CS/Math 240: Introduction to Discrete Mathematics Fall 2015

Reading 13 : Finite State Automata and Regular Expressions

Instructors: Beck Hasti, Gautam Prakriya

In this reading we study a mathematical model of computation called finite state automata.

13.1 Finite State Automata

A *finite state automaton* consists of a *finite control*, which we view as a set of *states* that the automata/ machine can be in. The machine runs in steps. In each step, it receives an input signal. Upon its receipt, the machine changes its state according to some rules.

One way of describing the functionality of a finite state machine is to draw a graph. Each state of the machine corresponds to one vertex of the graph. We draw an arrow pointing to the state in which the machine starts after being "powered on". We use edges to describe the *transitions* between states. For each transition, we list the inputs that cause it. For example, in Figure 13.1, there is an edge from state 5 to state 15. The edge is labeled 10, which means that this transition occurs when the machine receives the input 10 in state 5.

Example 13.1: Consider a vending machine. For simplicity, suppose it only sells one item, priced at 20 cents, and doesn't give any change. It accepts 5-cent and 10-cent coins.

The states of the machine represent how much money has been put in since the machine first started, or since the last item disposal, whichever came last. The machine reaches state ≥ 20, when at least 20 cents has been put into the machine. We represent this state with 2 circles, to indicate that once it is reached an item is dispensed.

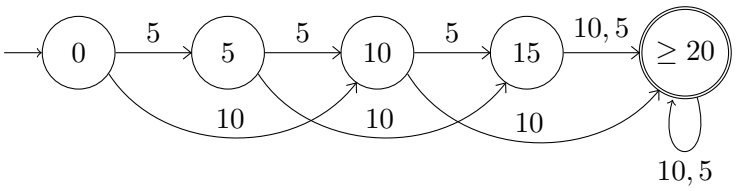


Figure 13.1: The transition diagram for the vending machine.

⊠

Before we define a finite state automaton more formally, let's see another example.

Example 13.2: Consider a machine that receives one of the letters A through Z in each step. The machine determines if the word NANO appears in the sequence of letters given to it.

We design the machine so that its states indicate how much of the word NANO it has recently seen. We label the states 0 through 4. State i indicates that the last i symbols received were the first i symbols from the word NANO.

If the machine is in state i for $i \in \{0, 1, 2, 3\}$, it goes to state $i + 1$ if the next letter given to it as input is the $(i + 1)$st letter of the word NANO. The machine enters and stays in state 4 if it sees the word NANO.

What if the machine receives an input that does not continue spelling the word NANO? Then the machine must go back to an earlier state, but not necessarily the starting state. For example, if the last three inputs received were NAN, the machine is in state 3. If the next input is A, the last

two letters received are also the first two letters of the word NANO, so the machine goes to state 2. Going to any other state could cause the machine to produce incorrect output. For example, if the next two letters after A were NO, the machine would not recognize that the word NANO appeared in the input if it went to state 0 instead of state 2 after seeing A.

Finally, if the machine is in state 4 after reading its input, then the input contains the word NANO. If the machine ends up at any other state after reading its input, then the input doesn't contain the word NANO. The state 4 is called an accept state. If the machine ends up at this state after reading the entire input, it accepts the input. If it ends up at any other state it rejects the input. We show all the transitions in Figure 13.2.

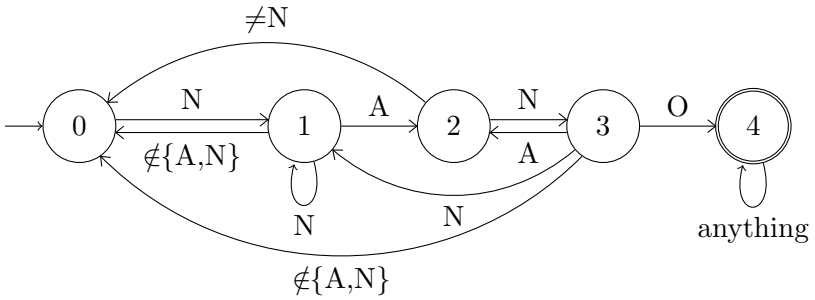


Figure 13.2: A finite state automaton that decides whether a sequence of inputs contains the word NANO in it.

⊠

We are now give a formal definition of a finite state automaton.

13.1.1 Formal Definition

A finite state automaton receives inputs from some finite set of symbols. We call this finite set an *alphabet*. The alphabet for Example 13.1 was the set $\{5, 10, 25\}$, and the alphabet for Example 13.2 was the set $\{\text{A},\ldots,\text{Z}\}$.

Definition 13.1. *A* finite state automaton *is a* 5*-tuple* $(S, \Sigma, \delta, s_0, A)$*, where*

- S *is a finite set of* states*.*
- Σ *is a finite set of symbols called the* alphabet*.*
- $\delta : S \times \Sigma \to S$ *is the* transition function*. The inputs to this function are the current state and the last input symbol. The function value* $\delta(s, x)$ *is the state the automaton goes to from state* s *after reading symbol* x*.*
- *The automaton starts in the* start state $s_0 \in S$*.*
- *The set of* accepting states A *indicates which states cause the automaton to output "yes". That is if the automaton ends up at a state in* A *after reading its input, then it accepts the input. If the automaton ends up at a state that is not in* A *after reading its input, it rejects the input.*

Example 13.3: We show what the five parts of the tuple $M_1 = (S, \Sigma, \delta, s_0, A)$ are for the finite state automaton in Figure 13.3.

- $S = \{s_0, s_1\}$.
- $\Sigma = \{0, 1\}$.
- $\delta(s_0, 0) = s_0$, $\delta(s_0, 1) = s_1$, $\delta(s_1, 0) = s_1$, $\delta(s_1, 1) = s_0$.
- The start state is s_0.
- $A = \{s_1\}$.

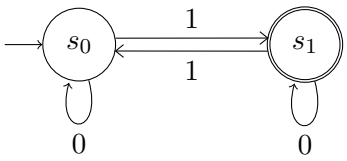


Figure 13.3: The automaton M_1 used in Example 13.3.

⊠

13.1.2 Strings and Languages

In each step, a finite state automaton processes some symbol from the alphabet. The input sequence could be infinite, thus causing the finite state automaton to run forever. For most computer programs, this is a situation we want to avoid, so let's focus on the case where the input sequences have finite length.

Definition 13.2. *A finite sequence of symbols from an alphabet Σ is called a* string *over Σ. An empty sequence of symbols is called an* empty string*, and is usually denoted by λ or ϵ. If x is a string over Σ, $|x|$ denotes the* length *of the string, that is, the length of the sequence of symbols x stands for.*

For example, the length of the empty string is $|\epsilon| = 0$. The length of the string "NANO" from Example 13.2 is 4.

We can further group strings into sets called languages.

Definition 13.3. *A* language *over alphabet Σ is a set of strings over Σ.*

Suppose you have a program in some text file. You can view this program as a string. Some strings represent valid programs, and some do not. It is the job of the compiler to distinguish between the former and the latter. For the former, it should say the program is valid, and for the latter, it should generate some error message. The set of all strings that represent valid programs is called a programming language. This is what motivates the definition above.

With every automaton, we associate the language of all strings accepted by the automaton.

Definition 13.4. *Given a finite automaton M, the set*

$$L(M) = \{x \mid \text{when } M \text{ is run on the string } x \text{ starting from the start state, the final state is in } A\}$$

is called the language decided by *M.*

Using this definition, we could say that the language decided by the compiler is the language of valid programs in some programming language. However, we need a stronger computational model than a finite state automaton to implement a compiler.

Let's now look at a language that can be decided by a finite state automaton.

Example 13.4: Let's find the language decided by the finite state automaton M_1 from Example 13.3. We start by finding some short strings in $L(M_1)$.

The empty string ϵ is not in the language because M_1 on input ϵ enters the state s_0, and terminates immediately because there are no more input symbols to read. Thus, M_1 ends in a non-accepting state on this input.

The string 0 is also not in the language. The machine M_1 starts in state s_0, and $\delta(s_0, 0) = s_0$, so M_1 stays in s_0 after reading the first and only input symbol. It outputs "no" after that.

The string 1 is in the language because the transition M_1 makes from the start state after reading 1 is to state s_1, which is an accepting state. It outputs "yes" after that.

The strings 01 and 10 are in the language. The sequences of states after reading the input symbols one by one are s_0, s_0, s_1 and s_0, s_1, s_1, respectively. The strings 00 and 11 are not in the language.

The strings $001, 010, 100, 111$ are all in $L(M_1)$, and no other strings of length 3 are.

We observe that M_1 changes states if and only if the next input symbol is 1. Therefore, the first time it gets to the accepting state s_1 is after seeing the first 1. It leaves this state after seeing the next 1, comes back after seeing another 1 after that, and so on. In other words, M_1 changes states if and only if it reads a 1. Because the start state is $s_0 \neq s_1$, M_1 ends in state s_1 if and only if it changes states an odd number of times, that is, if and only if the input contains an odd number of ones.

We now turn the intuitive explanation from the previous paragraph in a formal proof. We can think of it as proving the equality of two sets. One set is $L(M_1)$, and the other set is the set of strings with an odd number of ones in them. We do not pursue this direction, and use invariants instead.

First we introduce some notation that extends the transition function δ. Let x be a string over Σ. Then $\delta(s, x)$ is the state of the automaton after reading all symbols in x, assuming the automaton was in state s before processing the first symbol in x. Notice that if x is a single symbol from the alphabet, this coincides with the definition of the transition function.

We have $\delta(s, \epsilon) = \epsilon$ for any state s. If x is not the empty string, it looks like ya where y is a string of length $|x| - 1$ and $a \in \Sigma$. After processing y, the automaton is in state $\delta(s, y)$. The next input symbol is a, and M_1 goes to state $\delta(\delta(s, y), a)$ after processing it. Thus, we get a recursive definition of the form $\delta(s, ya) = \delta(\delta(s, y), a)$.

Now we prove the following invariant by induction on the number of steps: "M_1 is in state s_1 after n steps if and only if the number of ones in the first n symbols of x is odd".

We saw earlier that $\delta(s_0, \epsilon) = s_0$. Since ϵ has an even number of ones in it, this proves the base case.

Now suppose our invariant holds after the first n steps, and consider the $(n + 1)$st step. The first $n + 1$ symbols in the input have the form ya where $|y| = n$ and $a \in \Sigma$. There are two cases to consider.

Case 1: M_1 is in state s_1 after processing y. This happens if and only if y has an odd number of ones in it by the induction hypothesis. Now if $a = 1$, the state changes to s_0, and the additional 1 makes the number of ones in the first $n + 1$ symbols even. If $a = 0$, M_1 stays in state s_1, and the number of ones in the first $n + 1$ symbols stays odd. Thus, the invariant holds after $n + 1$ steps in this case.

Case 2: M_1 is in state s_0 after processing y. This happens if and only if y has an even number of ones in it by the induction hypothesis. Now if $a = 1$, the state changes to s_1, and the additional 1 makes the number of ones in the first $n+1$ symbols odd. If $a = 0$, M_1 stays in state s_0, and the number of ones in the first $n+1$ symbols stays even. Thus, the invariant holds after $n+1$ steps in this case too.

Then consider the situation after the entire input string x is processed, i.e., after $|x|$ steps. By the invariant we just proved, M_1 is in state s_1 if and only if the input string contains an odd number of ones. Since s_1 is the only accepting state, this implies that M_1 outputs "yes" on input x if and only if x contains an odd number of ones. ⊠

13.1.3 Finite State Automata as a Model of Computation

We can view a finite state automaton $M = (S, \Sigma, \delta, s_0, A)$ as a simple computer consisting of the following parts.

- A finite control that knows the the tuple M and stores the automaton's current state.
- A tape that has the input written on it.
- A tape head that is positioned above the tape and can read the symbol underneath it.

When the automaton starts, the finite control sets the current state to the start state s_0 and positions the tape head above the first symbol on the tape.

In each step, the automaton reads the symbol under its tape head. The finite control looks at that symbol and at the current state, changes states according to the transition function, and moves the tape head above the next symbol on the tape.

The finite state automaton looks like a very simple computing device. We show the setup in Figure 13.4.

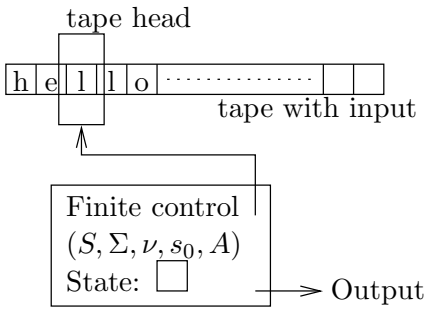


Figure 13.4: Components of a finite state automaton.

13.1.4 Designing Finite State Automata

We mentioned in an earlier reading that one often uses invariants to design algorithms. This is also true when designing finite state automata for deciding a given language L. In particular, invariants can describe all situations in which an automaton is in a particular state. Let's demonstrate this technique on an example.

Example 13.5: We design a finite state automaton M that decides the language consisting of all strings over the alphabet $\{0, 1\}$ that start and end with the same symbol. For example, the strings 101 and 0111010 are in the language, and the strings 110 and 0101 are not in the language.

We create a start state s. We make it so that the only way for the computation to end in this state is if the input is the empty string. Let's agree that the empty string starts and ends with the same symbol, so s is an accepting state.

Our automaton M must know what the first symbol in the input was because otherwise it will have no way of telling whether the last symbol read so far is the same as the first symbol. Thus, we create states s_0 and s_1, which indicate that the first symbol in the string was 0 and 1, respectively.

Next, we need to decide on the logic for each of the two situations mentioned in the previous paragraph. There cannot be any transitions from s_0 to s_1 or from s_1 to s_0 because such transition would lose information about the first symbol in the input.

Let's focus on the situation when the first symbol was zero first. Suppose M is in state s_0 before it reads the last symbol in the input. If this symbol is 0, M should transition to some accepting state, and should transition to a non-accepting state otherwise. We could use the state s_0 as that accepting state. This state must be accepting anyway because the string 0 starts and ends with zero, and M ends in state s_0 after processing this string. So we make s_0 an accepting state and add a transition from state s_0 to state s_0 on input 0. Now on input 1 in state s_0, M needs to transition away from s_0 to some non-accepting state because if this 1 were the last input in the string, staying in s_0 would cause M to accept incorrectly. So we add a state s_{01}. This state is reached if the first symbol in the string was 0, and the last symbol read was 1. We also redefine the meaning of the state s_0 to "the first and the last symbol read were 0". To complete this part of the picture, there is a transition from s_{01} to s_0 on input 0, and from s_{01} to s_{01} on input 1.

Likewise, we make the state s_1 accepting and redefine its meaning to "the first and the last symbol read were both 1". We also add the state s_{10} which indicates that the first symbol in the string was a 1 and the last symbol read was a zero. We add transitions similar to the case when the first input symbol was zero.

We show M in Figure 13.5. ⊠

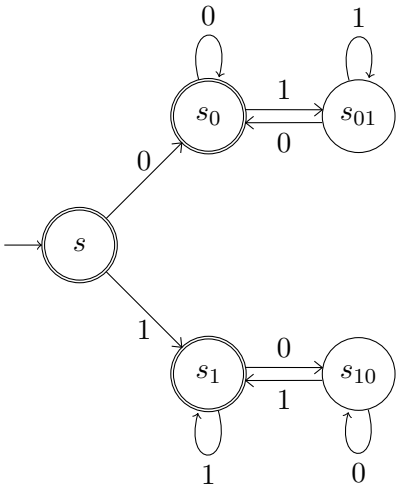


Figure 13.5: The automaton M for the language over $\{0, 1\}$ consisting of strings that start and end with the same symbol.

13.2 Regular Expressions

In this section we present regular expressions which are a means of describing languages.

13.2.1 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.

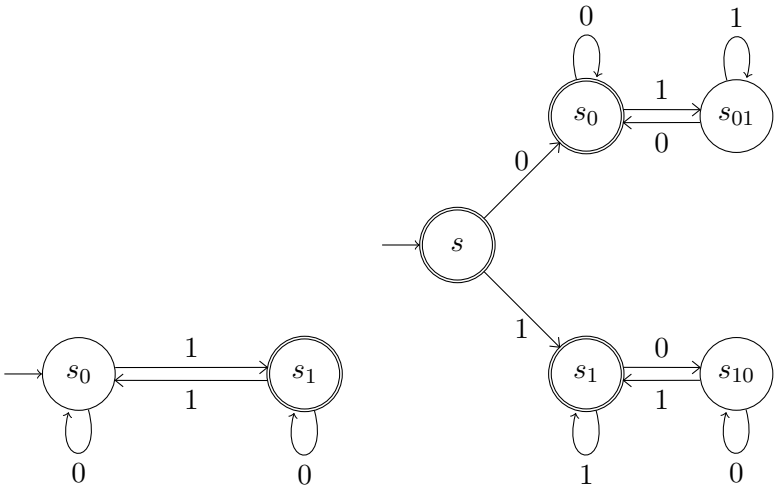
First note that languages are sets, so taking the *union* of two languages makes sense. If L_1 and L_2 are languages, $x \in L_1 \cup L_2$ if and only if $x \in L_1$ or $x \in L_2$.

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 13.5. *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 we described earlier. For completeness, they are shown in Figure 13.6.



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

Figure 13.6: Finite state automata

Example 13.6: 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 13.7: 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_1L_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 13.8: 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 13.9: 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 13.7 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 13.10: 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 13.6. 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.) ⊠

13.2.2 Formal Definition of a Regular Expression

A regular expression is a sequence of characters that define a pattern. Each regular expression describes a language. The language described by a regular expression R, denoted $L(R)$, can be viewed as the set of all strings that match the pattern.

Definition 13.6. *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, Then so are

- $R_1|R_2$ *("OR" or Union of R_1 and R_2) ,*
- R_1R_2 *(Concatenation of R_1 and R_2), and*
- R_1^* *("Kleene star of R_1" or just "R_1 star")*

We now define the language described by a regular expression

- The regular expression $\emptyset$ describes the empty language $\emptyset$.
- The regular expression ϵ describes the language containing just the empty string $\{\epsilon\}$.
- For each $a \in \Sigma$ the regular expression a describes the language $\{a\}$.
- Let R_1 and R_2 be regular expressions that describe the languages L_1 and L_2 respectively. Then $R_1|R_2$ describes the language $L_1 \cup L_2$ (the union of L_1 and L_2.
- Let R_1 and R_2 be regular expressions that describe the languages L_1 and L_2 then the language described by the regular expression R_1R_2 is the concatenation of L_1 and L_2 L_1L_2.
- Let R_1 be a regular expression that describes the language L_1 then R_1^* describes the language L_1^*.

The above is summarized in Table 13.1.

R	$L(R)$
$\emptyset$	$\emptyset$
ϵ	$\{\epsilon\}$
a	$\{a\}$
$R_1\|R_2$	$L(R_1) \cup L(R_2)$
R_1R_2	$L(R_1)L(R_2)$
R_1^*	$L(R_1)^*$

Table 13.1: Languages corresponding to regular expressions.

We remark that the notation for regular expressions isn't entirely standard. For example, sometimes you will see $+$ or $\cup$ for '|', 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.

13.3 Connection between Regular Expressions and Finite Automata

Regular expressions and finite automata define the same class of languages. We now formalize this equivalence of the expressive powers of regular expressions and finite automata.

Definition 13.7. *A language L is* regular *if and only if it can be defined by a regular expression, i.e., it can be written as $L(R)$ for some regular expression R.*

Theorem 13.8. *A language L is regular if and only if it is accepted by some finite automaton, i.e., there exists a finite automaton M such that $L = L(M)$.*

13.3.1 More finite state automata (Optional)

In this section we will design finite state automata for the following family of languages:

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

Given a binary string $x = x_1x_2 \dots x_{n+1}$, the number it represents is $\text{Val}(x) = \sum_{i=1}^{n} x_i 2^{n-i}$.

13.3.2 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

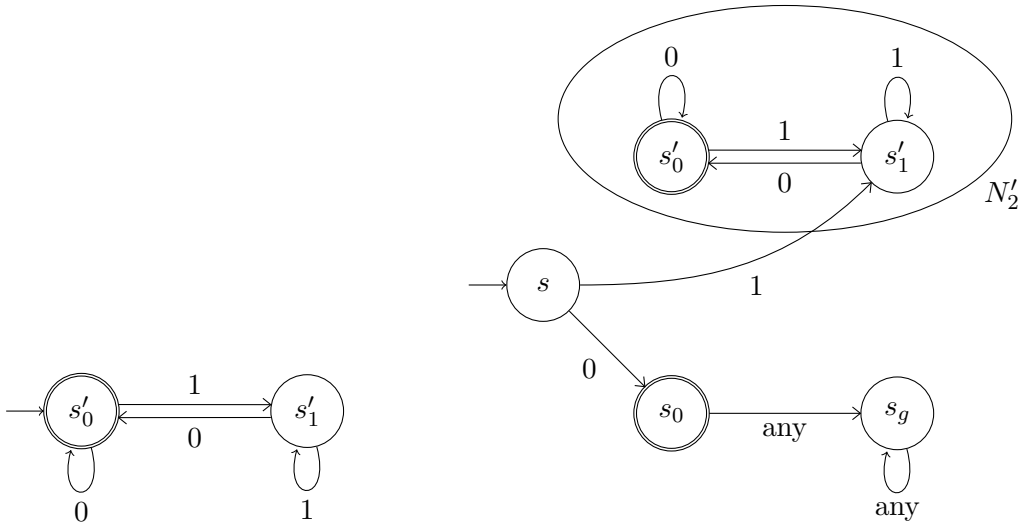
$$\begin{aligned} \mathrm{Val}(x_1x_2 \dots x_nx_{n+1}) &= \sum_{i=1}^{n+1} x_i 2^{n+1-i} \\ &= \sum_{i=1}^{n} x_i 2^{n+1-i} + x_{n+1} \\ &= 2\sum_{i=1}^{n} x_i 2^{n-i} + x_{n+1} \\ &= 2\mathrm{Val}(x_1x_2 \dots x_n) + x_{n+1}. \end{aligned} \tag{13.1}$$

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 (13.1) modulo k. The remainders of both sides after dividing by k have to be the same, so we have

$$\mathrm{Val}(x_1x_2 \dots x_nx_{n+1}) \bmod k = \big[2\mathrm{Val}(x_1x_2 \dots x_n) + x_{n+1}\big] \bmod k \tag{13.2}$$

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 (13.2) tells us what the transition function should be. We have $\nu'(s'_i, a) = s'_{(2i+a) \bmod k}$



(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 13.7: Constructing the automaton N_2.

for $0 \leq i < k$ 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 13.7a.

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.

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 13.7b. As promised earlier, N_k does not accept the empty string because the start state is rejecting.

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.