Faculty Notes

Fernández: 2.2 (pp. 13-20); Sipser: Chapter 1 (pp. 31-99)

Finite Automata and Regular Languages

As depicted in Figure 1, a Finite Automaton (FA) can be used to control an *abstract machine M* that *decides* whether a string w on its input tape is a member of the *regular* language $L_{\mathcal{R}}$ defined by the FA. For example, the FA in Figure 2 *accepts* (recognizes) a fractional binary language, $\mathcal{L}_{\mathcal{B}}$, and you should be able to play the role of M and perform the computation that solves the *decision problem* of whether the input word "10.01" should be accepted or rejected (i.e. whether "10.01" $\in \mathcal{L}_{\mathcal{B}}$).

Input symbol:	1	0	.	0	1
Tape cell:	0	1	2	3	4

As the initial state in our example is q_0, we begin in it with the tape head positioned to read the first (0^{th}) tape cell, which contains a '1'. This information is captured in a *configuration*.

Formally, we define a ***configuration*** for Finite Automaton machine M as a two-tuple,

$$C_i = (q_j, k), \qquad with\ i \geq 1, 0 \leq j \leq n, k \geq 0$$

where q_j is a state $(q_j \in Q)$ and k the location of the tape head (from the left side of the tape).

At this point in our example FA computation, our abstract machine is in the configuration,

$$C_1 = (q_0, 0)$$

Figure 1 | An abstract machine with a finite automaton control

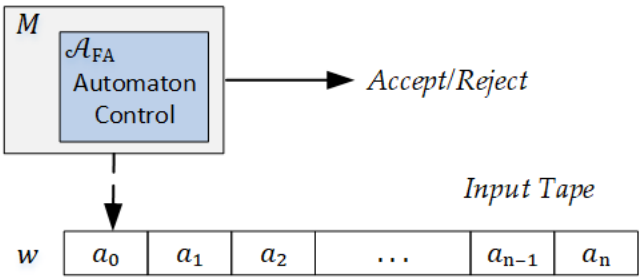
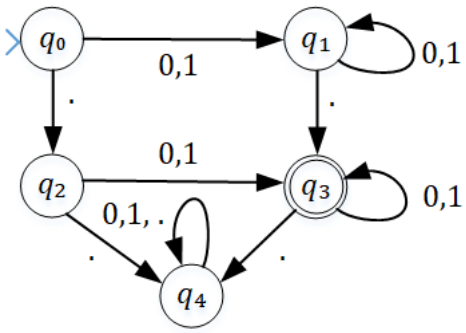


Figure 2 | An example finite automaton that accepts fractional binary numbers

$$\mathcal{A} = (\{.,0,1\}, \{q_0,q_1,q_2,q_3,q_4\}, q_0, \{q_3\}, \delta)$$

$$\delta = \begin{Bmatrix} (q_0,'0',q_1),(q_0,'1',q_1),(q_0,'.',q_2), \\ (q_1,'0',q_1),(q_1,'1',q_1),(q_1,'.',q_3), \\ (q_2,'0',q_3),(q_2,'1',q_3),(q_2,'.',q_4), \\ (q_3,'0',q_3),(q_3,'1',q_3),(q_3,'.',q_4), \\ (q_4,'0',q_4),(q_4,'1',q_4),(q_4,'.',q_4) \end{Bmatrix}$$



© Regis University, All Rights Reserved

As an FA doesn't have a mechanism to change the tape, it's not necessary to represent its contents in a configuration. However, when we examine more complex machines, it will be necessary to represent the entire tape contents within a configuration. For example,

$$C_1' = (q_0, k, 10.01)$$

Given the current state of q_0 and the current input symbol of '1' in the current configuration, the transition function is used to determine the next state, q_1,

$$\delta: (q_0, '1') \longrightarrow q_1$$

Additionally, the tape head is moved to the next cell. This step in the computation is referred to as a *move* of the machine and is represented by a relation '⊢' among configurations. For example,

$$(q_0, 0) \vdash (q_1, 1)$$

Here are all the moves (i.e. configurations) of our FA for this input example,

$$(q_0, 0) \vdash (q_1, 1) \vdash (q_1, 2) \vdash (q_3, 3) \vdash (q_3, 4)$$

Hence, we can speak about the third configuration entered by this machine as,

$$C_3 = (q_1, 2)$$

where the abstract machine M is in state q_1 and reading input tape cell two (i.e. the decimal point).

As this machine finishes in a final accepting state (e.g. q_3) after all input has been read, the machine is considered to have accepted the word "10.01" on the input tape. Consequently, this word is a member of the language accepted by this FA (machine).

Table 1 lists several additional words accepted/rejected by our example machine. We've simplified the configuration by only listing the states entered by the machine

Table 1 | Example finite automaton processing various inputs

Input	State Transitions	Decision	Input	State Transitions	Decision
ε	q_0	*Reject*	1.0	q_0, q_1, q_3, q_3	*Accept*
1	q_0, q_1	*Reject*	10.01	q_0, q_1, q_1, q_3, q_3	*Accept*
10.1.0	$q_0, q_1, q_1, q_3, q_4, q_4$	*Reject*	.01	q_0, q_2, q_3, q_3	*Accept*

Regular Languages

As stated in the textbook, a Finite Automata or Regular Expressions formally define regular languages. Additionally, regular languages can be formally defined by Type₃ phrase grammars. This implies there exist equivalent formal definitions for each of these approaches.

Type₃ Phrase Structure Grammars

A ***Type₃*** phrase structure grammar requires all productions (rules) to be in one of the two following forms:

$$A \rightarrow \alpha B \text{ where } \alpha \in T \text{ and } B \in N$$
$$A \rightarrow \alpha \quad \text{where } \alpha \in T \cup \varepsilon$$

© Regis University, All Rights Reserved

This restriction consists of a nonterminal rewritten to either:

(i) a terminal α followed by a nonterminal B or

(ii) a single terminal α, which may be the empty string ϵ.

As an example, consider the following *regular* grammar, where each production adheres to the preceding Type$_3$ restrictions given,

$S \rightarrow 0A \mid 1A \mid .B$	// *Essentially state* q_0
$A \rightarrow 0A \mid 1A \mid .C$	// *Essentially state* q_1
$B \rightarrow 0C \mid 1C \mid .D$	// *Essentially state* q_2
$C \rightarrow 0C \mid 1C \mid .D \mid \epsilon$	// *Essentially state* q_3
$D \rightarrow 0D \mid 1D \mid .D \mid \epsilon$	// *Essentially state* q_4

The regular language associated with this grammar is our example fractional binary language.

Regular Language Closure Properties

In addition to being closed under union, concatenation, and Kleene Closure, regular languages are also closed under the complement and intersection operations.

Complement Operation

Given an alphabet $\Sigma = \{0,1\}$, recall the *Kleene Closure* (star operation) is the language,

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots ,\} = \mathcal{L}_*$$

Now, consider the language of words over Σ starting with '1', as defined by the regular expression,

$$\mathcal{L}_1 = 1(0+1)^*$$

$\mathcal{L}_1$ is clearly a subset of $\mathcal{L}_*$. Thus, the ***complement*** of a language, $\bar{\mathcal{L}}$ or $\mathcal{L}'$, is the *difference* between $\mathcal{L}$ and Σ^*. For example, the complement of $\mathcal{L}_1$ with respect to Σ^* (i.e. the $\mathcal{L}_*$ language) is,

$$\overline{\mathcal{L}_1} = \{\epsilon, 0, 00, 01, 000, 001, 010, 011, 0000, \dots\}$$

which is the set of all words beginning with a zero and includes the empty string. The complement of a regular language results in a regular language.

Intersection Operation

Given two regular languages $\mathcal{L}_1$ *and* $\mathcal{L}_2$, the ***intersection*** language $\mathcal{L}_\cap = \mathcal{L}_1 \cap \mathcal{L}_2$ consists of all words in both $\mathcal{L}_1$ and $\mathcal{L}_2$. The *intersection* of two regular languages results in a regular language. A proof of this follows directly from the closure of union, complement, and DeMorgan's law,

$$\mathcal{L}_1 \cap \mathcal{L}_2 = (\mathcal{L}_1' \cup \mathcal{L}_2')'$$

It's worth noting, FAs form a Boolean Algebra since we've now defined complement, union, and intersection operations (i.e. analogues to a *complete set*, if these were truth-valued functions).

Additional Examples

If $\mathcal{L}_1 = \{a, b\}$ and $\mathcal{L}_2 = \{c, d, e\}$, then the *concatenation* (or *product*) of these is,

$$\mathcal{L}_1\mathcal{L}_2 = \{ac, ad, ae, bc, bd, be\}$$

© Regis University, All Rights Reserved

Cohen (1997) gives a nice example of *union*. Given $\Sigma = \{a, b\}$, using regular expressions define,

$\mathcal{L}_1 = a(a+b)^*a + b(a+b)^*b$ *all words beginning and ending with the same letter*
$\mathcal{L}_2 = (a+b)^*aba(a+b)^*$ *all words containing the substring aba*

then the union of these languages is

$$\mathcal{L}_\cup = \mathcal{L}_1 + \mathcal{L}_2 = [a(a+b)^*a + b(a+b)^*b] + [(a+b)^*aba(a+b)^*]$$

Notice, since Cohen only uses the regular expression alternation '+' operation on two existing regular expressions to create this union (on the right-hand side), it necessarily results in a regular expression, which defines a regular language. This isn't a proof, but nice to see an example.

Pumping Lemma and Nonregular Languages

Recall, the pumping lemma (Proposition 2.10) can be used to prove that regular languages have limited computational power (expressiveness). For example, consider the following language,

$$\mathcal{L} = \{0^n 1^n\} \mid n \geq 1$$

consisting of strings beginning and ending with equal number of 0s and 1s, respectively. Remember, these 0s and 1s are simply symbols and, just as easily, we could have used different symbols, such left '(' and right ')' parentheses. The following words are members of $\mathcal{L}$,

$$\{01, 0011, 000111, 00001111, \ldots, 0^n1^n\}$$

Notice, this language contains an infinite number of words and can be generated by the grammar,

$$S \rightarrow aSb \mid ab \tag{1}$$

In the Chomsky hierarchy, recall that regular languages (Type$_3$) are generated by regular grammars, which restrict the form of production rules to the following forms

$$P \rightarrow x \mid x \in T \cup \epsilon$$

$$P \rightarrow yz \mid y \in T, z \in N$$

with N the set of nonterminals, T the set of terminals, and ε the empty string. Notice, the grammar in (1) does not adhere to these restrictions and, hence, does not generate a regular language. This doesn't imply that another grammar, which is regular, might generate our example language. As it turns out, one doesn't exist, and we can use the pumping lemma to prove it.

We begin by assuming the 0^n1 is a regular language. Hence, we know there must be a finite automaton states that recognizes its member words. Furthermore, this automaton must have a fixed finite number states, say k, but our language must include words of the form (Parkes, 2002),

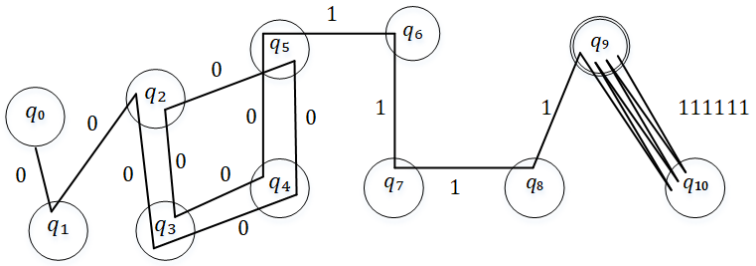
$$\{0^j 1^j\} \mid j > k$$

which results in a word w (in fact, infinitely many) with more than k symbols, thus requiring a circuit in our finite automaton. Let's look at a concrete example before returning to the lemma.

To understand the implications of adding a circuit to a FA, consider the following example, adapted from (Cohen, 1997), which considers the set of transitions for recognizing the word 0^91^9

© Regis University, All Rights Reserved

in a ten-state finite automaton that has been simplified by removing transition not germane to the argument, It's important to realize that any circuit for this language must consist of the same symbols on each transition, otherwise the automaton would recognize words not in 0^n1^n.



Focusing on the 0s circuit in this path, the word $0^9 1^9$ can be partitioned into the string,

$$0^9 1^9 = 0^2 0^{3+4*i} 1^9$$

where i is the number of loops through the circuit, which is one in the current situation,

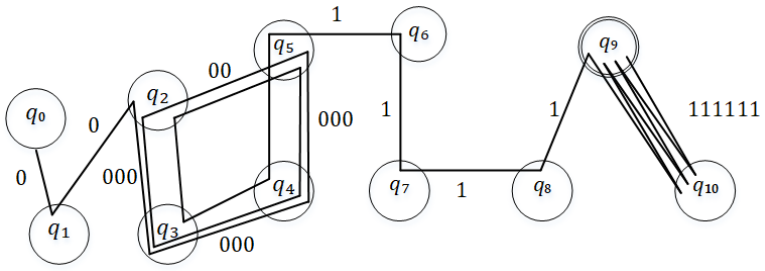
$$0^9 1^9 = 0^2 0^{3+4^1} 1^9$$

since the path completed one circuit with seven 0s,

To show the impact of the pumping lemma, Cohen asks what the path would be like for the word,

$$0^{13} 1^9$$

Here's the trace of the path for this second scenario,



This second scenario begins the same as with $0^9 1^9$, but must loop around the circuit twice.

$$0^{13} 1^9 = 0^2 0^{3+4*2} 1^9$$

Since the choice of 1 at state q_5 is the same in both scenarios, the finite automaton finishes in state q_9 in both cases since nine 1s are processed. However, the word $0^{13} 1^9$ should be rejected.

In fact, this ten-state automaton will accept any word of the form,

$$0^2 0^{3+4*i} 1^9$$

Of course, we can factor this equation differently,

$$0^2 0^{3+4*i} 1^9 = 0^5 0^{4*i} 1^9 = 0^5 (0000)^i 1^9$$

where the term $(0000)^i$ means repeat the four 0s for i number of times.

© Regis University, All Rights Reserved

Now, if we let $u = 00000$, $v^i = 0000$, and $w = 1^9$, we have the pumping lemma form given in the Fernández text. Notice, the conditions of the lemma are satisfied, namely,

1. the length of uv (e.g. $|000000000| = 9$) is less than or equal to n (i.e. $n = 10$),
2. the length of v (e.g. $|0000| = 4$) is greater than or equal to 1, and
3. for any $i \geq 0$, $uv^iw \in L$, where v^i represent the word v repeated i times

Parks (2002) demonstrates the pumping lemma using the following *repeatable substring* argument.

Assume $\mathcal{L} = \{0^n1^n\} \mid n \geq 1$ is a regular language with $uv^iw \in L$, then

- the '*repeatable string*' part v^i cannot consist of 0s followed by 1s since repeating this string would allows words not in $\mathcal{L}$ (e.g. 00**1010**11).
- Hence, the repeatable substring v^i must consist of as alone or bs alone
- The repeatable substring cannot consist of 0s alone since repeating it results in more 0s than 1s, once again resulting in a words not in $\mathcal{L}$ (e.g. 00**00**11). The same argument holds for a substring consisting only of 1s (e.g. 00**11**11).

With these three statements, we've reached a contradiction. As they logically follow from our assumption, this assumption must be wrong. Thus, $\mathcal{L} = \{0^n1^n\} \mid n \geq 1$ is **not** a regular language.

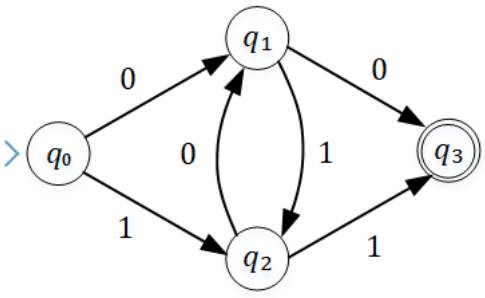
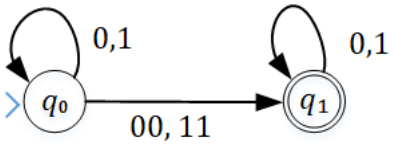
Recall, the pumping lemma argument is a form of the *pigeon hole principle*, which states: if we have n pigeon-pens and n +1 pigeons in these pens, then some pen must have more than one pigeon.

Related Abstract Machines

In addition to the equivalence of deterministic and nondeterministic Finite Automata, there are several equivalent abstract machine variations of finite automaton that you should also be familiar with (i.e. they all "recognize" regular languages).

Transition Graphs

To introduce transition graphs, let's compare them with finite automaton. For example, the FA and TG in the following table both recognize the same language consisting of all words with a double letter. The equivalent regular expression is: $(a + b)^*(aa + bb)(a + b)^*$

Finite Automation	Transition Graph
	

Notice, the transition graph allows transitions with multiple input symbols, which corresponds to reading multiple symbols from the input tape simultaneously. This can be viewed as a relaxation on the definition of a FA (i.e. FAs only read a single input symbol on the tape).

© Regis University, All Rights Reserved

As depicted in Figures 3a and 3b, transition graphs relax several other FA criteria including: allowing multiple initial states, allowing empty string transitions, and not requiring a transition from each state for each letter of the alphabet. They also allow multiple transitions with the same beginning input symbols as a non-deterministic finite automaton.

Figure 3a

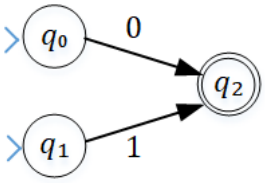
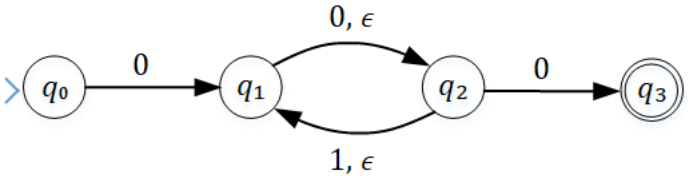


Figure 3b



As the TG in Figure 3a has two start states, this TG accepts the language with two words, a or b.

The TG in Figure 3b is a little stranger since it allows the empty string ϵ to be used to transition between states q_1 and q_2. For example, this TG accepts the word "00" with the following state transitions: begin in q_0 and transition on the first '0' input to q_1. Next, it transitions using the empty string ϵ to q_2. Finally, it transitions to q_3 using the second '0' input. You should convince yourself that this TG also accepts the word "0000". Notice, the states in this TGs do not include transitions for every possible input symbol.

Formally, a ***transition graph*** is a five-tuple $\mathcal{T} = (\Sigma, Q, S, F, \tau)$ with a finite set of *alphabet* symbols Σ, a finite set of *states* Q $\{q_0, \dots, q_n\}$, a finite set of *initial states* $S \in Q$, a set of *final* (*accept*) *states* $F \subseteq Q$; and a *transition function* $\tau: Q \times \Sigma^* \longrightarrow Q$ mapping a *current* state and one or more *input* symbols (including an empty string) to a *next* state.

Although transition graphs (TGs) appear more powerful than finite automata, they are not and only recognize the class of regular languages. Hence, Every transition graph has an equivalent deterministic finite automaton. The proof consists of showing how every construct in a transition graph can be represented in a corresponding finite automaton.

A state in a TG may not have a transition for every input symbol. If an input occurs in a state with a missing transition, the machine is said to ***crash*** and the corresponding input is not accepted.

Finite Automata and Output

In this section, we examine two approaches for adding *output* to a finite automation. Such automata are referred to as ***finite state transducers***.

Moore Machines

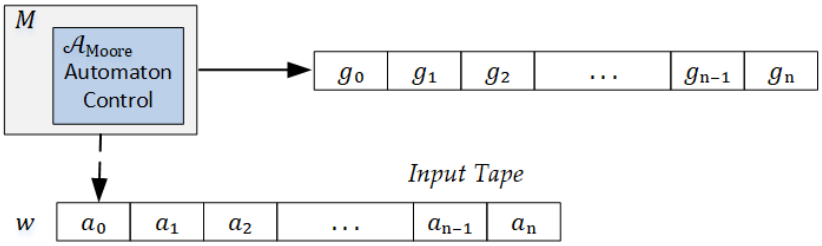
A ***Moore machine*** is formally defined as a six-tuple, $\mathcal{A}^{Moore} = (\Sigma, Q, q_0, \delta, \Gamma, \gamma)$

with a finite set of *input alphabet* symbols, Σ; a finite set of *states*, Q $\{q_0, \dots, q_n\}$; an *initial state* $q_0 \in Q$, a *transition function*, $\delta: Q \times \Sigma \longrightarrow Q$, a finite set of *output alphabet* symbols Γ, and an output function $\gamma: Q \longrightarrow \Gamma$. As depicted in Figure 5, a Moore machine can be viewed as writing symbols from Γ to an output tape, which occurs whenever a state is entered (except for the first state).

© Regis University, All Rights Reserved

Although a Moore machine doesn't define a set of final accepting states, it can be designed to perform recognition tasks (as shown later).

Figure 4 | Moore machine example



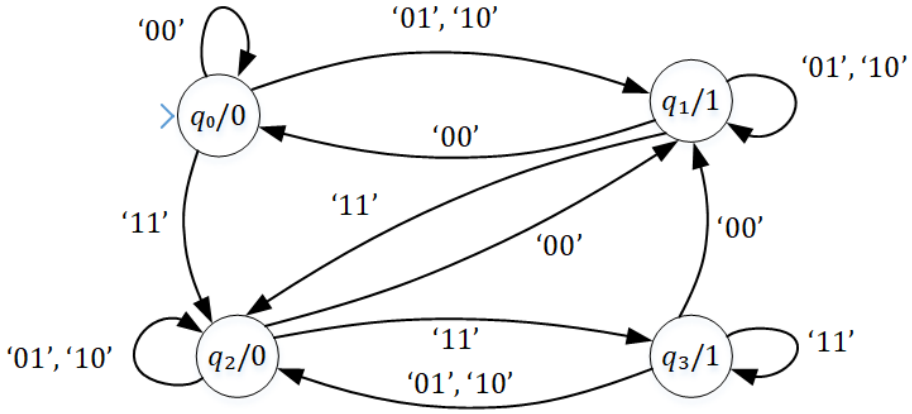
As an example, let's consider a Moore machine that performs binary addition (Figure 5),

$$\mathcal{A}_{+}^{Moore} = (\{00, 01, 10, 11\}, \{q_0, q_1, q_2, q_3\}, q_0, \delta, \gamma)$$

$$\delta = \begin{Bmatrix} (q_0, '00', q_0), (q_0, '01', q_1), (q_0, '10', q_1), (q_0, '11', q_2), \\ (q_1, '00', q_0), (q_1, '01', q_1), (q_1, '10', q_1), (q_1, '11', q_2), \\ (q_2, '00', q_1), (q_2, '01', q_2), (q_2, '10', q_2), (q_2, '11', q_3), \\ (q_3, '00', q_1), (q_3, '01', q_2), (q_3, '10', q_2), (q_3, '11', q_3) \end{Bmatrix}$$

$$\gamma = \{(q_0, 0), (q_1, 1), (q_2, 0), (q_3, 1)\}$$

Figure 5 | Example Moore machine for a binary carry-ripple adder, adapted from (Rosenberg, 2010)



In Figure 5, a state label such as "$q_3/1$" indicates to output a '1' when state q_3 is entered.

As an example, consider using our Moore machine to add fifteen $(1111)_2$ and six $(0110)_2$. to arrive at twenty-one $(10100)_2$. To ensure the machine handles any carry's in the most-significant digit, which occurs in this example, we initially pad each number with a leading zero.

$$\begin{array}{r} 01111 \\ 00110 \\ \hline 10101 \end{array}$$

Working from right-to-left, we select one binary digit from each number and encode it as the single character in the input alphabet. Hence, the input tape for this example is (where we use quote marks to indicate that we're dealing with a single atomic symbol in the tape in each cell),

© Regis University, All Rights Reserved

'10'	'11'	'11'	'10'	'00'

We begin in state q_0 with the initial input '10', which results in a transition to state q_1 and an output of '1' (written to the output tape of the machine). The next input symbol is '11', which results in a transition from q_1 to q_2 with an output of '0' at state q_2. Continuing in this fashion, the execution ends after all input is processed, the output tape contains the resulting sum (reversed):

1	0	1	0	1

Table 2 describes the states in our example Moore machine (i.e. what they correspond to as the input is processed).

Table 2 | Description of states in our example Moore machine

State	Comment
q_0	Sum is zero with no carry-in
q_1	Sum is one with no carry-in
q_2	Sum is zero with a carry-in of one
q_3	Sum is one with a carry-in of one

Mealy Machine

A Mealy machine is like a Moore machine except the output is based on the transitions.

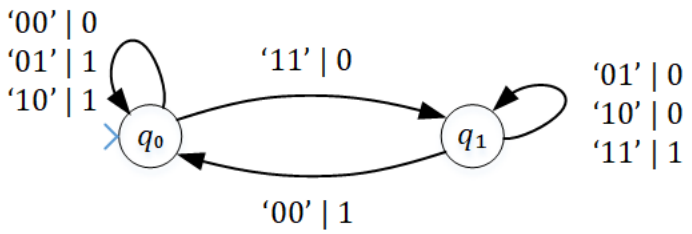
Formally, a ***Mealy machine*** is defined as a six-tuple, $\mathcal{A}^{Mealy} = (\Sigma, Q, q_0, \delta, \Gamma, \gamma)$

with a finite set of *input alphabet* symbols, Σ; a finite set of *states*, Q $\{q_0, \ldots, q_n\}$; an *initial state* $q_0 \in Q$, a *transition function*, $\delta: Q \times \Sigma \longrightarrow Q$, a finite set of *output alphabet* symbols Γ, and an output function $\gamma: Q \times \Sigma \longrightarrow \Gamma$. Notice, the difference between the Moore and Mealy machine definitions occurs in the output function γ. In the Moore machine, a state is mapped to an output symbol, while in the Mealy machine a state and input symbol is mapped to an output symbol.

As an example, consider the Mealy machine in Figure 6, which also performs binary addition. A transition label, such as "'01' | 1" indicates a transition on input symbol '01' with an output of '0'.

It's worth noting the term *transducer* results from the fact that Moore and Mealy machines can be used to design sequential (electrical) circuits

Figure 6 | A Mealy machine that performs binary addition



© Regis University, All Rights Reserved

Although Moore and Mealy machines are not formally designed as language recognizers, they can be constructed to perform this task. For example, we need only have every "non-recognizing" state output a '0' and every final "accepting" state output a '1'. If the last symbol output by the machine is a '1', the input word was accepted (recognized). When used as recognizers.

Moore and Mealey machines recognize regular languages

Two-Way Finite Automata

Thus far, each of the finite automata machines examined in this note only moves the input tape in one direction. The content of each cell is read only once, as the machine makes a single left to right pass through the input on the tape. A *two-way finite automaton* may move the tape in either direction. As a result, the machine may read the content of a tape cell more than once with the possibility of multiple passed through the input on the tape. Formally, a ***two-way finite automaton*** (Hopcroft & Ullman, 1969) is defined as a deterministic finite automaton, but with a *transition function* $\delta: Q \times \Sigma \longrightarrow Q \times \{L, R, S\}$ that maps a *current* state and *input* symbol to a *next* state and a control of the tape reading head: one cell to the left L, one cell right R (the default DFA), or remain reading the same cell, S. Because of this movement, when a two-way automaton moves its head off the right end of the tape it *accepts* its input, if it is in a final state. Whereas a DFA has only one case in which it rejects its input, (i) when it moves off the right end of the input tape in a non-accepting state, a two-way FA also rejects its input in two additional cases: (ii) if it moves off the left end of its input tape or (iii) it loops forever.

Although two-way finite automaton might appear more powerful than finite automata, they are not and recognize the class of regular languages.

Every two-way FA has an equivalent deterministic finite automaton

The proof is a little complex, but can be found in (Hopcroft and Ullman, 1969).

Summary

Table 3 summarizes the primary characteristics of regular languages presented in this topic. We'll extend this table in the next few topics, as we introduce additional types of automata. The final decidability column is included for completeness and presented in detail in Topic 7.

Table 3 | Major themes corresponding to languages [adapted from (Cohen, 1997)]

Language Defined by	Corresponding Acceptor	Nondeterminism = Determinism?	Language Closed Under	What Can be Decided
Regular Expression, Type_3 Grammar	Finite automaton, Transition graph	Yes	Union, Product, Kleene Star, Intersection, Complement	Equivalence, Emptiness, Finiteness, Membership

References

A complete set of references is given in the online course shell.

© Regis University, All Rights Reserved