CS/Math 240: Introduction to Discrete Mathematics 4/19/2011

Lecture 22 : Regular Expressions

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Last time we discussed a simple computational model called a finite state machine. The machine works on some input which it processes one symbol at a time. It can be in one of a finite number of states. After reading each input symbol, the machine can change states, and produces an output.

A special kind of a finite state machine is a finite state automaton which only outputs one bit after it finishes reading the entire input. We can think about this bit as meaning either "yes" or "no", or either "accept" or "reject".

We designed some finite state machines and automata last time. Today we design a family of finite state automata that recognize binary representations of multiples of integers, and then discuss regular expressions and their connection to finite state automata.

22.1 More Finite Automata

At the end of last lecture, we mentioned a class of languages

$$L_k = \{\text{binary representations of multiples of } k\}, \quad k \geq 2.$$

We design a finite state automaton N_k that accepts all binary representations of integers that are multiples of an integer k, where $k \geq 2$.

22.1.1 Notation for Today

We discussed binary representations of integers in Lecture 9. Today, we deviate from the notation used for representing integers in that lecture. In particular, we index the bits in the representation from left to right, i.e., we view the number as a string $x = x_1 x_2 \ldots x_n$ instead of $x_n x_{n-1} \ldots x_1 x_0$. Thus, unlike earlier, the most significant bit has the lowest index, which is 1 instead of 0, and the least significant bit has the highest index, n. With this notation, the numerical value of x is $\text{Val}(x) = \sum_{i=1}^{n} x_i 2^{n-i}$.

22.1.2 Designing the Automaton

There should be no leading zeros in the binary representation of a number, so the most significant bit should be one. The only exception to this is the number zero whose binary representation is 0. This makes designing the automaton for L_k a little more complicated.

As an initial attempt, let's allow leading zeros, and design an automaton N'_k that accepts binary representations of multiples of k that may have leading zeros. For example, 010 is the binary representation of 2 with one leading zero, so it's not in $L(N_2)$, but it is in $L(N'_2)$.

Later today, we will use N'_k to obtain N_k.

The states represent information N'_k has about the input. Since N_k is a finite state automaton, it cannot keep track of the entire input because the input could be arbitrarily large. Therefore, we

need to find some finite amount of information that is sufficient for the automaton to be able to decide whether a number is a multiple of k or not.

An integer is a multiple of k if and only if the remainder after dividing by k is zero. In other words,

$$\begin{aligned} x \in L(N'_k) &\iff k \mid \mathrm{Val}(x) \\ &\iff \mathrm{Val}(x) \equiv_k 0 \\ &\iff \mathrm{Val}(x) \bmod k = 0, \end{aligned}$$

where the notation $a \bmod b = c$ means that the remainder of a after division by b is c.

As we shall see, knowing the remainder of $\mathrm{Val}(x)$ after dividing by k is sufficient information. Our machine N'_k will have one state for each possible value of the remainder. Since k is a constant, this is a constant number of states.

We need to ensure that N'_k can maintain the information about the remainder as it goes through the input, symbol by symbol. Suppose the input is $x = x_1x_2 \dots x_N$ for some N. Say that N'_k has read $n < N$ bits so far, and knows the remainder of $x_1x_2 \dots x_n$ after dividing by k. This is a fair assumption because N'_k must be able to tell whether $x_1x_2 \dots x_n$ is a multiple of k or not.

The next bit in the input is x_{n+1}. Note that

$$\begin{aligned} \mathrm{Val}(x_1x_2 \dots x_nx_{n+1}) &= \sum_{i=1}^{n+1} x_i 2^{n+1-i} \\ &= \sum_{i=1}^{n} x_i 2^{n+1-i} + x_{n+1} \\ &= 2 \sum_{i=1}^{n} x_i 2^{n-i} + x_{n+1} \\ &= 2\mathrm{Val}(x_1x_2 \dots x_n) + x_{n+1}. \end{aligned} \tag{22.1}$$

For example, $\mathrm{Val}(010) = 2$, and $\mathrm{Val}(0101) = 2\mathrm{Val}(010) + 1 = 2 \cdot 2 + 1 = 5$.

First let's take both sides of (22.1) modulo k. The remainders of both sides after dividing by k have to be the same, so we have

$$\mathrm{Val}(x_1x_2 \dots x_nx_{n+1}) \bmod k = \big[2\mathrm{Val}(x_1x_2 \dots x_n) + x_{n+1}\big] \bmod k \tag{22.2}$$

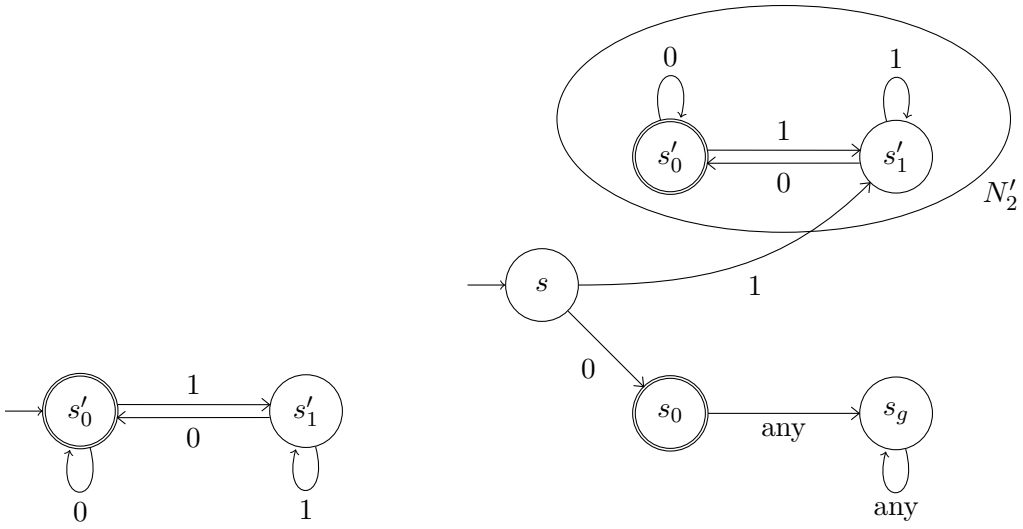
We remark that we can rewrite (22.2) as

$$\mathrm{Val}(x_1x_2 \dots x_nx_{n+1}) \bmod k = \big[2\big(\mathrm{Val}(x_1x_2 \dots x_n) \bmod k\big) + a\big] \bmod k.$$

We leave the proof as an exercise.

Let s'_i be a state indicating that the first n bits of he input x satisfy $\mathrm{Val}(x_1x_2 \dots x_n) = i \bmod k$. Equation (22.2) tells us what the transition function should be. We have $\nu'(s'_i, a) = s'_{(2i+a) \bmod k}$ for $0 \le i < k$ and $a \in \{0, 1\}$. We view the empty string as representing an integer whose remainder after dividing by k is zero. This makes sense because the remainder after dividing the first bit, x_1, by k is either 0 or 1. Thus, the start state is s'_0. The state s'_0 is also the only accepting state. This completes the description of the automaton. We show it in Figure 22.1a.

Note that N'_k also accepts the empty string, which is not a representation of any number. But that is not a problem because we'll never be in that situation when we make N'_k part of N_k.



(a) The automaton N_2' that allows leading zeros.

(b) The automaton N_2 that does not allow leading zeros. Note that this automaton contains a copy of N_2'.

Figure 22.1: Constructing the automaton N_2.

Now the question is what we can do to make N_k reject any string with leading zeros. We need to keep track of some additional information, namely the first symbol. Because zero is a multiple of k, the machine should accept if the first symbol is zero, but only if this first zero is also the last symbol in the input. Any other string that starts with a zero has at least one leading zero. Therefore, N_k should go to a reject state after reading another symbol after the leading zero, and should never leave that state after that.

Create a start state s which is rejecting. On input 0, N_k goes to the accepting state s_0 that indicates there is a leading zero. If an additional symbol is read when N_k is in state s_0, N_k goes to some garbage state s_g, which it never leaves. On input 1 in the start state, N_k goes to a copy of the machine N_k' and runs it on the rest of the input. Note it enters the machine in state s_1' because the remainder after reading the first bit of the input is 1. We show the machine for $k = 2$ in Figure 22.1b. As promised earlier, N_k does not accept the empty string because the start state is rejecting.

Let's reiterate one more time that the key to designing finite automata is answering the following question: What is the information about the string read so far that is necessary for the automaton to continue its computation correctly?" Furthermore, the amount of information should be finite.

22.2 Regular Expressions

There is a connection between regular expressions and finite automata. We will show that a language is decidable by a finite automaton if and only if it can be characterized by a regular expression.

22.2.1 Regular Operators on Languages

Before we define regular expressions, we need operators that work on languages. These operators take one or more languages, and produce a new language.

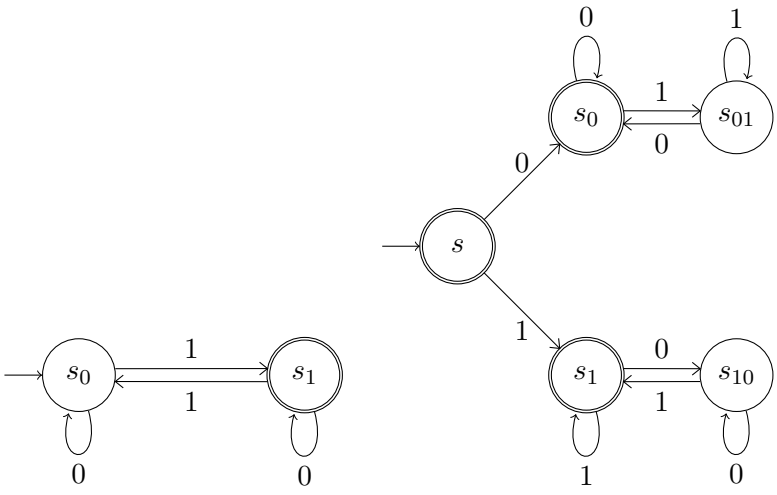
First note that languages are sets, so taking the *union* of two languages makes sense. If L_1 and L_2 are languages, $x \in L_1 \cup L_2$ if and only if $x \in L_1$ or $x \in L_2$.

The next operator is *concatenation*. The concatenation of strings x and y is obtained by writing down x followed by y right after it. To get a concatenation of two languages L_1 and L_2, we consider all pairs of strings, one from each L_1 and L_2, and concatenate them.

Definition 22.1. *Let L_1 and L_2 be languages. The* concatenation *of L_1 and L_2 is the set*

$$L_1 L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}.$$

We give some examples of concatenations of languages. We use the automata from Lecture 21. For completeness, we show them in Figure 22.2.



(a) The machine M_1. (b) The machine M_2.

Figure 22.2: Some finite state automata we designed in Lecture 21.

Example 22.1: Consider the concatenation $L(M_1)L(M_1)$ (where M_1 is the automaton from earlier that accepts strings with an odd number of ones). Concatenating a string with an odd number of ones with another string with an odd number of ones produces a string with an even number of ones. Also note that the number of ones in the result of the concatenation is at least two. Thus, all strings in $L(M_1)L(M_1)$ contain a positive even number of ones. In fact, this language contains all strings with a positive even number of ones. We leave the proof that every string with a positive even number of ones is in $L(M_1)L(M_1)$ to you as an exercise. You have to show that it's possible to decompose a string z with an even number of ones into two strings with an odd number of ones whose concatenation is z. ⊠

Example 22.2: Now consider $L(M_2)L(M_2)$. This is the set of all binary strings. Any language over the alphabet $\{0, 1\}$ is a subset of the set of all binary strings. For the other containment, we have to argue it's possible to write any binary string z as xy where $x, y \in L(M_2)$. For example, if z starts and ends with the same symbol, we can pick $x = z$ and $y = \epsilon$. We leave the proof to you as an exercise. ⊠

The two examples we gave both concatenate a language with itself. It is possible to concatenate two different languages as well. For example, we could consider the concatenation $L_1 L_2$, but it may be harder to figure out what the resulting language is.

The last regular operator is called *Kleene closure* (Kleene was actually a faculty member at UW-Madison) or *star* (because of the notation). We define

$$L^* = \bigcup_{k=0}^{\infty} L^k$$

where k is L concatenated with itself k times, i.e., $L^1 = L$, $L^2 = LL$, and so on. We can also define L^k inductively as $L^k = LL^{k-1}$ with the base case $L^0 = \{\epsilon\}$.

Example 22.3: The set $\{0,1\}^2$ is the set of all binary strings of length 2, i.e., $\{00, 01, 10, 11\}$. In general, $\{0,1\}^k$ is the set of all binary strings of length k. Taking the union of $\{0,1\}^k$ over all k gives us the set of all binary strings, $\{0,1\}^*$. ⊠

Example 22.4: Now let's find what $L(M_2)^*$ is. By definition, $L(M_2)^* = \bigcup_{k=0}^{\infty} L(M_2)^k$. Recall from Example 22.2 that $L(M_2)^2 = \{0,1\}^*$. Because the union $\bigcup_{k=0}^{\infty} L(M_2)^k$ contains $L(M_2)^2$, we have $\{0,1\}^* \subseteq L(M_2)^*$. But any language is a subset of $\{0,1\}^*$, so $L(M_2)^* = \{0,1\}^*$. ⊠

Example 22.5: What about $L(M_1)^*$? First, $L(M_1)^0 = \{\epsilon\}$. Now $L(M_1)^1$ is the set of strings with an odd number of ones, and $L(M_1)^2$ is the set of strings with a positive even number of ones by Example 22.1. The only strings from $\{0,1\}^*$ that are missing from $L(M_1)^0 \cup L(M_1)^1 \cup L(M_1)^2$ are the strings that have no ones in them and are not empty, i.e., all string consisting only of zeros.

Can $L(M_1)^k$ for some k contain a string consisting of only zeros? A string in $L(M_1)^k$ has at least k ones because it is a concatenation of k strings in $L(M_1)$, and each string in $L(M_1)$ contains at least one 1. Hence, $L(M_1)^* = (\{0,1\}^* - \{0\}^*) \cup \{\epsilon\}$. (Note we have to add ϵ back to the language using union because set difference eliminates it.) ⊠

22.2.2 A Formal Definition of a Regular Expression

Regular operators are the building blocks used to construct regular expressions out of a few small base cases.

Definition 22.2. *A* regular expression *over an alphabet Σ is any of the following:*

- $\emptyset$ *(the empty regular expression)*
- ϵ
- a *(for any $a \in \Sigma$)*

Furthermore, if R_1 and R_2 are regular expressions over Σ, $R_1 \cup R_2$, R_1R_2, and R_1^* are also regular expressions over Σ.

The constructor rules say that regular expressions are *closed* under regular operators. In general, a set S is closed under a set of operators if applying any of the operators to any elements of S produces another element of that set S.

With each regular expression R, we associate a language $L(R)$. For each part of Definition 22.2, we show what the corresponding language is in Table 22.1. Table 22.1 may look like syntactic sugar, but there is a difference in the meanings of the two columns.

We remark that the notation for regular expressions isn't entirely standard. For example, sometimes you will see + instead of $\cup$ for union, and you will often see $\cdot$ for concatenation. If we think of Kleene closure as taking powers, we get the natural precedence rules: To see what a language is, first take all Kleene closures, then evaluate all concatenations, and finally construct unions at the very end. To change these precedence rules, use parentheses.

R	$L(R)$
$\emptyset$	$\emptyset$
ϵ	$\{\epsilon\}$
a	$\{a\}$
$R_1 \cup R_2$	$L(R_1) \cup L(R_2)$
$R_1 R_2$	$L(R_1)L(R_2)$
R_1^*	$L(R_1)^*$

Table 22.1: Languages corresponding to regular expressions.