Formal Languages, Automata and Codes

Oleg Gutik

Lecture 3

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

1.3 SOME APPLICATIONS

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this lecture, we present some simple examples to give the student some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lowercase letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle \mid \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle \mid \langle \text{digit} \rangle \langle \text{rest} \rangle \mid \langle \text{undrscr} \rangle \langle \text{rest} \rangle \mid \lambda$$

$$\langle \text{letter} \rangle \to a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z$$

$$\langle \text{undrscr} \rangle \to \_$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b| \ldots |z|A|B| \ldots |Z$$

$$\langle \text{undrscr} \rangle \to \_$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b| \ldots |z|A|B| \ldots |Z$$

$$\langle \text{undrscr} \rangle \to \_$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a | b | \ldots | z | A | B | \ldots | Z$$

$$\langle \text{undrscr} \rangle \to \_$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a | b | \ldots | z | A | B | \ldots | Z$$

$$\langle \text{undrscr} \rangle \to \_$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a | b | \ldots | z | A | B | \ldots | Z$$

$$\langle \text{undrscr} \rangle \to \_$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a | b | \ldots | z | A | B | \ldots | Z$$

$$\langle \text{undrscr} \rangle \to \_$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a | b | \ldots | z | A | B | \ldots | Z$$

$$\langle \text{undrscr} \rangle \to \_$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b| \ldots |z|A|B| \ldots |Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b|\ldots|z|A|B|\ldots|Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b|\ldots|z|A|B|\ldots|Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b| \ldots |z|A|B| \ldots |Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b| \ldots |z|A|B| \ldots |Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a | b | \ldots | z | A | B | \ldots | Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b| \ldots |z|A|B| \ldots |Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b| \ldots |z|A|B| \ldots |Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15

The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

Formally, these rules can be described by a grammar.

$$\langle \text{id} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle$$

$$\langle \text{rest} \rangle \to \langle \text{letter} \rangle \langle \text{rest} \rangle | \langle \text{digit} \rangle \langle \text{rest} \rangle | \langle \text{undrscr} \rangle \langle \text{rest} \rangle | \lambda$$

$$\langle \text{letter} \rangle \to a|b| \ldots |z|A|B| \ldots |Z$$

$$\langle \text{undrscr} \rangle \to -$$

In this grammar, the variables are $\langle \text{id} \rangle$, $\langle \text{letter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{undrscr} \rangle$, and $\langle \text{rest} \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle \text{id} \rangle &\Rightarrow \langle \text{letter} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{rest} \rangle \\ &\Rightarrow a \langle \text{digit} \rangle \langle \text{rest} \rangle \\ &\Rightarrow a0 \langle \text{rest} \rangle \\ &\Rightarrow a0. \end{aligned}$$

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

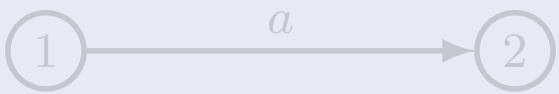
An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

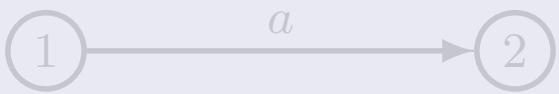


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

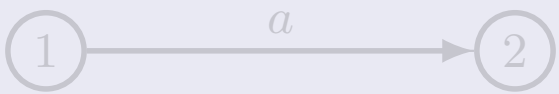


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

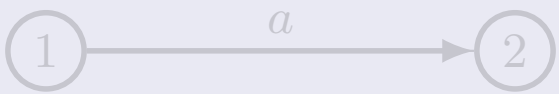


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

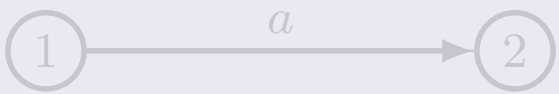


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

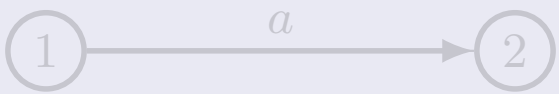


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

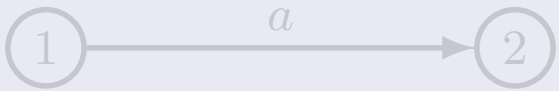


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

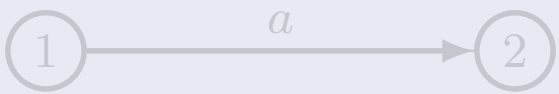


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

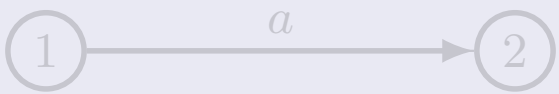


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

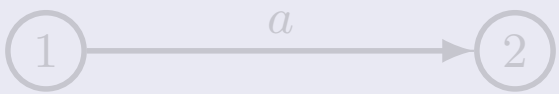


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

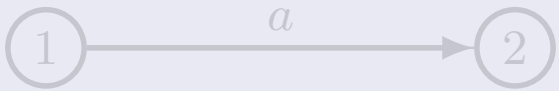


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

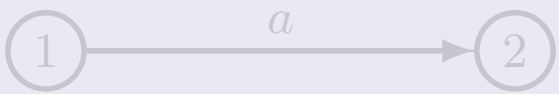


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

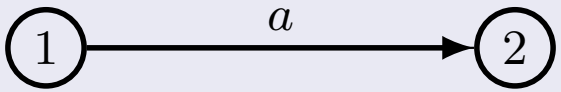


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

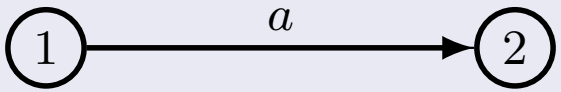


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.

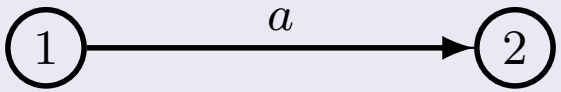


With this intuitive picture in mind, let us look at another way of describing C identifiers.

Example 1.15 (continuation)

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We shall do this shortly; for the moment, let us proceed in a more intuitive way.

An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, the following Figure represents a transition from State 1 to State 2, which is taken when the input symbol is a.



With this intuitive picture in mind, let us look at another way of describing C identifiers. ■

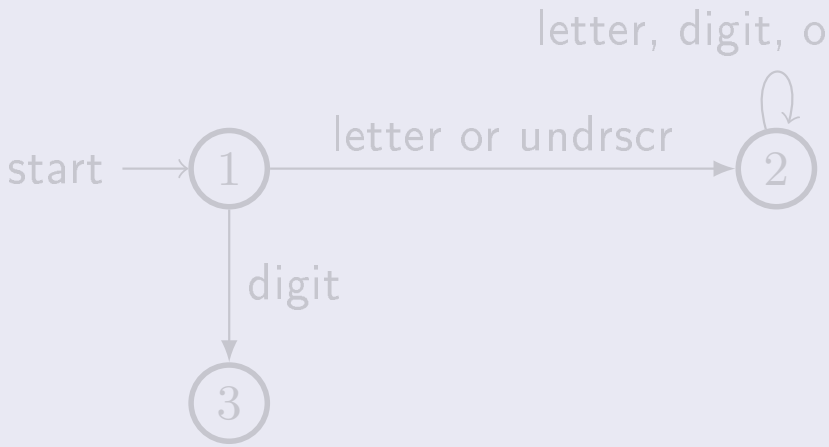
Example 1.16

The following Figure is an automaton that accepts all legal C identifiers.

We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

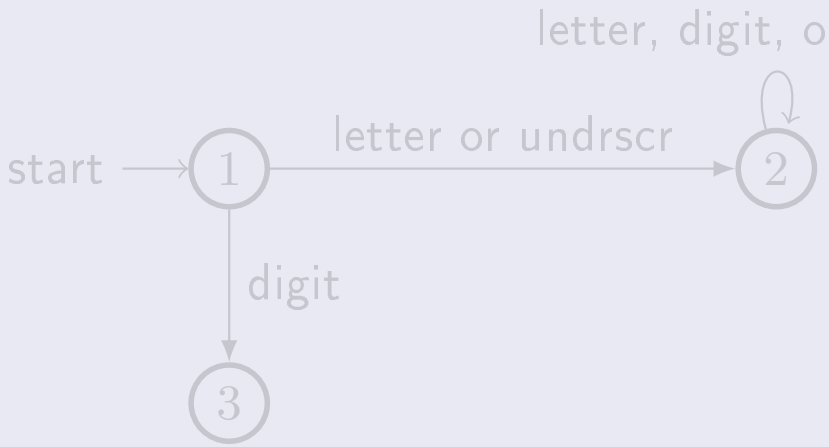
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

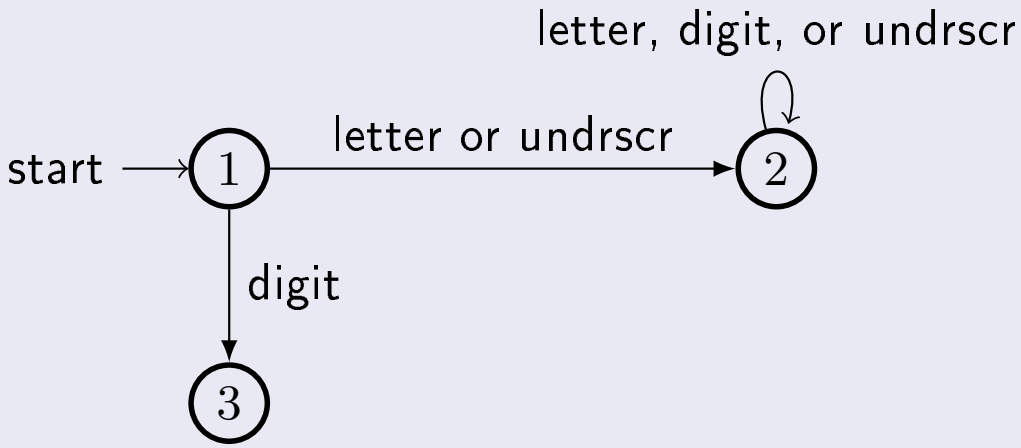
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

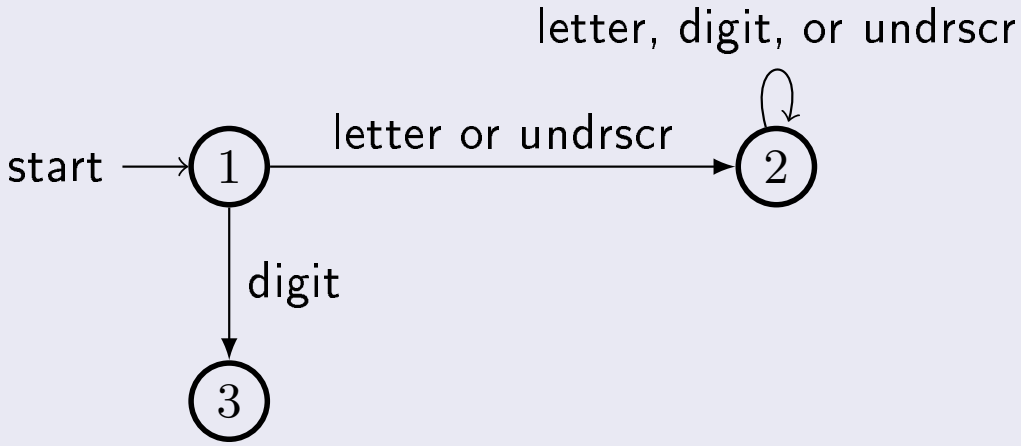
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

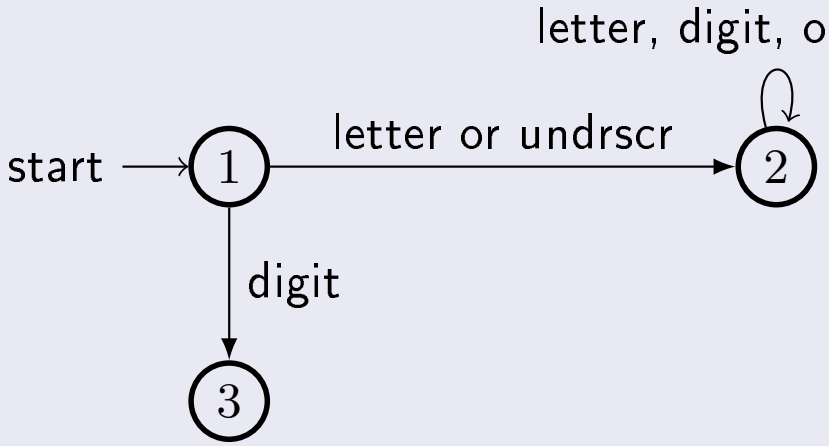
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

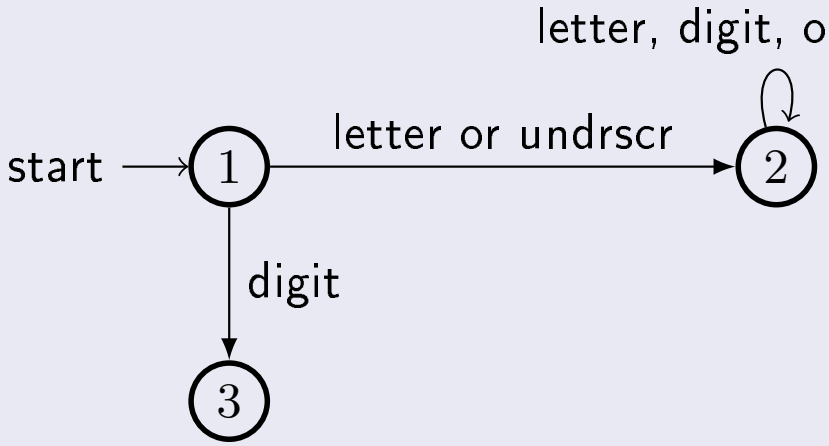
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

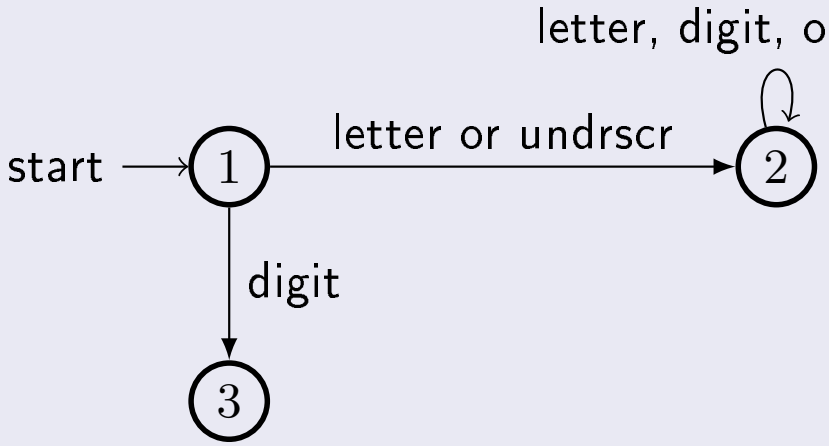
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

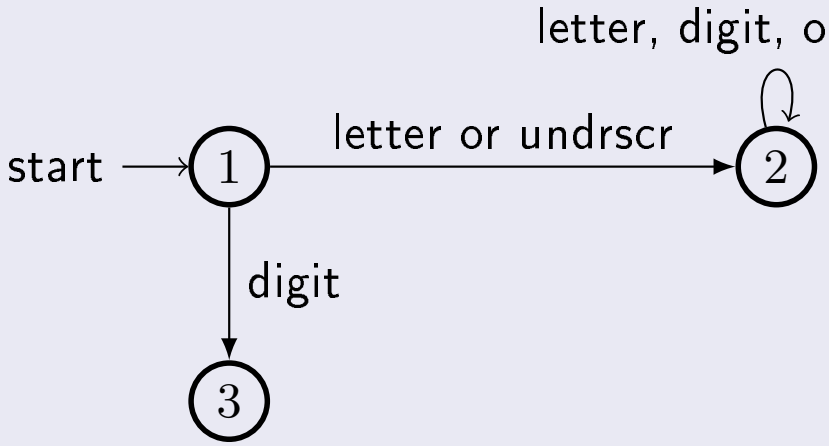
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

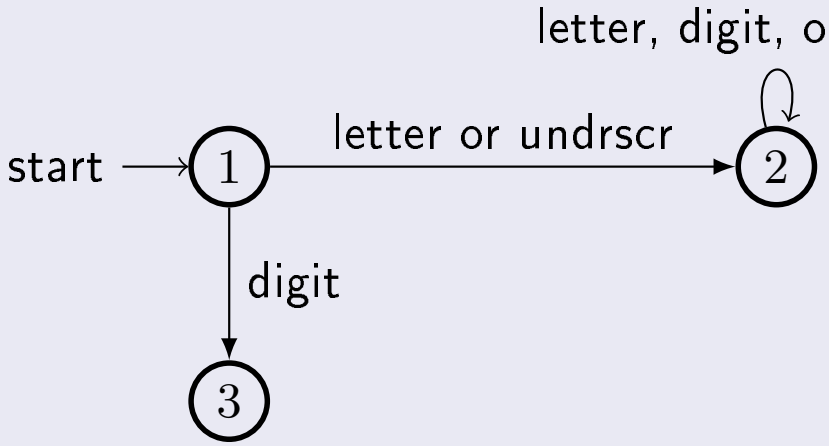
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

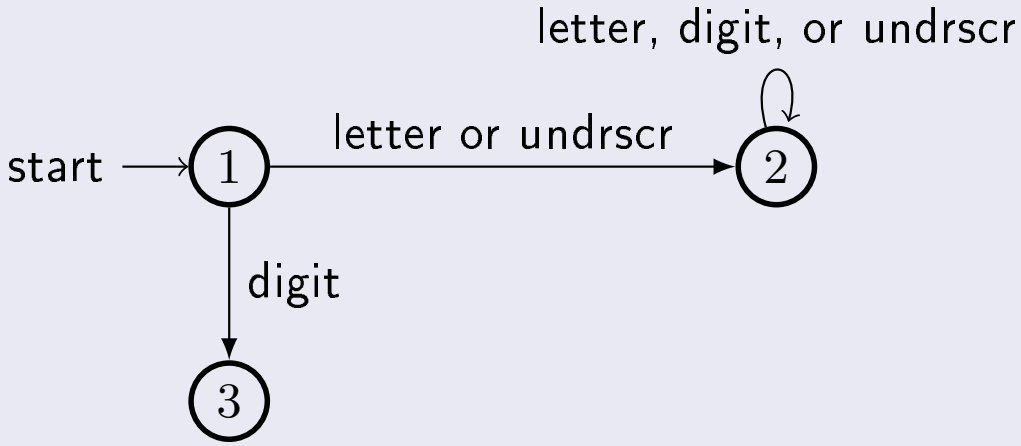
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

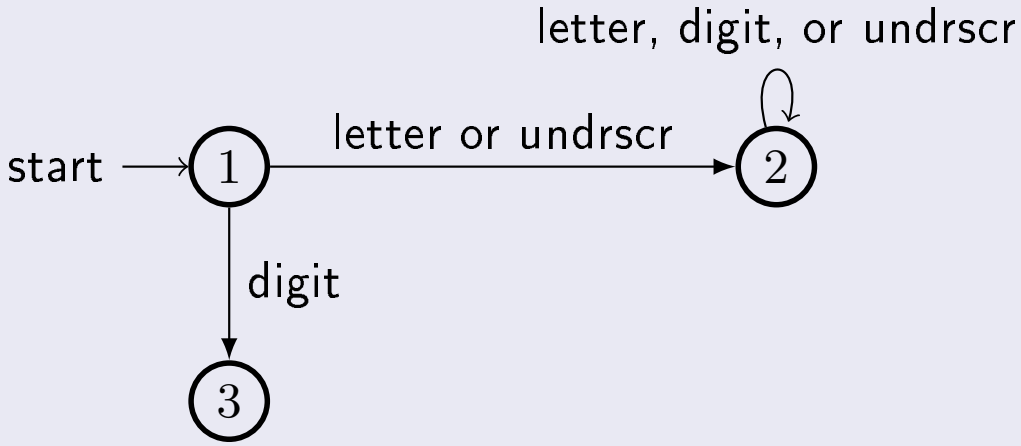
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

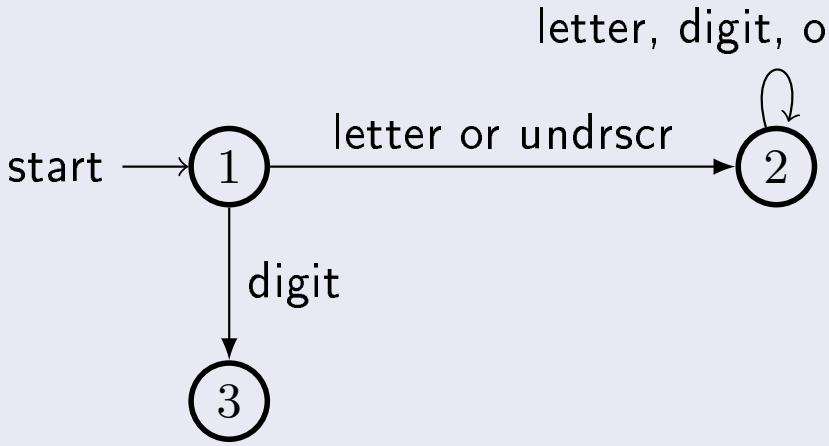
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

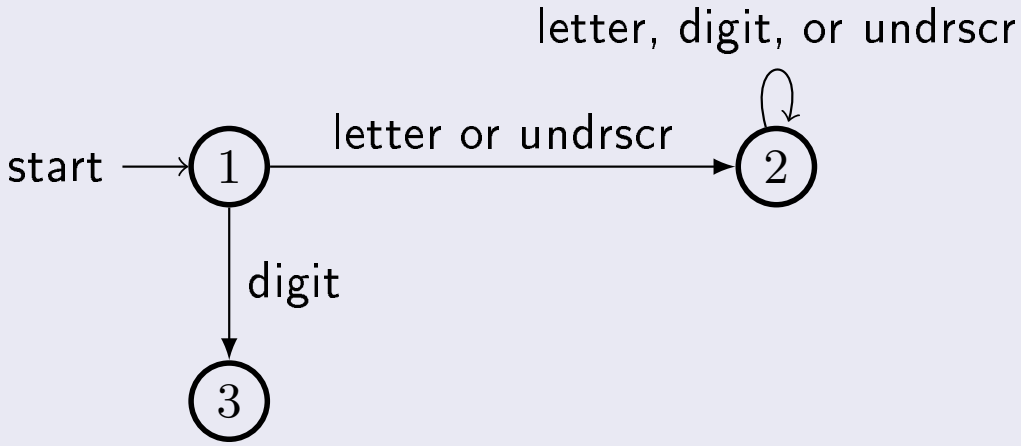
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

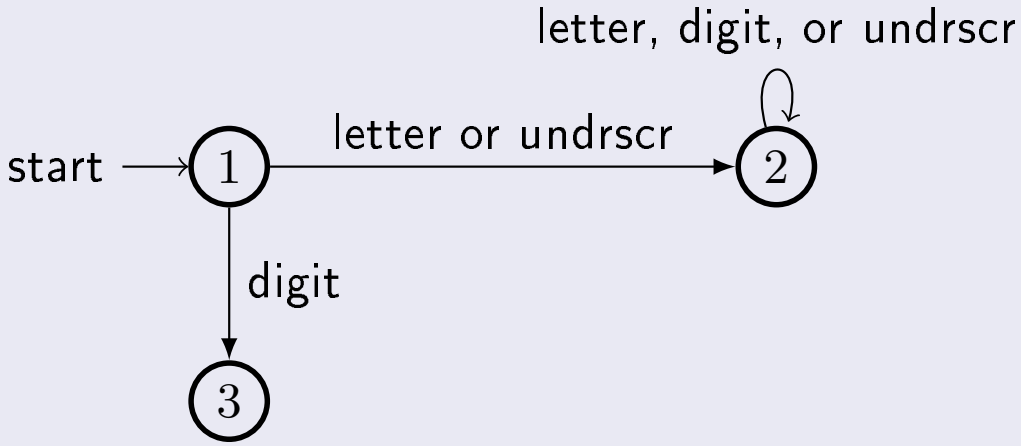
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

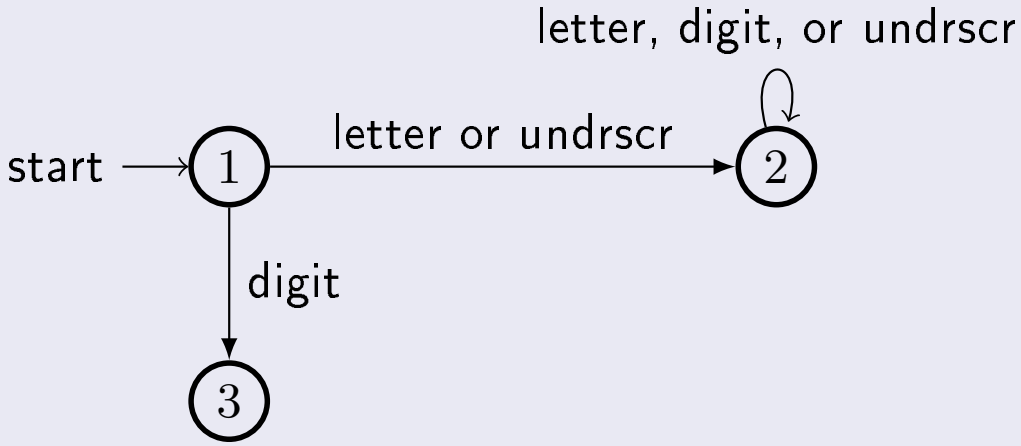
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Example 1.16

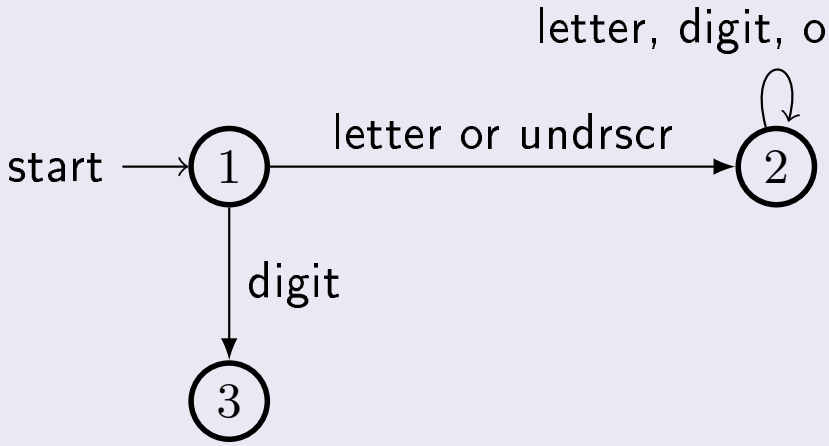
The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible.

Example 1.16

The following Figure is an automaton that accepts all legal C identifiers.



We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible. ■

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples.

Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

The following example previews transducers.

Example 1.17

A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

Example 1.17 (continuation)

A serial adder processes two such numbers $x = a_0 a_1 \cdots a_n$, and $y = b_0 b_1 \cdots b_n$, bit by bit, starting at the left end. Each bit addition creates a digit for the sum as well as a carry digit for the next higher position. A binary addition table (see Figure below) summarizes the process.

a_i \ b_i	0	1
0	0 No carry	1 No carry
1	1 No carry	0 Carry

Example 1.17 (continuation)

A serial adder processes two such numbers $x = a_0 a_1 \cdots a_n$, and $y = b_0 b_1 \cdots b_n$, bit by bit, starting at the left end. Each bit addition creates a digit for the sum as well as a carry digit for the next higher position. A binary addition table (see Figure below) summarizes the process.

	b_i = 0	b_i = 1
a_i = 0	0 No carry	1 No carry
a_i = 1	1 No carry	0 Carry

Example 1.17 (continuation)

A serial adder processes two such numbers $x = a_0 a_1 \cdots a_n$, and $y = b_0 b_1 \cdots b_n$, bit by bit, starting at the left end. Each bit addition creates a digit for the sum as well as a carry digit for the next higher position. A binary addition table (see Figure below) summarizes the process.

a_i \ b_i	0	1
0	0 No carry	1 No carry
1	1 No carry	0 Carry

Example 1.17 (continuation)

A serial adder processes two such numbers $x = a_0 a_1 \cdots a_n$, and $y = b_0 b_1 \cdots b_n$, bit by bit, starting at the left end. Each bit addition creates a digit for the sum as well as a carry digit for the next higher position. A binary addition table (see Figure below) summarizes the process.

a_i \ b_i	0	1
0	0 No carry	1 No carry
1	1 No carry	0 Carry

Example 1.17 (continuation)

A serial adder processes two such numbers $x = a_0 a_1 \cdots a_n$, and $y = b_0 b_1 \cdots b_n$, bit by bit, starting at the left end. Each bit addition creates a digit for the sum as well as a carry digit for the next higher position. A binary addition table (see Figure below) summarizes the process.

a_i \ b_i	0	1
0	0 No carry	1 No carry
1	1 No carry	0 Carry

Example 1.17 (continuation)

A serial adder processes two such numbers $x = a_0 a_1 \cdots a_n$, and $y = b_0 b_1 \cdots b_n$, bit by bit, starting at the left end. Each bit addition creates a digit for the sum as well as a carry digit for the next higher position. A binary addition table (see Figure below) summarizes the process.

a_i \ b_i	0	1
0	0 No carry	1 No carry
1	1 No carry	0 Carry

Example 1.17 (continuation)

A block diagram of the kind we saw when we first studied computers is given in the following Figure.

It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

Example 1.17 (continuation)

A block diagram of the kind we saw when we first studied computers is given in the following Figure.

It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

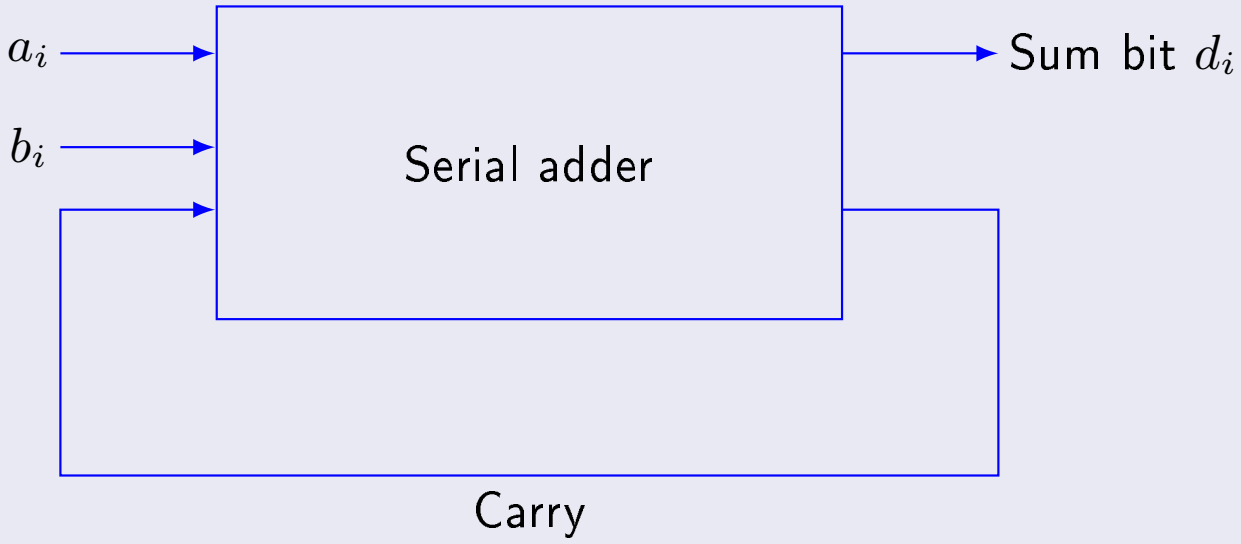
Example 1.17 (continuation)

A block diagram of the kind we saw when we first studied computers is given in the following Figure.

It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

Example 1.17 (continuation)

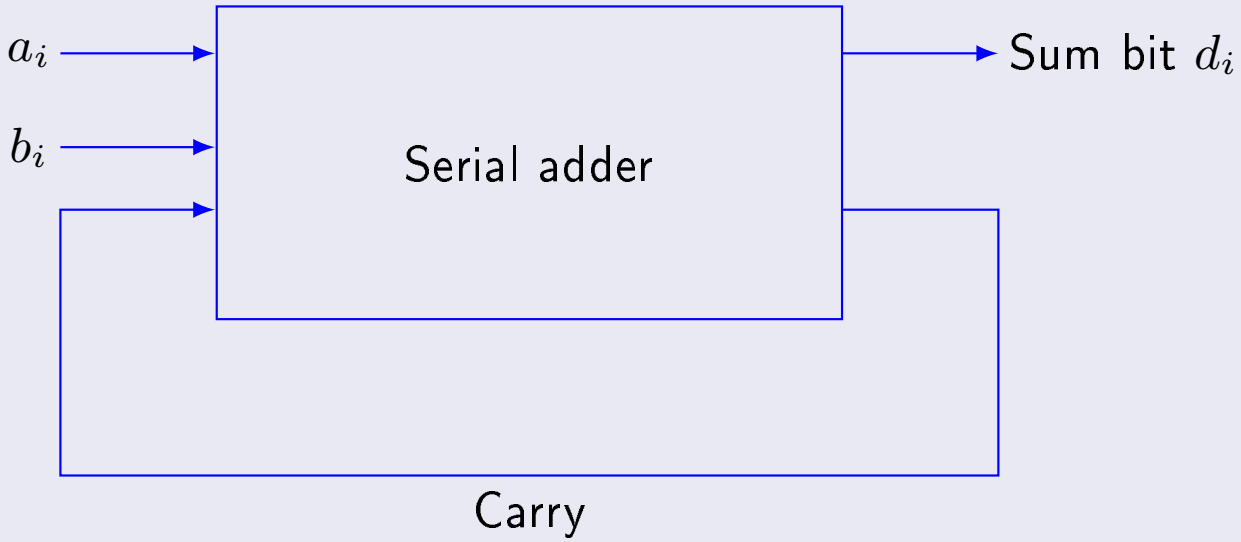
A block diagram of the kind we saw when we first studied computers is given in the following Figure.



It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

Example 1.17 (continuation)

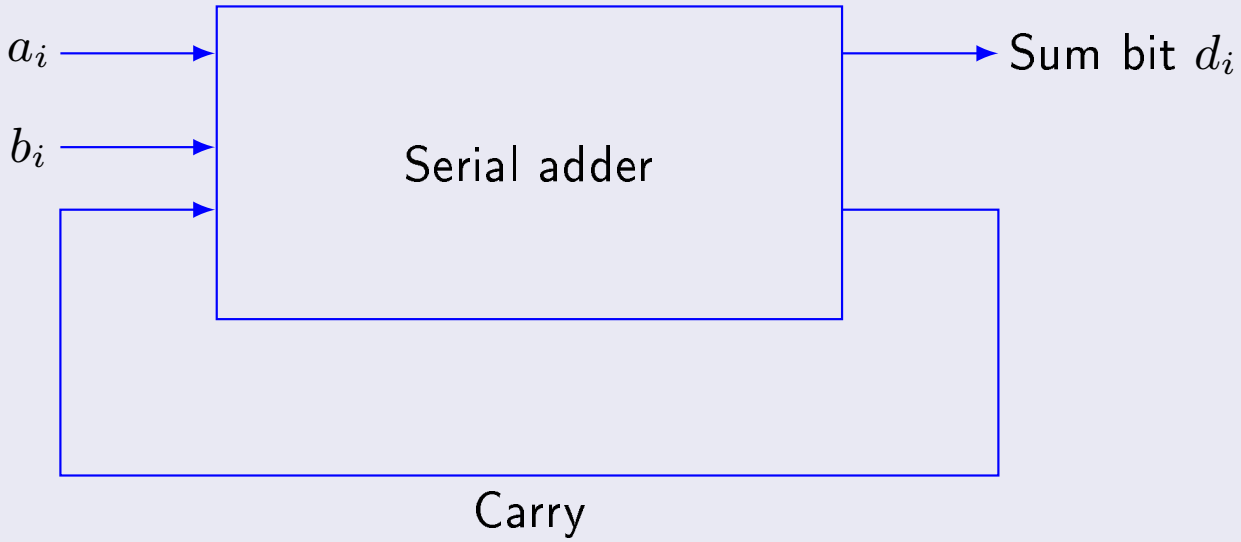
A block diagram of the kind we saw when we first studied computers is given in the following Figure.



It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

Example 1.17 (continuation)

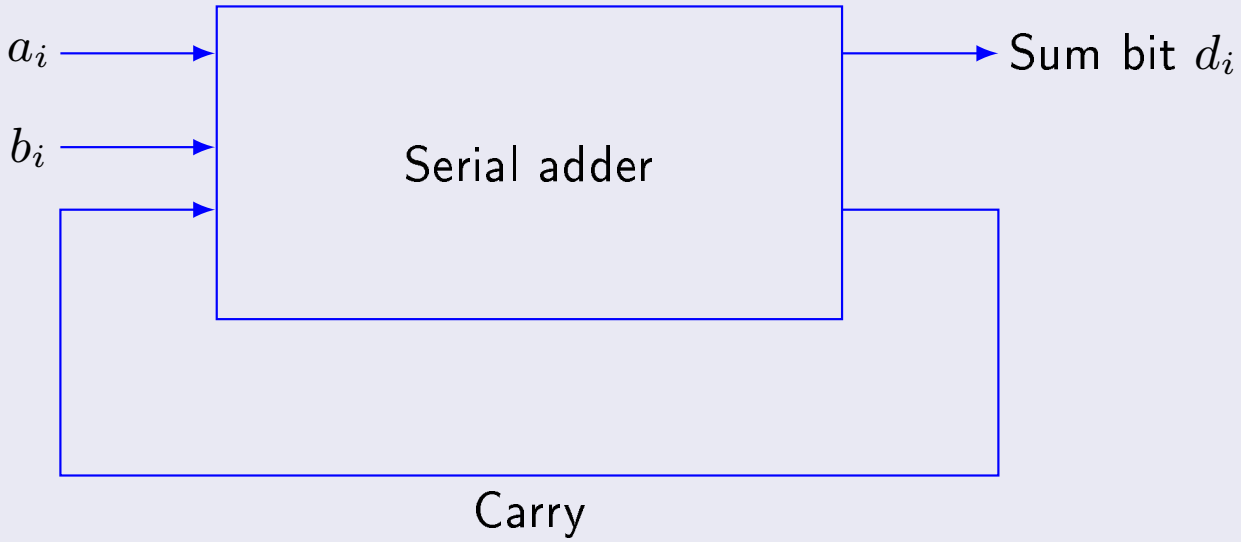
A block diagram of the kind we saw when we first studied computers is given in the following Figure.



It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

Example 1.17 (continuation)

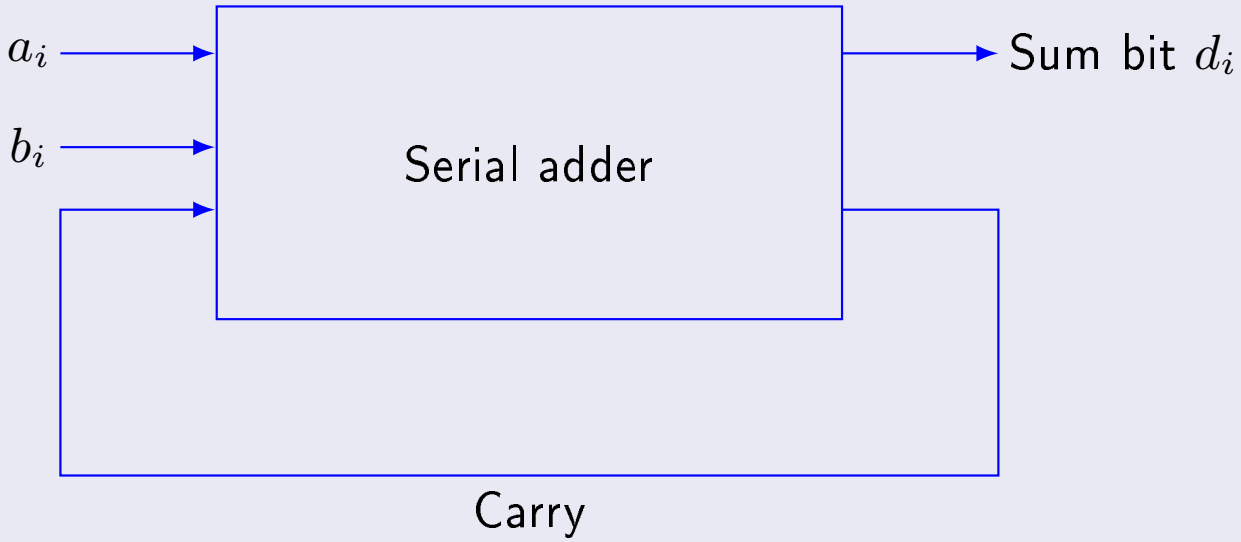
A block diagram of the kind we saw when we first studied computers is given in the following Figure.



It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

Example 1.17 (continuation)

A block diagram of the kind we saw when we first studied computers is given in the following Figure.



It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

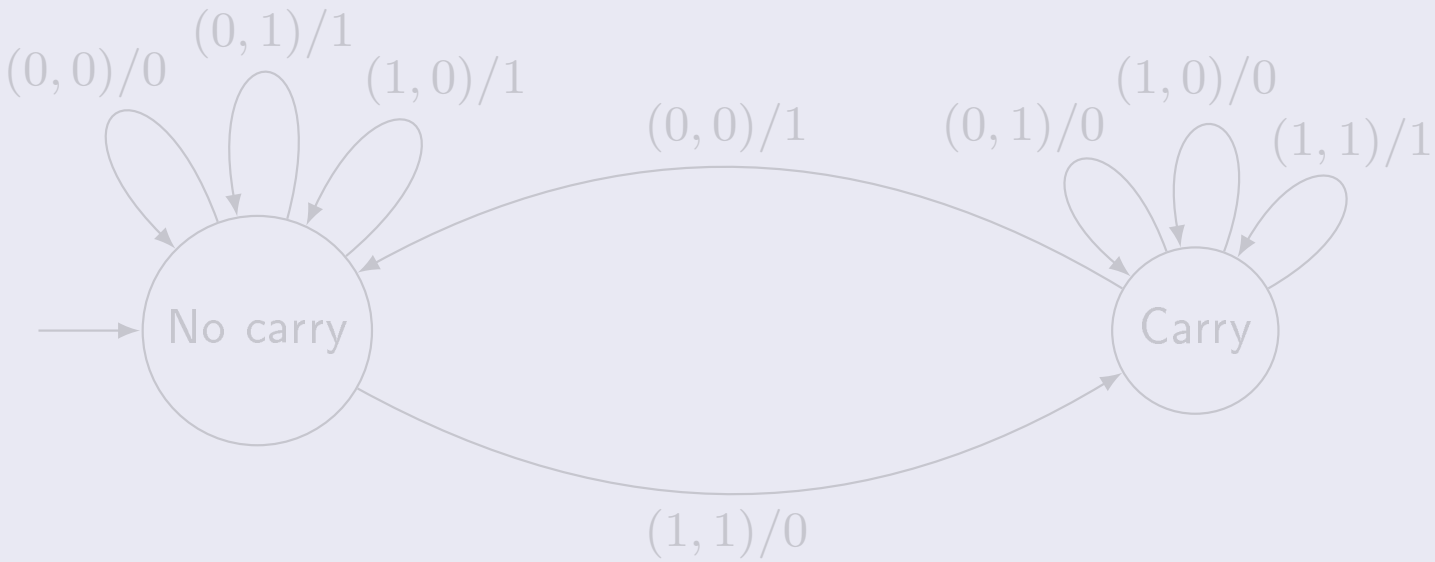
Example 1.17 (continuation)

The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.

Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

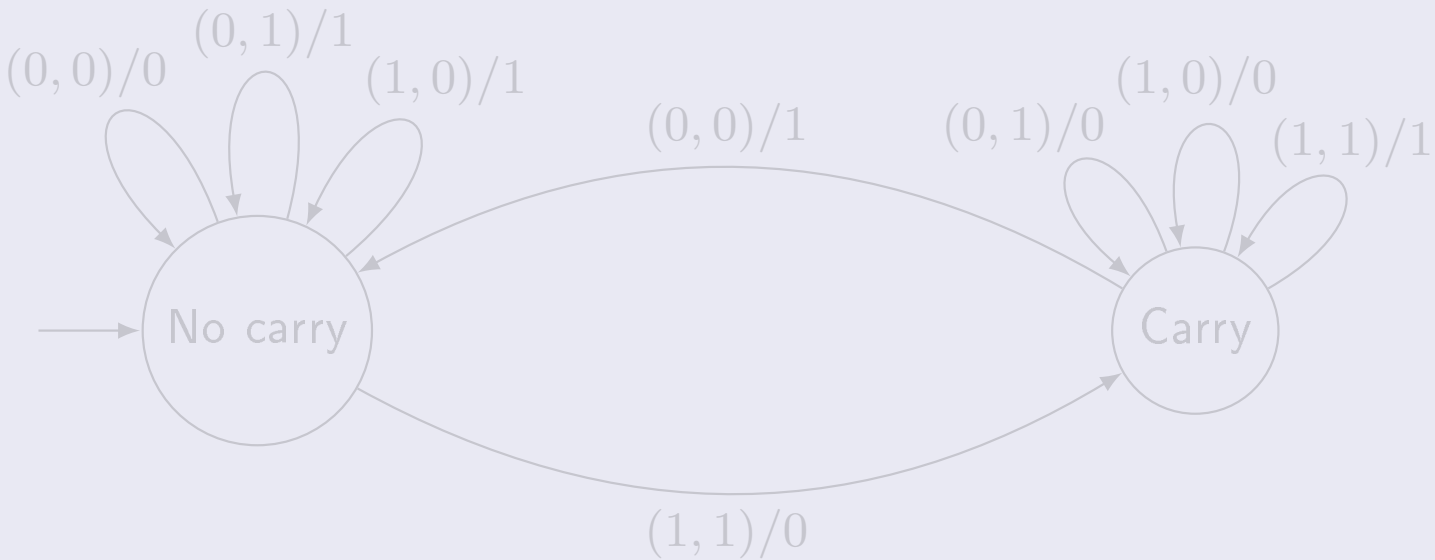
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

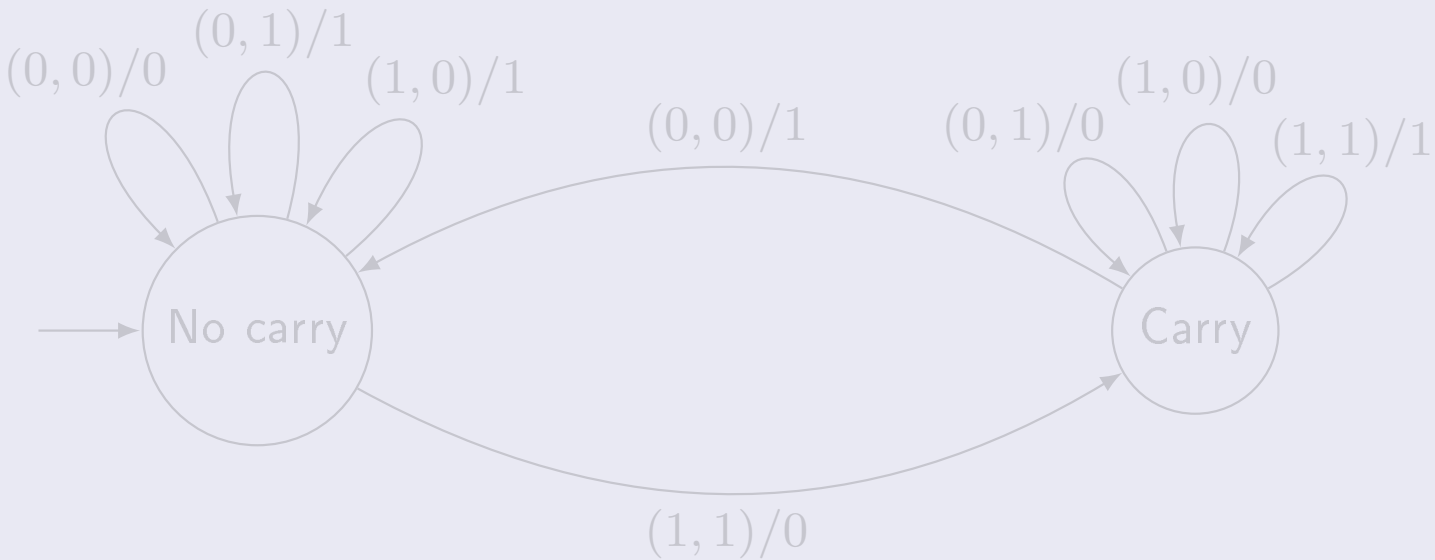
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

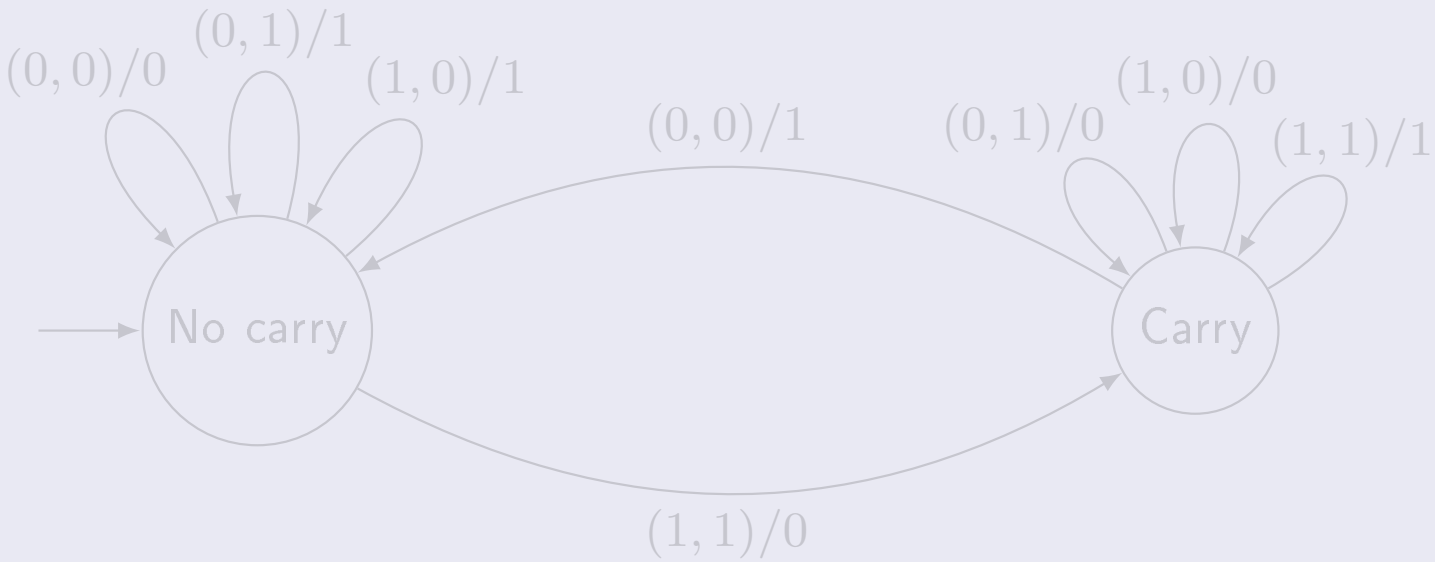
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

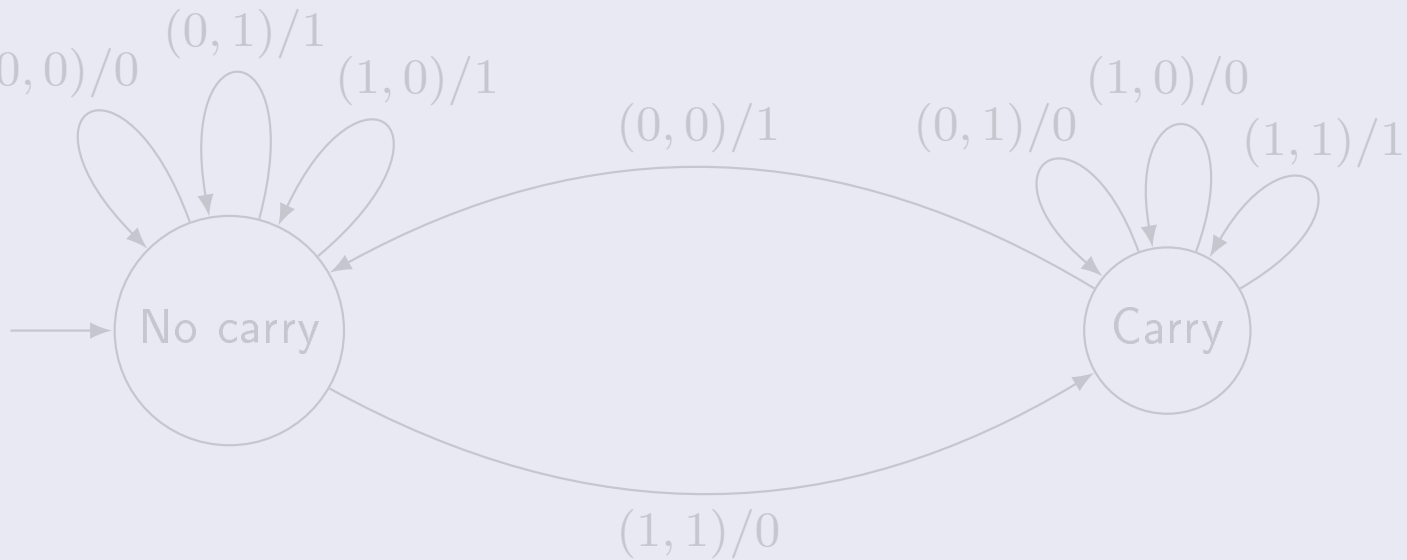
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

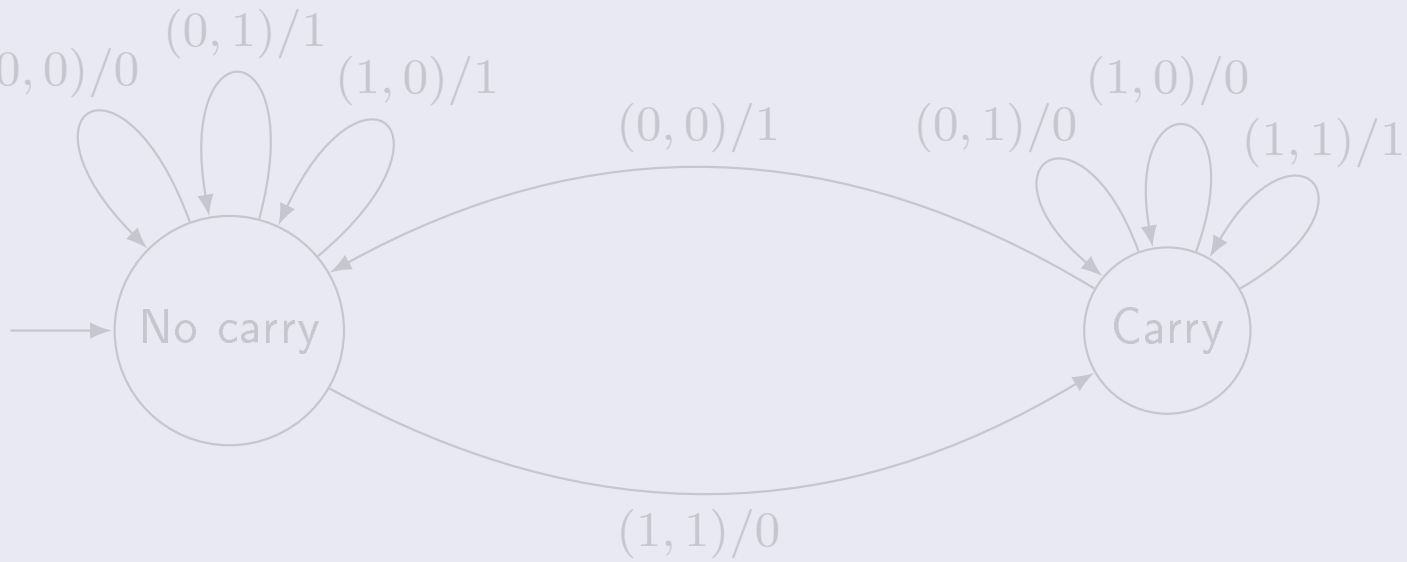
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

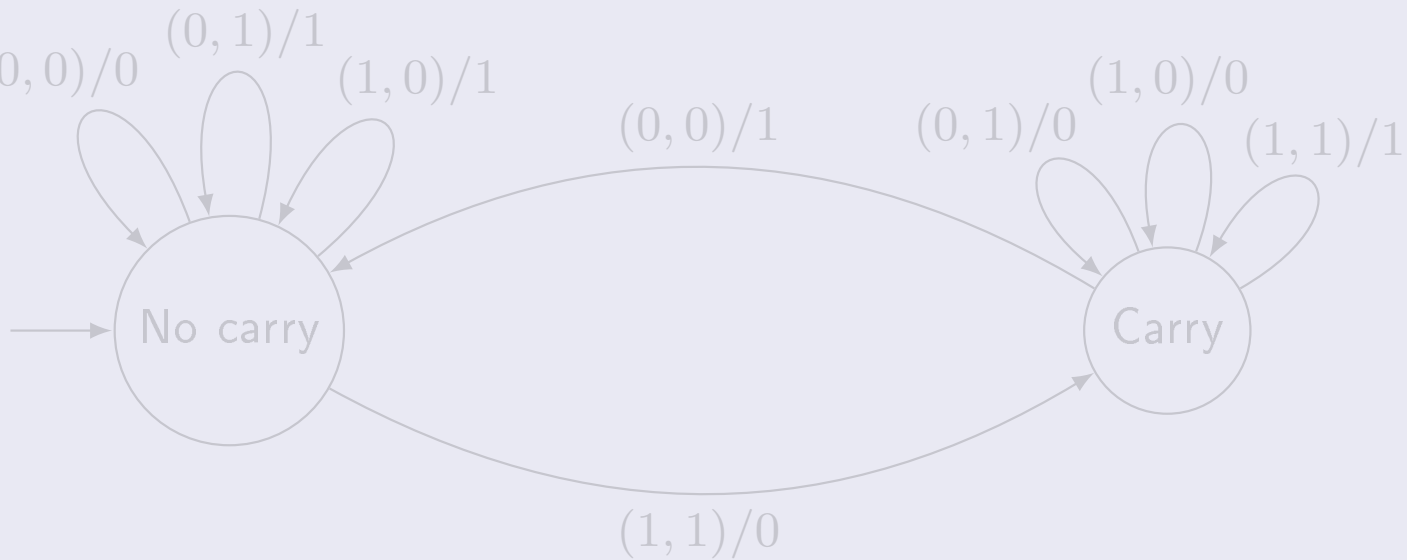
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

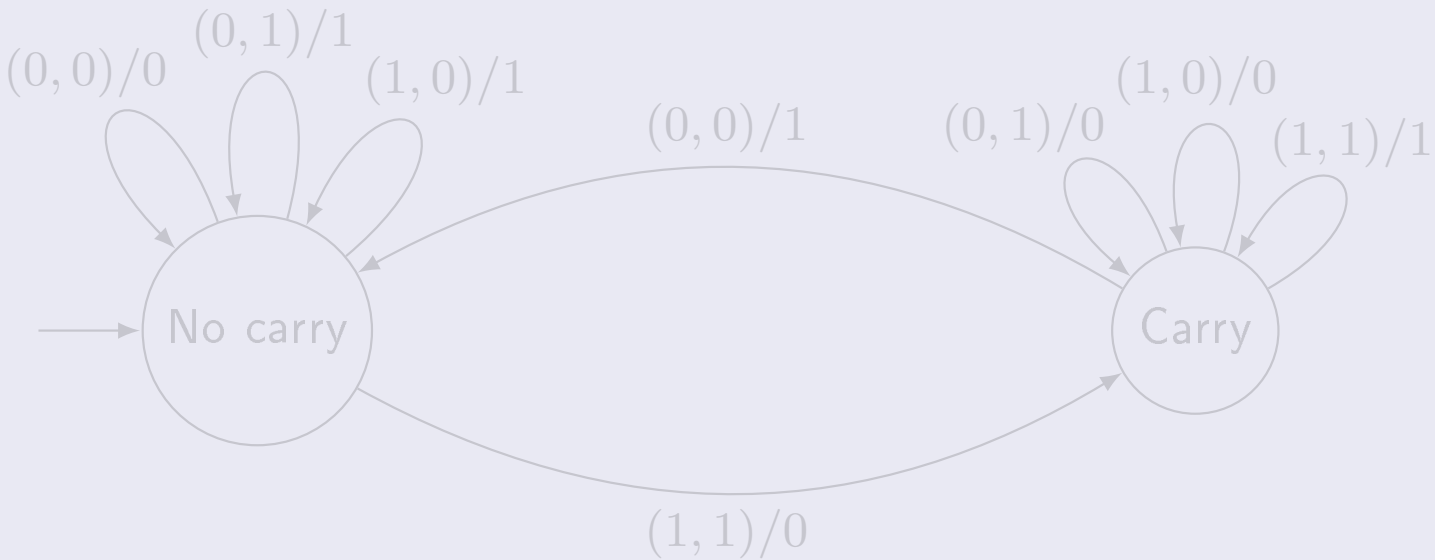
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

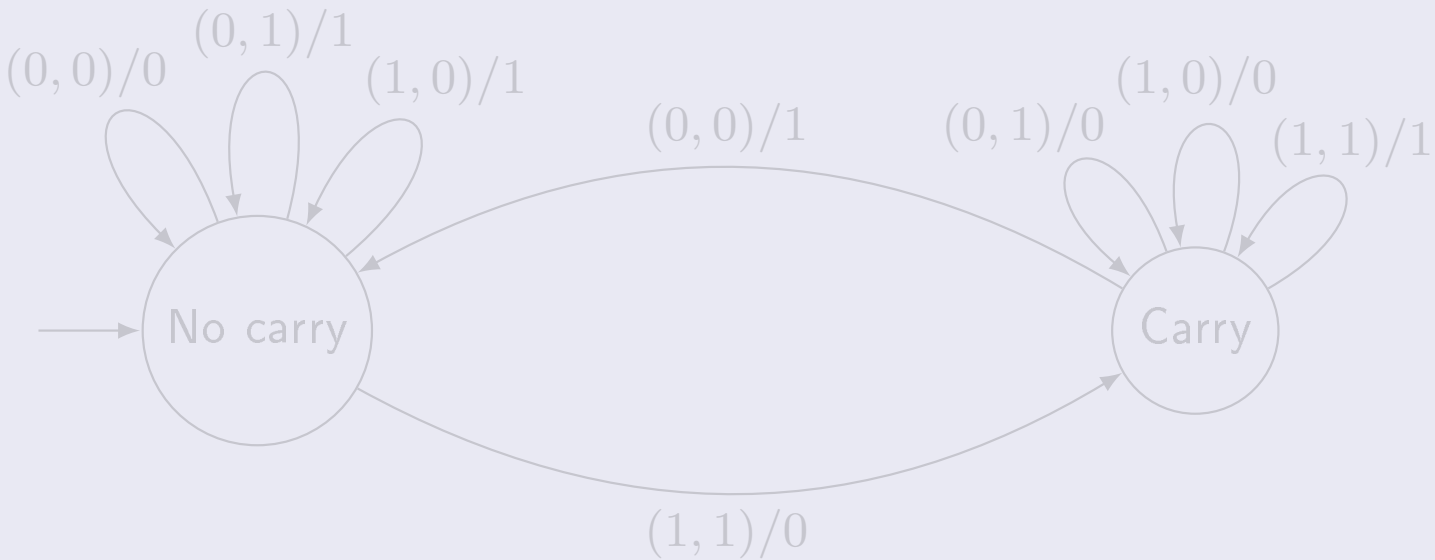
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

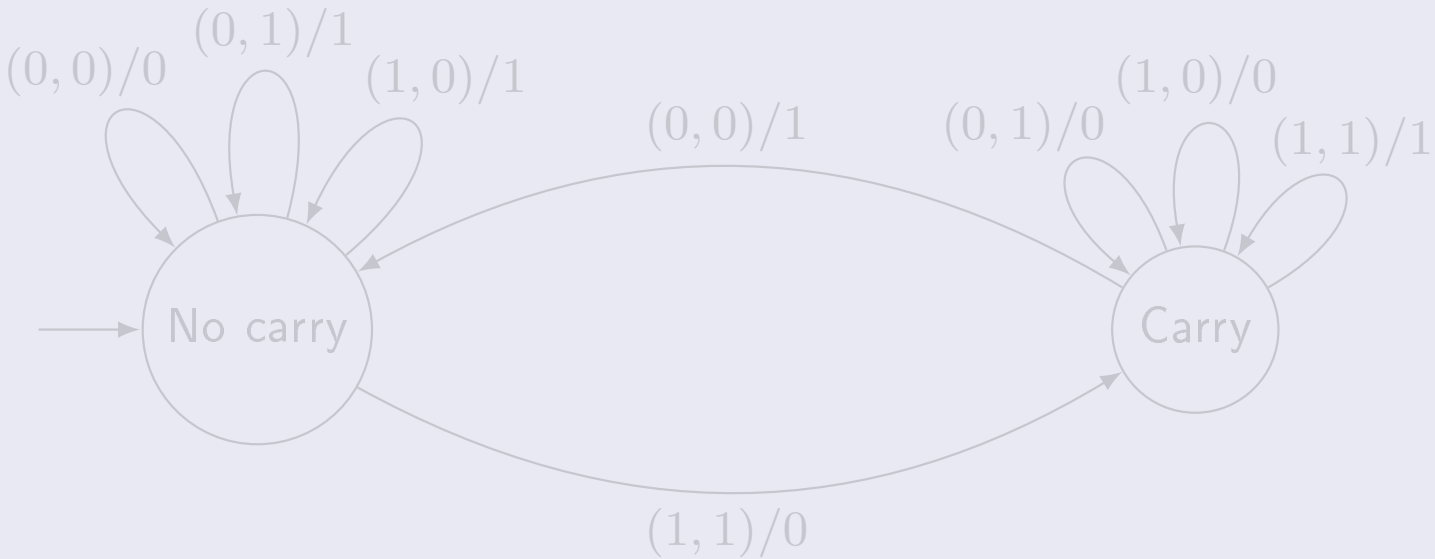
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

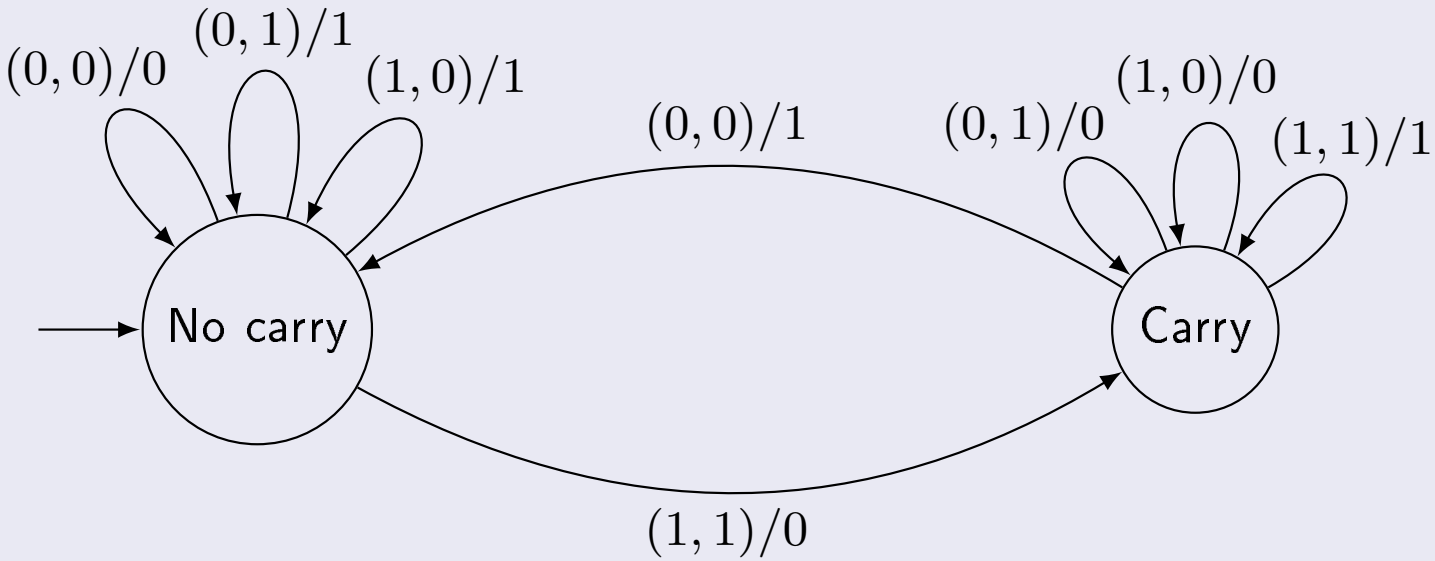
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

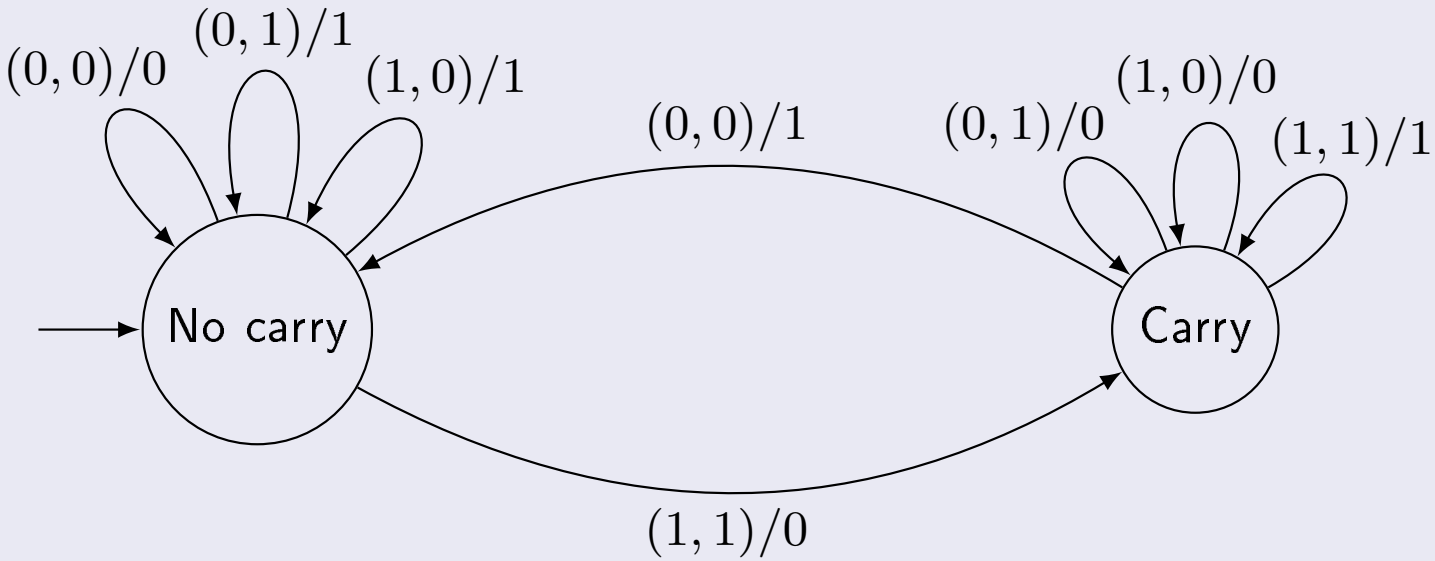
The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

The input to the transducer are the bit pairs (a_i, b_i); the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry". Initially, the transducer will be in state "no carry". It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in the Figure.



Follow this through with a few examples to convince yourself that it works correctly.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

Example 1.17 (continuation)

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise mathematical manipulation. For this reason, digital design methods rely heavily on concepts from automata theory.

■

Thank You for attention!