Lemma

Theorem 2.2 (Pumping Lemma restated.) *Consider a language L. If for any integer $p \geq 0$ there exists a word $w \in L$, such that $|w| \geq p$, and for any breakup of w into three strings x, y, z, such that:*

- *$w = xyz$,*
- *$|xy| \leq p$,*
- *$|y| \geq 1$,*

implies that there exists an i such that $xy^i z \notin L$, then the language L is not regular.

2.3.3 Proving that a language is not regular

Let us assume that we want to show that a language L is *not* regular.

Such a proof is done by contradiction. To prove L is not regular, we assume it is regular. This gives us a specific (but unknown) pumping length p. We then show that L satisfies the rest of the contrapositive version of the pumping lemma, so it can not be regular.

So the proof outline looks like:

- Suppose L is regular. Let p be its pumping length.
- Consider $w =$ [formula for a specific class of strings]
- By the Pumping Lemma, we know there exist x, y, z such that $w = xyz$, $|xy| \leq p$, and $|y| \geq 1$.
- Consider $i =$ [some specific value, almost always 0 or 2]
- $xy^i z$ is not in L. [explain why it can't be]

Notice that our adversary picks p. We get to pick w whose length depends on p. But then our adversary gets to pick the specific division of w into x, y, and z.

2.4 Examples

2.4.1 The language $L = \mathtt{a}^n\mathtt{b}^n$ is not regular

Claim 2.3 *The language $L = \mathtt{a}^n\mathtt{b}^n$ is not regular.*

Proof: For any $p \geq 0$, consider the word $w = \mathtt{a}^p\mathtt{b}^p$, and consider any breakup of w into three parts, such that $w = xyz$ $|y| \geq 1$, and $|xy| \leq p$. Clearly, xy is a prefix of w made out of only $\mathtt{a}$s. As such, the word $xyyz$ has more $\mathtt{a}$s in it than $\mathtt{b}$s, and as such, it is not in L.

But then, by the Pumping Lemma (Theorem 2.2), L is not regular. ■

2.4.2 The language $\{ww\}$ is not regular

Claim 2.4 *The language $L = \left\{ ww \;\middle|\; w \in \Sigma^* \right\}$ is not regular.*

Proof: For any $p \geq 0$, consider the word $w = \mathtt{0}^p\mathtt{1}\mathtt{0}^p\mathtt{1}$, and consider any breakup of w into three parts, such that $w = xyz$ $|y| \geq 1$, and $|xy| \leq p$. Clearly, xy is a prefix of w made out of only $\mathtt{0}$s. As such, the word $xyyz$ has more $\mathtt{0}$s in its first part than the second part. As such, $xyyz$ is not in L.

But then, by the Pumping Lemma (Theorem 2.2), L is not regular. ■

Consider the word w used in the above claim:

- It is concrete, made of specific characters, no variables left in it.
- These strings are a subset of L, chosen to exemplify what is not regular about L.
- Its length depends on p.
- The $\mathtt{1}$ in the middle serves as a barrier to separate the two groups of $\mathtt{0}$'s. (Think about why the proof would fail if it was not there.)
- The $\mathtt{1}$ at the end of w does not matter to the proof, but we nee it so that $w \in L$.

2.5 A note on finite languages

A language L is ***finite*** if has a bounded number of words in it. Clearly, a finite language is regular (since you can always write a finite regular expression that matches all the words in the language).

It is natural to ask why we can not apply the pumping lemma Theorem 2.1 to L? The reason is because we can always choose the threshold p to be larger than the length of the longest word in L. Now, there is no word in L with length larger than p in L. As such, the claim of the Pumping Lemma holds trivially for a finite language, but no word can be pumped - and as such L stays finite. So the pumping lemma makes sense even for finite languages!

3 Non-regularity via closure properties

If we know certain seed languages are not regular, then we can use closure properties to show other languages are not regular.

We remind the reader that *homomorphism* is a mapping $h : \Sigma_1 \rightarrow \Sigma_2^*$ (namely, every letter of Σ_1 is mapped to a string over Σ_2). We showed that if a language L over Σ_1 is regular, then the language $h(L)$ is regular. We referred to this property as *closure of regular languages under homomorphism.*

We know that the language $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ is not regular (by MNT or pumping lemma arguments). Now let us show this:

Claim 3.1 *The language $L' = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ is not regular.*

Proof: Assume for the sake of contradiction that L' is regular. Let h be the homomorphism that maps `0` to `a` and `1` to `b`. Then $h(L')$ must be regular (closure under homomorphism). But $h(L')$ is the language

$$L = \left\{\mathtt{a}^n\mathtt{b}^n \,\middle|\, n \geq 0\right\}, \tag{1}$$

which is not regular. A contradiction. As such, L' is not regular. ■

We remind the reader that regular languages are also closed under intersection.

Claim 3.2 *The language $L_2 = \left\{w \in \{\mathtt{a},\mathtt{b}\}^* \,\middle|\, w \text{ has an equal \# of } \mathtt{a}\text{'s and } \mathtt{b}\text{'s}\right\}$ is not regular.*

Proof: Suppose L_2 were regular. Consider $L_2 \cap \mathtt{a}^*\mathtt{b}^*$. This must be regular because L_2 and $\mathtt{a}^*\mathtt{b}^*$ are both regular and regular languages are closed under intersection. But $L_2 \cap \mathtt{a}^*\mathtt{b}^*$ is just the language $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$, which we know is not regular. The contradiction proves that L_2 is not regular. ■

Claim 3.3 *The language $L_3 = \left\{\mathtt{a}^n\mathtt{b}^n \,\middle|\, n \geq 1\right\}$ is not regular.*

Proof: Assume for the sake of contradiction that L_3 is regular. Consider $L_3 \cup \{\epsilon\}$. This must be regular because L_3 and $\{\epsilon\}$ are both regular and regular languages are closed under union. But $L_3 \cup \{\epsilon\}$ is just $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$, which is not regular. This contradiction shows that L_3 is not regular. ■

3.1 Being careful in using closure arguments

Most closure properties must be applied in the correct direction: We show (or assume) that all inputs to the operation are regular, therefore the output of the operation must be regular.

For example, consider (again) the language $L_B = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$, which is not regular.

Since L_B is not regular, $\overline{L_B}$ is also not regular. If $\overline{L_B}$ were regular, then L_B would also have to be regular because regular languages are closed under set complement. However, many similar lines of reasoning do not work for other closure properties.

For example, L_B and $\overline{L_B}$ are both non-regular, but their union is regular. Similarly, suppose that L_k is the set of all strings of length $\leq k$. Then $L_B \cap L_k$ is regular, even though L_B is not regular.

If you are not absolutely sure of what you are doing, always use closure properties in the forward direction. That is, establish that L and L' are regular, then conclude that L OP L' must be regular.

Also, be sure to apply only closure properties that we know to be true. In particular, regular languages are **not** closed under the subset and superset relations. Indeed, consider $L_1 = \{001, 00\}$, which is regular. But L_1 is a subset of L_B, which is not regular. Similarly, $L_2 = (0 + 1)^*$ is regular. And it is a superset of L (from Eq. (1) in the proof of Claim 3.1)). But you can not deduce that L is therefore regular. We know it is not.

So regular languages can be subsets of non-regular ones and vice versa.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 11: Algorithms on DFAs

23 February 2010

In this lecture, we will discuss several algorithms for regular languages presented as DFAs.

1 Emptiness

The emptiness problem for DFAs is decidable; i.e. we can write an algorithm to check whether the language of an input DFA A is non-empty. A DFA can be encoded as a string, say by encoding the tuple $(Q, \Sigma, \delta, q_0, F)$. The tuple can be appropriately encoded as it is finite (Q and Σ are finite, and hence $\delta : Q \times \Sigma \rightarrow Q$ is representable as a finite table, etc.

Now, consider the DFA as a graph with nodes as states, and (labeled) edges as transitions of the DFA. Then it is easy to see that $L(A)$ is non-empty iff there is a path from the initial state to some final state, in this graph. This problem is simply a reachability problem ($s - t$ reachability) and can be solved in linear time (in the number of nodes and edges), say using depth-first search. Hence the emptiness problem is decidable for DFAs.

A similar algorithm works for NFAs as well.

2 Inclusion, Equivalence, etc.

It now follows that inclusion and equivalence of regular languages, represented as DFAs, is decidable, using closure properties.

Given DFAs A_1 and A_2 over the same alphabet, we can decide if $L(A_1) \subseteq L(A_2)$ by noticing that this is true iff $L(A_1) \cap L(\bar{A}_2) = \emptyset$. Since DFAs are constructively closed under negation and intersection, we can complement A_2, and intersect the resulting automaton with A_1, and check the resulting automaton for emptiness, which is decidable.

Given DFAs A_1 and A_2 over the same alphabet, we can decide if $L(A_1) = L(A_2)$ by checking if $L(A_1) \subseteq L(A_2)$ and $L(A_2) \subseteq L(A_1)$. Hence equivalence is also decidable.

3 Infiniteness

Given a DFA A, can we decide if $L(A)$ is infinite? It turns out that we can. Let us first prove the following.

Theorem 3.1 *Let A be a DFA with n states. Then $L(A)$ is infinite iff A accepts some word of length l, where $n \leq l < 2n$.*

The proof of the above is simple. First, if $L(A)$ is infinite, then for any i, there is some word w in $L(A)$ whose length is at least i. Take a word w in $L(A)$ such that $|w| \geq n$, and let w be a shortest length word of this kind. Then we claim that $|w| < 2n$. Assume not. Then, by an argument similar to the pumping lemma, $w = uvx$ where $|uv| \leq n$ and $|v| > 0$, and we can "pump down" to get a word $w = ux$ in L. But since $|ux| > n$ and $|uv| < |w|$, this contradicts the fact that w was the shortest word longer than n in L. This proves that $w \in L$ and $n \leq |w| < 2n$. Now, turning to the other direction, assume that there is a word $w \in L$ with $n \leq |w| < 2n$. Then, by the pumping lemma, $w = uvx$, with $|v| > 0$ such that for every $i \in \mathbb{N}$, $uv^iw \in L$. Hence L is infinite.

We can now decide whether $L(A)$ is infinite simply by enumerating all words w of length l, where $n \leq l < 2n$, and checking whether A accepts any such w. If it does, then $L(A)$ is infinite, and otherwise, it is not.

4 Minimization algorithm

The Myhill-Nerode theorem gives us a way to minimize DFAs. Given a DFA A, we can simply compute the language L_q accepted by every state q (i.e. the language accepted when q is made the initial state), and we can check which of these languages are equal (using the language equivalence decidability result above), and merge all pairs of states q and q' that have the same language (i.e. if $L_q = L_{q'}$).

However, we now show a more efficient partition refinement algorithm that works in $O(nlogn)$ time.

A *partition* P of states is a set of subsets of Q, $P = \{S_1, \ldots S_k\}$ where each $S_i \subseteq Q$, $\bigcup_{i=1}^{k} S_i = Q$, and for any $i \neq j$, $S_i \cap S_j = \emptyset$. In other words, the sets in P divide Q into disjoint subsets.

A partition $R = \{T_1, \ldots T_m$ is said to refine $P = \{S_1, \ldots S_k\}$ if it is true that for every $q, q' \in Q$, if q and q' are in the same set in the partition R, then they are also in the same set in partition P. In other words, R is obtained by refining the partition P by dividing the sets in P into smaller sets.

We now describe an algorithm where we start with the initial partition $P_0 = \{F, Q \setminus F\}$ and keep refining it till we reach a partition P that can no longer be refined. This final partition P will then tell us which states can be merged to form a minimal DFA. Intuitively, all states that belong to the same set in P will be merged to a single state, and hence the minimal DFA will have only as many states as the number of sets in P.

Assume that we have a partition P. We define the *next* partition P' the unique partition that satisfies the following properties: Let $q, q' \in Q$.

- q and q' are in the same partition in P' iff they are in the same partition in P and for every $a \in \Sigma$, $\delta(q, a)$ and $\delta(q', a)$ are also in the same partition in P.

We keep refining the initial partition P using the above algorithm, till the refinement stabilizes, and does not give a new partition. Using this stabilized partition, we will build our minimal DFA.

Let us describe this on an example. Consider the DFA of Figure 2. It is more complex than it needs to be.

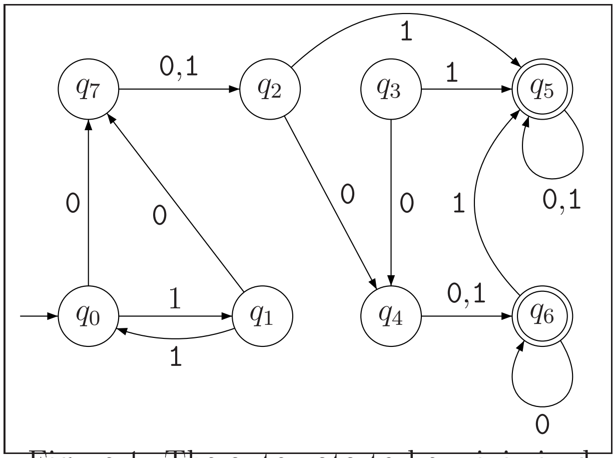


Figure 1: The automata to be minimized.

We start with the partition that groups together the non-final states together and the final states together. Intuitively, we know that a final state and a non-final state definitely have different suffix languages, and hence cannot be merged.

So $P_0 = \{\{q_0, q_1, q_2, q_3, q_4, q_7\}, \{q_5, q_6\}\}$.

Let us name the sets in P_0 as say $A = \{q_0, q_1, q_2, q_3, q_4, q_7\}$ and $B = \{q_5, q_6\}$.

Now, consider the state q_0. On reading 0 it goes to q_7 (which is in A) and on reading 1 it goes to q_1 (which is also in A). However, q_3, when reading 0 goes to q_4 (which is in A) and on reading 1 goes to q_5 (which is in B). Hence we must separate q_0 and q_3 in the next refinement. Also, states q_0 and q_5, since they are already separated in P_0, will continue to be separated in the next refinement.

Let us compute the sets where the states in A go to. q_0 goes to (A, A), q_1 goes to (A, A), q_2 goes to (A, B), q_3 goes to (A, B), q_4 goes to (B, B), and q_7 goes to (A, A). Hence we refine the set A into three sets: $\{q_0, q_1, q_7\}$, $\{q_2, q_3\}$ and $\{q_4\}$. Let us compute the sets where the states in B go to. q_5 goes to (B, B) and q_6 goes to (B, B). Hence we keep q_5 and q_6 in the same partition.

We hence get a new partition of states $P_1 = \{\{q_0, q_1, q_7\}, \{q_2, q_3\}, \{q_4\}, \{q_5, q_6\}\}$ which is a refinement of P_0.

Now we compute the next iteration of refinement. Let us name the sets in P_1. Let $C = \{q_0, q_1, q_7\}$, $D = \{q_2, q_3\}$, $E = \{q_4\}$, and $B = \{q_5, q_6\}$.

First let us examine the states in C. q_0 goes to (C, C), q_1 goes to (C, C), and q_7 goes to (D, D). Hence we refine C into two sets: $\{q_0, q_1\}$ and $\{q_7\}$.

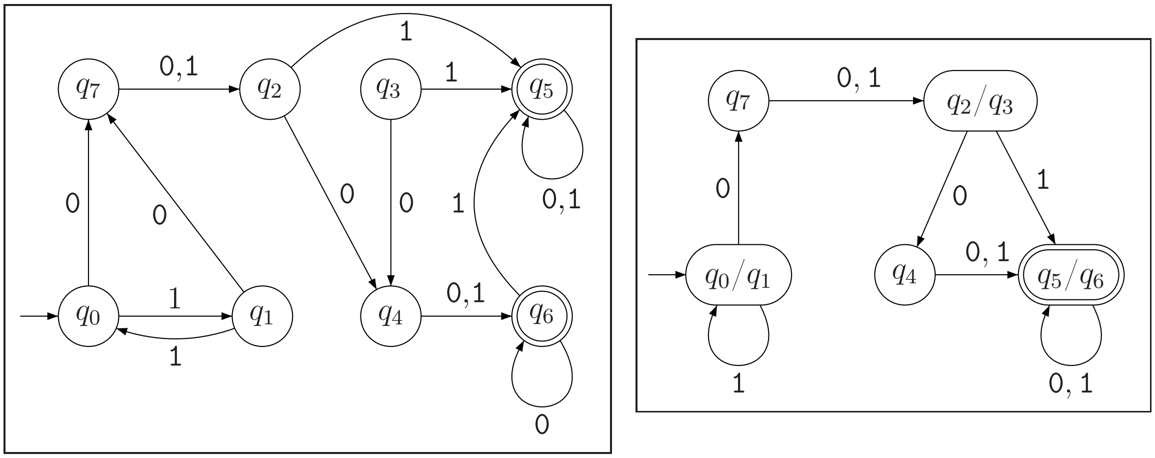
Now let us examine the states in D. q_2 goes to (E, B) and q_3 goes to (E, B). So q_2 and q_3 remain in the same partition.

Examining the states in B, also gives that q_5 and q_6 remain in the same partition.

So we can now form the new partition $P_2 = \{\{q_0, q_1\}, \{q_7\}, \{q_2, q_3\}, \{q_4\}, \{q_5, q_6\}\}$ which is a refinement of P_1.

We can now continue to refine the partition P_2. However, doing so gives the same partition as no other splitting of states happen. We are hence done and we can build the minimal DFA. The idea is to have a single state for each set in the partition of P_2. And have a transition on a letter d from one set S to another set S' provided there is some state $q \in S$ such that $\delta(q, d) \in S'$. Note that if there is some state $q \in S$ such that $\delta(q, d) \in S'$, then for *every* state $q' \in S$, $\delta(q', d) \in S'$, as we know otherwise that the partition would have been refined further. We also mark S to be a final state if some state in S (equivalently, all states in S) are final.

Doing so gives the minimal automaton depicted below.



(a) (b)

Figure 2: (a) Original automata, (b) minimized autoamta.

.

We omit the correctness argument for minimization. Intuitively, the algorithm ensures that states in different partitions at any time have distinct suffix languages (this is true in the beginning, and is maintained across each refinement). Hence the number of states in the final DFA certainly has no more states than a minimal DFA would have. Furthermore, we can prove that in the final partition, states in the same set have the same suffix languages, and hence indeed can be merged, giving a DFA that accepts the same language as the original DFA. Hence the constructed DFA must be a minimal DFA.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 12: Computability and Turing Machines

9 March 2010

> The Electric Monk was a labor-saving device, like a dishwasher or a video recorder. Dishwashers washed tedious dishes for you, thus saving you the bother of washing them yourself, video recorders watched tedious television for you, thus saving you the bother of looking at it yourself; Electric Monks believed things for you, thus saving you what was becoming an increasingly onerous task, that of believing all the things the world expected you to believe.
>
> – Dirk Gently's Holistic Detective Agency, Douglas Adams.

This lecture covers the beginning of section 3.1 from Sipser.

1 Computability

For the alphabet

$$\Sigma = \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}, +, -\},$$

consider the language

$$L = \left\{ a_n a_{n-1} \ldots a_0 + b_m b_{m-1} \ldots b_0 = c_r c_{r-1} \ldots c_0 \;\middle|\; \begin{array}{l} a_i, b_j, c_k \in [0,9] \text{ and} \\ \quad \langle a_n a_{n-1} \ldots a_0 \rangle \\ \qquad + \quad \langle b_m b_{m-1} \ldots b_0 \rangle \\ \qquad\qquad = \langle c_r c_{r-1} \ldots c_0 \rangle \end{array} \right\},$$

where $\langle a_n a_{n-1} \ldots a_0 \rangle = \sum_{i=0}^{n} a_i \cdot 10^i$ is the number represented in base ten by the string $a_n a_{n-1} \ldots a_0$. We are interested in the question of whether or not a given string belongs to this language. This is an example of a decision problem (where the output is either `yes` or `no`), which is easy in this specific case, but clearly too hard for a PDA to solve it[1].

Usually, we are interested in algorithms that compute something for their input and output the results. For example, given the strings $a_n a_{n-1} \ldots a_0$ and $b_m b_{m-1} \ldots b_0$ (i.e., two numbers) we want to compute the string representing their sum.

Here is another example for such a decision algorithm: Given a quadratic equation $ax^2 + bx + c = 0$, we would like to find the ***roots*** of this equation. Namely, two numbers r_1, r_2 such that $ax^2 + bx + c = a(x - r_1)(x - r_2) = 0$. Thus, given numbers a, b and c, the algorithm should output the numbers r_1 and r_2.

To see how subtle this innocent question can be, consider the question of computing the roots of a polynomial of degree 5. That is, given an equation

$$ax^5 + bx^4 + cx^3 + dx^2 + ex + f = 0,$$

[1] We use the world clearly here to indicate that the fact that this language is not context-free can be formally proven, but it is tedious and not the point of the discussion. The interested reader can try and prove this using the pumping lemma for CFGs.

can we compute the values of x for which is equation holds? Interestingly, if we limit our algorithm to use only the standard operators on numbers $+, -, *, /, \sqrt{}, \sqrt[k]{}$ then no such algorithm exists.[2]

In the final part of this course, we will look at the question of what (formally) is a computation? Or, in other words, what is (what we consider to be) a computer or an algorithm? A precise model for computation will allow us to prove that computers can solve certain problems but not others.

1.1 History

Early in this century, mathematicians (e.g. David Hilbert) thought that it might be possible to build formal algorithms that could decide whether any mathematical statement was true or false. For obvious reasons, there was great interest in whether this could really be done. In particular, he took upon himself the project of trying to formalize the known mathematics at the time. Gödel showed in 1929 that the project (of explicitly describing all of mathematics) is hopeless and there is no finite description of mathematical models.

In 1936, Alonzo Church and Alan Turing independently showed that this goal was impossible. In his paper, Alan Turing introduced the Turing machine (described below). Alonzo Church introduced the ***λ-calculus***, which formed the starting point for the development of a number of functional programming languages and also formal models of meaning in natural languages. Since then, these two models and some others (e.g. ***recursion*** theory) have been shown to be equivalent.

This has led to the Church-Turing Hypothesis.

> ***Church-Turing Hypothesis***: All reasonable models of (general-purpose) computers are equivalent. In particular, they are equivalent to a Turing machine.

This is not something you could actually prove is true (what is reasonable in the above statement, for example?). It could be proved false if someone found another model of computation that could solve more problems than a Turing machine, but no one has done this yet. Notice that we are ignoring how fast the computation can be done: it is certainly possible to improve on the speed of a Turing machine (in fact, every Turing machine can be speeded up by making it more complicated). We are only interested in what problems the machines can or can not solve.

2 Turing machines

2.1 Turing machines at a high level

So far, we have seen two simple models of computation:

- DFA/NFA: finite control, no extra memory, and

[2] This is the main result of Evariste Galois that died at the age of 20(!) in a duel. Niels Henrik Abel (which also died relatively young) proved this slightly before Galois, but Galois work lead to a more general theory.

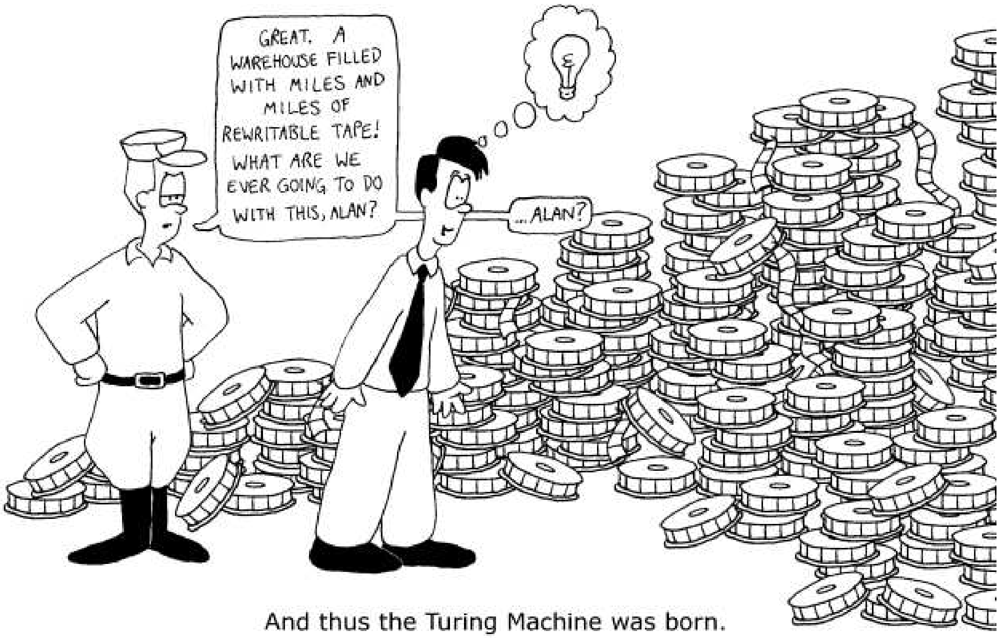

GREAT. A WAREHOUSE FILLED WITH MILES AND MILES OF REWRITABLE TAPE! WHAT ARE WE EVER GOING TO DO WITH THIS, ALAN?
...ALAN?
And thus the Turing Machine was born.

Figure 1: Comic by Geoff Draper.

- Recursive automatas/PDA: finite control, unbounded stack.

Both types of machines read their input left-to-right. They halt exactly when the input is exhausted. Turing machines are like a RA/PDA, in that they have a finite control and an unbounded one dimensional memory tape (i.e., stack). However, a Turing machine is different in the following ways.

(A) The input is delivered on the memory tape (not in a separate stream).

(B) The machine head can move freely back and forth, reading and writing on the tape in any pattern.

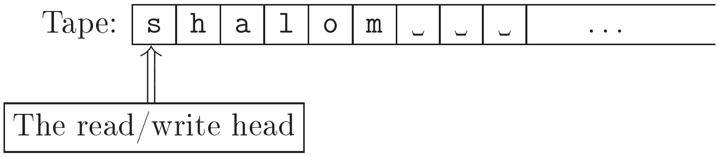
(C) The machine halts immediately when it enters an ***accept*** or ***reject*** state.

Notice condition (C) in particular. A Turing machine can read through its input several times, or it might halt without reading the whole input (e.g. the language of all strings that start with ab can be recognized by just reading two letters).

Moving back and forth along the tape allows a Turing machine to (somewhat slowly) simulate random access to memory. Surprisingly, this very simple machine can simulate all the features of "regular" computers. Here equivalent is meant only in the sense that whatever a regular computer can compute, so can a Turing machine compute. Of course, Turing machines do not have graphics/sound cards, internet connection and they are generally considered to be an inferior platform for computer games. Nevertheless, computationally, TMs can compute whatever a "regular" computer can compute.

2.2 Turing Machine in detail

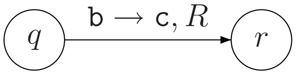
Specifically, a ***Turing machine*** (TM) has a finite control and an infinite tape. In this class, our basic model will have a tape that is infinite only in one direction. A Turing machine starts up with the input string written at the start of the tape. The rest of the tape is filled with a special blank character (i.e., '␣'). Initially, the head is located at the first tape position. Thus, the initial configuration of a Turing machine for the input `shalom` is as follows.



Each step of the Turing machine first reads the symbol on the cell of the tape under the head. Depending on the symbol and the current state of the controller, it then

- (optionally) writes a new symbol at the current tape position,
- moves either left or right, and
- (optionally) changes to a new state.

For example, the following transition is taken if the controller is in state q and the symbol under the read head is `b`. It replaces the `b` with the character `c` and then moves right, switching the controller to the state r.



Note, that Turing machines are deterministic. That is, once you know the state of the controller and which symbol is under the read/write head, there is exactly one choice for what the machine can (and must) do.

The controller has two special states q_{acc} and q_{rej}. When the machine enters one of these states, it halts. It either ***accepts*** or ***rejects***, depending on which of the two it entered.

Note 2.1 If the Turing machine is at the start of the tape and tries to move left, it simply stays put on the start position. This is not the only reasonable way to handle this case.

Note 2.2 Nothing guarantees that a Turing machine will eventually halt (i.e., stop). Like your favorite Java program, it can get stuck in an infinite loop[3]. This will have important consequences later, when we show that deciding if a program halts or not is in fact a task that computers can not solve.

Remark 2.3 Some authors define Turing machines to have a doubly-infinite tape. This does not change what the Turing machine can compute. There are many small variations on Turing machines which do not change the power of the machine. Later, we will see a few sample variations and how to prove they are equivalent to our basic model. The robustness of this model to minor changes in features is yet another reason computer scientists believe the Church-Turing hypothesis.

[3] Or just get stuck inside of Mobile with the Memphis blues again...

2.3 Turing machine examples

2.3.1 The language $w\$w$

For $\Sigma = \{\mathtt{a}, \mathtt{b}, \$\}$, consider the language

$$L = \left\{ w\$w \;\middle|\; w \in \Sigma^* \right\},$$

which is not context-free. So, let describe a TM that accepts this language.

One algorithm for recognizing L works as follows. It first

1. Cross off the first character `a` or `b` in the input (i.e. replace it with `x`, where `x` is some special character)) and remember what it was (by encoding the character in the current state). Let u denote this character.

2. Move right until we see a \$.

3. Read across any `x`'s.

4. Read the character (not `x`) on the tape. If this character is different from u, then it immediately rejects.

5. Cross off this character, and replace it by `x`.

6. Move left past the \$ and then keep going until we see an `x` on the tape.

7. Move one position right and go back to the first step.

We repeat this until the first step can not find any more `a`'s and `b`'s to cross off.

Figure 2 depicts the resulting TM. Observe, that for the sake of simplicity of exposition, we did not include the state q_{rej} in the diagram. In particular, all missing transitions in the diagram are transitions that go into the reject state.

Notice that we did not include the reject state in the diagram, because it is already too messy. If there is no transition shown, we will assume that one goes into the reject state.

Note 2.4 For most algorithms, the Turing machine code is complicated and tedious to write out explicitly. In particular, it is not reasonable to write it out as a state diagram or a transition function. This only works for the relatively simple examples, like the ones shown here. In particular, its important to be able to describe a TM in high level in pseudo-code, but yet be able to translate it into the nitty-gritty details if necessary.

2.3.2 Mark start position by shifting

Let $\Sigma = \{a, b\}$. Write a Turing machine that puts a special character `x` at the start of the tape, shifting the input over one position, then accepting the input.

Accepting or rejecting is not the point of this machine. Rather, marking the start of the input is a useful component for creating more complex algorithms. So you had normally see this machine used as part of a larger machine, and the larger machine would do the accepting or rejecting.

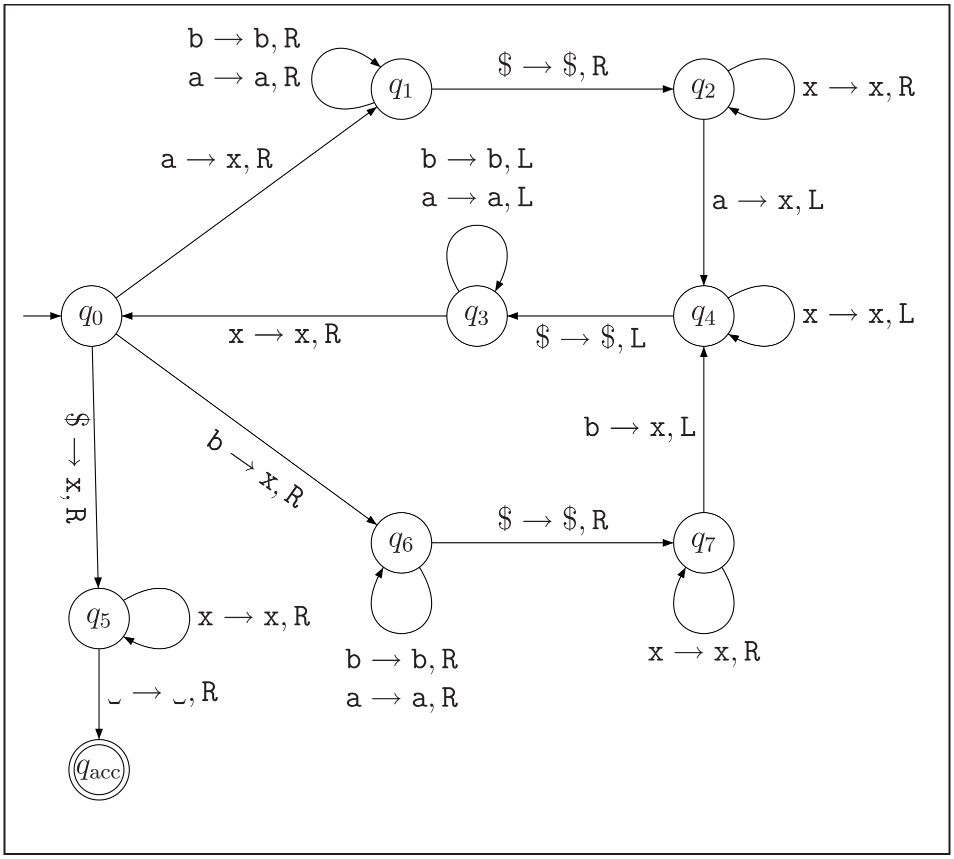


Figure 2: A TM for the language $w\$w$.

2.4 Formal definition of a Turing machine

A ***Turing machine*** is a 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

where

- Q: finite set of states.
- Σ: finite input alphabet.
- Γ: finite tape alphabet.
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}$: Transition function.

  As a concrete example, if $\delta(q, c) = (q', c', \mathtt{L})$ means that, that if the TM is at state q, and the head on the tape reads the character c, then it should move to state q', replace c on the tape by c', and move the head on the tape to the left.
- $q_0 \in Q$ is the initial state.
- $q_{\text{acc}} \in Q$ is the ***accepting/final*** state.

- $q_{\text{rej}} \in Q$ is the ***rejecting*** state.

This definition assumes that we've already defined a special blank character. In Sipser, the blank is written $\sqcup$ or ␣. A popular alternative is B. (If you use any other symbol for blank, you should write a note explaining what it is.)

The special blank character (i.e., ␣) is in the tape alphabet but it is not in the input alphabet.

2.4.1 Example

For the TM of Figure 2, we have the following $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, where

(i) $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_{\text{acc}}, q_{\text{rej}}\}$.

(ii) $\Sigma = \{\mathtt{a}, \mathtt{b}, \$\}$.

(iii) $\Gamma = \{\mathtt{a}, \mathtt{b}, \$, \text{␣}, \mathtt{x}\}$.

(iv) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}$.

	a	b	$	␣	x
q_0	$(q_1, \mathtt{x}, \mathtt{R})$	$(q_6, \mathtt{x}, \mathtt{R})$	$(q_5, \mathtt{x}, \mathtt{R})$	`reject`	`reject`
q_1	$(q_1, \mathtt{a}, \mathtt{R})$	$(q_1, \mathtt{b}, \mathtt{R})$	$(q_2, \$, \mathtt{R})$	`reject`	`reject`
q_2	$(q_4, \mathtt{x}, \mathtt{L})$	`reject`	`reject`	`reject`	$(q_2, \mathtt{x}, \mathtt{R})$
q_3	$(q_3, \mathtt{a}, \mathtt{L})$	$(q_3, \mathtt{b}, \mathtt{L})$	`reject`	`reject`	$(q_0, \mathtt{x}, \mathtt{R})$
q_4	`reject`	`reject`	$(q_3, \$, \mathtt{L})$	`reject`	$(q_4, x, \mathtt{L})$
q_5	`reject`	`reject`	`reject`	$(q_{\text{acc}}, \text{␣}, \mathtt{R})$	$(q_5, \mathtt{x}, \mathtt{R})$
q_6	$(q_6, \mathtt{a}, \mathtt{R})$	$(q_6, \mathtt{b}, \mathtt{R})$	$(q_7, \$, \mathtt{R})$	`reject`	`reject`
q_7	`reject`	$(q_4, \mathtt{x}, \mathtt{L})$	`reject`	`reject`	$(q_7, \mathtt{x}, \mathtt{R})$
q_{acc}	No need to define				
q_{rej}	No need to define				

Here, `reject` stands for $(q_{\text{rej}}, \mathtt{x}, \mathtt{R})$.

(Filling this table was fun, fun, fun!)

˜
˜

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 13: More on Turing Machines

9 March 2010

This lecture covers the formal definition of a Turing machine and related concepts such as configuration and Turing decidable. It surveys a range of variant forms of Turing machines and shows for one of them (multi-tape) why it is equivalent to the basic model.

1 A Turing machine

A ***Turing machine*** is a 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}),$$

where

- Q: finite set of states.
- Σ: finite input alphabet.
- Γ: finite tape alphabet.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}$.
- $q_0 \in Q$ is the initial state.
- $q_{\mathrm{acc}} \in Q$ is the ***accepting***/***final*** state.
- $q_{\mathrm{rej}} \in Q$ is the ***rejecting*** state.

TM has a working space (i.e., tape) and its deterministic. It has a reading/writing head that can travel back and forth along the tape and rewrite the content on the tape. TM halts immediately when it enters the accept state (i.e., q_{acc}) and then it accepts the input, or when the TM enters the reject state (i.e., q_{rej}), and then it rejects the input.

Example 1.1 Here we describe a TM that takes it input on the tape, shifts it to the right by one character, and put a \$ on the leftmost position on the tape.

So, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ (but the machine we describe would work for any alphabet). Let

$$Q = \{q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}\} \cup \left\{ q_c \;\middle|\; c \in \Sigma \right\}.$$

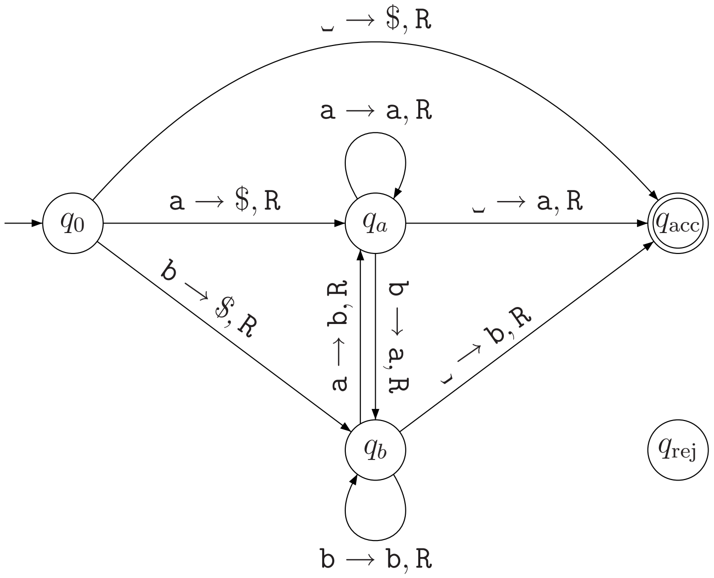


Figure 1: A TM that shifts its input right by one position, and inserts \$ in the beginning of the tape.

Now, the transitions function is

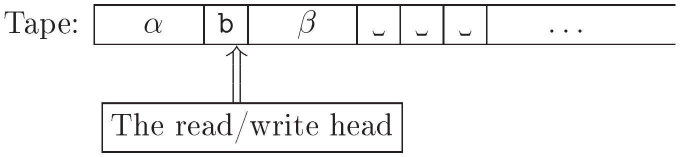
$$\begin{aligned} \forall s \in \Sigma \qquad & \delta(q_0, s) = (q_s, \$, \mathtt{R}) \\ \forall s, t \in \Sigma \qquad & \delta(q_s, t) = (q_t, s, \mathtt{R}) \\ \forall s \in \Sigma \qquad & \delta(q_s, \text{\textvisiblespace}) = (q_{\text{acc}}, \$, \mathtt{R})\,. \\ & \delta(q_0, \text{\textvisiblespace}) = (q_{\text{acc}}, \$, \mathtt{R}) \end{aligned}$$

The resulting machine is depicted in Figure 1, and here its pseudo-code:

```
Shift_Tape_Right
   At first tape position,
              remember character and write $
   At later positions,
              remember character on tape,
              and write previously remembered character.
   On blank, write remembered character and halt accepting.
```

2 Turing machine configurations

Consider a TM where the tape looks as follows,

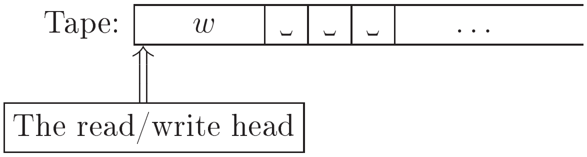


and the current control state of the TM is q_i. In this case, it would be convenient to write the TM ***configuration*** as

$$\alpha q_i \mathtt{b} \beta.$$

Namely, imagine that the head is just to the left of the cell its reading/writing, and $\mathtt{b}\beta$ is the string to the right of the head.

As such, the start ***configuration***, with a word w is



And this configuration is just $q_0 w$.

An ***accepting*** configuration for a TM is any configuration of the form $\alpha q_{\mathrm{acc}} \beta$.

We can now describe a transition of the TM using this configuration notation. Indeed, imagine the given TM is in a configuration $\alpha q_i \mathtt{a} \beta$ and its transition is

$$\delta(q_i, \mathtt{a}) = (q_j, \mathtt{c}, \mathtt{R}),$$

then the resulting configuration is $\alpha \mathtt{c} q_j \beta$. We will write the resulting transition as

$$\alpha q_i \mathtt{a} \beta \quad \Rightarrow \quad \alpha \mathtt{c} q_j \beta.$$

Similarly, if the given TM is in a configuration

$$\gamma \ \mathtt{d} \ q_k \ \mathtt{e} \ \tau,$$

where γ and τ are two strings, and $\mathtt{d}, \mathtt{e} \in \Sigma$. Assume the TM transition in this case is

$$\delta(q_k, \mathtt{e}) = (q_m, \mathtt{f}, \mathtt{L}),$$

then the resulting configuration is $\gamma \ q_m \ \mathtt{d} \ \mathtt{f} \ \tau$. We will write this transition as

$$\underbrace{\gamma \ \mathtt{d} \ q_k \ \mathtt{e} \ \tau}_{c} \quad \Rightarrow \quad \underbrace{\gamma \ q_m \ \mathtt{d} \ \mathtt{f} \ \tau}_{c'}.$$

In this case, we will say that c ***yields*** c', we will use the notation $c \mapsto c'$.

As we seen before, the ends of tape are special, as follows:

- You can not move off the tape from the left side. If the head is instructed to move to the left, it just stays where it is.

- The tape is padded on the right side with spaces (i.e., ␣). Namely, you can think about the tape as initially as being full with spaces (spaced out?), except for the input that is written on the beginning of the tape.

3 The languages recognized by Turing machines

Definition 3.1 For a TM M and a string w, the Turing machine M ***accepts*** w if there is a sequence of configurations

$$C_1, C_2, \ldots, C_k,$$

such that

(i) $C_1 = q_0 w$, where q_0 is the start state of M,

(ii) for all i, we have C_i yields C_{i+1} (using M transition function, naturally), and

(iii) C_k is an accepting configuration.

Definition 3.2 The ***language*** of a TM M (i.e., Turing machine M) is

$$L(M) = \left\{ w \,\middle|\, M \text{ accepts } w \right\}.$$

The language L is called ***Turing recognizable***.

Note, that if $w \in L(M)$ then M halts on w and accepts it. On the other hand, if $w \notin L(M)$ then either M halts and rejects w, or M loops forever on the input w. Specifically, for an input w a TM can either:

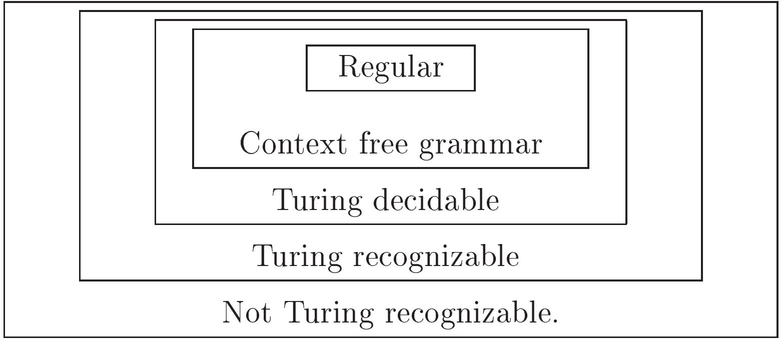
(a) accept (and then it halts),

(b) reject (and then it halts),

(c) or be in an infinite loop.

Definition 3.3 A TM that halts on all inputs is called a ***decider***.
As such, a language L is ***Turing decidable*** if there is a decider TM M, such that $L(M) = L$.

The hierarchy of languages looks as follows:



4 Variations on Turing Machines

There are many variations on the definition of a Turing machine which do not change the languages that can be recognized. Well-known variations include doubly-infinite tapes, a stay-put option, non-determinism, and multiple tapes. Turing machines can also be built with very small alphabets by encoding symbol names in unary or binary.

4.1 Doubly infinite tape

What if we allow the Turing machine to have an infinite tape on both sides? It turns out the resulting machine is not stronger than the original machine. To see that, we will show that a doubly infinite tape TM can be simulated on the standard TM.

So, consider a TM M that uses a doubly infinite tape. We will simulate this machine by a standard TM. Indeed, fold the tape of M over itself, such that location $i \in [-\infty, \infty]$ is mapped to location

$$h(i) = \begin{cases} 2|i| & i \leq 0 \\ 2i - 1 & i > 0. \end{cases}$$

on the usual tape. Clearly, now the doubly infinite tape becomes the usual one-sided infinite tape, and we can easily simulate the original machine on this new machine. Indeed, as long as we are far from the folding point on the tape, all we need to do is to just move in jumps of two (i.e., move `L` is mapped into move `LL`). Now, if we reach the beginning of the tape, we need to change between odd location and even location, but that's also easy to do with a bit of care. We omit the easy but tedious details.

Another approach would be to keep the working part of the doubly-infinite tape in its original order. When the machine tries to move off the lefthand end, push everything to the right to make more space.

4.2 Allow the head to stay in the same place

Allowing the read/write head to stay in the same place is clearly not a significant extension, since we can easily simulate this ability by moving the head to the right, and then moving it back to the left. Formally, we allow transitions to be of the form

$$\delta(q, \mathtt{c}) = (q', \mathtt{d}, \mathtt{S}),$$

where `S` denotes the command for the read/write head to stay where it is (rewriting the character on the tape from `c` to `d`).

4.3 Non-determinism

This does not buy you anything, but the details are not trivial, and we will delay the discussion of this issue to later.

4.4 Multi-tape

Consider a TM that has k tapes, where $k > 1$ is a some finite integer constant. Here each tape has its own read/write head, but there is only one finite control. The transition function of this machine, is a function

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}^k,$$

and the initial input is placed on the first tape.

5 Multiple tapes do not add any power

We next prove that one of these variations (multi-tape) is equivalent to a standard Turing machine. Proofs for most other variations are similar.

Claim 5.1 *A multi-tape* TM N *can be simulated by a standard* TM*.*

Proof: We will build a standard (single tape) TM simulating N.

Initially, the input $w$ is written on the (only) tape of M. We rewrite the tape so that it contains k strings, each string matches the content of one of the tapes of N. Thus, the rewriting of the input, would result in a tape that looks like the following:

$$\$w\underbrace{\$\,\text{␣}\,\$\,\text{␣}\ldots\$\,\text{␣}}_{k-1\text{times}}\$.$$

The string between the ith and $(i+1)$th \$ in this string, is going to be the content of the ith tape. We need to keep track on each of these tapes where the head is supposed to be. To this end, we create for each character $\texttt{a} \in \Gamma$, we create a dotted version, for example $\boxed{\overset{\bullet}{\texttt{a}}}$. Thus, if the initial input $w = xw'$, where x is a character, the new rewritten tape, would look like:

$$\$\overset{\bullet}{x}w'\underbrace{\$\overset{\bullet}{\text{␣}}\$\overset{\bullet}{\text{␣}}\ldots\$\overset{\bullet}{\text{␣}}}_{k-1\text{times}}\$.$$

This way, we can keep track of the head location in each one of the tapes.

For each move of N, we go back on M to the beginning of the tape and scan the tape from left to right, reading all the dotted characters and store them (encoding them in the current state), once we did that, we know which transition of N needs to be executed:

$$q_{\langle c_1,\ldots,c_k\rangle} \quad\rightarrow\quad q'_{\langle d_1,D_1,d_2,D_2,\ldots,d_k,D_k\rangle},$$

where $D_i \in \{\mathtt{L},\mathtt{R},\mathtt{S}\}$ is the instruction where the ith head must move. To implement this transition, we scan the tape from left to right (first moving the head to the start of the tape), and when we encounter the ith dotted character c_i, we replace it by (the undotted) d_i, and we move the head as instructed by D_i, by rewriting the relevant character (immidiately near the head location) by its dotted version. After doing that, we continue the scan to the right, to perform the operation for the remaining $i+1,\ldots,k$ tapes.

After completing this process, we might have $\overset{\bullet}{\$}$ on the tape (i.e., the relevant head is located on the end of the space allocated to its tape). We use the **Shift_Tape_Right** algorithm we describe above, to create space to the left of such a dotted dollar, and write in the newly created spot a dotted space. Thus, if the tape locally looked like

$$\ldots\texttt{ab}\,\overset{\bullet}{\$}\,\texttt{c}\ldots$$

then after the shifting right and dotting the space, the new tape would look like

$$\ldots\texttt{ab}\,\overset{\bullet}{\text{␣}}\,\$\texttt{c}\ldots$$

By doing this shift-right operation to all the dotted \$'s, we end up with a new tape that is guaranteed to have enough space if we decide to write new characters to any of the k tapes of N.

Its easy to now verify that we can now simulate N on this Turing machine M, which uses a single tape. In particular, any language that N recognizes is also recognized by M, which is a standard TM, establishing the claim. ■

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 14: Encoding problems and decidability

9 March 2010

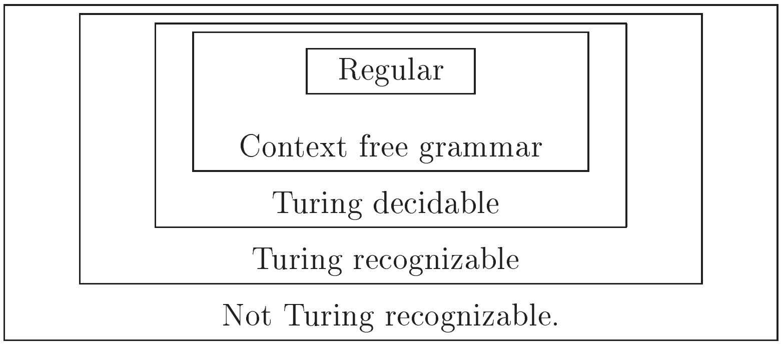
This lecture presents examples of languages that are ***Turing decidable***, and argues that existing “real” computers can be simulated by Turing machines.

1 Review and context

Remember that a Turing machine $\mathtt{D}$ can do three sorts of things on an input w. The TM $\mathtt{D}$ might halt and accept. It might halt and reject. Or it might never halt. A TM is a ***decider*** if it always halts on all inputs.

A TM ***recognizable*** language is a language L for which there is a TM $\mathtt{D}$, such that $\mathsf{L}(\mathtt{D}) = L$. A TM decidable language is a language L for which there is a decider TM $\mathtt{D}$, such that $\mathsf{L}(\mathtt{D}) = L$.

Here is a figure showing the hierarchy of languages.



Conceptually, when we think about algorithms in computer science, we are normally interested in code which is guaranteed to halt on all inputs. So, for questions about languages, our primary interest is in Turing decidable (not just recognizable) languages.

Any algorithmic task can be converted into decision problem about languages. Some tasks are naturally in this form, e.g. “Is the length of this string prime?”. In other cases, we have to restructure the question in one of two ways:

- A complex input object (e.g. a graph) may need to be encoded as a string.
- A construction task may have to be rephrased as a yes/no question.

2 TM example: Adding two numbers

2.1 A simple decision problem

For example, consider the task of adding two decimal numbers. The obvious algorithm might take two numbers a and b as input, and produce a number c as output. We can rephrase this as a question about languages by asking "Given inputs a, b, and c, is $c = a + b$".

For the alphabet

$$\Sigma = \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}, +, -\},$$

consider the language

$$L = \left\{ a_n a_{n-1} \ldots a_0 + b_m b_{m-1} \ldots b_0 = c_r c_{r-1} \ldots c_0 \;\middle|\; \begin{array}{l} a_i, b_j, c_k \in [0,9] \text{ and} \\ \quad \langle a_n a_{n-1} \ldots a_0 \rangle \\ \qquad + \quad \langle b_m b_{m-1} \ldots b_0 \rangle \\ \qquad\qquad = \langle c_r c_{r-1} \ldots c_0 \rangle \end{array} \right\},$$

where $\langle a_n a_{n-1} \ldots a_0 \rangle = \sum_{i=0}^{n} a_i \cdot 10^i$ is the number represented in base ten by the string $a_n a_{n-1} \ldots a_0$.

We then ask whether we can build a TM which decides the language L.

2.2 A decider for addition

To build a decider for this addition problem, we will use a multi-tape TM. We showed (last class) that a multi-tape TM is equivalent to a single tape TM. First, let us build a useful helper function, which reverses the contents of one tape.

2.2.1 Reversing a tape

Given the content of tape ✇$_1$, we can reverse it easily in two steps using a temporary tape. First, we put a marker onto the temporary tape. Moving the heads on both tapes to the right, we copy the contents of ✇$_1$ onto the temporary tape.

Next, we put the ✇$_1$ head at the start of its tape, but the temporary tape head remains at the end of this tape. We copy the material back onto ✇$_1$, but in reverse order, moving the ✇$_1$ head rightwards and the temporary tape head leftwards.

Let **ReverseTape**(t) denote the TM mechanism (i.e. procedure) that reverses the tth tape. We are going to buildup TM by putting together such procedures.

2.2.2 Adding two numbers

Now, let us assemble the addition algorithm. We will use five tapes: the input (✇$_1$), three tapes to hold numbers (✇$_2$, ✇$_3$, and ✇$_4$), and a scratch tape used for the reversal operation.

The TM will first scan the input tape (i.e., ✇$_1$), and copy the first number to ✇$_2$, and the second number to ✇$_3$. Next, we do **ReverseTape**(2) and **ReverseTape**(3). Now, we move the head of ✇$_2$ and ✇$_3$ to the beginning of the tapes, and we start moving them together computing the sum of the digits under the two heads, writing the output to ✇$_4$, and moving the three heads to the right. Naturally, we have a carry over digit, which we encode in the current state of the TM controller (the carry over digit is either $0, 1$ or 2).

If one of the heads of ✇$_2$ or ✇$_3$ reaches the end of the tape, then we continue moving it, interpreting ␣ as a 0. We halt when the heads on both tapes see ␣.

Next, we move the head of ✇$_4$ back to the beginning of the tape, and do **ReverseTape**(4). Finally, we compare the content of ✇$_4$ with the number written on ✇$_1$ after the = character. If they are equal, the TM accepts, otherwise it rejects.

3 Encoding a graph problem

As the above example demonstrates, the ***coding scheme*** used for the input has big impact on the complexity of our algorithm. The addition algorithm would have been easier if the numbers were written in reverse order, or if they had been in binary. Such details may affect the running time of the algorithm, but they do not change whether the problem is Turing decidable or not.

When algorithms operate on objects that are not strings, these objects need to be ***encoded*** into strings before we can make the algorithm into a decision problem. For example, consider the following situation. We are given a ***directed graph*** $G = (V, E)$, and two vertices $s, t \in V$, and we would like to decide if there is a way to reach t from s.

All sorts of encodings are possible. But it is easiest to understand if we use encodings that look like standard `ASCII` file, of the sort you might use as input to your `Java` or `C++` program. `ASCII` files look like they are two-dimensional. But remember that they are actually one-dimensional strings inside the computer. Line breaks display in a special way, but they are underlyingly just a special separator character (<`NL`> on a unix system), very similar to the \$ or # that we've used to subdivide items in our string examples.

To make things easy, we will number the vertices of V from 1 to $n = |V|$. To specify that there is an edge between two vertices u and v, we then specify the two indices of u and v. We will use the notation (u, v). Thus, to specify a graph as a text file, we could use the following format, where n is the number of vertices and m is the number of edges in the graph.

$$\begin{array}{l} n \\ m \\ (n_1, n'_1) \\ (n_2, n'_2) \\ \vdots \\ (n_m, n'_m) \end{array}$$

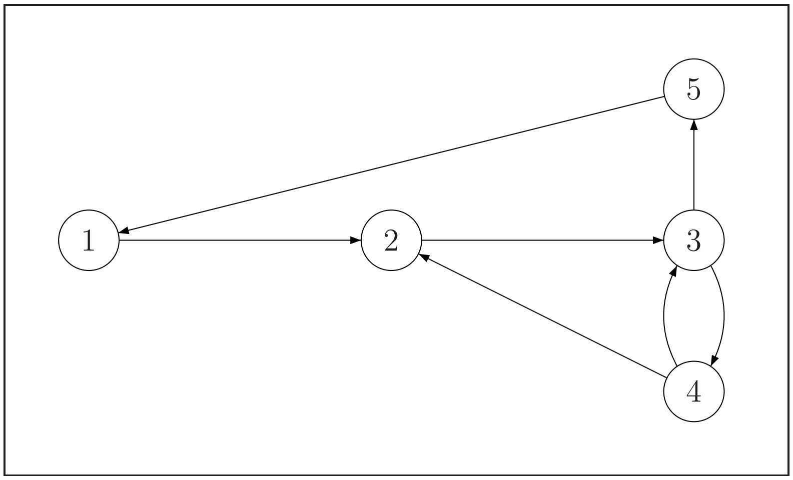
Namely, the first line of the file, will contain the number (written explicitly using `ASCII`), next the second line is the number of edges of G (i.e., m). Then, every line specify one edge of the graph, by specifying the two numbers that are the vertices of the edge. As a concrete example, consider the following graph.

The number of edges is a bit redundant, because we could just stop reading at the end of the file. But it is convenient for algorithm design.

See Figure 1, for an example of a graph its encoding using these scheme.

4 Algorithm for graph reachability

To encode an instance of the ***s, t-reachability problem***, our `ASCII` file will need to contain not only the graph but also the vertices s and t. The input tape for our TM would contain all this information, laid out in 1D (i.e. imagine the line break displayed as an ordinary separator character).



Graph encoding

```
5
7
(1,2)
(2,3)
(3,5)
(5,1)
(3,4)
(4,3)
(4,2)
```

Figure 1: A graph encoded as text. The string encoding the graph is in fact "`5`⟨`NL`⟩`7`⟨`NL`⟩`(1,2)`⟨`NL`⟩`(2,3)`⟨`NL`⟩`(3,5)`⟨`NL`⟩`(5,1)`⟨`NL`⟩`(3,4)`⟨`NL`⟩`(4,3)`⟨`NL`⟩`(4,2)`". Here ⟨`NL`⟩ denotes the spacial new-line character.

To solve this problem, we will need to search the graph, starting with node s. The TM accepts iff this search finds the node t. We will store information on four TM tapes, in addition to the input tape. The TM would have the following tapes:

✇$_1$: Input tape

✇$_2$: Target node t.

✇$_3$: Edge list.

✇$_4$: *Done list*: list of nodes that we've finished processing

✇$_5$: *To-do list*: list of nodes whose outgoing edges have not been followed

Given the graph, the TM reads the graph (checking that the input is in the right format). It puts the list of edges onto tape ✇$_3$, puts t onto its own tape (i.e., ✇$_2$), and puts the node s onto the to-do list tape (i.e., ✇$_5$).

Next, the TM loops. In each iteration, it removes the first node x from the to-do list. If $x = t$, the TM halts and accepts. Otherwise, x is added to the done list (i.e., ✇$_4$). Then the TM searches the Edge list for all edges going outwards from x. Suppose an outgoing edge goes from x to y. Then if y is not already on the finished list or the to-do list, then y is added to the to-do list.

If there is nothing left on on the to-do list, the TM halts and rejects.

This algorithm is a ***graph search*** algorithm. It is breadth-first search if the new nodes are added to the end of the to-do list and depth-first search if they are added in the start of the list. (Or, said another way, the to-do list operates as either a queue or a stack.)

The separate visited list is necessary to prevent the algorithm from going into an infinite loop if the graph contains cycles.

5 Some decidable DFA problems

If $\mathtt{D}$ is a DFA, the string encoding of $\mathtt{D}$ is written as $\langle \mathtt{D} \rangle$.

The string encoding of a DFA is similar to the encoding of a directed graph except that our encoding has to have a label for each edge, specify the start state, and list the final states.

Emptiness of DFA. Consider the language

$$\mathsf{E}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D} \rangle \,\middle|\, \mathtt{D} \text{ is a } \mathsf{DFA}\text{, and } \mathsf{L}(\mathtt{D}) = \emptyset \right\}.$$

This language is decidable. Namely, given an instance $\langle \mathtt{D} \rangle$, there is a TM that reads $\langle \mathtt{D} \rangle$, this TM always stops, and accepts if and only if $\mathsf{L}(\mathtt{D})$ is empty. Indeed, do a graph search on the DFA (as above) starting at the start state of $\mathtt{D}$, and check whether any of the final states is reachable. If so, the $\mathsf{L}(\mathtt{D}) \neq \emptyset$.

Lemma 5.1 *The language $\mathsf{E}_{\mathsf{DFA}}$ is decidable.*

Emptiness of NFA. Consider the following language

$$\mathsf{E}_{\mathsf{NFA}} = \left\{ \langle \mathtt{D} \rangle \,\middle|\, \mathtt{D} \text{ is a } \mathsf{NFA}\text{, and } \mathsf{L}(\mathtt{D}) = \emptyset \right\}.$$

This language is decidable. Indeed, convert the given NFA into a DFA (as done in class, long time ago) and then call the code for E_{DFA} on the encoded DFA. Notice that the first step in this algorithm takes the encoded version of $\mathtt{D}$ and writes the encoding for the corresponding DFA. You can imagine this as taking a state diagram as input and producing a new state diagram as output.

Equal languages for DFAs. Consider the language

$$\mathsf{EQ}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D}, \mathtt{C} \rangle \,\middle|\, \mathtt{D} \text{ and } \mathtt{C} \text{ are } \mathsf{NFA}\text{s, and } \mathsf{L}(\mathtt{D}) = \mathsf{L}(\mathtt{C}) \right\}.$$

This language is also decidable. Remember that the ***symmetric difference*** of two sets X and Y is $X \oplus Y = (X \cap \overline{Y}) \cup (Y \cap \overline{X})$. The set $X \oplus Y$ is empty if and only if the two sets are equal. But, given a DFA, we know how to make a DFA recognizing the complement of its language. And we also know how to take two DFA's and make a DFA recognizing the union or intersection of their languages.

So, given the encodings for $\mathtt{D}$ and $\mathtt{C}$, our TM will construct the encoding of a DFA $\langle \mathtt{B} \rangle$ recognizing the symmetric difference of their languages. Then it would call the code for deciding if $\langle \mathtt{B} \rangle \in \mathsf{E}_{\mathsf{DFA}}$.

Informally, problems involving regular languages are always decidable, because they are so easy to manipulate. Problems involving context-free languages are sometimes decidable. And only the simplest problems involving Turing machines are decidable.

6 The acceptance problem for DFA's

The following language is also decidable:

$$\mathsf{A}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D}, w \rangle \;\middle|\; \mathtt{D} \text{ is a } \mathsf{DFA},\ w \text{ is a word, and } \mathtt{D} \text{ accepts } w. \right\}.$$

As before, the notation $\langle \mathtt{D}, w \rangle$ is the encoding of the DFA $\mathtt{D}$ and the word w; that is, it is the pair $\langle \mathtt{D} \rangle$ and $\langle w \rangle$. For example, if $\langle w \rangle$ is just w (it's already a string), then $\langle \mathtt{D}, w \rangle$ might be $\langle \mathtt{D} \rangle \# w$ where # is some separator character. Or it might be $(\langle \mathtt{D} \rangle, w)$. Or anything similar that encodes the input well. We will just assume that it is in some such reasonable encoding of a pair and that the low-level code for our TM (which we will not spell out in detail) knows what it is.

A Turing machine deciding $\mathsf{A}_{\mathsf{DFA}}$ needs to be able to take the code for some arbitrary DFA, plus some arbitrary string, and decide if that DFA accepts that string. So it will need to contain a general-purpose DFA simulator. This is called the ***acceptance problem*** for DFA's.

It's useful to contrast this with a similar-sounding claim. If $\mathtt{D}$ is any DFA, then $\mathsf{L}(\mathtt{D})$ is Turing-decidable. Indeed, to build a TM that accepts $\mathsf{L}(\mathtt{D})$, we simply move the TM head to the right over the input, using the TM's controller to simulate the controller of the DFA directly.

In this case, we are given a specific fixed DFA $\mathtt{D}$ and we only need to cook up a TM that recognizes strings from this one particular language. This is much easier than $\mathsf{A}_{\mathsf{DFA}}$.

To decide $\mathsf{A}_{\mathsf{DFA}}$, our TM will use five tapes:

✇$_1$: input: $\langle \mathtt{D}, w \rangle$,

✇$_2$: state,

✇$_3$: final states

✇$_4$: transition triples

✇$_5$: input string.

The simulator then runs as follows:

(1) Check the format of the input. Copy the start state to tape ✇$_2$. Copy the input string to tape ✇$_5$. Copy the transition triples and final states of the input machine $\langle \mathtt{D} \rangle$ to tapes ✇$_3$ and ✇$_4$.

(2) Put the tape ✇$_5$ head at the beginning of the tape.

(3) Find a transition triple $p \xrightarrow{\mathsf{c}} q$ (written on tape ✇$_4$) whose input state and character match the state written on tape ✇$_1$ (i.e., p) and the character (i.e., c) under the head on tape ✇$_5$.

(4) Change the current state of the simulated DFA from p to q.

Specifically, copy the state q (written on the triple we just found on ✇$_4$), to tape ✇$_2$.

(5) Move the tape ✇$_5$ head to the right (i.e., the simulation handled this input character).

(6) Goto step (3).

(7) Halt the loop when the tape ✇$_5$ head sees a blank. Accept if and only if the state on tape ✇$_2$ is one of the states on list of final states, stored on tape ✇$_3$.

7 Simulating a real computer with a Turing machine

We would like to argue that we can simulate a “real” world computer on a Turing machine. Here are some key program features that we would like to simulate on a TM.

- **Numbers & arithmetic**: We already saw in previous lecture how some basic integer operations can be handled. It is not too hard to extend these to negative integers and perform all required numerical operations if we allow a TM with multiple tapes. As such, we can assume that we can implement any standard numerical operation.

  Of course, can also do floating point operations on a TM. The details are overwhelming but they are quite doable. In fact, until 20 years[1] ago, many computers implemented floating point operations using integer arithmetic. Hardware implementation of floating point-operations became mainstream, when Intel introduced the i486 in 1989 that had FPU (floating-point unit). You would probably will see/seen how floating point arithmetic works in computer architecture courses.

- **Stored constant strings**: The program we are trying to translate into a TM might have strings and constants in it. For example, it might check if the input contains the (all important) string `UIUC`. As we saw above, we can encode such strings in the states. Initially, on power-up, the TM starts by writing out such strings, onto a special tape that we use for this purpose.

- **Random-access memory**: We will use an associative memory. Here, consider the memory as having a unique label to identify it (i.e., its address), and content. Thus, if cell 17 contains the value `abc`, we will consider it as storing the pair $(17, \texttt{abc})$. We can store the memory on a tape as a list of such pairs. Thus, the tape might look like:

  $$(17, \texttt{abc})\$(1, \texttt{samuel})\$(85, \texttt{noclue})\$\ldots(11, \texttt{stamp})\$\sqcup\sqcup\sqcup\sqcup\sqcup\ldots$$

  Here, address 17 stores the string `abc`, address 1 stores the string `samuel`, and so on.

  Reading the value of address x from the tape is easy. Suppose x is written on ✇$_i$, and we would like to find the value associated with x on the memory tape and write it onto ✇$_j$. To do this, the TM scans ✇$_{mem}$ the memory tape (i.e., the tape we use to simulate the associative memory) from the beginning, till the TM encounter a pair in ✇$_{mem}$ having x as its first argument. It then copies the second part of the pair to the output tape ✇$_j$.

[1] This number keep changing. Very irritating.

Storing new value (x, y) in memory is almost as easy. If a pair having x as first element exists you delete it out (by writing a special cross-out character over it), and then you write the new pair (x, y) in the end of the tape ✇$_{mem}$.

If you wanted to use memory more efficiently, the new value could be written into the original location, whenever the original location had enough room. You could also write new pairs into crossed-out regions, if they have enough room. Implementations of C `malloc`/`free` and Java garbage collection use slightly more sophisticated versions of these ideas. However, TM designers rarely care about efficiency.

- **Subroutine calls**: To simulate a real program, we need to be able to do calls (and recursive calls). The standard way to implement such things is by having a stack. It is clear how to implement a stack on its own TM tape.

  We need to store three pieces of information for each procedure call:

  (i) private working space,

  (ii) the return value,

  (iii) and the name of the state to return to after the call is done.

  The private working space needs to be implemented with a stack, because a set of nested procedure calls might be active all at once, including several recursive calls to the same procedure.

  The return value can be handled by just putting it onto a designated register tape, say ✇$_{24}$.

  Right before we give control over to a procedure, we need to store the name of the state it should return to when it is done. This allows us to call a single fixed piece of code from several different places in our TM. Again, these return points need to be put on a stack, to handle nested procedure calls.

  After it returns from a procedure, the TM reads the state name to return to. A special set of TM states handle reading a state name and transitioning to the corresponding TM state.

These are just the most essential features for a very simple general-purpose computer. In some computer architecture class, you will see how to implement fancier program features (e.g. garbage collection, objects) on top of this simple model.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 16: Undecidability using diagonalization

16 March 2010

> ‘There must be some mistake,’ he said, ‘are you not a greater computer than the Milliard Gargantubrain at Maximegalon which can count all the atoms in a star in a millisecond?’
>
> ‘The Milliard Gargantubrain?’ said Deep Thought with unconcealed contempt. ‘A mere abacus - mention it not.’
>
> – The Hitch Hiker’s Guide to the Galaxy, by Douglas Adams.

In this lecture we will show that the problem of checking, given an input Turing machine M and a word w, whether M will accept w, is ***undecidable***. We will show this using a proof based on ***diagonalization***. This covers most of Sipser section 4.2.

We first give a two-step proof, and then combine them to give a one-step proof. We think this is less “cryptic” than the proof given in Sipser. (The two-step proof explains clearly the diagonalization; and the one-step proof, though similar to Sipser, doesn’t involve feeding a machine’s description to itself, etc. as it is not really needed.)

1 Liar’s Paradox

There’s a widespread fascination with logical paradoxes. For example, in the Deltora Quest novel “The Lake of Tears” (author Emily Rodda), the hero Lief has just incorrectly answered the trick question posed by the giant guardian of a bridge.

> “We will play a game to decide which way you will die,” said the man. “You may say one thing, and one thing only. If what you say is true, I will strangle you with my bare hands. If what you say is false, I will cut off your head.”

After some soul-searching, Lief replies “My head will be cut off.” At this point, there’s no way for the giant to make good on his threat, so the spell he’s under melts away, he changes back to his original bird form, and Lief gets to cross the bridge.

The key problem for the giant is that, if he strangles Lief, then Lief’s statement will have been false. But he said he would strangle him only if his statement was true. So that does not work. And cutting off his head does not work any better. So the giant’s algorithm sounded good, but it turned out not to work properly for certain inputs.

A key property of this paradox is that the input (Lief’s reply) duplicates material used in the algorithm. We’ve fed part of the algorithm back into itself.

2 The Turing machine acceptance problem

Consider the following language

$$\mathrm{A}_{\mathsf{TM}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } M \text{ accepts } w \right\}.$$

In the above, we fix a particular alphabet Σ (say $\Sigma = \{0, 1\}$, and encode all Turing machines M with input alphabet Σ into words over Σ, denoted $\langle M \rangle$. Hence all Turing machines in A_{TM} are over a particular alphabet Σ. Also, by $\langle, M, w \rangle$, where M is a TM (with input alphabet Σ) and $w \in \Sigma^*$, is simply an encoding of the pair $\langle M \rangle$ and w, into a single word over Σ.

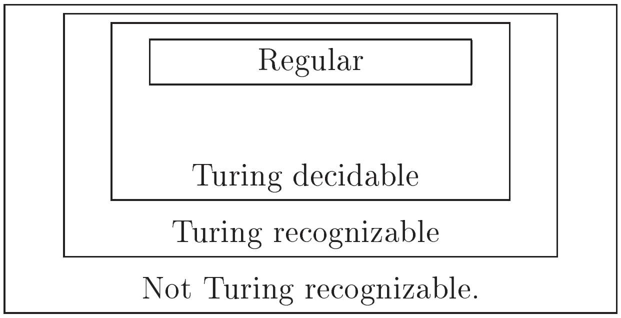
Note that for any TM M (with input alphabet Σ), $\langle M \rangle \in \Sigma^*$, but in general not every string $w \in \Sigma^*$ may correspond to the encoding of a TM.

We saw in the previous lecture, that one can build a universal Turing machine U_{TM} that can *recognize* the above language, by simulating the input Turing machine M on the input word w. Hence, using U_{TM}, we have the following TM recognizing A_{TM}:

```
Recognize-$\mathrm{A}_{\mathsf{TM}}$( $\langle M, w \rangle$)
    Simulate $M$ using $U_{\mathsf{TM}}$ till it halts
    if $M$ halts and accepts then
            accept
    else
            reject
```

Note, that if M goes into an infinite loop on the input w, then the TM **Recognize-A_{TM}** would run forever. This means that this TM is only a recognizer, not a decider. A decider for this problem would call a halt to simulations that will loop forever. So the question of whether A_{TM} is TM decidable is equivalent to asking whether we can tell if a TM M will halt on input w. Because of this, both versions of this question are typically called the ***halting*** problem.

We remind the reader that the language hierarchy looks as depicted on the right.



2.1 Implications

So, let us suppose that the Halting problem (i.e., deciding if a word in is in A_{TM}) were decidable. Namely, there is an algorithm that can solves it (for any input). this seems somewhat hard to believe since even humans can not solve this problem (and we still live under the delusion that we are smarter than computers).

If we could decide the Halting problem, then we could build compilers that would automatically prevent programs from going into infinite loops and other very useful debugging tools. We could also solve a variety of hard mathematical problems. For example, consider the following program.

```
Percolate ( $n$)
    for $p < q < n$ do
        if $p$ is prime and $q$ is prime, and $p + q = n$ then
            return

    If program reach this point then Stop!!!


Main:
    $n \leftarrow 4$
    while true do
        Percolate (n)
        $n \leftarrow n + 2$
```

Does this program stops? We do not know. If it does stop, then the ***Strong Goldbach conjecture*** is false.

Conjecture 2.1 (Strong Goldbach conjecture.) *Every even integer greater than 2 can be written as a sum of two primes.*

This conjecture is still open and its considered to be one of the major open problems in mathematics. It was stated in a letter on 7 of June 1742, and it is still open. Its seems unlikely that a computer program would be able to solve this, and a larger number of other mathematical conjectures. If A_{TM} is decidable, then we can write a program that would try to generate all possible proofs of a conjecture and verify each proof. Now, if we can decide if programs stop, then we can discover whether or not a mathematical conjecture is true or not, and this seems extremely unlikely. We will now prove that A_{TM} is undecidable.

3 A language that is not Turing recognizable

Let us show a proof that not all languages are Turing recognizable. This is true because there are fewer Turing machines than languages.

Fix an alphabet Σ and define the lexicographic order on Σ^* to be: first order strings by length, within each length put them in dictionary order.

Lexicographic order gives us a mapping from the integers to all strings, e.g. s_1 is the first string in our ordered list, and s_i is the ith string.

The encoding of each Turing machine is a finite-length string. So we can put all Turing machines into an ordered list by sorting their encodings in lexicographic order. Let us call the Turing machines in our list M_1, M_2, and so forth.

We can make an (infinite) table of how each Turing machine behaves on each input string. This table in depicted on the right. Here, the ith row represents the ith TM M_i, where the jth entry in the row is acc if M_i accepts the jth word s_j.

The idea is now to define a language from the table. Consider the language L_{diag} which is the language formed by taking the diagonal of this table.

	s_1	s_2	s_3	s_4	$\dots$
M_1	**acc**	acc	¬acc	¬acc	$\dots$
M_2	¬acc	**acc**	¬ acc	acc	$\dots$
M_3	acc	¬acc	**acc**	acc	$\dots$
M_4	¬acc	acc	¬acc	¬**acc**	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Formally, the word $s_i \in L_{\mathrm{diag}}$ if and only if M_i accepts the string s_i. Now, consider the complement language $L = \overline{L_{\mathrm{diag}}}$.

This language cannot be recognized by an of the Turing machines on the list $M_1, M_2, \dots$. Indeed, if M_k recognized the language L, then consider s_k. There are two possibilities.

- If M_k accepts s_k then the k'th entry in the k'th row of this infinite table is acc. Which implies in turn that $s_k \notin L$ (since L is the complement language), but then M_k (which recognizes L) must not accept s_k. A contradiction.

- If M_k does not accept s_k then the kth entry in the kth row of this infinite table is ¬acc. Which implies in turn that $s_k \in L$ (since L is the complement language), but then M_k (which recognizes L) must accept s_k. A contradiction.

Thus, our assumption that all languages have a TM that recognizes them is false. Let us summarize this very surprising result.

Theorem 3.1 *Not all languages have a TM that recognize them. In particular, $\overline{L_{\mathrm{diag}}}$ is not TM-recognizable (and hence not TM-decidable).*

Intuitively, the above claim is a statement above infinities: There are way more languages (uncountably many) than TMs, as the number of TMs is countable (i.e., as numerous as integer numbers). Since the cardinality of real numbers (i.e., $\aleph$) is strictly larger than the cardinality of integer numbers (i.e., $\aleph_0$), it follows that there must be an orphan language without a machine recognizing it.

A limitation of the preceding proof is that it does not identify any particular *interesting* tasks that are not TM recognizable or decidable. Perhaps the problem tasks are only really obscure problems of interest only to mathematicians. Sadly, that is not true.

4 Undecidability of Turing-machine membership

We will now show that a particular concrete problem is not TM decidable. This will let us construct particular concrete problems that are not even TM recognizable.

Theorem 4.1 (Undecidability of Turing-machine membership) *The language A_{TM} is not Turing-decidable.*

Proof: Assume A_{TM} is Turing-decidable. We will show then that $\overline{L_{\mathrm{diag}}}$ (defined above) is decidable, which contradicts Theorem 3.1.

Let M be a Turing machine that decides A_{TM}. Construct the following machine M' that decides $\overline{L_{\mathrm{diag}}}$:

1. Input: w
2. Compute the index of w, i.e. find i such that $s_i = w$.
3. Compute the i'th Turing machine M_i.
4. Feed $\langle M_i, s_i\rangle$ to M.
5. If M accepts, then reject w; if M rejects, accept w.

Since M is a decider for A_{TM}, the Turing machine above is a decider for $\overline{L_{\mathrm{diag}}}$, which contradicts Theorem 3.1. This contradiction proves that our assumption that A_{TM} is Turing-decidable is wrong. Hence A_{TM} must be undecidable.

■

5 Undecidability of Turing-machine membership: Combining the two proofs

We can *combine* the above two proofs to give a simpler (albeit more cryptic) proof that A_{TM} is undecidable.

Theorem 5.1 (Undecidability of Turing-machine membership) *The language A_{TM} is not Turing-decidable.*

Proof: Assume A_{TM} is TM decidable, and let $\widehat{M}$ be this TM deciding A_{TM}. That is, $\widehat{M}$ is a TM that always halts, and works as follows

$$\begin{cases} \widehat{M} \text{ accepts } \langle M, w\rangle & \text{if } M \text{ accepts } w \\ \widehat{M} \text{ rejects } \langle M, w\rangle & \text{if } M \text{ does not accept } w. \end{cases}$$

We will now build a new TM **Flipper**, such that on the input w, if $w = s_i$ (i.e. w is the i'th word), runs $\widehat{M}$ on the input $\langle M_i, s_i\rangle$. If $\widehat{M}$ accepts $\langle M_i, s_i\rangle$ then **Flipper** rejects w, and if $\widehat{M}$ rejects $\langle M, s_i\rangle$, then **Flipper** accepts w. Formally

```
Flipper ($w$)
  Compute $i$ such that $w = s_i$.
  Compute $M_i$. res $\leftarrow \widehat{M}(\langle M_i, s_i\rangle)$
  if res is accept then
          reject
  else
          accept
```

The key observation is that **Flipper** *always halts*. Indeed, it uses $\widehat{M}$ as a subroutine and $\widehat{M}$, by our assumptions, always halts as it is a decider. In particular, we have the following: for any $i \in \mathbb{N}$

$$\textbf{Flipper} \text{ on input } s_i : \begin{cases} \text{rejects} & \text{if } M_i \text{ accepts } s_i \\ \text{accepts} & \text{if } M_i \text{ does not accept } s_i. \end{cases}$$

Now **Flipper** is itself a TM (duh!). Let **Flipper** be the j'th Turing machine, i.e. $DHalt = M_j$. Now, consider running **Flipper** on s_j. We get the following

$$\textbf{Flipper}(\text{i.e. } M_j) \text{ on input } s_j : \begin{cases} \text{rejects} & \text{if } M_j \text{ accepts } s_j \\ \text{accepts} & \text{if } M_j \text{ does not accept } s_j. \end{cases}$$

This is absurd. Ridiculous even! Indeed, if **Flipper** (which is M_j) accepts s_j, then it must not accept it (by the above definition), which is impossible. Also, if **Flipper** rejects s_j (note that **Flipper** always stops!), then by the above definition it must accept $\langle$**Flipper**$\rangle$, which is also impossible.

Thus, it must be that our assumption that $\widehat{M}$ exists is false. We conclude that A_{TM} is not TM decidable. ■

Corollary 5.2 *The language* A_{TM} *is TM recognizable but not TM decidable.*

6 More Implications

From this basic result, we can derive a huge variety of problems that can not be solved. Spinning out these consequences will occupy us for most of the rest of the term.

Theorem 6.1 *There is no C program that reads a C program P and input w, and decides if P "accepts" w.*

The proof of the above theorem is identical to the halting theorem - we just perform our rewriting the C program.

Also, notice that being able to recognize a language and its complement implies that the language is decidable, as the following theorem testifies.

Theorem 6.2 *A language is TM decidable iff it is TM recognizable and its complement is also TM recognizable.*

Proof: It is obvious that decidability implies that the language and its complement are recognizable. To prove the other direction, assume that L and $\overline{L}$ are both recognizable. Let M and N be Turing machines recognizing them, respectively. Then we can build a decider for L by running M and N in parallel.

Specifically, suppose that w is the string input to M. Simulate both M and N using U_{TM}, but single-step the simulations. Advance each simulation by one step, alternating between the two simulations. Halt when either of the simulations halts, returning the appropriate answer.

If w is in L, then the simulation of M must eventually halt. If w is not in L, then the simulation of N must eventually halt. So our combined simulation must eventually halt and, therefore, it is a decider for L. ■

A quick consequence of this theorem is that:

Theorem 6.3 *The set complement of A_{TM} is not TM recognizable.*

If it were recognizable, then we could build a decider for A_{TM} by Theorem 6.2.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 17: Reductions

18 March 2010

1 What is a reduction?

Last lecture we proved that A_{TM} is undecidable. Now that we have one example of an undecidable language, we can use it to prove other problems to be undecidable.

Meta definition: Problem A ***reduces*** to problem B, if given a solution to B, then it implies a solution for A. Namely, we can solve B then we can solve A. We will denote this by A $\implies$ B.

An ***oracle*** ORAC for a language L is a function that receives as a word w, and it returns true if and only if $w \in L$. An oracle can be thought of as a black box that can solve membership in a language without requiring us to consider the question of whether L is computable or not. Alternatively, you can think about an oracle as a provided library function that computes whatever it requires to do, and it always return (i.e., it never goes into an infinite loop).

Intuitively, a TM decider for a language L is the ultimate oracle. Not only it can decide if a word is in L, but furthermore, it can be implemented as a TM that always stops.

In the context of showing languages are undecidable, the following more specific definition would be useful.

Definition 1.1 A language X ***reduces*** to a language Y, if one can construct a TM decider for X using a given oracle ORAC_Y for Y.

We will denote this fact by $X \implies Y$.

In particular, if X reduces to Y then given a decider for the language Y (i.e., an oracle for Y), then there is a program that can decide X. So Y must be at least as "hard" as X. In particular, if X is undecidable, then it must be that Y is also undecidable.

Warning. It is easy to get confused about which of the two problems "reduces" to the other. Do not get hung up on this. Instead, concentrate on getting the right outline for your proofs (proving them in the right direction, of course).

Reduction proof technique. Formally, consider a problem B that we would like to prove is undecidable. We will prove this via reduction, that is a proof by contradiction, similar in outline to the ones we have seen for regular and context-free languages. You assume that your new language L (i.e., the language of B) is decided by some TM M. Then you use M as a component to create a decider for some language known to be undecidable (typically A_{TM}).

This is would imply that we have a decider for A (i.e., A_{TM}). But this is a contradiction since A (i.e., A_{TM}) is not decidable. As such, we must have been wrong in assuming that L was decidable.

We will concentrate on using reductions to show that problems are undecidable. However, the technique is actually very general. Similar methods can be used to show problems to be not TM recognizable. We have used similar proofs to show languages to be not regular or not context-free. And reductions will be used in CS 473 to show that certain problems are "NP complete", i.e. these problems (probably) require exponential time to solve.

1.1 Formal argument

Lemma 1.2 *Let X and Y be two languages, and assume that $X \implies Y$. If Y is TM decidable then X is TM decidable.*

Proof: Let $\mathtt{T}$ be the TM decider for Y. Since X reduces to Y, it follows that there is a procedure $\mathtt{T}_{X|Y}$ (i.e., TM decider) for X that uses an oracle for Y as a subroutine. We replace the calls to this oracle in $\mathtt{T}_{X|Y}$ by calls to $\mathtt{T}$. The resulting TM $\mathtt{T}_X$ is a TM decider and its language is X. Thus X is TM decidable. ■

The counter-positive of this lemma, is what we will use.

Lemma 1.3 *Let X and Y be two languages, and assume that $X \implies Y$. If X is TM undecidable then Y is TM undecidable.*

2 Halting

We remind the reader that A_{TM} is the language

$$\mathrm{A}_{\mathsf{TM}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a TM and } M \text{ accepts } w \right\}.$$

This is the problem that we showed (last class) to be undecidable (via diagonalization). Right now, it is the only problem we officially know to be undecidable.

Consider the following slight modification, which is all the pairs $\langle M, w \rangle$ such that M ***halts*** on w. Formally,

$$A_{\mathrm{Halt}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a TM and } M \text{ stops on } w \right\}.$$

Intuitively, this is very similar to A_{TM}. The big obstacle to building a decider for A_{TM} was deciding whether a simulation would ever halt or not.

To show formally that A_{Halt} is undecidable, we show that we can use a oracle for A_{Halt} to build a decider for A_{TM}. This construction looks like the following.

Lemma 2.1 *The language A_{TM} reduces to A_{Halt}. Namely, given an oracle for A_{Halt} one can build a decider (that uses this oracle) for A_{TM}.*

Proof: Let ORAC_{Halt} be the given oracle for A_{Halt}. We build the following decider for A_{TM}.

```
Decider-A_TM(⟨M, w⟩)
        res ← ORAC_Halt(⟨M, w⟩)
        // if M does not halt on w then reject.
        if res = reject then
                halt and reject.

        // M halts on w since res =accept.
        // Thus, simulating M on w would terminate in finite time.
        res_2 ←Simulate M on w (using U_TM).

        return res_2.
```

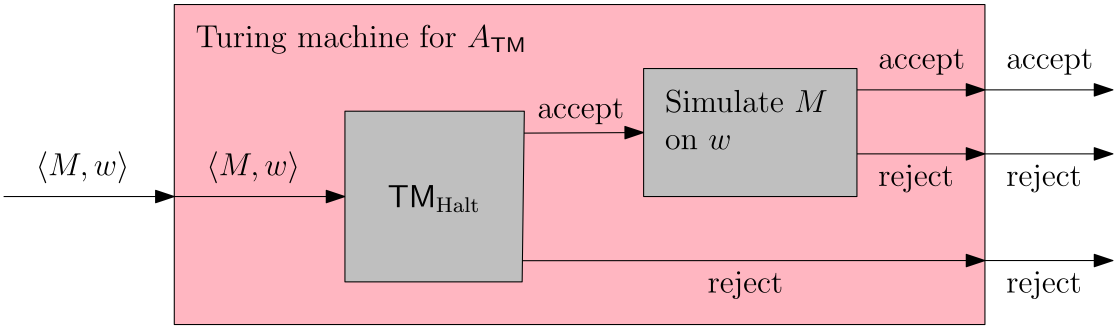
Clearly, this procedure always return and as such its a decider for A_{TM}. ■

Theorem 2.2 *The language A_{Halt} is not decidable.*

Proof: Assume, for the sake of contradiction, that A_{Halt} is decidable. As such, there is a TM, denoted by $\mathsf{TM}_{\mathrm{Halt}}$, that is a decider for A_{Halt}. We can use $\mathsf{TM}_{\mathrm{Halt}}$ as an implementation of an oracle for A_{Halt}, which would imply by Lemma 2.1 that one can build a decider for A_{TM}. However, A_{TM} is undecidable. A contradiction. It must be that A_{Halt} is undecidable. ■

We will be usually less formal in our presentation. We will just show that given a TM decider for A_{Halt} implies that we can build a decider for A_{TM}. This would imply that A_{TM} is undecidable.

Thus, given a black box (i.e., decider) $\mathsf{TM}_{\mathrm{Halt}}$ that can decide membership in A_{Halt}, we build a decider for A_{TM} is follows.



This would imply that if A_{Halt} is decidable, then we can decide A_{TM}, which is of course impossible.

3 Emptiness

Now, consider the language

$$E_{\mathsf{TM}} = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } L(M) = \emptyset \right\}.$$

Note: In class (see slides/video), we proved the above slightly differently, where the decider for A_{TM} worked by constructing, on input $\langle M, w \rangle$ a new TM M' whose language was either $\emptyset$ or $\{w\}$. Below is a variant.

Again, we assume that we have a decider for E_{TM}. Let us call it $\mathsf{TM}_{E_{\mathsf{TM}}}$. We need to use the component $\mathsf{TM}_{E_{\mathsf{TM}}}$ to build a decider for A_{TM}.

A decider for A_{TM} is given M and w and must decide whether M accepts w. We need to restructure this question into a question about some Turing machine having an empty language. Notice that the decider for E_{TM} takes only one input: a Turing machine. So we have to somehow make the second input (w) disappear.

The key trick here is to hard-code w into M, creating a TM M_w which runs M on the fixed string w. Specifically the code for M_w might look like:

```
TM $M_w$:

    1. Input = $x$ (which will be ignored)
    2. Simulate $M$ on $w$.
    3. If the simulation accepts, accept. If the simulation rejects, reject.
```

Its important to understand what is going on. The input is $\langle M \rangle$ and w. Namely, a string encoding M and a the string w. The above shows that we can write a procedure (i.e., TM) that accepts this two strings as input, and outputs the string $\langle M_w \rangle$ which encodes M_w. We will refer to this procedure as **EmbedString**. The algorithm **EmbedString**$(\langle M, w \rangle)$ as such, is a procedure reading its input, which is just two strings, and outputting a string that encodes the TM $\langle M_w \rangle$.

It is natural to ask, what is the language of the machine encoded by the string $\langle M_w \rangle$; that is, what is $L(M_w)$?

Because we are ignoring the input x, the language of M_w is either Σ^* or $\emptyset$. It is Σ^* if M accepts w, and it is $\emptyset$ if M does not accept w.

We are now ready to prove the following theorem.

Theorem 3.1 *The language E_{TM} is undecidable.*

Proof: We assume, for the sake of contradiction, that E_{TM} is decidable, and let $\mathsf{TM}_{E_{\mathsf{TM}}}$ be its decider. Next, we build our decider **AnotherDecider-A_{TM}** for A_{TM}, using the **EmbedString** procedure described above.

```
**AnotherDecider-**$\mathrm{A}_{\mathsf{TM}}$$(\langle M, w\rangle)$
        $\langle M_w\rangle \leftarrow$ **EmbedString** $(\langle M, w\rangle)$
        $r \leftarrow \mathsf{TM}_{E_{\mathsf{TM}}}(\langle M_w\rangle)$.
        **if** $r =$ accept **then**
                reject.

        // $\mathsf{TM}_{E_{\mathsf{TM}}}(\langle M_w\rangle)$ rejected its input

        **return** accept
```

Consider the possible behavior of **AnotherDecider-**A_{TM} on the input $\langle M, w\rangle$.

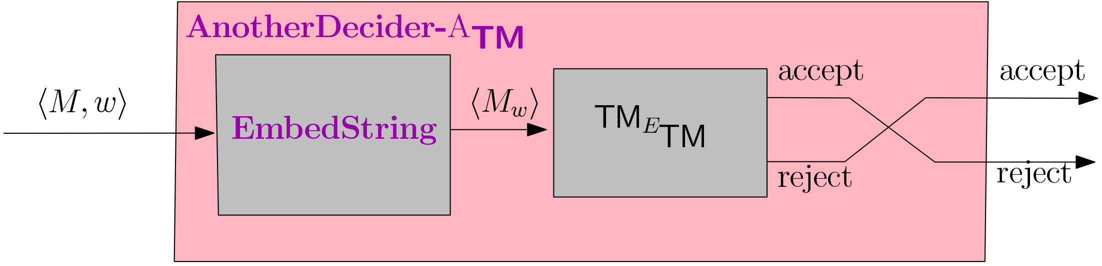
- If $\mathsf{TM}_{E_{\mathsf{TM}}}$ accepts $\langle M_w\rangle$, then $L(M_w)$ is empty. This implies that M does not accept w. As such, **AnotherDecider-**A_{TM} rejects its input $\langle M, w\rangle$.

- If $\mathsf{TM}_{E_{\mathsf{TM}}}$ accepts $\langle M_w\rangle$, then $L(M_w)$ is not empty. This implies that M accepts w. So **AnotherDecider-**A_{TM} accepts $\langle M, w\rangle$.

Namely, **AnotherDecider-**A_{TM} is indeed a decider for A_{TM}, (its a decider since it always stops on its input). But we know that A_{TM} is undecidable, and as such it must be that our assumption that E_{TM} is decidable is false. ■

In the above proof, note that **AnotherDecider-**A_{TM} is indeed a decider, so it always halts, either accepting or rejecting. By contrast, M_w might not always halt. So, when we do our analysis, we need to think about what happens if M_w never halts. In this example, if M never halts on w, then w will be treated just like the explicit rejection cases and this is what we want.

Here is the code for **AnotherDecider-**A_{TM} in flow diagram form.



Observe, that **AnotherDecider-**A_{TM} never actually runs the code for M_w. It hands the code to a function $\mathsf{TM}_{E_{\mathsf{TM}}}$ which analyzes what the code would do if we ever did choose to run it. But we never run it. So it does not matter that M_w might go into an infinite loop.

Also notice that we have two input strings floating around our code: w (one input to the decider for A_{TM}) and x (input to M_w). Be careful to keep track of which strings are input to which functions. Also be careful about how many inputs, and what types of inputs, each function expects.

CS 373: Theory of Computation

Madhusudan Parthasarathy

Lecture 18: More reductions; Rice's Theorem

30 March 2010

This lecture covers more reductions for undecidability as well as Rice's theorem.

1 Equality

An easy corollary of the undecidability of E_{TM} is the undecidability of the language

$$EQ_{\mathsf{TM}} = \left\{ \langle M, N \rangle \;\middle|\; M \text{ and } N \text{ are } \mathsf{TM}\text{'s and } L(M) = L(N) \right\}.$$

Lemma 1.1 *The language EQ_{TM} is undecidable.*

Proof: Suppose that we had a decider **DeciderEqual** for EQ_{TM}. Then we can build a decider for E_{TM} as follows:

TM R:

1. Input $= \langle M \rangle$
2. Include the (constant) code for a TM T that rejects all its input. We denote the string encoding T by $\langle T \rangle$.
3. Run **DeciderEqual** on $\langle M, T \rangle$.
4. If **DeciderEqual** accepts, then accept.
5. If **DeciderEqual** rejects, then reject.

■

Since the decider for E_{TM} (i.e., $\mathsf{TM}_{E_{\mathsf{TM}}}$) takes one input but the decider for EQ_{TM} (i.e. **DeciderEqual**) requires two inputs, we are tying one of **DeciderEqual**'s input to a constant value (i.e., T).

There are many Turing machines that reject all their input and could be used as T. Building code for R just requires writing code for one such TM.

2 Regularity

It turns out that almost any property defining a TM language induces a language which is undecidable, and the proofs all have the same basic pattern. Let us do a slightly more complex example and study the outline in more detail.

Let

$$\mathrm{Regular}_{\mathsf{TM}} = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } L(M) \text{ is regular} \right\}.$$

Suppose that we have a TM **DeciderRegL** that decides $\mathrm{Regular}_{\mathsf{TM}}$. In this case, doing the reduction from halting, would require to turn a problem about deciding whether a TM M accepts w (i.e., is $w \in \mathrm{A}_{\mathsf{TM}}$) into a problem about whether some TM accepts a regular set of strings.

Given M and w, consider the following TM M'_w:

TM M'_w:

(i) Input $= x$

(ii) If x has the form $\texttt{a}^n\texttt{b}^n$, halt and accept.

(iii) Otherwise, simulate M on w.

(iv) If the simulation accepts, then accept.

(v) If the simulation rejects, then reject.

Again, we are ***not*** going to execute M'_w directly ourself. Rather, we will feed its description $\langle M'_w \rangle$ (which is just a string) into **DeciderRegL**. Let **EmbedRegularString**denote this algorithm, which accepts as input $\langle M \rangle$ and w, and outputs $\langle M'_w \rangle$, which is the encoding of the machine M'_w.

If M accepts w, then every input x will eventually be accepted by the machine M'_w. Some are accepted right away and some are accepted in step (i). So if M accepts w then the language of M'_w is Σ^*.

If M does not accept w, then some strings x (that are of the form $\texttt{a}^n\texttt{b}^n$) will be accepted in step (ii) of M'_w. However, after that, either step (iii) will never halt or step (iv) will reject. So the rest of the strings (that are in the set $\Sigma^* \setminus \left\{ \texttt{a}^n\texttt{b}^n \;\middle|\; n \geq 0 \right\}$) will not be accepted. So the language of M'_w is $\texttt{a}^n\texttt{b}^n$ in this case.

Since $\texttt{a}^n\texttt{b}^n$ is not regular, we can use our decider **DeciderRegL** on M'_w to distinguish these two cases.

Notice that the test in step (ii) was cooked up specifically to match the capabilities of our given decider **DeciderRegL**. If **DeciderRegL** had been testing whether our language contained the string "uiuc", step (ii) would be comparing x to see if it was equal to "uiuc". This test can be anything that a TM can compute without the danger of going into an infinite loop.

Specifically, we can build a decider for A_{TM} as follows.

```
YetAnotherDecider-$\mathrm{A}_{\mathsf{TM}}$($\langle M, w \rangle$)
        $\langle M'_w \rangle \leftarrow$ EmbedRegularString ($\langle M, w \rangle$)
        $r \leftarrow$ DeciderRegL($\langle M'_w \rangle$).
        return $r$
```

The reason why **YetAnotherDecider-**A_{TM} does the right thing is that:

— If **DeciderRegL** accepts, then $L(M'_w)$ is regular. So it must be Σ^*. This implies that M accepts w. So **YetAnotherDecider-A_{TM}** should accept $\langle M, w \rangle$.

— If **DeciderRegL** rejects, then $L(M'_w)$ is not regular. So it must be $\mathtt{a}^n\mathtt{b}^n$. This implies that M does not accept w. So **YetAnotherDecider-A_{TM}** should reject $\langle M, w \rangle$.

3 Rice's Theorem

3.1 Another Example - The language L_3

Let us consider another reduction with a very similar outline. Suppose we have the following language

$$\mathsf{L}_3 = \left\{ \langle M \rangle \;\middle|\; |L(M)| = 3 \right\}.$$

That is L_3 contains all Turing machines whose languages contain exactly three strings.

Lemma 3.1 *The language L_3 is undecidable.*

Proof: Proof by reduction from A_{TM}. Assume, for the sake of contradiction, that L_3 was decidable and let $\mathtt{decider}_{\mathsf{L}_3}$ be a TM deciding it. We use $\mathtt{decider}_{\mathsf{L}_3}$ to construct a Turing machine $\mathtt{decider}_9\text{-}\mathrm{A}_{\mathsf{TM}}$ deciding A_{TM}. The decider $\mathsf{TM}\mathtt{decider}_9\text{-}\mathrm{A}_{\mathsf{TM}}$ is constructed as follows:

```
decider_9-A_TM ( ⟨M, w⟩ )
        Construct a new Turing machine $M_w$:

                $M_w$( $x$ ): // $x$:  input
                        $res \leftarrow$ Run $M$ on $w$
                        if  ($res$ = reject) then
                             reject
                        if $x$ = UIUC or $x$ = Iowa or $x$ = Michigan then
                             accept

                        reject

        return decider_{L_3} (⟨$M_w$⟩).
```

(We emphasize here, again, that constructing M_w involve taking the encoding of $\langle M \rangle$ and w, and generating the encoding of $\langle M_w \rangle$.)

Notice that the language of M_w has only two possible values. If M loops or rejects w, then $L(M_w) = \emptyset$. If M accepts w, then th the language of M_w contains exactly three strings: "`UIUC`", "`Iowa`", and "`Michigan`".

So $\mathtt{decider}_9\text{-}\mathrm{A}_{\mathsf{TM}}(\langle M_w \rangle)$ accepts exactly when M accepts w. Thus, $\mathtt{decider}_9\text{-}\mathrm{A}_{\mathsf{TM}}$ is a decider for A_{TM} But we know that A_{TM} is undecidable. A contradiction. As such, our assumption that L_3 is decidable is false. ■

3.2 Rice's theorem

Notice that these two reductions have very similar outlines. Our hypothetical decider **decider** looks for some property P. The auxiliary TM's tests x for membership in an example set with property P. The big difference is whether we simulate M on w before or after testing x and, consequently, whether the second possibility for $L(M_w)$ is $\emptyset$ or Σ^*.

It's easy to cook up many examples of reductions similar to this one, all involving sets of TM's whose *languages* share some property (e.g. they are regular, they have size three). Rice's Theorem generalizes all these reductions into a common result.

Theorem 3.2 (Rice's Theorem.) *Suppose that* L *is a language of Turing machines; that is, each word in* L *encodes a TM. Furthermore, assume that the following two properties hold.*

(a) Membership in L *depends only on the Turing machine's language, i.e. if* $L(M) = L(N)$ *then* $\langle M \rangle \in \mathsf{L} \Leftrightarrow \langle N \rangle \in \mathsf{L}$.

(b) The set L *is "non-trivial," i.e.* $\mathsf{L} \neq \emptyset$ *and* L *does not contain all Turing machines.*

Then L *is a undecidable.*

Proof: Assume, for the sake of contradiction, that L is decided by TM**deciderForL**. We will construct a TMDecider$_4$-A_{TM} that decides A_{TM}. Since Decider$_4$-A_{TM} does not exist, we will have a contradiction, implying that **deciderForL** does not exist.

Remember from last class that $\mathsf{TM}_\emptyset$ is a TM (pick your favorite) which rejects all input strings. Assume, for the time being, that $\mathsf{TM}_\emptyset \notin \mathsf{L}$. This assumption will be removed shortly.

Since L is non-trivial, also choose some other TM $Z \in \mathsf{L}$. Now, given $\langle M, w \rangle$ Decider$_4$-A_{TM} will construct the encoding of the following TM M_w.

TM M_w:

(1) Input $= x$.

(2) Simulate M on w.

(3) If the simulation rejects, halt and reject.

(4) If the simulation accepts, simulate Z on x and accept if and only if T halts and accepts.

If M loops or rejects w, then M_w will get stuck on line (2) or stop at line (3). So $L(M_w)$ is $\emptyset$. Because membership in L depends only on a Turing machine's language and $\langle \mathsf{TM}_\emptyset \rangle$ is not in L, this means that M_w is not in L. So M_w will be rejected by N.

If M accepts w, then M_w will proceed to line (4), where it simulates the behavior of Z. So $L(M_w)$ will be $L(Z)$. Because membership in L depends only on a Turing machine's language and T is L, this means that M_w is in L. So M_w will be accepted by N.

As usual, our decider for A_{TM} looks like:

```
Decider$_4$-$\mathrm{A}_{\mathsf{TM}}$ ($\langle M, w \rangle$)
        Construct $\langle M_w \rangle$ from $\langle M, w \rangle$
        return deciderForL ($\langle M_w \rangle$)
```

So $\texttt{Decider}_4\text{-}\mathrm{A}_{\mathsf{TM}}$ $(\langle M, w\rangle)$ will accept $\langle M, w\rangle$ iff **deciderForL** accepts M_w. But we saw above that **deciderForL** accepts M_w iff M accepts w. So $\texttt{Decider}_4\text{-}\mathrm{A}_{\mathsf{TM}}$ is a decider for A_{TM}. Since such a decider cannot exist, we must have been wrong in our assumption that there was a decider for L.

Now, let us remove the assumption that $\mathsf{TM}_\emptyset \notin \mathsf{L}$. The above proof showed that L is undecidable, assuming that $\langle \mathsf{TM}_\emptyset\rangle$ was not in L. If $\mathsf{TM}_\emptyset \in \mathsf{L}$, then we run the above proof using $\overline{\mathsf{L}}$ in place of L. At the end, we note that $\overline{\mathsf{L}}$ is decidable iff L is decidable. ■

A More examples

The following examples weren't presented in lecture, but may be helpful to students.

A.1 The language L_{UIUC}

Here's another example of a reduction that fits the Rice's Theorem outline.

Let

$$L_{\text{UIUC}} = \left\{ \langle M\rangle \;\middle|\; L(M) \text{ contains the string “UIUC”} \right\}.$$

Lemma A.1 *L_{UIUC} is undecidable.*

Proof: Proof by reduction from A_{TM}. Suppose that L_{UIUC} were decidable and let R be a Turing machine deciding it. We use R to construct a Turing machine deciding A_{TM}. S is constructed as follows:

- Input is $\langle M, w\rangle$, where M is the code for a Turing Machine and w is a string.
- Construct code for a new Turing machine M_w as follows:
  - Input is a string x.
  - Erase the input x and replace it with the constant string w.
  - Simulate M on w.
- Feed $\langle M_w\rangle$ to R. If R accepts, accept. If R rejects, reject.

If M accepts w, the language of M_w contains all strings and, thus, the string “UIUC”. If M does not accept w, the language of M_w is the empty set and, thus, does not contain the string “UIUC”. So R($\langle M_w\rangle$) accepts exactly when M accepts w. Thus, S decides A_{TM}

But we know that A_{TM} is undecidable. So S does not exist. Therefore we have a contradiction. So L_{UIUC} must have been undecidable. ■

A.2 The language $\mathsf{Halt\_Empty\_TM}$

Here's another example which isn't technically an instance of Rice's Theorem, but has a very similar structure.

Let

$$\mathsf{Halt\_Empty\_TM} = \Big\{ \langle M \rangle \;\Big|\; M \text{ halts on blank input} \Big\}.$$

Lemma A.2 $\mathsf{Halt\_Empty\_}$*TM is undecidable.*

Proof: By reduction from A_{TM}. Suppose that $\mathsf{Halt\_Empty\_TM}$ were decidable and let R be a Turing machine deciding it. We use R to construct a Turing machine deciding A_{TM}. S is constructed as follows:

- Input is $\langle M, w \rangle$, where M is the code for a Turing Machine and w is a string.
- Construct code for a new Turing machine M_w as follows:
  - Input is a string x.
  - Ignore the value of x.
  - Simulate M on w.
- Feed $\langle M_w \rangle$ to R. If R accepts, then accept. If R rejects, then reject.

If M accepts w, the language of M_w contains all strings and, thus, in particular the empty string. If M does not accept w, the language of M_w is the empty set and, thus, does not contain the empty string. So $R\big(\langle M_w \rangle\big)$ accepts exactly when M accepts w. Thus, S decides A_{TM}

But we know that A_{TM} is undecidable. So S can not exist. Therefore we have a contradiction. So $\mathsf{Halt\_Empty\_TM}$ must have been undecidable. ■

A.3 The language L_{111}

Here is another example of an undecidable language defined by a Turing machine's behavior, to which Rice's Theorem does not apply.

Let

$$L_{111} = \Big\{ \langle M \rangle \;\Big|\; M \text{ prints three one's in a row on blank input} \Big\}.$$

Lemma A.3 *The language L_{111} is undecidable.*

Proof: Suppose that L_{111} were decidable. Let R be a Turing machine deciding L_{111}. We will now construct a Turing machine S that decides A_{TM}.

The decider S for A_{TM} is constructed as follows:

- Input is $\langle M, w \rangle$, where M is the code for a Turing Machine and w is a string.
- Construct the code for a new Turing machine M', which is just like M except that
  - every use of the character $\mathtt{1}$ is replaced by a new character $\mathtt{1}'$ which M does not use.

  - when M would accept, M' first prints `111` and then accepts

- Similarly, create a string w' in which every character `1` has been replaced by $\texttt{1}'$.
- Create a second new Turing machine M'_w which simulates M' on the hard-coded string w'.
- Run R on $\langle M'_w \rangle$. If R accepts, accept. If R rejects, then reject.

If M accepts w, then M'_w will print `111` on any input (and thus on a blank input). If M does not accept w, then M'_w is guaranteed never to print `111` accidently. So R will accept $\langle M'_w \rangle$ exactly when M accepts w. Therefore, S decides A_{TM}.

But we know that A_{TM} is undecidable. So S can not exist. Therefore we have a contradiction. So $L_{\texttt{111}}$ must have been undecidable. ■

˜

˜

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 19: Dovetailing and non-deterministic Turing machines

6 April 2010

This lecture covers dovetailing, a method for running a gradually expanding set of simulations in parallel. We use it to demonstrate that non-deterministic TMs can be simulated by deterministic TMs.

1 Dovetailing

1.1 Interleaving

We have seen that you can run two Turing machines in parallel, to compute some function of their outputs, e.g. recognize the union of their languages.

Suppose that we had Turing machines $M_1, \ldots, M_k$ recognizing languages $L_1, \ldots, L_k$, respectively. Then, we can build a TM M which recognizes $\bigcup_{i=1}^{k} L_i$. To do this, we assume that M has k simulation tapes, plus input and working tapes. The TM M cycles through the k simulations in turn, advancing each one by a single step. If any of the simulations halts and accepts, then M halts and accepts.

We could use this same method to run a single TM M on a set of k input strings $w_1, \ldots, w_k$; that is, accept the input list if M accepts any of the strings $w_1, \ldots, w_k$.

The limitation of this approach is that the number of tapes is finite and fixed for any particular Turing machine.

1.2 Interleaving on one tape

Suppose that TM M recognizes language L and consider the language

$$\widehat{L} = \left\{ w_1 \# w_2 \# \ldots \# w_k \;\middle|\; M \text{ accepts } w_k \text{ for some } k \right\}.$$

The language $\widehat{L}$ is recognizable, but we have to be careful how we construct its recognizer $\widehat{M}$. Because M is not necessarily a decider, we can not process the input strings one after another, because one of them might get stuck in an infinite loop. Instead, we need to run all k simulations in parallel. But k is different for different inputs to $\widehat{M}$, so we can not just give k tapes to $\widehat{M}$.

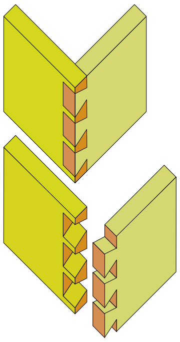
Instead, we can store all the simulations on a single tape ☢$_T$. Divide up

☢$_T$

into k sections, one for each simulation. If a simulation runs out of space in its section, push everything over to the right to make more room.

1.3 Dovetailing

Dovetailing (in carpentry) is a way of connecting two pieces of wood by interleaving them, see picture on the right.



Dovetailing is an interleaving technique for simulating many (in fact, infinite number of) TM together. Here, we would like to interleave an infinite number of simulations, so that if any of them stops, our simulation of all of them would also stop.

Consider the language:

$$J = \left\{ \langle M \rangle \;\middle|\; M \text{ accepts at least one string in } \Sigma^* \right\}.$$

It is tempting to design our recognizer for J as follows.

```
algBuggyRecog (⟨M⟩)
    $x = \epsilon$
    while True do
            simulated $M$ on $x$ (using $U_{\mathsf{TM}}$)
            if $M$ accepts then
                    halt and accept
            $x \leftarrow$ next string in lexicographic order
```

Unfortunately, if M never halts on one of the strings, this process will get stuck before it even reaches the string that M does accept. So we need to run our simulations in parallel. Since we can not start up an infinite number of simulations all at once, we use the following idea.

Dovetailing is the idea of running k simulations in parallel, but keep dynamically increasing k. So, for our example, suppose that we store all our simulations on tape ☢$_T$ and x lives on some other tape. Then our code might look like:

```
algDovetailingRecog (⟨M⟩)
    $x = \epsilon$
    while True do
            On ☢$_T$, start up the simulation of $M$ on $x$
            Advance all simulations on ☢$_T$ by one step.
            if any simulation on ☢$_T$ accepted then
                    halt and accept
            $x \leftarrow$ Next($x$)
```

Here **Next**(x) yields the next string in the lexicographic ordering.

In each iteration through the loop, we only start up one new simulation. So, at any time, we are only running a finite number of simulations. However, the set of simulations keeps expanding. So, for any string $w \in \Sigma^*$, we will eventually start up a simulation on w.

1.3.1 Increasing resource bounds

The effect of dovetailing can also be achieved by running simulations with a resource bound and gradually increasing it. For example, the following code can also recognize J.

```
Increasing resource bound

for $i = 0, 1, \ldots$
  (1) Generate the first $i$ strings (in lexicographic order) in $\Sigma^*$
  (2) On tape $T$, start up simulations of $M$ on these $i$ input strings
  (3) Run the set of simulations for $i$ steps.
  (4) If any simulation has accepted, halt and accept
  (5) Otherwise increment $i$ and repeat the loop
```

Each iteration of the loop does only a finite amount of work: i steps for each of i simulations. However, because i increases without bound, the loop will eventually consider every string in Σ^* and will run each simulation for more and more steps. So if there is some string w which is accepted by M, our procedure will eventually simulate M on w for enough steps to see it halt.

2 Nondeterministic Turing machines

A ***non-deterministic Turing machine*** (NTM) is just like a normal TM, except that the transition function generates a set of possible moves (not just a single one) for a given state and character being read. That is, the output of the transition function is a set of triples (r, c, D) where r is the new state, c is the character to write onto the tape, and D is either L or R. That is

$$\delta(q, \mathtt{c}) = \left\{ (r, \mathtt{d}, D) \;\middle|\; \text{for some } r \in Q, \mathtt{d} \in \Gamma \text{ and } D \in \{\mathtt{L}, \mathtt{R}\} \right\}.$$

An NTM M accepts an input w if there is some possible run of M on w which reaches the accept state. Otherwise, M does not accept w.

This works just like non-determinism for the simpler automata. That is, you can either imagine searching through all possible runs, or you can imagine that the NTM magically makes the right guess for what option to take in each transition.

For regular languages, the deterministic and non-deteterministic machines do the same thing. For context-free languages, they do different things. We claim that non-deterministic Turing machines can recognize the same languages as ordinary Turing machines.

2.1 NTMs recognize the same languages

Theorem 2.1 *NTMs recognize the same languages as normal TMs.*

Suppose that a language L is recognized by an NTM M. We need to construct a deterministic TM that recognizes L. We can do this using dovetailing to search all possible choices that the NTM could make for its moves.

Our simulation will use two simulation tapes S and T in a similar way. In this case, flicker isn't a big problem. But the double buffering makes the algorithm slightly easier to understand.

```
simulate NTM $M$ on input $w$

  (1) put the start configuration $q_0w$ onto tape $S$
  (2) for each configuration $C$ on $S$
        – generate all options $D_1, D_2, \ldots D_k$ for the next configuration
        – if any of $D_1, D_2, \ldots D_k$ is an accept configuration, halt and accept
        – otherwise, erase all the $D_i$ that have halted and rejected
        – copy the rest onto the end of $T$
  (3) if tape $T$ is empty, halt and reject
  (4) copy the contents of $T$ to $S$, overwriting what was there
  (5) erase tape $T$.
  (6) goto step 2
```

You can think about the set of possible configurations as a tree. The root is the start configuration. Its children are the configurations that could be reached in one step. Its grandchildren are the configurations that could be reached in two steps. And so forth. Our simulator is then doing a breadth-first search of this tree of possibilities, looking for a branch that ends in acceptance.

2.2 Halting and deciders

Like a regular TM, an NTM can cleanly reject an input string w or it can implicitly reject it by never halting. An NTM halts if all possible runs eventually halt. Once it halts, the NTM accepts the input if some run ended in the accept state, and rejects the input if all runs ended in the reject state.

A NTM is a ***decider*** if it halts on all possible inputs on all branches. Formally, if you think about all possible configurations that a TM might generate for a specific input as a tree (i.e., a branch represents a non-deterministic choice) then an NTM is a decider if and only if this tree is finite, for all inputs.

The simulation we used above has the property that the simulation halts exactly if the NTM would have halted. So we have also shown that

Theorem 2.2 *NTMs decide the same languages as normal TMs.*

2.3 Enumerators

A language can be ***enumerated***, if there exists a TM with an output tape (in addition to its working tape), that the TM prints out on this tape all the words in the language (assume that between two printed words we place a special separator character). Note, that the output tape is a write only tape.

Definition 2.3 (Lexicographical ordering.) For two strings s_1 and s_2, we have that $s_1 < s_2$ in lexicographical ordering if $|s_1| < |s_2|$ or $|s_1| = |s_2|$ then s_1 appears before s_2 in the dictionary ordering.

(That is, lexicographical ordering is a dictionary ordering for strings of the same length, and shortest strings appear before longer strings.)

Claim 2.4 *A language L is TM recognizable iff it can be enumerated.*

Proof: Let $\mathtt{T}$ the TM recognizer for L, and we need to build an enumerator for this language. Using dovetailing, we "run" $\mathtt{T}$ on all the strings in $\Sigma^* = \{w_1, w_2, \ldots\}$ (say in lexicographical ordering). Whenever one of this executions stops and accepts on a string w_i, we print this string w_i to the enumerator output string. Clearly, all the words of L would be sooner or later printed by this enumerated. As such, this language can be enumerated.

As for the other direction, assume that we are given an enumerate $\mathtt{T}_{\mathrm{enum}}$ for L. Given a word $x \in \Sigma^*$, we can recognize if it is in L, by running the enumerator and reading the strings it prints out one by one. If one of these strings is x, then we stop and accept. Otherwise, this TM would continue running. Clearly, if $x \in \mathsf{L}$ sooner or later the enumerator would output x and our TM would stop and accept it. ∎

Claim 2.5 *A language L is decidable iff it can be enumerated in lexicographic order.*

Proof: If L is finite the claim trivially hold, so we assume that L is infinite.

If L is decidable, then there is a decider **deciderFor**L for it. Generates the words of Σ^* in lexicographic ordering, as $w_1, w_2, \ldots$. In the ith stage, check if $w_i \in \mathsf{L}$ by calling **deciderFor**L on w_i. If **deciderFor**L accepts w_i then we print it to the output tape. Clearly, this procedure never get stuck since **deciderFor**L always stop. More importantly, the output is sorted in lexicographical ordering. Note, that if $x \in \mathsf{L}$, then there exists an i such that $w_i = x$. Thus, in the ith stage, the program would output x. Thus, **deciderFor**L indeed prints all the words in L.

Similarly, assume we are given a lexicographical enumerator $\mathtt{T}_{\mathrm{enum}}$ for L. Consider a word $x \in \Sigma^*$. Clearly, the number of words in Σ^* that appear before x in the lexicographical ordering is finite. As such, if x is the ith word in Σ^* in the lexicographical ordering, then if $\mathtt{T}_{\mathrm{enum}}$ outputs i words and none of them is x, then we know it would never x, and as such x is not in the language. As such, we stop and reject. Similarly, if x is output in the first i words of the output of $\mathtt{T}_{\mathrm{enum}}$ then we stop and accept. Clearly, since $\mathtt{T}_{\mathrm{enum}}$ enumerates an infinite language, it continuously spits out new strings. As such, sooner or later it would output i words of L, and at this point our procedure would stop. Namely, this procedures accepts the language L, and it always stop; namely, it is a decider for L. ∎

Lecture 20: Context-free grammars

8 April 2010

This lecture introduces context-free grammars, covering section 2.1 from Sipser.

1 Context-free grammars

1.1 Introduction

Regular languages are efficient but very limited in power. For example, not powerful enough to represent the overall structure of a `C` program.

As another example, consider the following language

$$L = \{\text{all strings formed by properly nested parenthesis}\}.$$

Here, the string (()()) is in L. ())(is not.

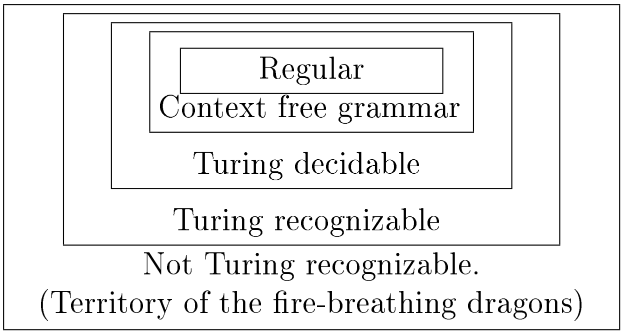
Lemma 1.1 *The language L is not regular.*

Proof: Assume for the sake of contradiction that L is regular. Then consider $L' = L \cap '('^{*})^{*}$. Since L is regular and regular languages are closed under intersection, L' must be regular. But L is just $\left\{ (^{n})^{n} \,\middle|\, n \geq 0 \right\}$. We can map this, with a homomorphism, to $\texttt{0}^n\texttt{1}^n$, which is not regular (as we seen before). A contradiction. ■

Our purpose is to come up with a way to describe the above language L in a compact way. It turns out that context-free grammars are one possible way to capture such languages.

Here is a diagram demonstrating the classes of languages we will encounter in this class. Currently, we only saw the weakest class – regular language. Next, we will see context free grammars.

!



A compiler or a natural language understanding program, use these languages as follows:

- It uses regular languages to convert character strings to tokens (e.g. words, variables names, function names).

- It uses context-free languages to parse token sequences into functions, programs, sentences.

Just as for regular languages, context-free languages have a procedural and a declarative representation, which we will show to be equivalent.

procedural	declarative
NFAs/DFAs	regular expressions
pushdown automata (PDAs)	context-free grammar

1.2 Deriving the context-free grammars by example

So, consider our old arch-nemesis, the language

$$L = \left\{ \mathtt{a}^n \mathtt{b}^n \;\middle|\; n \geq 0 \right\}.$$

we would like to come up with a recursive definition for a word in the language.

So, let S denote any word we can generate in the language, then a word w in this language can be generated as

$$w = \mathtt{a}^n\mathtt{b}^n = \mathtt{a}\ \underbrace{\boxed{\mathtt{a}^{n-1}\mathtt{b}^{n-1}}}_{=w'}\ \mathtt{b},$$

where $w' \in L$. Thus, we have a compact recursive way to generate L. It is the language containing the empty word, and one can generate a new word, by taking a word w' already in the language and padding it with a before, and b after it. Thus, generating the new word $\mathtt{a}w'\mathtt{b}$. This suggests a random procedure S to generate such a word. It either return without generating anything, or it prints a a, generates a word recursively by calling S, and then it outputs a b. Naturally, the procedure has to somehow guess which of the two options to perform. We demonstrate this idea in the C program on the right, where S uses randomization to decide which action to take. As such, running this program would generate a random word in this language.

```
#include  <stdlib.h>
#include  <stdio.h>
#include  <time.h>
int  guess()
{ return  random() % 16; }
void  S() {
    if  ( guess() == 0 ) return;
    else {
        printf( "a" );
        S();
        printf( "b" );
    }
}
int  main() {
    srand( time( 0 ) );
    S();
}
```

The way to write this recursive generation algorithm using context free grammar is by specifying

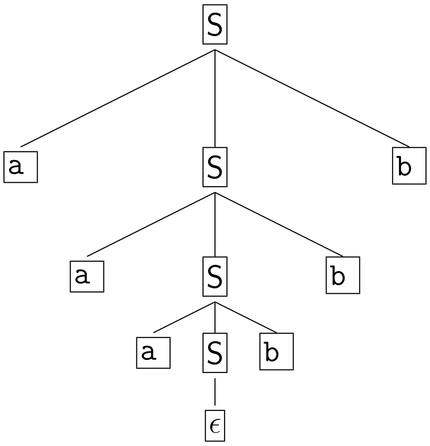
$$\mathsf{S} \rightarrow \epsilon \;\mid\; \mathsf{aSb}. \tag{1}$$

Thus, CFG can be taught of as a way to specify languages by a recursive means: We can build sole basic words, and then we can build up together more complicated words by recursively building fragments of words and concatenating them together.

For example, we can derive the word aaabbb from the grammar of Eq. (1), as follows:

$$\mathsf{S} \rightarrow \mathsf{aSb} \rightarrow \mathsf{aaSbb} \rightarrow \mathsf{aaaSbbb} \rightarrow \mathsf{aaa}\epsilon\mathsf{bbb} = \mathsf{aaabbb}.$$

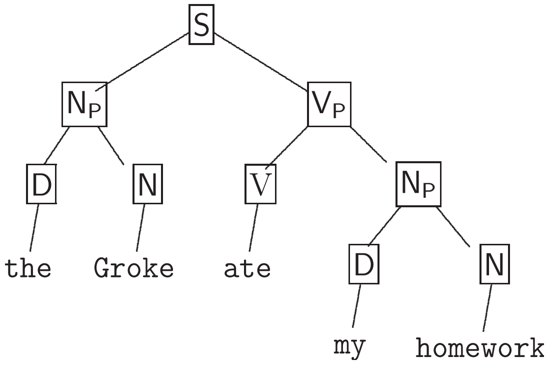
Alternatively, we can think about the recursion tree used by our program to generate this string.



This tree is known as the ***parse tree*** of the grammar of Eq. (1) for the word aaabbb.

1.2.1 Deriving the context-free grammars by constructing sentence structure

A context-free grammar defines the syntax of a program or sentence. The structure is easiest to see in a parse tree.



The interior nodes of the tree contain "variables", e.g. $\mathsf{N_P}$, D. The leaves of the tree contain "terminals". The "yield" of the tree is the string of terminals at the bottom. In this case, "the Groke at my homework." [1]

The grammar for this has several components:

(i) A start symbol: S.

(ii) A finite set of variables: $\{\mathsf{S}, \mathsf{N_P}, \mathsf{D}, \mathsf{N}, \mathsf{V}, \mathsf{V_P}\}$

(iii) A finite set of terminals = $\{\mathtt{the}, \mathtt{Groke}, \mathtt{ate}, \mathtt{my}, \ldots\}$

[1] Groke – Also known in Norway as Hufsa, in Estonia as Urr and in Mexico as La Coca is a fictional character in the Moomin world created by Tove Jansson.

(iv) A finite set of rules.

Example of how rules look like

(i) $\mathsf{S} \rightarrow \mathsf{N_P}\ \mathsf{V_P}$

(ii) $\mathsf{N_P} \rightarrow \mathsf{D}\ \mathsf{N}$

(iii) $\mathsf{V_P} \rightarrow \mathsf{V}\ \mathsf{N_P}$

(iv) $\mathsf{N} \rightarrow |$ `Groke` | homework | lunch ...

(v) $\mathsf{D} \rightarrow$ `the` | `my` ...

(vi) $\mathsf{V} \rightarrow$ `ate` | `corrected` | washed ...

If projection is working, show a sample computer-language grammar from the net. (See pointers on web page.)

1.2.2 Synthetic examples

In practical applications, the terminals are often whole words, as in the example above. In synthetic examples (and often in the homework problems), the terminals will be single letters.

Consider $L = \left\{ \mathsf{0}^n \mathsf{1}^n \;\middle|\; n \geq 0 \right\}$. We can capture this language with a grammar that has start symbol S and rule

$$\mathsf{S} \rightarrow \mathsf{0S1} \mid \epsilon\ .$$

For example, we can derive the string `000111` as follows:

$$\mathsf{S} \rightarrow 0\mathsf{S}1 \rightarrow 00S11 \rightarrow 000S111 \rightarrow 000\epsilon111 = 000111.$$

Or, consider the language of palindromes $L = \left\{ w \in \{a,b\}^* \;\middle|\; w = w^R \right\}$. Here is a grammar with start symbol P. for this language

$$\mathsf{P} \rightarrow \mathsf{aPa} \;\mid\; \mathsf{bPb} \;\mid\; \epsilon \;\mid\; \mathsf{a} \;\mid\; \mathsf{b}.$$

A possible derivation of the string abbba is

$$\mathsf{P} \rightarrow \mathsf{aPa} \rightarrow \mathsf{abPba} \rightarrow \mathsf{abbba}.$$

2 Derivations

Consider our Groke example again. It has only one parse tree, but multiple derivations: After we apply the first rule, we have two variables in our string. So we have two choices about which to expand first:

$$S \rightarrow \mathsf{N_P}\ \mathsf{V_P} \rightarrow \ldots$$

If we expand the leftmost variable first, we get this derivation:

$$S \to \mathsf{N_P}\ \mathsf{V_P} \to \mathsf{D}\ N\ \mathsf{V_P} \to \textit{the}\ N\ \mathsf{V_P} \to \textit{the Groke}\ \mathsf{V_P} \to \textit{the Groke}\ V\ \mathsf{N_P} \to \ldots$$

If we expand the rightmost variable first, we get this derivation:

$$\begin{aligned} \mathsf{S} &\to \mathsf{N_P}\ \mathsf{V_P} \to \mathsf{N_P}\ V\ \mathsf{N_P} \to \mathsf{N_P}\ V\ \mathsf{D}\ N \to \mathsf{N_P}\ V\ \mathsf{D}\ \texttt{homework} \\ &\to \mathsf{N_P}\ V\ \texttt{my homework}\ldots \end{aligned}$$

The first is called the ***leftmost derivation***. The second is called the ***rightmost derivation***. There are also many other possible derivations. Each parse tree has many derivations, but exactly one rightmost derivation, and exactly one leftmost derivation.

2.1 Formal definition of context-free grammar

Definition 2.1 (CFG) A ***context-free grammar*** (CFG) is a 4-tuple $\mathcal{G} = (V, \Sigma, R, \mathsf{S})$, where

(i) $\mathsf{S} \in V$ is the ***start variable***,

(ii) Σ is the alphabet (as such, we refer to $c \in \Sigma$ as a character or ***terminal***),

(iii) V is a finite set of ***variables***, and

(iv) R is a finite set of rules, each is of the form $\mathsf{B} \to w$ where $\mathsf{B} \in V$ and $w \in (V \cup \Sigma)^*$ is a word made out of variables and terminals..

Definition 2.2 (CFG yields.) Suppose x, y, and w are strings in $(V \cup \Sigma)^*$ and B is a variable. Then $x\mathsf{B}y$ ***yields*** xwy, written as

$$x\mathsf{B}y \Rightarrow xwy,$$

if there is a rule in R of the form $\mathsf{B} \to w$.

Notice that $x \Rightarrow x$, for any x and any set of rules.

Definition 2.3 (CFG derives.) If x and y in $(V \cup \Sigma)^*$, then w ***derives*** x, written as

$$w \overset{*}{\Rightarrow} x$$

if you can get from w to x in zero or more yields steps.

That is, there is a sequence of strings $y_1, y_2, \ldots y_k$ in $(V \cup \Sigma)^*$ such that

$$w = y_1 \Rightarrow y_2 \Rightarrow \ldots \Rightarrow y_k = x.$$

Definition 2.4 If $\mathcal{G} = (V, \Sigma, R, S)$ is a grammar, then $L(\mathcal{G})$ (the ***language*** of $\mathcal{G}$) is the set

$$L(\mathcal{G}) = \left\{ w \in \Sigma^* \;\middle|\; S \overset{*}{\Rightarrow} w \right\}..$$

That is, $L(G)$ is all the strings containing only terminals which can be derived from the start symbol of $\mathcal{G}$.

2.2 Ambiguity

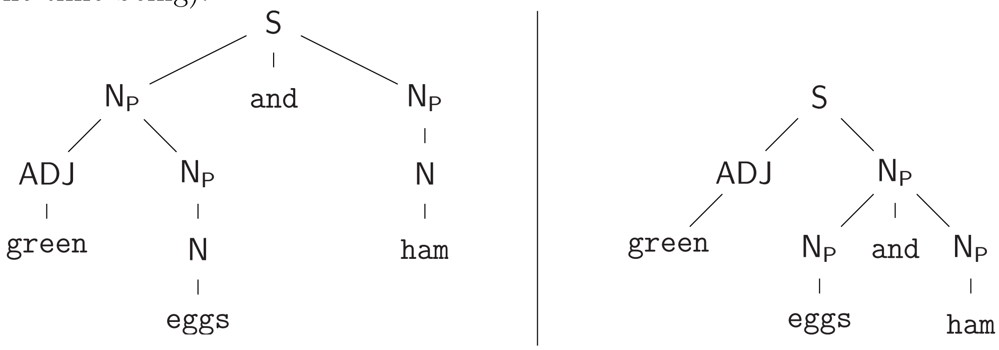
Consider the following grammar $\mathcal{G} = (V, \Sigma, R, \mathsf{S})$. Here

$$V = \{\mathsf{S}, N, \mathsf{N_P}, \mathsf{ADJ}\} \qquad \text{and} \qquad \Sigma = \{\texttt{and}, \texttt{eggs}, \texttt{ham}, \texttt{pencilgreen}, \texttt{cold}, \texttt{tasty}, \ldots\}.$$

The set R contains the following rules:

- $\mathsf{N_P} \rightarrow \mathsf{N_P}$ `and` $\mathsf{N_P}$
- $\mathsf{N_P} \rightarrow \mathsf{ADJ}\ \mathsf{N_P}$
- $\mathsf{N_P} \rightarrow \mathsf{N}$
- $\mathsf{N} \rightarrow$ `eggs` | `ham` | `pencil` | ...
- $\mathsf{ADJ} \rightarrow$ `green` | `cold` | `tasty` | ...
- ...

Here are two possible parse trees for the string `green eggs and ham` (ignore the spacing for the time being).



The two parse trees group the words differently, creating a different meaning. In the first case, only the eggs are green. In the second, both the eggs and the ham are green.

A string w is ***ambiguous*** with respect to a grammar $\mathcal{G}$ if w has more than one possible parse tree using the rules in $\mathcal{G}$.

Most grammars for practical applications are ambiguous. This is a source of real practical issues, because the end users of parsers (e.g. the compiler) need to be clear on which meaning is intended.

2.2.1 Removing ambiguity

There are several ways to remove ambiguity:

(A) Fix grammar so it is not ambiguous. (Not always possible or reasonable or possible.)

(B) Add grouping/precedence rules.

(C) Use semantics: choose parse that makes the most sense.

Grouping/precedence rules are the most common approach in programming language applications. E.g. "else" goes with the closest "if", * binds more tightly than $+$.

Invoking semantics is more common in natural language applications. For example, "The policeman killed the burgler with the knife." Did the burgler have the knife or the policeman? The previous context from the news story or the mystery novel may have made this clear.

E.g. perhaps we have already been told that the burgler had a knife and the policeman had a gun.

Fixing the grammar is less often useful in practice, but neat when you can do it. Here's an ambiguous grammar with start symbol E. N stands for "number" and E stands for "expression".

$$\mathsf{E} \rightarrow \mathsf{E} \times \mathsf{E} \mid \mathsf{E}{+}\mathsf{E} \mid \mathsf{N}$$

$$\mathsf{N} \rightarrow \mathsf{0N} \mid \mathsf{1N} \mid \mathsf{0} \mid \mathsf{1}$$

An expression like $\mathsf{0110} \times \mathsf{110} + \mathsf{01111}$ has two parse trees and, therefore, we do not know which operation to do first when we evaluate it.

We can remove this ambiguity as follows, by rewriting the grammar as

$$\mathsf{E} \rightarrow \mathsf{E} + \mathsf{T} \mid \mathsf{T}$$

$$\mathsf{T} \rightarrow \mathsf{N} \times \mathsf{T} \mid \mathsf{N}$$

$$\mathsf{N} \rightarrow \mathsf{0N} \mid \mathsf{1N} \mid \mathsf{0} \mid \mathsf{1}$$

Now, the expression $\mathsf{0110} \times \mathsf{110} + \mathsf{01111}$ must be parsed with the $+$ as the topmost operation.

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 21: Chomsky Normal form

13 April 2010

In this lecture, we are interested in transforming a given grammar into a cleaner form, known as the Chomsky Normal Form. First, we will give an algorithm that decides if the language of a CFG is empty or not. Similar algorithms will be used in the conversion of a grammar to CFG.

1 Emptiness of a context-free grammar

1.1 Finding useless variables: variables that do not generate anything

In the next step we remove variables that do not generate any string.

Given a grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, we would like to find all variables that do not derive any string (not even ϵ. To this end, consider the following algorithm, which is a *fixed-point* algorithm, that builds larger and larger subsets of variables that can generate some word.

```
**compGeneratingVars** $\Big(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})\Big)$
    $V_{\text{old}} \leftarrow \emptyset$
    $V_{\text{new}} \leftarrow V_{\text{old}}$
    do
        $V_{\text{old}} \leftarrow V_{\text{new}}.$
        for $\mathsf{X} \in \mathcal{V}$ do
            for $(\mathsf{X} \rightarrow w) \in \mathcal{R}$ do
                if $w \in (\Sigma \cup V_{\text{old}})^*$ then
                    $V_{\text{new}} \leftarrow V_{\text{new}} \cup \{\mathsf{X}\}$
    while $(V_{old} \neq V_{\text{new}})$
    $V' \leftarrow V_{\text{new}}$
    return $V'$.
```

Lemma 1.1 *Given a context-free grammar (CFG) $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ we can compute the subset of variables V' consisting of all variables that can generate some word in the language, i.e. $V' = \{X \mid X \Rightarrow^* w, w \in \Sigma^*\}$.*

Note, that if a grammar $\mathcal{G}$ generates an empty language iff the start variable $S \notin V'$, where V' is the set computed by the above algorithm. Hence, the emptiness problem for CFGs is decidable.

Theorem 1.2 (CFG emptiness.) *Given a CFG $\mathcal{G}$, there is an algorithm that decides if the language of $\mathcal{G}$ is empty.*

2 Removing ϵ-productions and unit rules from a grammar

Next, we would like to remove ***ϵ-production*** (i.e., a rule of the form $\mathsf{X} \rightarrow \epsilon$) and ***unit-rules*** (i.e., a rule of the form $\mathsf{X} \rightarrow \mathsf{Y}$) from the language. This is somewhat subtle, and one needs to be careful in doing this removal process.

2.1 Discovering nullable variables

Given a grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, we are interested in discovering all the nullable variables. A variable $\mathsf{X} \in \mathcal{V}$ is ***nullable***, if there is a way derive the empty string from X in $\mathcal{G}$. This can be done with the following algorithm.

```
compNullableVars $\left(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})\right)$
        $V_{\text{null}} \leftarrow \emptyset$
        do
            $V_{\text{old}} \leftarrow V_{\text{null}}$.
            for  $\mathsf{X} \in \mathcal{V}$ do
                for  $(\mathsf{X} \rightarrow w) \in \mathcal{R}$ do
                    if $w = \epsilon$  or  $w \in (V_{\text{null}})^*$ then
                        $V_{\text{null}} \leftarrow V_{\text{null}} \cup \{\mathsf{X}\}$
        while $(V_{\text{null}} \neq V_{\text{old}})$
        return $V_{\text{null}}$.
```

2.2 Removing ϵ-productions

A rule is an ***ϵ-production*** if it is of the form $VX \rightarrow \epsilon$. We would like to remove all such rules from the grammar (or almost all of them).

To this end, we run **compNullableVars** on the given grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, and get the set of all nullable variable V_{null}. If the start variable is nullable (i.e., $\mathsf{S} \in V_{\text{null}}$), then we create a new start state S', and add the rules to the grammar

$$\mathsf{S}' \rightarrow \mathsf{S} \mid \epsilon.$$

We also now remove all the other rules of the form $\mathsf{X} \rightarrow \epsilon$ from $\mathcal{R}$. Let $\mathcal{G}' = (\mathcal{V}', \Sigma, \mathcal{R}', \mathsf{S}')$ be the resulting grammar. The grammar $\mathcal{G}'$ is not equivalent to the original rules, since we missed some possible productions. For example, if we had the rule

$$\mathsf{X} \rightarrow \mathsf{ABC},$$

where B is nullable, then since B is no longer nullable (we removed all the ϵ-productions form the language), we missed the possibility that $\mathsf{B} \overset{*}{\Longrightarrow} \epsilon$. To compensate for that, we need to add back the rule

$$\mathsf{X} \rightarrow \mathsf{AC},$$

to the set of rules.

So, for every rule $\mathsf{A} \rightarrow X_1 X_2 \ldots X_m$ is in $\mathcal{R}'$, we add the rules of the form $\mathsf{A} \rightarrow \alpha_1 \ldots \alpha_m$ to the grammar, where

(i) If X_i is not nullable (its a character or a non-nullable variable), then $\alpha_i = X_i$.

(ii) If X_i is nullable, then α_i is either X_i or ϵ.

(iii) Not all α_is are ϵ.

Let $\mathcal{G}'' = (\mathcal{V}, \Sigma, \mathcal{R}', \mathsf{S}')$ be the resulting grammar. Clearly, no variable is nullable, except maybe the start variable, and there are no ϵ-production rules (except, again, for the special rule for the start variable).

Note, that we might need to feed $\mathcal{G}''$ into our procedures to remove useless variables. Since this process does not introduce new rules or variables, we have to do it only once.

3 Removing unit rules

A ***unit rule*** is a rule of the form $\mathsf{X} \rightarrow \mathsf{Z}$. We would like to remove all such rules from a given grammar.

3.1 Discovering all unit pairs

We have a grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ that has no useless variables or ϵ-predictions. We would like to figure out all the unit pairs. A pair of variables Y and X is a ***unit pair*** if $\mathsf{X} \overset{*}{\Longrightarrow} \mathsf{Y}$ by $\mathcal{G}$. We will first compute all such pairs, and their we will remove all unit

Since there are no ϵ transitions in $\mathcal{G}$, the only way for $\mathcal{G}$ to derive Y from X, is to have a sequence of rules of the form

$$\mathsf{X} \rightarrow \mathsf{Z}_1, \mathsf{Z}_1 \rightarrow \mathsf{Z}_2, \ldots, \mathsf{Z}_{k-1} \rightarrow \mathsf{Z}_z = \mathsf{Y},$$

where all these rules are in $\mathcal{R}$. We will generate all possible such pairs, by generating explicitly the rules of the form $\mathsf{X} \rightarrow \mathsf{Y}$ they induce.

```
compUnitPairs $\Big(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})\Big)$
    $R_{\text{new}} \leftarrow \left\{ \mathsf{X} \rightarrow \mathsf{Y} \,\middle|\, (\mathsf{X} \rightarrow \mathsf{Y}) \in \mathcal{R} \right\}$
    do
        $R_{\text{old}} \leftarrow R_{\text{new}}$.
        for $(\mathsf{X} \rightarrow \mathsf{Y}) \in R_{\text{new}}$ do
            for $(\mathsf{Y} \rightarrow \mathsf{Z}) \in R_{\text{new}}$ do
                $R_{\text{new}} \leftarrow R_{\text{new}} \cup \{\mathsf{X} \rightarrow \mathsf{Z}\}$.
    while $(R_{\text{new}} \neq R_{\text{old}})$
    return $R_{\text{new}}$.
```

3.2 Removing unit rules

If we have a rule $X \rightarrow Y$, and $Y \rightarrow w$, then if we want to remove the unit rule $X \rightarrow Y$, then we need to introduce the new rule $X \rightarrow w$. We want to do that for all possible unit pairs.

```
removeUnitRules $\left(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, S)\right)$
        $U \leftarrow$ compUnitPairs($\mathcal{G}$)
        $\mathcal{R} \leftarrow \mathcal{R} \setminus U$
        for  $(X \rightarrow A) \in U$ do
            for  $(A \rightarrow w) \in R_{\text{old}}$ do
                $\mathcal{R} \leftarrow \mathcal{R} \cup \{X \rightarrow w\}$.
        return $(\mathcal{V}, \Sigma, \mathcal{R}, S)$.
```

We thus established the following result.

Theorem 3.1 *Given an arbitrary CFG, one can compute an equivalent grammar $\mathcal{G}'$, such that $\mathcal{G}'$ has no unit rules, no ϵ-productions (except maybe a single ϵ-production for the start variable), and no useless variables.*

4 Chomsky Normal Form

Chomsky Normal Form requires that each rule in the grammar is either

(C1) of the form $A \rightarrow BC$, where A, B, C are all variables and neither B nor C is the start variable.

(That is, a rule has exactly two variables on its right side.)

(C2) $A \rightarrow a$, where A is a variable and a is a terminal.

(A rule with terminals on its right side, has only a single character.)

(C3) $S \rightarrow \epsilon$, where S is the start symbol.

(The start variable can derive ϵ, but this is the only variable that can do so.)

Note, that rules of the form $A \rightarrow B$, $A \rightarrow BCD$ or $A \rightarrow aC$ are all illegal in a CNF.

Also a grammar in CNF never has the start variable on the right side of a rule.

Why should we care for CNF? Well, its an effective grammar, in the sense that every variable that being expanded (being a node in a parse tree), is guaranteed to generate a letter in the final string. As such, a word w of length n, must be generated by a parse tree that has $O(n)$ nodes. This is of course not necessarily true with general grammars that might have huge trees, with little strings generated by them.

4.1 Outline of conversion algorithm

All context-free grammars can be converted to CNF. We did most of the steps already. Here is an outline of the procedure:

(i) Create a new start symbol S_0, with new rule $\mathsf{S}_0 \rightarrow \mathsf{S}$ mapping it to old start symbol (i.e., S).

(ii) Remove nullable variables (i.e., variables that can generate the empty string).

(iii) Remove unit rules (i.e., variables that can generate each other).

(iv) Restructure rules with long righthand sides.

The only step we did not describe yet is the last one.

4.2 Final restructuring of a grammar into CNF

Assume that we already cleaned up a grammar by applying the algorithm of Theorem 3.1 to it. So, we now want to convert this grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ into CNF.

Removing characters from right side of rules. As a first step, we introduce a variable V_c for every character $c \in \Sigma$ and it to $\mathcal{V}$. Next, we add the rules $\mathsf{V}_c \rightarrow c$ to the grammar, for every $c \in \Sigma$.

Now, for any string $w \in (\mathcal{V} \cup \Sigma)^*$, let $\widehat{w}$ denote the string, such that any appearance of a character c in w, is replaced by V_c.

Now, we replace every rule $\mathsf{X} \rightarrow w$, such that $|w| > 1$, by the rule $\mathsf{X} \rightarrow \widehat{w}$.

Clearly, (C2) and (C3) hold for the resulting grammar, and furthermore, any rule having variables on the right side, is made only of variables.

Making rules with only two variables on the right side. The only remaining problem, is that in the current grammar, we might have rules that are too long, since they have long string on the right side. For example, we might have a rule in the grammar of the form

$$\mathsf{X} \rightarrow \mathsf{B}_1\mathsf{B}_3 \ldots \mathsf{B}_k.$$

To make this into a binary rule (with only two variables on the right side, we remove this rule from the grammar, and replace it by the following set of rules

$$\begin{aligned}
&\mathsf{X} \rightarrow \mathsf{B}_1\mathsf{Z}_1 \\
&\mathsf{Z}_1 \rightarrow \mathsf{B}_2\mathsf{Z}_2 \qquad\qquad\qquad\qquad \mathsf{Z}_2 \rightarrow \mathsf{B}_3\mathsf{Z}_3 \\
\ldots& \\
&\mathsf{Z}_{k-3} \rightarrow \mathsf{B}_{k-2}\mathsf{Z}_{k-2} \\
&\mathsf{Z}_{k-2} \rightarrow \mathsf{B}_{k-1}\mathsf{B}_k,
\end{aligned}$$

where $\mathsf{Z}_1, \ldots, \mathsf{Z}_{k-2}$ are new variables.

We repeat this process, till all rules in the grammar is binary. This gramamr is now in CNF. We summarize our result.

Theorem 4.1 (CFG → CNF.) *Any context-free grammar can be converted into Chomsky normal form.*

4.3 An example of converting a CFG into CNF

Let us look at an example grammar with start symbol S.

$$\text{(G0)} \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{aB} \\ & \mathsf{A} \rightarrow \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \rightarrow \mathsf{b} \mid \epsilon \end{array}}$$

After adding the new start symbol S_0, we get the following grammar.

$$\text{(G1)} \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{S} \\ & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{aB} \\ & \mathsf{A} \rightarrow \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \rightarrow \mathsf{b} \mid \epsilon \end{array}}$$

Removing nullable variables In the above grammar, both A and B are the nullable variables. We have the rule $\mathsf{S} \rightarrow \mathsf{ASA}$. Since A is nullable, we need to add $\mathsf{S} \rightarrow \mathsf{SA}$ and $\mathsf{S} \rightarrow \mathsf{AS}$ and $\mathsf{S} \rightarrow \mathsf{S}$ (which is of course a silly rule, so we will not waste our time putting it in). We also have $\mathsf{S} \rightarrow \mathsf{aB}$. Since B is nullable, we need to add $\mathsf{S} \rightarrow \mathsf{a}$. The resulting grammar is the following.

$$\text{(G2)} \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{S} \\ & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{A} \rightarrow \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \rightarrow \mathsf{b} \end{array}}$$

Removing unit rules. The unit pairs for this grammar are $\{\mathsf{A} \rightarrow \mathsf{B}, \mathsf{A} \rightarrow \mathsf{S}, \mathsf{S}_0 \rightarrow \mathsf{S}\}$. We need to copy the productions for S up to S_0, copying the productions for S down to A, and copying the production $\mathsf{B} \rightarrow \mathsf{b}$ to $\mathsf{A} \rightarrow \mathsf{b}$.

$$\text{(G3)} \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{ASA} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{A} \rightarrow \mathsf{b} \mid \mathsf{ASA} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{B} \rightarrow \mathsf{b} \end{array}}$$

Final restructuring. Now, we can directly patch any places where our grammar rules have the wrong form for CNF. First, if the rule has at least two symbols on its righthand side but some of them are terminals, we introduce new variables which expand into these terminals. For our example, the offending rules are $\mathsf{S}_0 \rightarrow \mathsf{aB}$, $\mathsf{S} \rightarrow \mathsf{aB}$, and $\mathsf{A} \rightarrow \mathsf{aB}$. We can fix these by replacing the a's with a new variable U, and adding a rule $\mathsf{U} \rightarrow \mathsf{a}$.

$$\text{(G4)} \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{ASA} \mid \mathsf{UB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{UB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{A} \rightarrow \mathsf{b} \mid \mathsf{ASA} \mid \mathsf{UB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{B} \rightarrow \mathsf{b} \\ & \mathsf{U} \rightarrow \mathsf{a} \end{array}}$$

Then, if any rules have more than two variables on their righthand side, we fix that with more new variables. For the grammar (G4), the offending rules are $\mathsf{S}_0 \rightarrow \mathsf{ASA}$, $\mathsf{S} \rightarrow \mathsf{ASA}$, and $\mathsf{A} \rightarrow \mathsf{ASA}$. We can rewrite these using a new variable Z and a rule $\mathsf{Z} \rightarrow \mathsf{SA}$. This gives us the CNF grammar shown on the right.

$$
\boxed{
\text{(G5)} \quad
\begin{array}{rl}
\Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\
 & \mathsf{S} \rightarrow \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\
 & \mathsf{A} \rightarrow \mathtt{b} \mid \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\
 & \mathsf{B} \rightarrow \mathtt{b} \\
 & \mathsf{U} \rightarrow \mathtt{a} \\
 & \mathsf{Z} \rightarrow \mathsf{SA}
\end{array}
}
$$

We are done!

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 23: Repetition in context free languages

20 April 2010

1 Generating new words

We are interested in phenomena of repetition in context free languages. We had seen that regular languages repeat themselves if the strings are sufficiently long. We would like to make a similar statement about regular languages, but unfortunately, while the general statement is correct, the details are somewhat more involved.

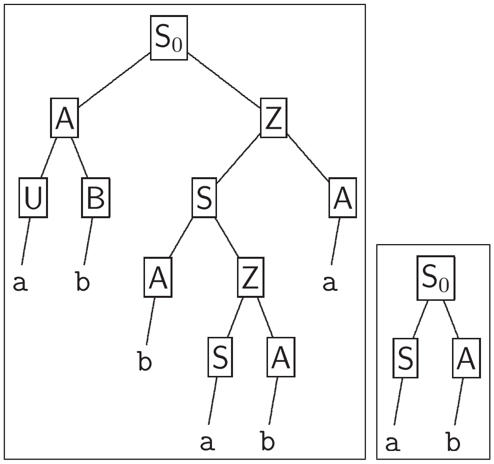
1.1 Example of repetition

As a concrete example, consider the following context-free grammar which is in Chomsky normal form (CNF). (We remind the reader that any context free grammar can be converted into CNF, as such assuming that we have a grammar in CNF form does not restrict our discussion.)

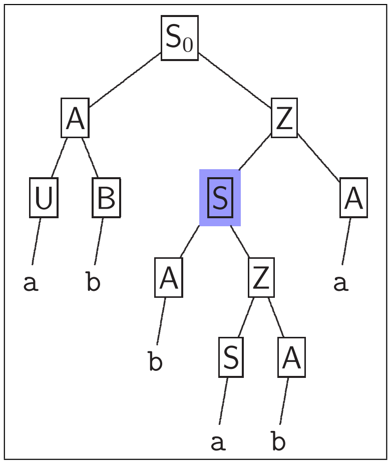
As a concrete example, consider the following grammar from the previous lecture:

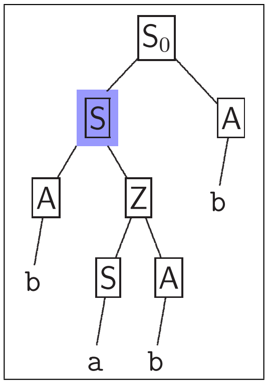
$$\text{(G5)}\qquad \begin{array}{|ll|} \hline \Rightarrow & \mathsf{S_0} \to \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{S} \to \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{A} \to \mathtt{b} \mid \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{B} \to \mathtt{b} \\ & \mathsf{U} \to \mathtt{a} \\ & \mathsf{Z} \to \mathsf{SA} \\ \hline \end{array}$$

Next, consider the two words created from this grammar, as depicted on the right.

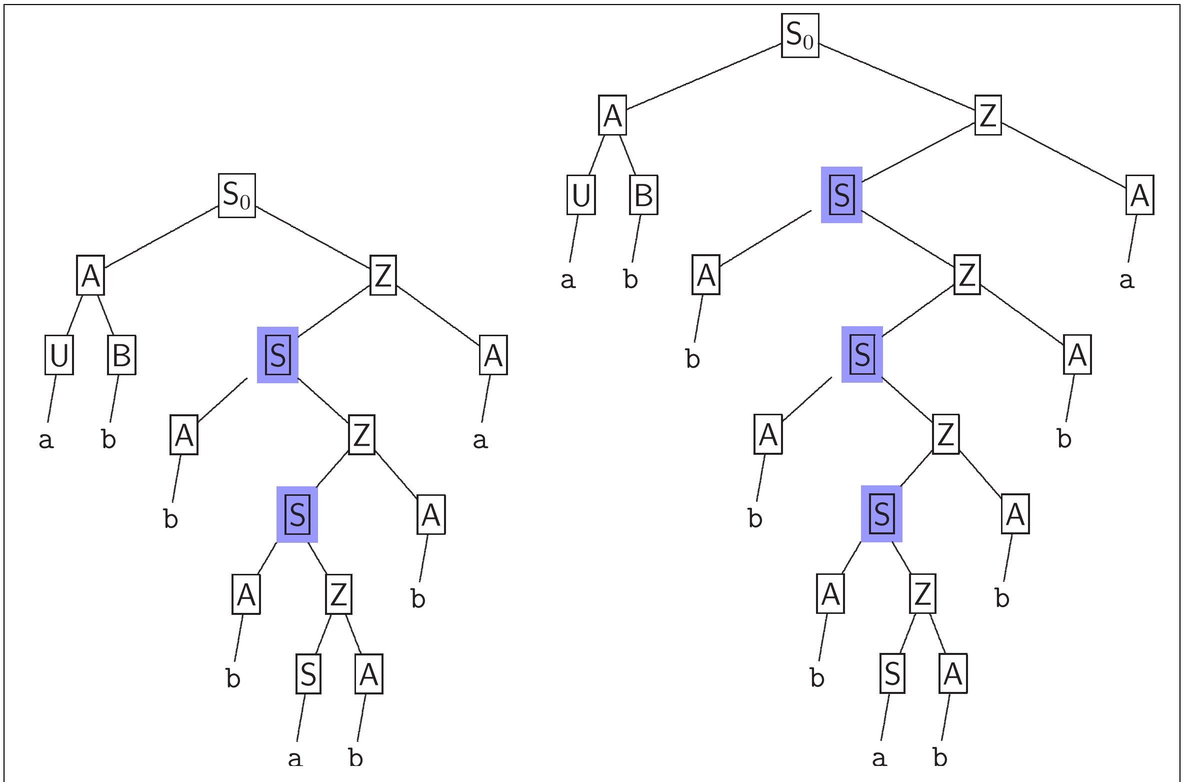


What if we wanted to make the second word longer without thinking too much? Well, both parse trees have subtrees with nodes generated by the variable S. As such, we could just cut the subtree of the first word using the variable S, and replace the subtree of S in the second word by this subtree, which would like the following:





Even more interestingly, we can do this cut and paste on the original tree:



(Pumping once.) (Pumping twice.)

Naturally, we can repeat this pumping operation (cutting and pasting a subtree) as many times as want, see for example Figure 1. In particular, we get that the word $\mathtt{abb}^i\mathtt{ab}^i\mathtt{a}$, for any i, is in the language of the grammar (G5). Notice that unlike the pumping lemma for regular languages, here the repetition happens in two places in the string. We claim that such a repetition (in two places) in the word must happen for any context free language, once we take a word which is sufficiently long.

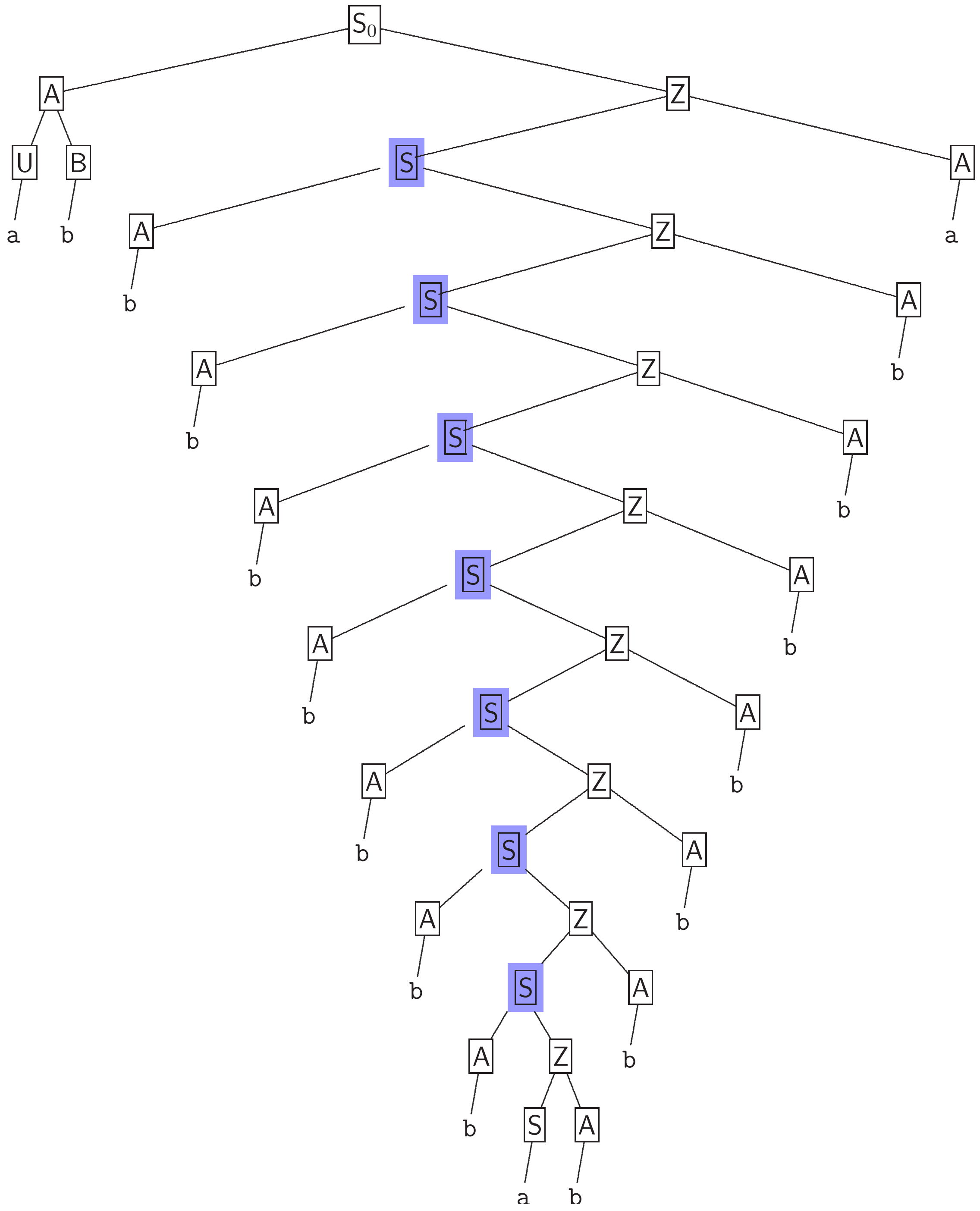


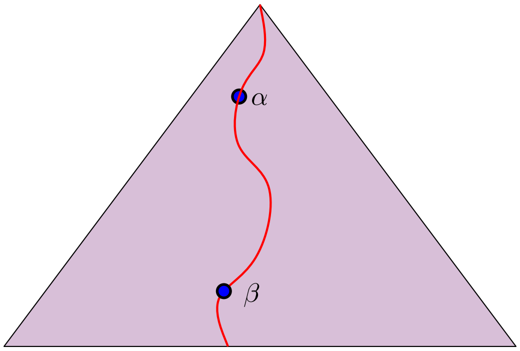
Figure 1: Naturally, one can pump the string as many times as one want, to get a longer and longer string.

2 The pumping lemma for CFG languages

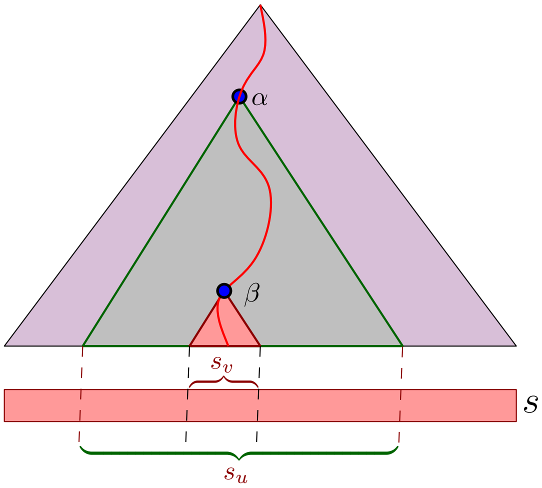
So, assume we are given a context free grammar $\mathcal{G}$ which is in CNF, and it has m variables.

2.1 If a variable repeats

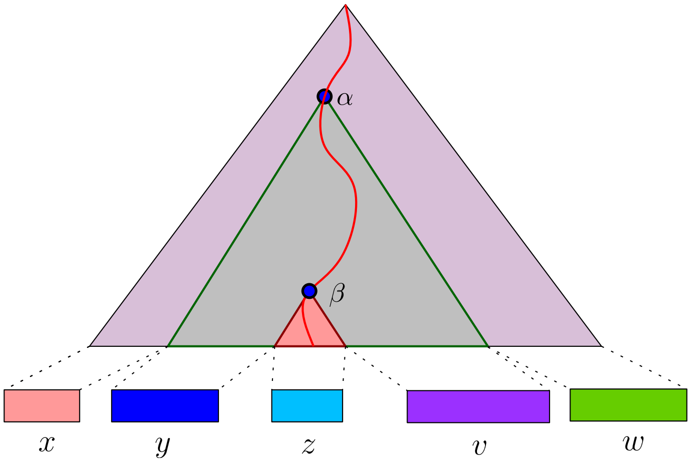
So, assume we have a parsing tree T for a word s (where the underlying grammar is in CNF), and there is a path in T from the root to a leaf, such that a variables repeats twice. So, say nodes α and β have the same variable (say S) stored in them:



The subtrees rooted at α and β corresponds to substrings of s:

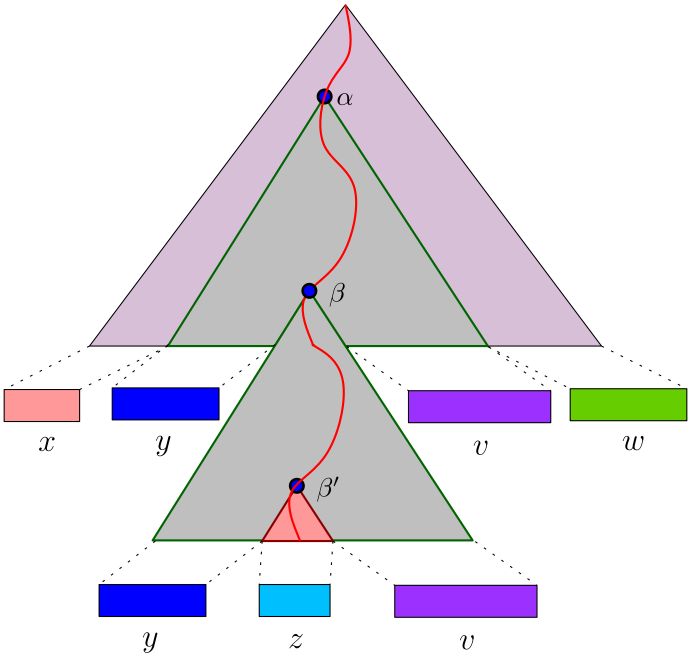


In particular, the substrings s_u and s_v break s into 5 parts:



Namely, s can be written as $s = xyzvw$.

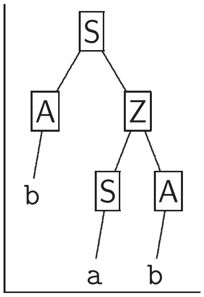
Now, if we copy the subtree rooted at α and copy it to β, we get a new parse tree:



The new tree is much bigger, and the new string it represents is $s = xyyzvvw$. In general, if we do this cut & paste operation $i - 1$ times, we get the string

$$xy^i zv^i w.$$

2.2 How tall the parse tree have to be?



We will refer to a parse generated from a context free grammar in CNF form as a ***CNF tree***. A CNF tree has the special property that the parent of a leaf as a single child which is a terminal. The ***height*** of a tree is the maximum number of edges on a path from the root of the tree to a leaf. Thus, the tree depicted on the right has height 3.

The grammar $\mathcal{G}$ has m variables. As such, if the parse tree T has a path π from the root of length k, and $k > m$ (i.e., the path has k edges), then it must contain at least $m + 1$ variables (the last edge is between a variable and a terminal). As such, by the pigeon hole principle, there must be a repeated variable along π. In particular, a parse tree that does not have a repeated variable have height at most m.

Since $\mathcal{G}$ is in CNF, its a binary tree, and a variable either has two children, or a single child which is a leaf (and that leaf contains a single character of the input). As such, a tree of height at most m, contains at most 2^m leaves[1], and represents as such a string of length at most 2^m.

[1] In fact, a CNF tree of height m can have at most 2^{m-1} leaves (figure out why), but thats a subtlety we will ignore that anyway works in our favor.

We restate the above observation formally for the record.

Observation 2.1 *If a* CNF *parse tree (or a subtree of such a tree) has height* h*, then the string it generates is of length at most* 2^h*.*

Lemma 2.2 *Let* $\mathcal{G}$ *be a grammar given in Chomsky Normal Form (*CNF*), and consider a word* $s \in \mathsf{L}(\mathcal{G})$*, such that* $\ell = |s|$ *is strictly larger than* 2^m *(i.e.,* $\ell > 2^m$*). Then, any parse tree* T *for* s *(generated by* $\mathcal{G}$*) must have a path from the root to some leaf with a repeated variable on it.*

Proof: Assume for the sake of contradiction that T has no repeated variable on any path from the root, then the height of T is at most m. But a parse tree of height m for a CNF can generate a string of length at most 2^m. A contradiction, since $\ell = |s| > 2^m$. ■

2.3 Pumping Lemma for CNF grammars

We need the following observation.

Lemma 2.3 (CNF is effective.) *In a* CNF *parse tree* T*, if* u *and* v *are two nodes, both storing variables in them, and* u *is an ancestor of* v*, then the string* S_u *generated by the subtree of* u *is strictly longer than the substring* S_v *generated by the subtree of* u*. Namely,* $|S_u| > |S_v|$*. (Of course,* S_v *is a substring of* S_u*.)*

Proof: Assume that the node u stores a variable X, and that we had used the rule $\mathsf{X} \rightarrow \mathsf{BC}$ to generate its two children u_L and u_R. Furthermore, assume that u_L and u_R generated the strings S_L and S_R, respectively. The string generated by v must be a substring of either S_L or S_R. However, CNF has the property that no variable[2] can generate the empty word ϵ. As such $|S_R| > 0$ and $|S_L| > 0$.

In particular, assume without loss of generality, that v is in the left subtree of u, and as such S_v is a substring of S_L. We have that

$$|S_v| \leq |S_L| < |S_L| + |S_R| = |S_u|\,.$$

■

Lemma 2.4 *Let* T *be a tree, and* π *be the longest path in the tree realizing the height* h *of* T*. Fix* $k \geq 0$*, and let* u *be the* k*th node from the end of* π *(i.e.,* u *is in distance* $h - k$ *from the root of* T*). Then the tree rooted at* u *has height at most* k*.*

Proof: Let r be the root of T, and assume, for the sake of contradiction, that T_u (i.e., the subtree rooted at u) has height larger than k, and let σ be the path from u to the leaf γ of T_u realizing this height (i.e., the length of σ is $> k$). Next, consider the path formed by concatenating the path in T from r to u with the path σ. Clearly, this is a new path of length $h - k + |\sigma| > h$ that leads from the root of T into a leaf of T. As such, the height of T is larger than h, which is a contradiction. ■

[2] Except the start variable, but this not relevant here.

Lemma 2.5 (Pumping lemma for Chomsky Normal Form (CNF).) *Let $\mathcal{G}$ be a CNF context-free grammar with m variables in it. Then, given any word $\mathcal{S}$ in $\mathsf{L}(\mathcal{G})$ of length $> 2^m$, one can break $\mathcal{S}$ into 5 substrings $\mathcal{S} = xyzvw$, such that for any $i \geq 0$, we have that xy^izv^iw is a word in $\mathsf{L}(\mathcal{G})$. In addition, the following holds:*

1. *The strings y and v are not both empty (i.e., the pumping is getting us new words).*
2. *$|yzv| \leq 2^m$.*

Proof: Let T be a CNF parse tree for $\mathcal{S}$ (generated by $\mathcal{G}$). Since $\ell = |s| > 2^m$, by Lemma 2.2, there is a path in T from its root to a leaf which has a repeated variable (and its length is longer than m). In fact, let π be the longest path in T from the root to a leaf (i.e., π is the path realizing the height of the tree T). We know that T has more than $m+1$ variables on it and as such it has a repetition.

We need to be a bit careful in picking the two nodes α and β on π to apply the pumping to. In particular, let α be the last node on π such that there is a repeated appearance of the symbol stored in u later in the path. Clearly, the length of the subpath τ of π starting at α till the end of π has at most m symbols on it (because otherwise, there would be another repetition on π). Let β be the node of $\tau \subseteq \pi$ which has repetition of the symbol stored in α.

By Lemma 2.4 the subtree T_α (i.e., the subtree of T rooted at α) has height at most m. As above, T_α and T_β generate two strings $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$, respectively. By Observation 2.1, we have that $|\mathcal{S}_\alpha| \leq 2^m$. By Lemma 2.3, we have that $|\mathcal{S}_\alpha| > |\mathcal{S}_\beta|$. As such, the two substrings $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ breaks $\mathcal{S}$ into 5 substrings $\mathcal{S} = xyzvw$. Here, we have

$$\mathcal{S} = x \;\; \overbrace{y \quad \underbrace{z}_{=\mathcal{S}_\beta} \quad v}^{\mathcal{S}_\alpha=} \;\; w.$$

As such, we know that $|yv| = |\mathcal{S}_\alpha| - |\mathcal{S}_\beta| > 0$. Namely, the strings y and v are not both empty. Furthermore, $|yzv| = |\mathcal{S}_\alpha| \leq 2^m$.

The remaining task is to show the pumping. Indeed, if we replace T_β by the tree T_α we get a parse tree generating the string xy^2zv^2w. If we repeat this process $i-1$ times, we get the word

$$xy^izv^iw \in \mathsf{L}(\mathcal{G}),$$

for any i, establishing the lemma. ■

Lemma 2.6 (Pumping lemma for context-free languages.) *If L is a context-free language, then there is a number p (the pumping length) where, if $\mathcal{S}$ is any string in L of length at least p, then $\mathcal{S}$ may be divided into five pieces $\mathcal{S} = xyzvw$ satisfying the conditions:*

1. *for any $i \geq 0$, we have $xy^izv^iw \in L$,*
2. *$|yv| > 0$,*
3. *and $|yzv| \leq p$.*

Proof: Since L is context free it has a CNF grammar $\mathcal{G}$ that generates it. Now, if m is the number of variables in $\mathcal{G}$, then for $p = 2^m + 1$, the lemma follows by Lemma 2.5. ■

We now prove a corollary to the above lemma, which is easy to show, and which is easier to use to show languages are not context-free.

Lemma 2.7 *If L is a context-free language, then there is a number n such that, for any word $\mathcal{S}$ in L of length at least n, $\mathcal{S}$ can be divided into three words $\mathcal{S} = xtw$, where $|t| \leq n$, and there exists a strict contiguous substring t' of t such that $xt'z \in L$.*

Proof: Choose n to be p, where p is the number assured by the pumping lemma for CFLs. Let $\mathcal{S}$ in L of length at least n. Then by the pumping lemma, there is a split $\mathcal{S} = xyzvw$ such that $|yv| > 0$, $|yzv| \leq p$, and for any $i \geq 0$, $xy^izv^iw \in L$. In particular, for $i = 0$, $xzw \in L$.

Now, choose $t = yzv$ and $t' = z$. Then $\mathcal{S} = xtw$, t' is a strict contiguous substring of t (since $|yv| > 0$), $|t| \leq n$, and $xt'w \in L$. ■

Intuitively, the above lemma says that if L is a CFL, then there is an n such that if w is a word in L of length greater than n, then there is a small substring t of z (of length at most n) that can be *contracted* by replacing t with a smaller string t' that is a contiguous substring of t.

3 Languages that are not context-free

3.1 The language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ is not context-free

Lemma 3.1 *The language $L = \left\{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \,\middle|\, n \geq 0 \right\}$ is not context-free.*

Proof: We give two proofs. The first is based on the pumping lemma, and the second based on the corollary to the pumping lemma.

Proof I: Using the pumping lemma:
Assume, for the sake of contradiction, that L is context-free, and apply the Pumping Lemma to it (Lemma 2.6). As such, there exists $p > 0$ such that any word in L longer than p can be pumped. So, consider the word $\mathcal{S} = \mathtt{a}^{p+1}\mathtt{b}^{p+1}\mathtt{c}^{p+1}$. By the pumping lemma, it can be written as $\mathtt{a}^{p+1}\mathtt{b}^{p+1}\mathtt{c}^{p+1} = xyzvw$, where $|yzv| \leq p$.

We claim, that yzv can made out of only two characters. Indeed, if yzv contained all three characters, it would have to contain the string $\mathtt{b}^{p+1}$ as a substring (as $\mathtt{b}^{p+1}$ separates all the appearances of $\mathtt{a}$ from all the appearances of $\mathtt{c}$ in $\mathcal{S}$). This would require that $|yzv| > p$ but we know that $|yzv| \leq p$.

In particular, let $i_\mathtt{a}, i_\mathtt{b}$ and $i_\mathtt{c}$ be the number of $\mathtt{a}$s, $\mathtt{b}$s and $\mathtt{c}$s in the string yv, respectively. All we know is that $i_\mathtt{a} + i_\mathtt{b} + i_\mathtt{c} = |yv| > 0$ and that $i_\mathtt{a} = 0$ or $i_\mathtt{c} = 0$. Namely, $i_\mathtt{a} \neq i_\mathtt{b}$ or $i_\mathtt{b} \neq i_\mathtt{c}$ (the case $i_\mathtt{a} \neq i_\mathtt{c}$ implies one of these two cases). In particular, by the pumping lemma, the word

$$\mathcal{S}_2 = xy^2zv^2w \quad \in \quad L.$$

We have the following:

character	how many times it appears in $\mathcal{S}_2$
$\mathtt{a}$	$p+1+i_{\mathtt{a}}$
$\mathtt{b}$	$p+1+i_{\mathtt{b}}$
$\mathtt{c}$	$p+1+i_{\mathtt{c}}$

If $i_{\mathtt{a}} \neq i_{\mathtt{b}}$ then $\mathcal{S}_2$, by the above table, does not have the same number of $\mathtt{as}$ and $\mathtt{bs}$ and as such it is not in L.

If $i_{\mathtt{b}} \neq i_{\mathtt{c}}$ then $\mathcal{S}_2$, by the above table, does not have the same number of $\mathtt{bs}$ and $\mathtt{cs}$ and as such it is not in L.

In either case, we get that $\mathcal{S}_2 \notin L$, which is a contradiction. Namely, our assumption that L is context-free is false.

Proof II: Using the corollary to pumping lemma:
Assume, for the sake of contradiction, that L is context-free, and apply the corollary to the Pumping Lemma to it (Lemma 2.7). Then there exists $n > 0$ such that any word in L longer than n can be contracted. Consider the word $\mathcal{S} = \mathtt{a}^{n+1}\mathtt{b}^{n+1}\mathtt{c}^{n+1}$. By the corollary to the pumping lemma, it can be written as $\mathtt{a}^{n+1}\mathtt{b}^{n+1}\mathtt{c}^{n+1} = xtw$, where $|t| \leq n$, and there is a strict contiguous substring t' of t such that $xt'w \in L$.

Since $|t| \leq n$, t cannot contain all of the letters a, b, and c. Hence contracting t to t' will reduce the number of letters of at most two kinds (e.g. a and b, or, b and c, etc.) but not the third. Hence $xt'w$ cannot have the same number of a's, b's, and c's, and hence is not in L, which is a contradiction. Hence our assumption must be wrong, and L cannot be context-free. ■

4 Closure properties

4.1 Context-free languages are not closed under intersection

We know that the languages

$$L_1 = \left\{ \mathtt{a}^*\mathtt{b}^n\mathtt{c}^n \;\middle|\; n \geq 0 \right\} \text{ and } L_2 = \left\{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^* \;\middle|\; n \geq 0 \right\}$$

are context-free (prove this). But

$$L = \left\{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \;\middle|\; n \geq 0 \right\} = L_1 \cap L_2$$

is not context-free by Lemma 3.1. We conclude that the intersection of two context-free languages is not necessarily context-free.

Lemma 4.1 *Context-free languages are not closed under intersection.*

4.2 Context-free languages are closed under union

Lemma 4.2 *Context-free languages over Σ are closed under union.*

Let L_1 and L_2 be two context-free languages, and let $G_1 = (V_1, \Sigma, P_1, S_1)$ and $G_2 = (V_2, \Sigma, P_2, S_2)$ be grammars generating these languages, respectively. Assume that $V_1 \cap V_2 = \emptyset$ (if not, rename the variables so that the sets are disjoint). Then it is easy to see that the grammar $G = (V, \Sigma, P, S)$ where $V = V_1 \cup V_2 \cup \{S\}$ and $P = P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}$ generates $L_1 \cup L_2$.

Proof: If $w \in L_i$ (where $i = 1$ or $i = 2$), then $S_i \Rightarrow^* w$ in the CFG G_i. Then, $w \in L(G)$ as we can derive $S \Rightarrow S_i \Rightarrow^* w$. Conversely, let $w \in L(G)$, and hence $S \Rightarrow^* w$. Since the only rules involving S are $S \rightarrow S_1$ and $S \rightarrow S_2$, the first rule used in the derivation of w must use such a rule. So let $S \Rightarrow S_i \Rightarrow^* w$ (where $i = 1$ or $i = 2$). Since the rules that can be used in the derivation from S_i in G are all from G_i, we can derive $S_i \Rightarrow^* w$ in G_i as well, and hence $w \in L_i$. Hence $L(G) = L(G_1) \cup L(G_2)$.

Lemma 4.3 *Context-free languages are not closed under complement.*

Proof: Since intersection can be written using union and complement operations,
i.e. $L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$, and since CFLs are closed under union, if CFLs were closed under complement, then CFLs will be closed under intersection as well. Since CFLs are not closed under intersection, it follows that they can't be closed under complement.

■

CS 373: Theory of Computation
Madhusudan Parthasarathy

Lecture 24: Recursive automata

22 April 2010

1 Recursive automata

A finite automaton can be seen as a program with only a finite amount of memory. A recursive automaton is like a program which can use *recursion* (calling procedures recursively), but again over a finite amount of memory in its variable space. Note that the recursion, which is typically handled by using a stack, gives a limited form of *infinite* memory to the machine, which it can use to accept certain non-regular languages. It turns out that the recursive definition of a language defined using a context-free grammar precisely corresponds to recursion in a finite-state recursive automaton.

1.1 Formal definition of RAs

A recursive automaton (RA) over Σ is made up of a finite set of NFAs that can call each other (like in a programming language), perhaps recursively, in order to check if a word belongs to a language.

Definition 1.1 A ***recursive automaton*** (RA) over Σ is a tuple

$$\left(M, \textbf{main}, \left\{ \mathtt{D}_M \,\middle|\, m \in M \right\} \right),$$

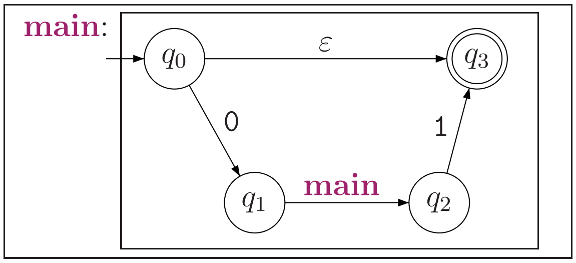
where

- M is a finite set of module names,
- $\textbf{main} \in M$ is the initial module,
- For each $m \in M$, there is an associated automaton $\mathtt{D}_m = (Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)$ which is an NFA over the alphabet $\Sigma \cup M$. In other words, Q_m is a finite set of states, $q_0^m \in Q_m$ is the initial state (of the module m), $F_m \subseteq Q_m$ is the set of final states of the module m (from where the module can return), and $\delta_m : Q_m \times (\Sigma \cup M \cup \{\epsilon\}) \rightarrow 2^{Q_m}$ is the (non-deterministic) transition function.
- For any $m, m' \in M$, $m \neq m'$ we have $Q_m \cap Q_{m'} = \emptyset$ (the set of states of different modules are disjoint).

Intuitively, we view a recursive automaton as a set of procedures/modules, where the execution starts with the **main**-module, and the automaton processes the word by calling modules recursively.

1.1.1 Example of a recursive automata

Let $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ and let $L = \left\{ \mathtt{0}^n \mathtt{1}^n \;\middle|\; n \in \mathbb{N} \right\}$. The language L is accepted by the following recursive automaton.



Why? The recursive automaton consists of single module, which is also the **main** module. The module either accepts ϵ, or reads $\mathtt{0}$, calls itself, and after returning from the call, reads $\mathtt{1}$ and reaches a final state (at which point it can return if it was called). In order to accept, we require the run to return from all calls and reach the final state of the module **main**.

For example, the recursive automaton accepts $\mathtt{01}$ because of the following execution

$$q_0 \xrightarrow{\mathtt{0}} q_1 \xrightarrow{\text{call } \mathbf{main}} q_0 \xrightarrow{\epsilon} q_3 \xrightarrow{\text{return}} q_2 \xrightarrow{\mathtt{1}} q_3.$$

Note that using a transition of the form $q \xrightarrow{\mathtt{m}} q'$ calls the module m; the module m starts with its initial state, will process letters (perhaps recursively calling more modules), and when it reaches a final state will return to the state q' in the calling module.

1.1.2 Formal definition of acceptance

Stack. We first need the concept of a ***stack***. A stack s is a list of elements. The ***top of the stack*** (TOS) is the first element in the list, denoted by topOfStack(() s). Pushing an element x into a stack (i.e., ***push*** operation) s, is equivalent to creating a new list, with the first element being x, and the rest of the list being s. We denote the resulting stack by push(s, x). Similarly, popping the stack s (i.e., ***pop*** operation) is the list created from removing the first element of s. We will denote the resulting stack by pop(s).

We denote the empty stack by $\langle\rangle$. A stack containing the elements x, y, z (in this order) is written as $s = \langle x, y, z \rangle$. Here topOfStack$(s) = x$, pop$(s) = \langle y, z \rangle$ and push$(s, b) = \langle b, x, y, z \rangle$.

Acceptance. Formally, let $\mathtt{C} = \left(M, \mathbf{main}, \left\{ (Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m) \;\middle|\; m \in M \right\} \right)$ be a recursive automaton.

We define a run of $\mathtt{C}$ on a word w. Since the modules can call each other recursively, we define the run using a stack. When $\mathtt{C}$ is in state q and calls a module m using the transition $q \overset{*}{\Longrightarrow} mq'$, we push q' onto the stack so that we know where to go to when we return from the call. When we return from a call, we pop the stack and go to the state stored on the top of the stack.

Formally, let $Q = \bigcup_{m \in M} Q_m$ be the set of all states in the automaton $\mathtt{C}$. A ***configuration*** of $\mathtt{C}$ is a pair (q, s) where $q \in Q$ and s is a stack.

We say that a word w is ***accepted*** by $\mathtt{C}$ provided we can write $w = y_1 \ldots y_k$, such that each $y_i \in \Sigma \cup \{\epsilon\}$, and there is a sequence of $k + 1$ configurations $(q_0, s_0), \ldots (q_k, s_k)$, such that

- $q_0 = q_0^{\textbf{main}}$ and $s_0 = \langle\rangle$.

  We start with the initial state of the main module with the stack being empty.

- $q_k \in F_{\textbf{main}}$ and $s_k = \langle\rangle$.

  We end with a final state of the main module with the stack being empty (i.e. we expect all calls to have returned).

- For every $i < k$, one of the following must hold:

  Internal: $q_i \in Q_m$, $q_{i+1} \in \delta_m(q_i, y_{i+1})$, and $s_{i+1} = s_i$.

  Call: $q_i \in Q_m$, $y_{i+1} = \epsilon$, $q' \in \delta_m(q_i, m')$, $q_{i+1} = q_0^{m'}$ and $s_{i+1} = \text{push}(s_i, q')$.

  Return: $q_i \in F_m$, $y_{i+1} = \epsilon$, $q_{i+1} = \text{topOfStack}(s_i)$ and $s_{i+1} = \text{pop}(s_i)$.

2 CFGs and recursive automata

We will now show that context-free grammars and recursive automata accept precisely the same class of languages.

2.1 Converting a CFG into a recursive automata

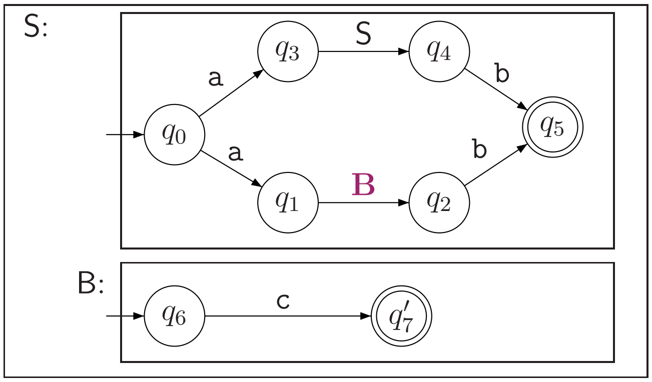
Given a CFG, we want to construct a recursive automaton for the language generated by the CFG. Let us first do this for an example.

Consider the grammar (where S is the start variable) which generates $\left\{ \mathtt{a}^n \mathtt{c} \mathtt{b}^n \;\middle|\; n \in \mathbb{N} \right\}$:

$$\begin{aligned} \Longrightarrow \mathsf{S} &\rightarrow \mathtt{aSb} \mid \mathtt{aBb} \\ \mathsf{B} &\rightarrow \mathtt{c}. \end{aligned}$$

Each variable in the CFG corresponds to a language; this language is recursively defined using other variables. We hence look upon each variable as a module; and define modules that accept words by calling other modules recursively.

For example, the recursive automaton for the above grammar is:



(Here S is the main modules of the recursive automaton.)

Formal construction. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ be the given context free grammar.

Let $\mathtt{D}_{\mathcal{G}} = \left(M, \mathsf{S}, \left\{(Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m) \,\middle|\, m \in M\right\}\right)$ where $M = \mathcal{V}$, and the main module is S. Furthermore, for each $\mathsf{X} \in M$, let $\mathtt{D}_{\mathsf{X}} = \left(Q_{\mathsf{X}}, \Sigma \cup M, \delta_{\mathsf{X}}, q_0^{\mathsf{X}}, F_{\mathsf{X}}\right)$ be an NFA that accepts the (finite, and hence) regular language $L_{\mathsf{X}} = \left\{w \,\middle|\, (\mathsf{X} \to w) \in \mathcal{R}\right\}$.

Let us elaborate on the construction of $\mathtt{D}_{\mathsf{X}}$. We create two special states $q_{\text{init}}^{\mathsf{X}}$ and $q_{\text{final}}^{\mathsf{X}}$. Here $q_{\text{init}}^{\mathsf{X}}$ is the initial state of $\mathtt{D}_{\mathsf{X}}$ and $q_{\text{final}}^{\mathsf{X}}$ is the accepting state of $\mathtt{D}_{\mathsf{X}}$. Now, consider a rule $(\mathsf{X} \to w) \in \mathcal{R}$. We will introduce a path of length $|w|$ in $\mathtt{D}_{\mathsf{X}}$ (corresponding to w) leading from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$. Creating this path requires introducing new "dummy" states in the middle of the path, if $|w| > 1$. The ith transition along this path reads the ith character of w. Naturally, if this ith character is a variable, then this edge would correspond to a recursive call to the corresponding module. As such, if the variable X has k rules in the grammar $\mathcal{G}$, then $\mathtt{D}_{\mathsf{X}}$ would contain k disjoint paths from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$, corresponding to each such rule. For example, if we have the derivation $(\mathsf{X} \to \epsilon) \in \mathcal{R}$, then we have an ϵ-transition from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$.

2.2 Converting a recursive automata into a CFG

Let $\mathtt{C} = (M, \textbf{main}, \{(Q_m, \Sigma \cup M, \delta_m, q_{\text{init}}^m, F_m)\}_{m \in M})$ be a recursive automaton. We construct a CFG $\mathcal{G}_{\mathtt{C}} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ with $\mathcal{V} = \left\{\mathsf{X}_q \mid q \in \bigcup_{m \in M} Q_m\right\}$.

Intuitively, the variable X_q will represent the set of all words accepted by starting in state q and ending in a final state of the module q is in (however, on recursive calls to this module, we still enter at the original initial state of the module).

The set of rules R is generated as follows.

- **Regular transitions.** For any $m \in M$, $q, q' \in Q_m$, $c \in \Sigma \cup \{\epsilon\}$, if $q' \in \delta_m(q, c)$, then the rule $\mathsf{X}_q \to c\mathsf{X}_{q'}$ is added to $\mathcal{R}$.

  Intuitively, a transition within a module is simulated by generating the letter on the transition and generating a variable that stands for the language generated from the next state.

- **Recursive call transitions.** for all $m, m' \in M$ and $q, q' \in Q_m$, if $q' \in \delta_m(q, m')$, then the rule $\mathsf{X}_q \to \mathsf{X}_{q_{\text{init}}^{m'}} \mathsf{X}_{q'}$ is in R,

  Intuitively, if $q' \in \delta_m(q, m')$, then X_q can generate a word of the form xy where x is accepted using a call to module m and y is accepted from the state q'.

- **Acceptance/return rules.**

  For any $q \in \bigcup_{m \in M} F_m$, we add $\mathsf{X}_q \to \epsilon$ to $\mathcal{R}$.

  When arriving at a final state, we can stop generating letters and return from the recursive call.

The initial variable S is $\mathsf{X}_{q_{\text{init}}^{\textbf{main}}}$; that is, the variable corresponding to the initial state of the main module.

We have a CFG and it is not too hard to see intuitively that the language generated by this grammar is equal to the RA C language. We will not prove it formally here, but we state the result for the sake of completeness.

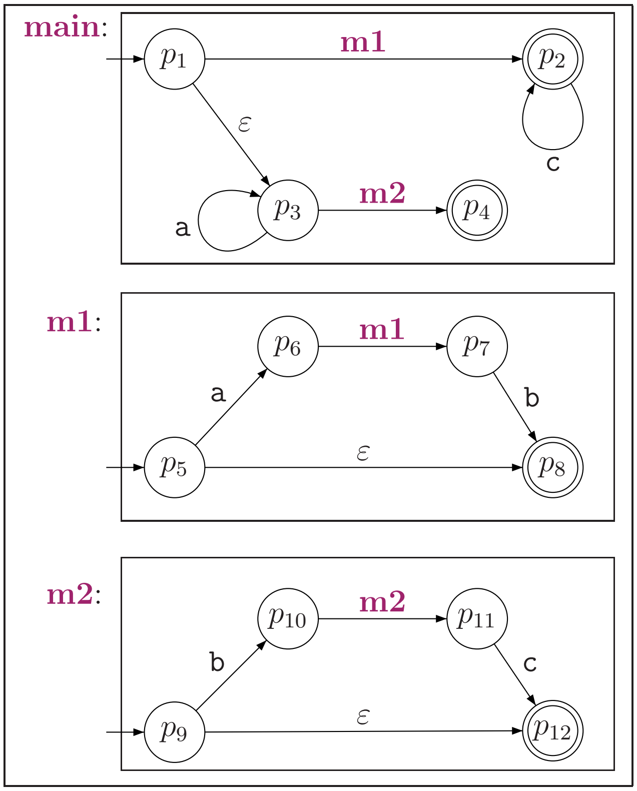
Lemma 2.1 $\mathsf{L}(\mathcal{G}_{\mathsf{C}}) = \mathsf{L}(\mathsf{C})$.

2.2.1 An example of conversion of a RA into a CFG

Consider the following recursive automaton, which accepts the language

$$\left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \;\middle|\; i = j \textit{ or } j = k \right\},$$

and the grammar generating it.



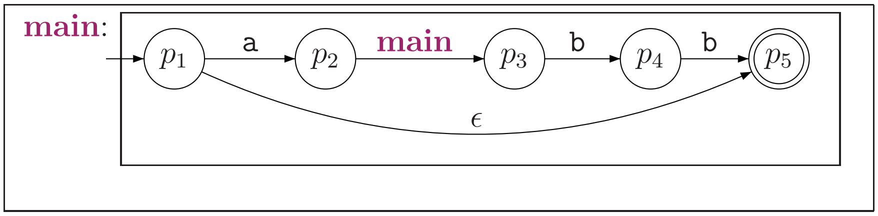
$$\begin{aligned}
\mathsf{X}_{p1} &\rightarrow \mathsf{X}_{p5}\mathsf{X}_{p2} \mid \mathsf{X}_{p3}\\
\mathsf{X}_{p2} &\rightarrow \mathtt{c}\mathsf{X}_{p2} \mid \epsilon\\
\mathsf{X}_{p3} &\rightarrow \mathtt{a}X_{p3} \mid \mathsf{X}_{p9}X_{p4}\\
\mathsf{X}_{p4} &\rightarrow \epsilon\\
\mathsf{X}_{p5} &\rightarrow \mathtt{a}\mathsf{X}_{p6} \mid \mathsf{X}_{p8}\\
\mathsf{X}_{p6} &\rightarrow \mathsf{X}_{p5}\mathsf{X}_{p7}\\
\mathsf{X}_{p7} &\rightarrow \mathtt{b}X_{p8}\\
\mathsf{X}_{p8} &\rightarrow \epsilon\\
\mathsf{X}_{p9} &\rightarrow \mathtt{b}\mathsf{X}_{p10} \mid \mathsf{X}_{p12}\\
\mathsf{X}_{p10} &\rightarrow \mathsf{X}_{p9}\mathsf{X}_{p11}\\
\mathsf{X}_{p11} &\rightarrow \mathtt{c}X_{p12}\\
\mathsf{X}_{p12} &\rightarrow \epsilon
\end{aligned}$$

The start variable is X_{p1}.

3 More examples

3.1 Example 1: RA for the language $\mathtt{a}^n\mathtt{b}^{2n}$

Let us design a recursive automaton for the language $L = \left\{ \mathtt{a}^n\mathtt{b}^{2n} \;\middle|\; n \in \mathbb{N} \right\}$. We would like to generate this recursively. How do we generate $\mathtt{a}^{n+1}\mathtt{b}^{2n+2}$ using a procedure to generate $\mathtt{a}^n\mathtt{b}^{2n}$? We read $\mathtt{a}$ followed by a call to generate $\mathtt{a}^n\mathtt{b}^{2n}$, and follow that by generating two $\mathtt{b}$'s. The "base-case" of this recursion is when $n = 0$, when we must accept ϵ. This leads us to the following automaton:

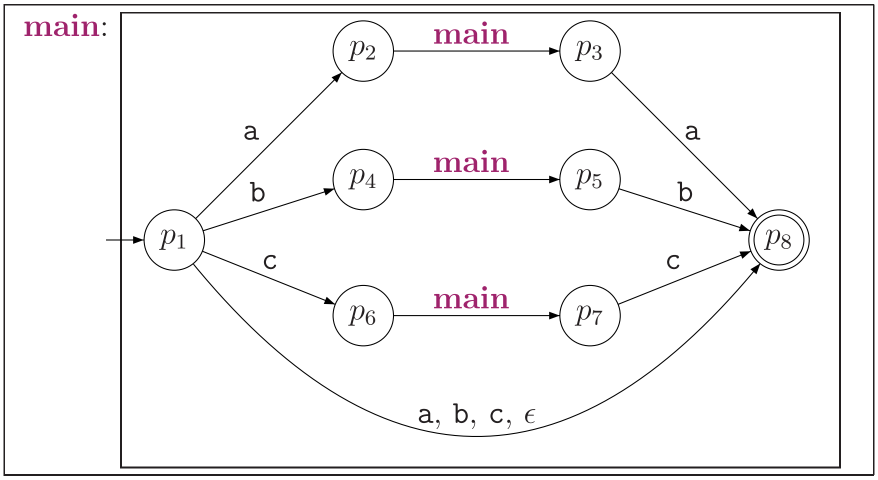


3.2 Example 2: Palindrome

Let us design a recursive automaton for the language

$$L = \left\{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \;\middle|\; w \text{ is a palindrome} \right\}.$$

Thinking recursively, the smallest palindromes are ϵ, a, b, c, and we can construct a longer palindrome by generating awa, bwb, cwc, where w is a smaller palindrome. This give us the following recursive automaton:



3.3 Example 3: $\#_a = \#_b$

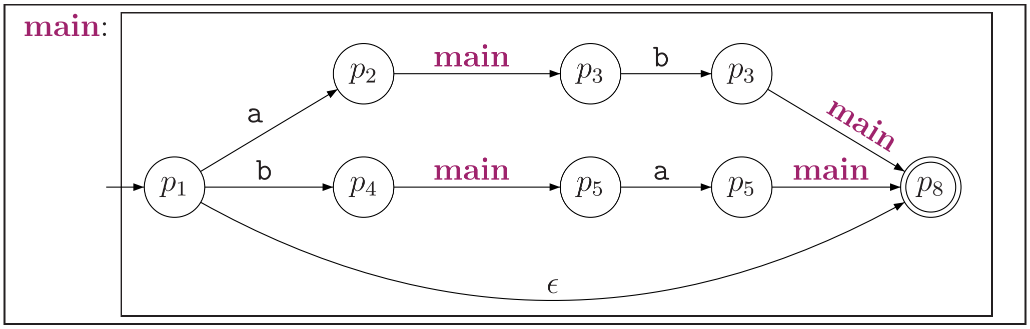
Let us design a recursive automaton for the language L containing all strings $w \in \{\mathtt{a}, \mathtt{b}\}^*$ that has an equal number of a's and b's.

Let w be a string, of length at least one, with equal number of a's and b's.

Case 1: w starts with a. As we read longer and longer prefixes of w, we have the number of a's seen is more than the number of b's seen. This situation can continue, but we must reach a place when the number of a's seen is precisely the number of b's seen (at worst at the end of the word). Let us consider some prefix longer than a where this happens. Then we have that $w = \mathtt{a}w_1\mathtt{b}w_2$, where the number of a's and b's in $\mathtt{a}w_1\mathtt{b}$ is the same, i.e. the number of a's and b's in w_1 are the same. Hence the number of a's and b's in w_2 are also the same.

Case 2: If w starts with b, then by a similar argument as above, $w = \mathtt{b}w_1\mathtt{a}w_2$ for some (smaller) words w_1 and w_2 in L.

Hence any word w in L of length at least one is of the form $\mathtt{a}w_1\mathtt{b}w_2$ or $\mathtt{b}w_1\mathtt{a}w_2$, where $w_1, w_2 \in L$, and they are strictly shorter than w. Also, note ϵ is in L. So this gives us the following recursive automaton.



4 Recursive automata and pushdown automata

The definition of acceptance of a word by a recursive automaton employs a ***stack***, where the target state gets pushed on a call-transition, and gets popped when the called module returns. An alternate way (and classical) way of defining automata models for context-free languages directly uses a stack. A ***pushdown automaton*** (PDA) is a non-deterministic automaton with a finite set of control states, and where transitions are allowed to push and pop letters from a finite alphabet Γ (Γ is fixed, of course) onto the stack. It should be clear that a recursive automaton can be easily simulated by a pushdown automaton (we simply take the union of all states of the recursive automaton, and replace call transitions $q \xrightarrow{\mathtt{m}} q'$ with an explicit push-transition that pushes q' onto the stack and explicit pop transitions from the final states in F_m to q' on popping q'.

It turns out that pushdown automata can be converted to recursive automata (and hence to CFGs) as well. This is a fact worth knowing! But we will not define pushdown automata formally, nor show this direction of the proof.