~ :
~

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) $\{\texttt{0}, \texttt{1}\}$.

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

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

(iv) $\{\texttt{a}, ...\texttt{z}, \texttt{A}, ...\texttt{Z}\}$: 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 $\textvisiblespace$ is 32.

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

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) $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ 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 $w = \texttt{abcdef}$ 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 $x = \texttt{cat}$, $w = \texttt{nip}$ and the concatenated strings $xw = \texttt{catnip}$ and $wx = \texttt{nipcat}$.

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

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

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

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

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

Next, consider the following set

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

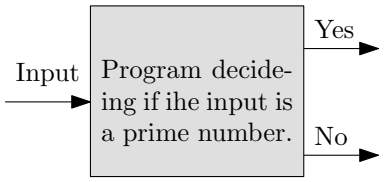
So L_2 is the language made out of all prefixes of L_2. We can write L_2 explicitly, but its tedious. Indeed,

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

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

$$abbaabaabbbabababbbabababbabbbabab \ldots\ldots\ldots$$

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.