chapter 14

Monoids and Automata

GOALS
At first glance, the two topics that we will discuss in this chapter seem totally unrelated. The first is monoid theory, which we touched upon in Chapter 11. The second is automata theory, in which computers and other machines are described in abstract terms. After short independent discussions of these topics, we will describe how the two are related in the sense that each monoid can be viewed as a machine and each machine has a monoid associated with it.

14.1 Monoids

Recall the definition of a monoid:

Definition: Monoid. A monoid is a set \( M \) together with a binary operation \( * \) with the properties

\[
\begin{align*}
(a) \quad & * \text{ is associative; } (a*b)*c = a*(b*c) \text{ for all } a, b, c \in M, \\
(b) \quad & * \text{ has an identity; there exists } e \in M \text{ such that for all } a \in M, \ a*e = e*a = a.
\end{align*}
\]

Note: Since the requirements for a group contain the requirements for a monoid, every group is a monoid.

Example 14.1.1.

(a) The power set of any set together with any one of the operations intersection, union, or symmetric difference is a monoid.

(b) The set of integers, \( \mathbb{Z} \), with multiplication, is a monoid. With addition, \( \mathbb{Z} \) is also a monoid.

(c) The set of \( n \times n \) matrices over the integers, \( M_n(\mathbb{Z}) \), \( n \geq 2 \), with matrix multiplication, is a monoid. This follows from the fact that matrix multiplication is associative and has an identity, \( I_n \). This is an example of a noncommutative monoid since there are matrices, \( A \) and \( B \), for which \( A B \neq B A \).

(d) \( [\mathbb{Z}_n, \times_n] \), \( n \geq 2 \), is a monoid with identity 1.

(e) Let \( X \) be a nonempty set. The set of all functions from \( X \) into \( X \), often denoted \( X^X \), is a monoid over function composition. In Chapter 7, we saw that function composition is associative. The function \( i : X \rightarrow X \) defined by \( i(a) = a \) is the identity element for this system. This is another example of a noncommutative monoid, provided \( |X| \) is greater than 1.

If \( X \) is finite, \( |X^X| = |X|^{|X|} \). For example, if \( B = \{0, 1\} \), \( |B^B| = 4 \). The functions \( z, u, i, \) and \( t \), defined by the graphs in Figure 14.1.1, are the elements of \( B^B \). This monoid is not a group. Do you know why?

One reason that \( B^B \) is noncommutative is that \( tz \neq zt \), since \( (tz)(0) = 1 \) and \( (zt)(0) = 0 \).
Chapter 14 - Monoids and Automata

GENERAL CONCEPTS AND PROPERTIES OF MONOIDS

Virtually all of the group concepts that were discussed in Chapter 11 are applicable to monoids. When we introduced subsystems, we saw that a submonoid of monoid \( M \) is a subset of \( M \) — that is, it itself is a monoid with the operation of \( M \). To prove that a subset is a submonoid, you can apply the following algorithm.

**Theorem/Algorithm 14.1.1.** Let \( [M; \ast] \) be a monoid and \( K \) is a nonempty subset of \( M \), \( K \) is a submonoid of \( M \) if and only if:

(a) If \( a, b \in K \), then \( a \ast b \in K \) (i.e., \( K \) is closed under \( \ast \)), and

(b) the identity of \( M \) belongs to \( K \).

Often we will want to discuss the smallest submonoid that includes a certain subset \( S \) of a monoid \( M \). This submonoid can be defined recursively by the following definition.

**Definition: Submonoid Generated by a Set.** If \( S \) is a subset of monoid \( [M; \ast] \), the submonoid generated by \( S \), \(( S \), is defined by:

(a) (Basis) \( i \in S \Rightarrow a \in \langle S \rangle \), and \( (ii) \) the identity of \( M \) belongs to \( \langle S \rangle \);

(b) (Recursion) \( a, b \in \langle S \rangle \Rightarrow a \ast b \in \langle S \rangle \).

Note: If \( S = \{a_1, a_2, \ldots, a_n\} \), we write \( \langle a_1, a_2, \ldots, a_n \rangle \) in place of \( \langle \{a_1, a_2, \ldots, a_n\} \rangle \).

**Example 14.1.2.**

(a) In \([\mathbb{Z}; +]\), \( (2) = \{0, 2, 4, 6, 8, \ldots\} \).

(b) The power set of \( \mathbb{Z}, \mathcal{P}(\mathbb{Z}) \), over union is a monoid with identity \( \emptyset \). If \( S = \{1, 2, 3\} \), then \( \langle S \rangle \) is the power set of \( \{1, 2, 3\} \). If \( S = \{n \in \mathbb{Z} \} \), then \( \langle S \rangle \) is the set of finite subsets of the integers.

**MONOID ISOMORPHISMS**

Two monoids are **isomorphic** if and only if there exists a translation rule between them so that any true proposition in one monoid is translated to a true proposition in the other.

**Example 14.1.3.** \( M = [\mathcal{P}\{1, 2, 3\}, \cap] \) is isomorphic to \( M_2 = [\mathbb{Z}_2; +] \), where the operation in \( M_2 \) is componentwise \( \text{mod } 2 \) multiplication. A translation rule is that if \( A \subseteq \{1, 2, 3\} \), then it is translated to \((d_1, d_2, d_3)\) where \( d_i = \begin{cases} 1 & \text{if } i \in A \\ 0 & \text{if } i \notin A \end{cases} \). Two cases of how this translation rule works are:

\[
\begin{align*}
\{1, 2, 3\} & \text{ is the identity for } M_1, \quad \text{and} \quad \{1, 2\} \cap \{2, 3\} = \{2\} \\
\uparrow & \quad \uparrow \quad \uparrow \quad \uparrow \\
(1, 1, 1) & \text{ is the identity for } M_2, \quad \text{and} \quad (1, 1, 1) \cdot (0, 1, 1) = (0, 1, 0).
\end{align*}
\]

A more precise definition of a monoid isomorphism is identical to the definition of a group isomorphism (see Section 11.7).

**EXERCISES FOR SECTION 14.1**

**A Exercises**

1. For each of the subsets of the indicated monoid, determine whether the subset is a sub monoid.

(a) \( S_1 = \{0, 2, 4, 6\} \) and \( S_2 = \{1, 3, 5, 7\} \) in \([\mathbb{Z}_8; \times_8]\).

(b) \( \{f \in \mathbb{N}^\mathbb{N} : f(n) \leq n, \forall n \in \mathbb{N}\} \) and \( \{f \in \mathbb{N}^\mathbb{N} : f(1) = 2\} \) in \( \mathbb{N}^\mathbb{N} \).
Chapter 14 - Monoids and Automata

(c) \{A \subseteq \mathbb{Z} : A \text{ is finite}\} and \{A \subseteq \mathbb{Z} : A' \text{ is finite}\} in \{\mathcal{P}(\mathbb{Z}); \cup\}.

2. For each subset, describe the submonoid that it generates.

(a) \{3\} and \{0\} in \{\mathbb{Z}_2; \times_{\mathbb{Z}_2}\}

(b) \{5\} in \{\mathbb{Z}_{25}; \times_{\mathbb{Z}_{25}}\}

(c) the set of prime numbers and \{2\} in \{\mathcal{P}; \cdot\}

(d) \{3, 5\} in \{\mathbb{N}; +\}

B Exercises

3. Definition: Stochastic Matrix. An \(n \times n\) matrix of real numbers is called stochastic if and only if each entry is nonnegative and the sum of entries in each column is 1. Prove that the set of stochastic matrices is a monoid over matrix multiplication.


14.2 Free Monoids and Languages

In this section, we will introduce the concept of a language. Languages are subsets of a certain type of monoid, the free monoid over an alphabet. After defining a free monoid, we will discuss languages and some of the basic problems relating to them. We will also discuss the common ways in which languages are defined.

Let A be a nonempty set, which we will call an alphabet. Our primary interest will be in the case where A is finite; however, A could be infinite for most of the situations that we will describe. The elements of A are called letters or symbols. Among the alphabets that we will use are \(B = [0, 1]\), ASCII = the set of ASCII characters, and \(PAS = \) the Pascal character set (whichever one you use).

**Definition:** Strings over an Alphabet. A string of length \(n, n \geq 1\), over A is a sequence of n letters from A: \(a_1 a_2 \ldots a_n\). The null string, \(\lambda\), is defined as the string of length zero containing no letters. The set of strings of length n over A is denoted by \(A^n\). The set of all strings over A is denoted \(A^*\).

Notes:

(a) If the length of string s is n, we write \(|s| = n\).

(b) The null string is not the same as the empty set, although they are similar in many ways.

(c) \(A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \ldots\) and if \(i \neq j, A^i \cap A^j = \emptyset\); that is, \(\{A^0, A^1, A^2, A^3, \ldots\}\) is a partition of \(A^*\).

(d) An element of A can appear any number of times in a string.

**Theorem 14.2.1.** If A is countable, then \(A^*\) is countable.

Proof: Case 1. Given the alphabet \(B = [0, 1]\), we can define a bijection from the positive integers into \(B^*\). Each positive integer has a binary expansion \(d_0 d_1 \cdots d_k\), where each \(d_i\) is 0 or 1 and \(d_0 = 1\). If \(n\) has such a binary expansion, then \(2^k < n < 2^{k+1}\). We define \(f: P \rightarrow B^*\) by \(f(n) = d_0 d_1 \cdots d_k = d_0 d_1 \cdots d_k\), where \(f(1) = \lambda\). Every one of the 2\(^k\) strings of length \(k\) are the images of exactly one of the integers between 2\(^k\) and 2\(^{k+1}\) - 1. From its definition, \(f\) is clearly a bijection; therefore, \(B^*\) is countable.

Case 2: A is Finite. We will describe how this case is handled with an example first and then give the general proof. If A = \{a, b, c, d, e\}, then we can code the letters in A into strings from \(B^5\). One of the coding schemes (there are many) is a \(\leftrightarrow 000\), b \(\leftrightarrow 001\), c \(\leftrightarrow 010\), d \(\leftrightarrow 011\), and e \(\leftrightarrow 100\). Now every string in \(A^*\) corresponds to a different string in \(B^*\); for example, ace would correspond with 000010100. The cardinality of \(A^*\) is equal to the cardinality of the set of strings that can be obtained from this encoding system. The possible coded strings must be countable, since they are a subset of a countable set (\(B^5\)); therefore, \(A^*\) is countable.

If \(|A| = n\), then the letters in A can be coded using a set of fixed-length strings from \(B^m\). If \(2^{m-1} < m \leq 2^k\), then there are at least as many strings of length \(k\) in \(B^m\) as there are letters in A. Now we can associate each letter in A with an element of \(B^m\). Then any string in \(A^*\) corresponds to a string in \(B^m\). By the same reasoning as in the example above, \(A^*\) is countable.

Case 3: A is Countably Infinite. We will leave this case as an exercise.

**FREE MONOIDS OVER AN ALPHABET**

The set of strings over any alphabet is a monoid under concatenation.

**Definition:** Concatenation. Let \(a = a_1 a_2 \cdots a_m\) and \(b = b_1 b_2 \cdots b_n\) be strings of length \(m\) and \(n\), respectively. The concatenation of \(a\) with \(b\), \(a \cdot b\), is the string of length \(m + n\) : \(a_1 a_2 \cdots a_m b_1 b_2 \cdots b_n\).

Notes:

(a) The null string is the identity element of \([A^*; \cdot]\). Henceforth, we will denote the monoid of strings over A by \(A^*\).

(b) Concatenation is noncommutative, provided \(|A| > 1\).

(c) If \(|A_1| = |A_2|\), then the monoids \(A_1^*\) and \(A_2^*\) are isomorphic. An isomorphism can be defined using any bijec-
The Algorithm

Recall that a function $f : A \rightarrow A^*$. If $a = a_1 a_2 \cdots a_n \in A^*$, \( f(a) = f(a_1) f(a_2) \cdots f(a_n) \) defines a bijection from $A^*$ into $A$. We will leave it to the reader to convince him or herself that for all $a, b, c \in A^*$, \( f(a) b = f(a) c \) if and only if \( f(a) = f(b) \).

**Languages**

The languages of the world—English, German, Russian, Chinese, and so forth—are called natural languages. In order to communicate in writing in any one of them, you must first know the letters of the alphabet and then know how to combine the letters in meaningful ways. A **formal language** is an abstraction of this situation.

**Definition:** Formal Language. *If $A$ is an alphabet, a formal language over $A$ is a subset of $A^*$.*

**Example 14.2.1.**

(a) English can be thought of as a language over the set of letters $A, B, \cdots Z$ (upper and lower case) and other special symbols, such as punctuation marks and the blank. Exactly what subset of the strings over this alphabet defines the English language is difficult to pin down exactly. This is a characteristic of natural languages that we try to avoid with formal languages.

(b) The set of all ASCII stream files can be defined in terms of a language over ASCII. An ASCII stream file is a sequence of zero or more lines followed by an end-of-file symbol. A line is defined as a sequence of ASCII characters that ends with the two characters CR (carriage return) and LF (line feed). The end-of-file symbol is system-dependent; for example, CTRL/C is a common one.

(c) The set of all syntactically correct expressions in *Mathematica* is a language over the set of ASCII strings.

(d) A few languages over $B$ are

\[ L_1 = \{ s \in B^* \mid s \text{ has exactly as many 1's as it has 0's} \}, \]

\[ L_2 = \{ 1 \rightarrow s \leftrightarrow 0 \mid s \in B^* \}, \] and

\[ L_3 = \{ (0, 01) \} = \text{the submonoid of } B^* \text{ generated by } \{0, 01\}. \]

**Two Fundamental Problems: Recognition and Generation**

The generation and recognition problems are basic to computer programming. Given a language, $L$, the programmer must know how to write (or generate) a syntactically correct program that solves a problem. On the other hand, the compiler must be written to recognize whether a program contains any syntax errors.

**The Recognition Problem:** Design an algorithm that determines the truth of $s \in L$ in a finite number of steps for all $a \in A^*$. Any such algorithm is called a **recognition algorithm**.

**Definition:** Recursive Language. *A language is recursive if there exists a recognition algorithm for it.*

**Example 14.2.2.**

(a) The language of syntactically correct *Mathematica* expressions is recursive.

(b) The three languages in Example 14.2.1 (d) are all recursive. Recognition algorithms for $L_1$ and $L_2$ should be easy for you to imagine. The reason a recognition algorithm for $L_1$ might not be obvious is that $L_1$’s definition is more cryptic. It doesn’t tell us what belongs to $L_1$, just what can be used to create strings in $L_1$. This is how many languages are defined. With a second description of $L_3$, we can easily design a recognition algorithm. $L_3 = \{ s \in B^* \mid s = \lambda \text{ or } s \text{ starts with 0 and has no consecutive 1's} \}$.

**Algorithm 14.2.1:** Recognition Algorithm for $L_3$. Let $s = s_1 s_2 \cdots s_n \in B^*$. This algorithm determines the truth value of $s \in L_3$. The truth value is returned as the value of Word.

1. \( \text{Word := true} \)
2. \( \text{If } n > 0 \text{ then} \)
   \( \text{if } s_1 = 1 \text{ then } \text{Word := false} \)
   \( \text{else for } i := 3 \text{ to } \)
   \( \text{if } s_{i-1} = 1 \text{ and } s_i = 1 \text{ then } \text{Word := false} \)

**The Generation Problem.** Design an algorithm that generates or produces any string in $L$. Here we presume that $A$ is either finite or countably infinite; hence, $A^*$ is countable by Theorem 14.2.1, and $L \subseteq A^*$ must be countable. Therefore, the generation of $L$ amounts to creating a list of strings in $L$. The list may be either finite or infinite, and you must be able to show that every string in $L$ appears somewhere in the list.

**Theorem 14.2.2.**

(a) If $A$ is countable, then there exists a generating algorithm for $A^*$.

(b) If $L$ is a recursive language over a countable alphabet, then there exists a generating algorithm for $L$.

**Proof:**

(a) Part (a) follows from the fact that $A^*$ is countable; therefore, there exists a complete list of strings in $A^*$.

(b) To generate all strings of $L$, start with a list of all strings in $A^*$ and an empty list, $W$, of strings in $L$. For each string $s$, use a recognition algorithm (one exists since $L$ is recursive) to determine whether $s \in L$. If $s$ is in $L$, add it to $W$; otherwise "throw it out." Then go to the next string in the list of $A^*$.■
Example 14.2.3. Since all of the languages in Example 14.2.2 are recursive, they must have generating algorithms. The one given in the proof of Theorem 14.2.2 is not generally the most efficient. You could probably design more efficient generating algorithms for $L_2$ and $L_3$; however, a better generating algorithm for $L_1$ is not quite so obvious.

The recognition and generation problems can vary in difficulty depending on how a language is defined and what sort of algorithms we allow ourselves to use. This is not to say that the means by which a language is defined determines whether it is recursive. It just means that the truth of "$L$ is recursive" may be more difficult to determine with one definition than with another. We will close this section with a discussion of grammars, which are standard forms of definition for a language. When we restrict ourselves to only certain types of algorithms, we can affect our ability to determine whether $s \in L$ is true. In defining a recursive language, we do not restrict ourselves in any way in regard to the type of algorithm that will be used. In Section 14.3, we will consider machines called finite automata, which can only perform simple algorithms.

**PHRASE STRUCTURE GRAMMARS AND LANGUAGES**

One common way of defining a language is by means of a phrase structure grammar (or grammar, for short). The set of strings that can be produced using the grammar rules is called the phrase structure language (of the grammar).

**Example 14.2.4.** We can define the set of all strings over $B$ for which all 0s precede all 1s as follows. Define the starting symbol $S$ and establish rules that $S$ can be replaced with any of the following: $\lambda$, 0$S$, or 1$S$. These replacement rules are usually called production (or rewriting) rules and are usually written in the format $S \rightarrow \lambda$, $S \rightarrow 0S$, and $S \rightarrow 1S$. Now define $L$ to be the set of all strings that can be produced by starting with $S$ and applying the production rules until $S$ no longer appears. The strings in $L$ are exactly the ones that are described above.

**Definition:** Phrase Structure Grammar. A phrase structure grammar consists of four components:

1. A nonempty finite set of terminal characters, $T$. If the grammar is defining a language over $A$, $T$ is a subset of $A^*$.
2. A finite set of nonterminal characters, $N$.
3. A starting symbol, $S \in N$.
4. A finite set of production rules, each of the form $X \rightarrow Y$, where $X$ and $Y$ are strings over $A \cup N$ such that $X \neq Y$ and $X$ contains at least one nonterminal symbol.

If $G$ is a phrase structure grammar, $L(G)$ is the set of strings that can be obtained by starting with $S$ and applying the production rules a finite number of times until no nonterminal characters remain. If a language can be defined by a phrase structure grammar, then it is called a phrase structure language.

**Example 14.2.5.** The language over $B$ consisting of strings of alternating 0s and 1s is a phrase structure language. It can be defined by the following grammar:

1. Terminal characters: $\lambda$, 0, and 1.
4. Production rules:

$$
S \rightarrow T, \quad S \rightarrow U, \quad S \rightarrow \lambda, \quad S \rightarrow 0, \quad S \rightarrow 1, \quad S \rightarrow 0T,
$$

$$
S \rightarrow 1U, \quad T \rightarrow 10T, \quad T \rightarrow 10, \quad U \rightarrow 01U, \quad U \rightarrow 01
$$

These rules can be visualized more easily with a graph:
Chapter 14 - Monoids and Automata

![Figure 14.2.1](image)

Production rules for the language of alternating 0's and 1's.

We can verify that a string such as 10101 belongs to the language by starting with $S$ and producing 10101 using the production rules a finite number of times: $S \to 1 \ U \to 101 \ U \to 10101$.

**Example 14.2.6.** Let $G$ be the grammar with components:

1. Terminal symbols = all letters of the alphabet (both upper and lower case) and the digits 0 through 9,
2. Nonterminal symbols = $\{I, X\}$,
3. Starting symbol: $I$
4. Production rules: $I \to a$, where $a$ is any letter, $I \to aX$ for any letter $a$, $X \to \beta X$ for any letter or digit $\beta$, and $X \to \beta$ for any letter or digit $\beta$.

There are a total of 176 production rules for this grammar. The language $L(G)$ consists of all valid Mathematica names.

**Backus-Naur form (BNF).** A popular alternate form of defining the production rules in a grammar is BNF. If the production rules $A \to B_1, A \to B_2, \ldots A \to B_n$ are part of a grammar, they would be written in BNF as $A ::= B_1 | B_2 | \cdots | B_n$. The symbol $|$ in BNF is read as "or," while the :: is read as "is defined as." Additional notations of BNF are that $[x]$ represents zero or more repetitions of $x$ and $[y]$ means that $y$ is optional.

**Example 14.2.7.** A BNF version of the production rules for a Mathematica name is

```
letter ::= a | b | c \cdots | z | A | B | \cdots | Z
digit ::= 0 | 1 | \cdots | 9
I ::= letter | digit
```

**Example 14.2.8.** An arithmetic expression can be defined in BNF. For simplicity, we will consider only expressions obtained using addition and multiplication of integers. The terminal symbols are ( ), +, *, - , and the digits 0 through 9. The nonterminal symbols are $E$ (for expression), $T$ (term), $F$ (factor), and $N$ (number). The starting symbol is $E$.

```
E ::= E + T | T
```
Chapter 14 - Monoids and Automata

\[
T ::= T \ast F \mid F \\
F ::= (E) \mid N \\
N ::= [-] \text{digit (digit).}
\]

One particularly simple type of phrase structure grammar is the regular grammar.

**Definition:** Regular Grammar. A **regular (right-hand form) grammar** is a grammar whose production rules are all of the form \( A \rightarrow t \) and \( A \rightarrow tB \), where \( A \) and \( B \) are nonterminal and \( t \) is terminal. A left-hand form grammar allows only \( A \rightarrow t \) and \( A \rightarrow Br \), a language that has a regular phrase structure language is called a regular language.

**Example 14.2.9.**

(a) The set of Mathematica names is a regular language since the grammar by which we defined the set is a regular grammar.

(b) The language of all strings for which all 0s precede all 1s (Example 14.2.4) is regular; however, the grammar by which we defined this set is not regular. Can you define these strings with a regular grammar?

(c) The language of arithmetic expressions is not regular.

**EXERCISES FOR SECTION 14.2**

**A Exercises**

1. (a) If a computer is being designed to operate with a character set of 350 symbols, how many bits must be reserved for each character? Assume each character will use the same number of bits.

(b) Do the same for 3,500 symbols.

2. It was pointed out in the text that the null string and the null set are different. The former is a string and the latter is a set, two different kinds of objects. Discuss how the two are similar.

3. What sets of strings are defined by the following grammar?
   
   (a) Terminal symbols: \( \lambda \), 0 and 1
   
   (b) Nonterminal symbols: \( S \) and \( E \)
   
   (c) Starting symbol: \( S \)
   
   (d) Production rules: \( S \rightarrow 0 S 0, \ S \rightarrow 1 S 1, \ S \rightarrow E, \ E \rightarrow \lambda, \ E \rightarrow 0, \ E \rightarrow 1. \)

4. What sets of strings are defined by the following grammar?
   
   (a) Terminal symbols: \( \lambda, \ a, \ b, \) and \( c \)

   (b) Nonterminal symbols: \( S, \ T, \ U \) and \( E \)

   (c) Starting symbol: \( S \)

   (d) Production rules: \( S \rightarrow a S, \ S \rightarrow T, \ T \rightarrow b T, \ T \rightarrow U, \ U \rightarrow c U, \ U \rightarrow E, \ E \rightarrow \lambda. \)

5. Define the following languages over \( B \) with phrase structure grammars. Which of these languages are regular?
   
   (a) The strings with an odd number of characters.

   (b) The strings of length 4 or less.

   (c) The palindromes, strings that are the same backwards as forwards.

6. Define the following languages over \( B \) with phrase structure grammars. Which of these languages are regular?
   
   (a) The strings with more 0s than 1s.

   (b) The strings with an even number of 1s.

   (c) The strings for which all 0s precede all 1s.

7. Prove that if a language over \( A \) is recursive, then its complement is also recursive.

8. Use BNF to define the grammars in Exercises 3 and 4.
B Exercise

9. (a) Prove that if \( X_1, X_2, \ldots \) is a countable sequence of countable sets, the union of these sets, \( \bigcup_{i=1}^{\infty} X_i \), is countable.

(b) Using the fact that the countable union of countable sets is countable, prove that if \( A \) is countable, then \( A^\ast \) is countable.

14.3 Automata, Finite-State Machines

In this section, we will introduce the concept of an abstract machine. The machines we will examine will (in theory) be capable of performing many of the tasks associated with digital computers. One such task is solving the recognition problem for a language. We will concentrate on one class of machines, finite-state machines (finite automata). And we will see that they are precisely the machines that are capable of recognizing strings in a regular grammar.

Given an alphabet \( X \), we will imagine a string in \( X^\ast \) to be encoded on a tape that we will call an input tape. When we refer to a tape, we might imagine a strip of material that is divided into segments, each of which can contain either a letter or a blank.

The typical abstract machine includes an input device, the read head, which is capable of reading the symbol from the segment of the input tape that is currently in the read head. Some more advanced machines have a read/write head that can also write symbols onto the tape. The movement of the input tape after reading a symbol depends on the machine. With a finite-state machine, the next segment of the input tape is always moved into the read head after a symbol has been read. Most machines (including finite-state machines) also have a separate output tape that is written on with a write head. The output symbols come from an output alphabet, \( Z \), that may or may not be equal to the input alphabet. The most significant component of an abstract machine is its memory structure. This structure can range from a finite number of bits of memory (as in a finite-state machine) to an infinite amount of memory that can be sorted in the form of a tape that can be read from and written on (as in a Turing machine).

**Definition:** Finite-State Machine. A finite-state machine is defined by a quintet \( (S, X, Z, w, \ell) \) where

1. \( S = \{s_1, s_2, \ldots, s_n\} \) is the state set, a finite set that corresponds to the set of memory configurations that the machines can have at any time.
2. \( X = \{x_1, x_2, \ldots, x_m\} \) is the input alphabet.
3. \( Z = \{z_1, z_2, \ldots, z_k\} \) is the output alphabet.
4. \( w : X \times S \rightarrow Z \) is the output function, which specifies which output symbol \( w(x, s) \in Z \) is written onto the output tape when the machine is in state \( s \) and the input symbol \( x \) is read.
5. \( \ell : X \times S \rightarrow S \) is the next-state (or transition) function, which specifies which state \( \ell(x, s) \in S \) the machine should enter when it is in state \( s \) and it reads the symbol \( x \).

**Example 14.3.1.** Many mechanical devices, such as simple vending machines, can be thought of as finite-state machines. For simplicity, assume that a vending machine dispenses packets of gum, spearmint (\( S \)), peppermint (\( P \)), and bubble (\( B \)), for 25c each. We can define the input alphabet to be \( \{\text{deposit } 25c, \text{ press } S, \text{ press } P, \text{ press } B\} \) and the state set to be \( \{\text{Locked}, \text{ Select}\} \). The machine will not disburse gum unless the player deposits at least 25c. We will leave it to the reader to imagine what the output alphabet, output function, and next-state function would be. You are also invited to let your imagination run wild and include such features as a coin-return lever and change maker.

**Example 14.3.2.** The following machine is called a parity checker. It recognizes whether or not a string in \( B^\ast \) contains an even number of 1s. The memory structure of this machine reflects the fact that in order to check the parity of a string, we need only keep track of whether an odd or even number of 1s has been detected.

1. The input alphabet is \( B = \{0, 1\} \).
2. The output alphabet is also \( B \).
3. The state set is \( \{\text{even}, \text{odd}\} \).

(4, 5) The following table defines the output and next-state functions:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( s )</th>
<th>( w(x, s) )</th>
<th>( \ell(x, s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>even</td>
<td>0</td>
<td>even</td>
</tr>
<tr>
<td>0</td>
<td>odd</td>
<td>1</td>
<td>odd</td>
</tr>
<tr>
<td>1</td>
<td>even</td>
<td>1</td>
<td>odd</td>
</tr>
<tr>
<td>1</td>
<td>odd</td>
<td>0</td>
<td>even</td>
</tr>
</tbody>
</table>

Note how the value of the most recent output at any time is an indication of the current state of the machine. Therefore, if we start in the even state and read any finite input tape, the last output corresponds to the final state of the parity checker and tells us the parity of the string on the input tape. For example, if the string 11001010 is read from left to right, the output tape, also from left to right, will be 10001100. Since the last character is a 0, we know that the input string has even parity.

An alternate method for defining a finite-state machine is with a transition diagram. A **transition diagram** is a directed graph that contains a node for each state and edges that indicate the transition and output functions. An edge \( (s_i, s_j) \) that is labeled \( x/z \) indicates that in state \( s_i \) the input \( x \) results in an output of \( z \) and the next state is \( s_j \). That is, \( w(x, s_i) = z \) and \( \ell(x, s_i) = s_j \). The transition diagram for the parity checker
appears in Figure 14.3.1. In later examples, we will see that if different inputs, \( x_i \) and \( x_j \), while in the same state, result in the same transitions and outputs, we label a single edge \( x_i \), \( x_j/z \) instead of drawing two edges with labels \( x_i/z \) and \( x_j/z \).

One of the most significant features of a finite-state machine is that it retains no information about its past states that can be accessed by the machine itself. For example, after we input a tape encoded with the symbols 01101010 into the parity checker, the current state will be even, but we have no indication within the machine whether or not it has always been in even state. Note how the output tape is not considered part of the machine’s memory. In this case, the output tape does contain a "history" of the parity checker's past states. We assume that the finite-state machine has no way of recovering the output sequence for later use.

\[
\begin{array}{c}
\text{Odd} \quad \text{Even} \\
0/1 \quad 1/0 \quad 1/1 \quad 0/0
\end{array}
\]

**Figure 14.3.1** Transition Diagram for a parity checker

**Example 14.3.3.** Consider the following simplified version of the game of baseball. To be precise, this machine describes one half-inning of a simplified baseball game. Suppose that in addition to home plate, there is only one base instead of the usual three bases. Also, assume that there are only two outs per inning instead of the usual three. Our input alphabet will consist of the types of hits that the batter could have: out (O), double play (DP), single (S), and home run (HR). The input DP is meant to represent a batted ball that would result in a double play (two outs), if possible. The input DP can then occur at any time. The output alphabet is the numbers 0, 1, and 2 for the number of runs that can be scored as a result of any input. The state set contains the current situation in the inning, the number of outs, and whether a base runner is currently on the base. The list of possible states is then 00 (for 0 outs and 0 runners), 01, 10, 11, and end (when the half-inning is over). The transition diagram for this machine appears in Figure 14.3.2.

Let's concentrate on one state. If the current state is 01, 0 outs and 1 runner on base, each input results in a different combination of output and next-state. If the batter hits the ball poorly (a double play) the output is zero runs and the inning is over (the limit of two outs has been made). A simple out also results in an output of 0 runs and the next state is 11, one out and one runner on base. If the batter hits a single, one run scores (output = 1) while the state remains 01. If a home run is hit, two runs are scored (output = 2) and the next state is 00. If we had allowed three outs per inning, this graph would only be marginally more complicated. The usual game with three bases would be quite a bit more complicated, however.

\[
\begin{array}{c}
\text{Start} \\
\text{End}
\end{array}
\]

**Figure 14.3.2** Transition Diagram for a simplified game of baseball

**RECOGNITION IN REGULAR LANGUAGES**

As we mentioned at the outset of this section, finite-state machines can recognize strings in a regular language. Consider the language \( L \) over \( \{a, b, c\} \) that contains the strings of positive length in which each \( a \) is followed by \( b \) and each \( b \) is followed by \( c \). One such string is \( bcbabc \). This language is regular. A grammar for the language would be nonterminal symbols \( \{A, B, C\} \) with starting symbol \( C \) and production rules
A → bB, B → cC, C → aA, C → bC, C → cC and C → c. A finite-state machine (Figure 14.3.3) that recognizes this language can be constructed with one state for each nonterminal symbol and an additional state (Reject) that is entered if any invalid production takes place. At the end of an input tape that encodes a string in \( \{a, b, c\}^* \), we will know when the string belongs to \( L \) based on the final output. If the final output is 1, the string belongs to \( L \) and if it is 0, the string does not belong to \( L \). In addition, recognition can be accomplished by examining the final state of the machine. The input string belongs to the language if and only if the final state is \( C \).

The construction of this machine is quite easy: note how each production rule translates into an edge between states other than Reject. For example, \( C → bB \) indicates that in State \( C \), an input of \( b \) places the machine into State \( B \). Not all sets of production rules can be as easily translated to a finite-state machine. Another set of production rules for \( L \) is \( A → aB, B → bC, C → cA, C → cB, C → cC \) and \( C → c \). Techniques for constructing finite-state machines from production rules is not our objective here. Hence we will only expect you to experiment with production rules until appropriate ones are found.

![Figure 14.3.3](image)

**Example 14.3.4.** A finite-state machine can be designed to add positive integers of any size. Given two integers in binary form, \( a = a_n a_{n-1} \ldots a_1 a_0 \) and \( b = b_n b_{n-1} \ldots b_1 b_0 \), the machine will read the input sequence, which is obtained from the digits of \( a \) and \( b \) reading from right to left,

\[
a_0 b_0 (a_0 + 2 b_0) + \ldots + a_n b_n (a_n + 2 b_n),
\]

followed by the special input 111. Note how all possible inputs except the last one must even parity (contain an even number of ones). The output sequence is the sum of \( a \) and \( b \), starting with the units digit, and comes from the set \( \{0, 1, \lambda\} \). The transition diagram for this machine appears in Figure 14.3.4.

![Figure 14.3.4](image)

**EXERCISES FOR SECTION 14.3**

A Exercises

1. Draw a transition diagram for the vending machine described in Example
14.3.1.
2. Construct finite-state machines that recognize the regular languages that you identified in Section 14.2.
3. What is the input set for the machine in Example 14.3.4?
4. What input sequence would be used to compute the sum of 1101 and 0111 (binary integers)? What would the output sequence be?

B Exercise
5. The Gray Code Decoder. The finite-state machine defined by the following figure has an interesting connection with the Gray Code (Section 9.4).

![Gray Code Decoder Diagram](image)

Given a string \( x = x_1x_2 \cdots x_n \in B^n \), we may ask where \( x \) appears in \( G_n \). Starting in Copy state, the input string \( x \) will result in an output string \( z \in B^n \), which is the binary form of the position of \( x \) in \( G_n \). Positions are numbered from 0 to \( 2^n - 1 \).

(a) In what positions \( (0 \sim 31) \) do 0110, 00100, and 11111 appear in \( G_3 \)?

(b) Prove that the Gray Code Decoder always works.

### 14.4 The Monoid of a Finite-State Machine

In this section, we will see how every finite-state machine has a monoid associated with it. For any finite-state machine, the elements of its associated monoid correspond to certain input sequences. Because only a finite number of combinations of states and inputs is possible for a finite-state machine there is only a finite number of input sequences that summarize the machine. This idea is illustrated best with a few examples.

**Example 14.4.1.** Consider the parity checker. The following table summarizes the effect on the parity checker of strings in \( B^1 \) and \( B^2 \).

The row labeled "Even" contains the final state and final output as a result of each input string in \( B^1 \) and \( B^2 \) when the machine starts in the even state. Similarly, the row labeled "Odd" contains the same information for input sequences when the machine starts in the odd state.

<table>
<thead>
<tr>
<th>Input String</th>
<th>0</th>
<th>1</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Even</td>
<td>(Even, 0)</td>
<td>(Odd, 1)</td>
<td>(Even, 0)</td>
<td>(Odd, 1)</td>
<td>(Odd, 1)</td>
<td>(Even, 0)</td>
</tr>
<tr>
<td>Odd</td>
<td>(Odd, 1)</td>
<td>(Even, 1)</td>
<td>(Odd, 1)</td>
<td>(Even, 1)</td>
<td>(Even, 0)</td>
<td>(Odd, 1)</td>
</tr>
<tr>
<td>Same Effect</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note how, as indicated in the last row, the strings in \( B^2 \) have the same effect as certain strings in \( B^1 \). For this reason, we can summarize the machine in terms of how it is affected by strings of length 1. The actual monoid that we will now describe consists of a set of functions, and the operation on the functions will be based on the concatenation operation.

Let \( T_0 \) be the final effect (state and output) on the parity checker of the input 0. Similarly, \( T_1 \) is defined as the final effect on the parity checker of the input 1. More precisely,

\[
T_0(\text{even}) = (\text{even}, 0) \quad \text{and} \quad T_0(\text{odd}) = (\text{odd}, 1),
\]

while

\[
T_1(\text{even}) = (\text{odd}, 1) \quad \text{and} \quad T_1(\text{odd}) = (\text{even}, 0).
\]

In general, we define the operation on a set of such functions as follows: if \( s, t \) are input sequences and \( T_s \) and \( T_t \) are functions as above, then \( T_s \circ T_t = T_{st} \), that is, the result of the function that summarizes the effect on the machine by the concatenation of \( s \) with \( t \). Since, for example, 01 has the same effect on the parity checker as 1, \( T_0 \circ T_1 = T_{01} = T_1 \). We don’t stop our calculation at \( T_{01} \) because we want to use the shortest string of inputs to describe the final result. A complete table for the monoid of the parity checker is

\[
\begin{align*}
& s & T_0 & T_1 \\
& T_0 & T_0 & T_1 \\
& T_1 & T_1 & T_0
\end{align*}
\]

What is the identity of this monoid? The monoid of the parity checker is isomorphic to the monoid \([Z_2, +_2]\).

This operation may remind you of the composition operation on functions, but there are two principal differences. The domain of \( T_s \) is not the codomain of \( T_t \) and the functions are read from left to right unlike in composition, where they are normally read from right to left.

You may have noticed that the output of the parity checker echoes the state of the machine and that we could have looked only at the effect on the machine as the final state. The following example has the same property, hence we will only consider the final state.

**Example 14.4.2.** The transition diagram for the machine that recognizes strings in \( B^* \) that have no consecutive 1’s appears in Figure 14.4.1. Note how it is similar to the graph in Figure 9.1.1. Only a "reject state" has been added, for the case when an input of 1 occurs while
in State $a$. We construct a similar table to the one in the previous example to study the effect of certain strings on this machine. This time, we must include strings of length 3 before we recognize that no ‘new effects’ can be found.

![Diagram of a machine with states a, b, and s and transitions labeled 0 and 1 for inputs 'a' and 'b'.]

The following table summarizes how combinations of the strings 0, 1, 01, 10, and 11 affect this machine.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>0</th>
<th>1</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$01^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$001^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$010^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$101^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$011^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$110^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$111^*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results in this table can be obtained using the previous table. For example,

$$ T_{10} \ast T_{01} = T_{1001} = T_{10} \ast T_{1} = T_{101} = T_{1} $$

and

$$ T_{01} \ast T_{01} = T_{0101} = T_{0} \ast T_{1} = T_{0} T_{1} = T_{01} $$

Note that none of the elements that we have listed in this table serves as the identity for our operation. This problem can always be remedied by including the function that corresponds to the input of the null string, $T_X$. Since the null string is the identity for concatenation of strings, $T_{0} T_{X} = T_{X} T_{0} = T_{X}$ for all input strings $s$.

**Example 14.4.3.** A finite-state machine called the unit-time delay machine does not echo its current state, but prints its previous state. For this reason, when we find the monoid of the unit-time delay machine, we must consider both state and output. The transition diagram of this machine appears in Figure 14.4.2.

![Diagram of a unit-time delay machine with states 0 and 1 and transitions labeled 0 and 1 for inputs '0' and '1'.]

<table>
<thead>
<tr>
<th>Input</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>(0, 0)</td>
<td>(1, 0)</td>
<td>(0, 0)</td>
<td>(0, 1)</td>
<td>(0, 0)</td>
<td>(1, 1)</td>
<td>(0, 0)</td>
<td>(1, 1)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>$1$</td>
<td>(0, 1)</td>
<td>(1, 1)</td>
<td>(0, 0)</td>
<td>(1, 0)</td>
<td>(0, 1)</td>
<td>(1, 1)</td>
<td>(0, 0)</td>
<td>(1, 1)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>Same as</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Again, since no new outcomes were obtained from strings of length 3, only strings of length 2 or less contribute to the monoid of the machine. The table for the strings of positive length shows that we must add $T_X$ to obtain a monoid.
**Chapter 14 - Monoids and Automata**

### EXERCISES FOR SECTION 14.4

**A Exercise**

1. For each of the transition diagrams in Figure 14.4.3, write out tables for their associated monoids. Identify the identity in terms of a string of positive length, if possible. *(Hint: Where the output echoes the current state, the output can be ignored.)*

![Diagram](a)

![Diagram](b)

Figure 14.4.3

**B Exercise**

2. What common monoids are isomorphic to the monoids obtained in the previous exercise?

**C Exercise**

3. Can two finite-state machines with nonisomorphic transition diagrams have isomorphic monoids?

### 14.5 The Machine of a Monoid

Any finite monoid \([M, \ast]\) can be represented in the form of a finite-state machine with input and state sets equal to \(M\). The output of the machine will be ignored here, since it would echo the current state of the machine. Machines of this type are called state machines. It can be shown that whatever can be done with a finite-state machine can be done with a state machine; however, there is a trade-off. Usually, state machines that perform a specific function are more complex than general finite-state machines.

**Definition:** Machine of a Monoid. If \([M, \ast]\) is a finite monoid, then the machine of \(M\), denoted \(m(M)\), is the state machine with state set \(M\), input set \(M\), and next-state function \(t : M \times M \rightarrow M\) defined by \(t(s, x) = s \ast x\).

**Example 14.5.1.** We will construct the machine of the monoid \([\mathbb{Z}_2, +_2]\). As mentioned above, the state set and the input set are both \(\mathbb{Z}_2\). The next state function is defined by \(t(s, x) = s +_2 x\). The transition diagram for \(m(\mathbb{Z}_2)\) appears in Figure 14.5.1. Note how it is identical to the transition diagram of the parity checker, which has an associated monoid that was isomorphic to \([\mathbb{Z}_2, +_2]\).

![Diagram](14.5.1)

**Example 14.5.2.** The transition diagram of the monoids \([\mathbb{Z}_2, \times_2]\) and \([\mathbb{Z}_3, \times_3]\) appear in Figure 14.5.2.
Chapter 14 - Monoids and Automata

Example 14.5.3. Let $U$ be the monoid that we obtained from the unit-time delay machine (Example 14.4.3). We have seen that the machine of the monoid of the parity checker is essentially the parity checker. Will we obtain a unit-time delay machine when we construct the machine of $U$? We can't expect to get exactly the same machine because the unit-time delay machine is not a state machine and the machine of a monoid is a state machine. However, we will see that our new machine is capable of telling us what input was received in the previous time period. The operation table for the monoid serves as a table to define the transition function for the machine. The row headings are the state values, while the column headings are the inputs. If we were to draw a transition diagram with all possible inputs, the diagram would be too difficult to read. Since $U$ is generated by the two elements, $T_0$ and $T_1$, we will include only those inputs. Suppose that we wanted to read the transition function for the input $T_{01}$. Since $T_{01} = T_0 T_1$, in any state $s$, $t(s, T_{01}) = t(t(s, T_0), T_1)$. The transition diagram appears in Figure 14.5.3.

![Transition Diagram](image)

Figure 14.5.3

If we start reading a string of 0s and 1s while in state $T_0$, and are in state $T_{00}$ at any one time, the input from the previous time period (not the input that sent us into $T_{00}$, the one before that) is $a$. In states $T_0$, $T_0$, and $T_1$, no previous input exists.

EXERCISES FOR SECTION 14.5

A Exercise

1. Draw the transition diagrams for the machines of the following monoids:
   
   (a) $[\mathbb{Z}_4; +_4]$
   
   (b) The direct product of $[\mathbb{Z}_2; \times_2]$ with itself.
B Exercise
2. Even though a monoid may be infinite, we can visualize it as an infinite-state machine provided that it is generated by a finite number of elements. For example, the monoid $B^*$ is generated by 0 and 1. A section of its transition diagram can be obtained by allowing input only from the generating set (Figure 14.5.4a). The monoid of integers under addition is generated by the set $\{-1, 1\}$. The transition diagram for this monoid can be visualized by drawing a small portion of it, as in Figure 14.5.4b.

(a) Draw a transition diagram for $\{a, b, c\}^*$.  
(b) Draw a transition diagram for $[\mathbb{Z} \times \mathbb{Z}, \text{componentwise addition}]$. 

![Diagram](image-url)