The Chomsky Hierarchy
Regular Languages

CS 331 Programming Languages

Lecture Slides

Wednesday, January 14, 2026

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

© 2017–2026 Glenn G. Chappell

Unit Overview
Formal Languages & Grammars

Topics

- ✓ Basic concepts
- ✓ Introduction to formal languages & grammars
- The Chomsky hierarchy
- Regular languages
- Regular expressions
- Context-free languages
- Programming language syntax specification

Review

Review
Basic Concepts

Dynamic: *at runtime*.

Static: *before runtime*.

Syntax: the correct *structure* of code.

Semantics: the *meaning* of code.

Coming Up

- How the syntax of a programming language is specified.
- How such specifications are used.
- Writing a lexer & parser; the latter checks syntactic correctness.
- Later, a brief study of semantics.

Review
Intro. To Formal Lang's & Grammars — Formal Languages [1/2]

A (**formal**) **language** is a *set of strings*.

Not the same as a programming language!

Alphabet: the set of characters that may appear in the strings.

For now, we write strings without quotes (for example, *abc*). We represent the empty string with a lower-case Greek epsilon (ε).

Example of a language over {0, 1}:

{ε, 01, 0101, 010101, 01010101, …}

Important examples of formal languages:

- The set of all lexemes in some category, for some programming language (e.g., the set of all legal C++ identifiers).
- The set of all syntactically correct programs, in some programming language (e.g., the set of all syntactically correct Lua programs).

Review
Intro. To Formal Lang's & Grammars — Formal Languages [2/2]

Two ways to describe a formal language:

- With a **generator**: something that can produce the strings in a formal language—all of them, and nothing else.
- With a **recognizer**: a way of determining whether a given string lies in the formal language.

Typically:

- Generators are easier to construct.
- Recognizers are more useful.

It is common begin with a generator and then construct a recognizer based on it. This construction process might be automated (but in this class it will not be).

Review
Intro. To Formal Lang's & Grammars — Grammars [1/4]

A (**phrase-structure**) **grammar** is a kind of language generator.

Needed

- A collection of **terminal symbols**. This is our alphabet.
- A collection of **nonterminal symbols**. These are like variables that eventually turn into something else. One nonterminal symbol is the **start symbol**.

Our conventions for symbols—*for now*:

- Terminal symbols are lower-case letters (a b x)
- Nonterminal symbols are upper-case letters (C Q S)
- The start symbol is S.

Review
Intro. To Formal Lang's & Grammars — Grammars [2/4]

A **grammar** is a list of one or more *productions*. A **production** is a rule for altering strings by substituting one substring for another. The strings are made of terminal and nonterminal symbols.

Here is a grammar with three productions.

$S \rightarrow xSy$

$S \rightarrow a$

$S \rightarrow \varepsilon$

An important application of grammars is specifying programming-language syntax. Since the 1970s, nearly all PLs have used some form of grammar for their syntax specification.

Intro. To Formal Lang's & Grammars — Grammars [3/4]

Grammar

1. $S \rightarrow xSy$
2. $S \rightarrow a$
3. $S \rightarrow \varepsilon$

The numbers and underlining are annotations that I find helpful. They are not actually part of the derivation.

A derivation is a list of strings.

Derivation of *xxxyyy*

$\underline{S}$
1
$x\underline{S}y$
1
$xx\underline{S}yy$
1
$xxx\underline{S}yyy$
3
$xxxyyy$

No "ε" appears here.

Using a grammar:

- Begin with the start symbol.
- Repeat:
  - Apply a production, replacing the left-hand side of the production (which must be a contiguous collection of symbols in the current string) with the right-hand side.
- We can stop only when there are no more nonterminals.

The result is a **derivation** of the final string.

Grammar

$S \rightarrow xSy$

$S \rightarrow a$

$S \rightarrow \varepsilon$

The language **generated** by a grammar consists of all strings for which there is a derivation.

- So *xxxyyy* lies in the language generated by the above grammar.

Q. What language does this grammar generate?

A. The set of all strings that consist of zero or more *x*'s, followed by an optional *a*, followed by the same number of *y*'s as *x*'s.

$\{\varepsilon, xy, xxyy, xxxyyy, \ldots, a, xay, xxayy, xxxayyy, \ldots\}$

Grammar D

1. $S \rightarrow AB$
2. $A \rightarrow x$
3. $B \rightarrow y$

Exercises

6. Based on Grammar D, write a derivation for *xy*.

Intro. To Formal Lang's & Grammars — TRY IT #3 (Answers)

Grammar D

1. $S \rightarrow AB$
2. $A \rightarrow x$
3. $B \rightarrow y$

Derivation #1 of *xy*

$\underline{S}$
1 $\underline{A}B$
2 $x\underline{B}$
3 xy

Derivation #2 of *xy*

$\underline{S}$
1 $A\underline{B}$
3 $\underline{A}y$
2 xy

Answers

6. Based on Grammar D, write a derivation for *xy*.

See above. Either answer is correct.

Exercises

7. Write a grammar that generates the following language:
   { *ab*, *abb*, *abbb*, *abbbb*, *abbbbb*, *abbbbbb*, … }

8. How could we change the grammar from Exercise 7 so that the language it generates also contains the string "*a*"?

Answers

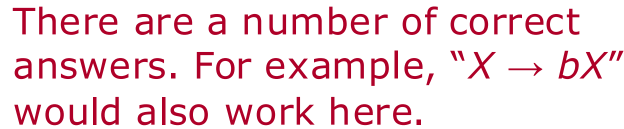
7. Write a grammar that generates the following language:
{ *ab*, *abb*, *abbb*, *abbbb*, *abbbbb*, *abbbbbb*, … }

Grammar

$S \rightarrow aX$

$X \rightarrow Xb$

$X \rightarrow b$



8. How could we change the grammar from Exercise 7 so that the language it generates also contains the string “*a*”?

Replace the production “$X \rightarrow b$” with “$X \rightarrow \varepsilon$”.

The Chomsky Hierarchy

In the mid-1950s, linguist Noam Chomsky described a hierarchy of categories of formal languages, defined in terms of the kinds of grammars that could generate them. Chomsky aimed to develop a framework for studying natural languages; however, his hierarchy has proved to be useful in the theory of computation.

The Chomsky hierarchy includes four categories of languages. He called them types 3, 2, 1, and 0. More modern names are **regular**, **context-free**, **context-sensitive**, and **computably enumerable**.

For each language category, there is an associated category of grammars that can generate that kind of language. The same names are used for the grammar categories (for example, a **regular grammar** generates a **regular language**).

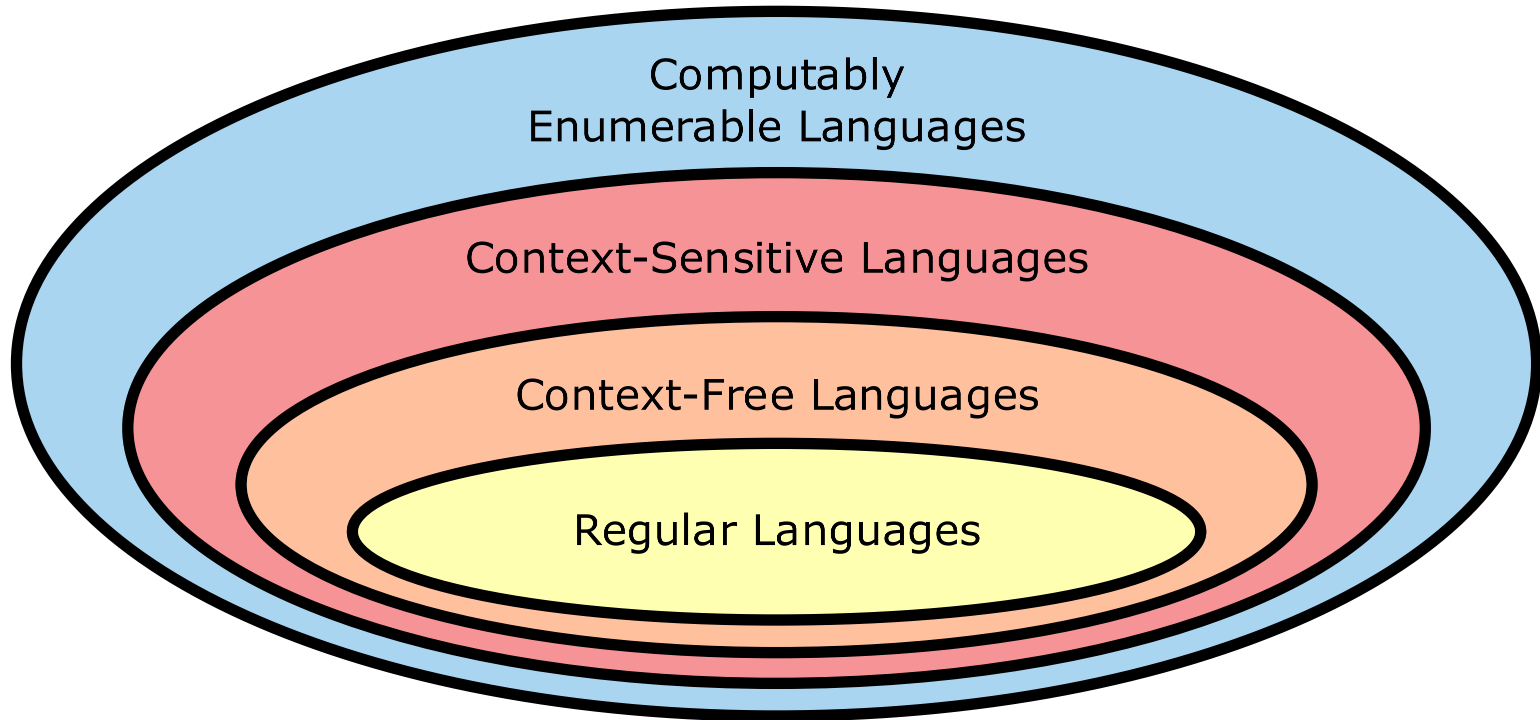
The Chomsky Hierarchy
The Hierarchy [1/2]

Here is the Chomsky hierarchy.

Language Category		Generator	Recognizer
Number	**Name**		
Type 3	Regular	Grammar in which each production has one of the following forms. • $A \rightarrow \varepsilon$ • $A \rightarrow b$ • $A \rightarrow bC$ Another kind of generator: *regular expressions* (covered later).	Deterministic Finite Automaton Think: Program that uses a small, fixed amount of memory.
Type 2	Context-Free	Grammar in which the left-hand side of each production consists of a single nonterminal. • $A \rightarrow$ [anything]	Nondeterministic Push-Down Automaton Think: Finite Automaton + Stack (roughly).
Type 1	Context-Sensitive	*Don't worry about it.*	*Don't worry about it.*
Type 0	Computably Enumerable	Grammar (no restrictions).	Turing Machine Think: Computer Program

The Chomsky Hierarchy
The Hierarchy [2/2]

Each category of languages in the Chomsky hierarchy is contained in the next. So every regular language is context-free, etc.



Next we look briefly at each category in the Chomsky hierarchy, how it is defined, and why we care about it.

The Chomsky Hierarchy
Why We Care [1/5]

A **regular language** is one that can be generated by a grammar in which each production has one of the following forms.

- $A \to \varepsilon$
- $A \to b$
- $A \to bC$

Alternative generator: **regular expression** (covered later).

A regular language can be recognized by a **deterministic finite automaton**.

- Think: a program using only a small, fixed amount of memory.

Regular languages generally describe lexeme categories.

- The set of all legal C++ identifiers forms a regular language.

Thus, these languages encompass the level of computation required for **lexical analysis**: breaking a program into lexemes.

Regular languages are also used in text **search/replace**.

The Chomsky Hierarchy
Why We Care [2/5]

A **context-free language** is one that can be generated by a grammar in which the left-hand each production consists of a single nonterminal.

- $A \rightarrow$ [anything]

A context-free language can be recognized by a **nondeterministic push-down automaton**.

- Roughly: a finite automaton plus a memory that acts as a stack.

Context-free languages generally describe programming-language syntactic correctness.

- The set of all syntactically correct Lua programs (for example) is a context-free language.

Thus, these languages encompass the level of computation required for **parsing**: determining whether a program is syntactically correct, and, if so, how it is structured.

The Chomsky Hierarchy
Why We Care [3/5]

As for **context-sensitive languages**: we generally do *not* care.

- I would call this category a mistake—an idea that Chomsky thought would be fruitful, but turned out not to be.
- I mention this category only for historical interest. You do not need to know anything about context-sensitive languages.

You may stop reading this slide here.

In case anyone is interested: Context-sensitive grammars—which generate context-sensitive languages—allow restricting the expansion of a nonterminal to a particular context. For example, such a grammar might include the following production.

$$xAy \to xBcy$$

So A can be expanded to Bc, as long as it lies between x and y.

And the recognizer for a context-sensitive language is called a *linear bounded automaton*. Google it, if you wish. Or don't.

The Chomsky Hierarchy
Why We Care [4/5]

A **computably enumerable language** is one that can be described by a grammar. We place no restrictions on the productions in the grammar.

The recognizer for a computably enumerable language is a **Turing machine**—a formalization of a computer program.

We care about computably enumerable languages because they encompass the things that computer programs can do.

- If a language is computably enumerable, then some computer program is a recognizer for it.
- Otherwise, *no such program exists*.

Computably enumerable languages are important, but we will not discuss them any further in this class.

Note. This kind of language is also called a **recursively enumerable language**. This terminology comes from a branch of mathematics called *recursive function theory*.

The Chomsky Hierarchy
Why We Care [5/5]

Summary

- A lexeme category (e.g., C++ identifiers) usually forms a **regular language**. Recognition of a regular language is thus the level of computation required for lexical analysis—and text search/replace.
- In most programming languages, the set of all syntactically correct programs forms a **context-free language**. Recognition of context-free languages is thus the level of computation required for parsing.
- **Context-sensitive languages** are a largely impractical curiosity.
- Recognition of **computably enumerable** languages encompasses the tasks that computer programs are capable of. These languages are important in the theory of computation.

Our next topic is *Regular Languages*. We will cover ideas to be used in lexical analysis. We will also look at *regular expressions*, which are used, for example, in text search & replace.

After that, we study *Context-Free Languages*, covering ideas to be used in parsing.

Regular Languages

Regular Languages
Introduction

Now we look closer at the smallest of the four categories of languages in the Chomsky hierarchy: the *regular languages*.

Regular languages have two important applications.

- In most programming languages, the set of all lexemes (words, roughly) of a particular kind forms a regular language. Thus we make use of regular languages in the early stages of compilation or interpretation, when we break up a program into lexemes—a process called **lexical analysis**, or **lexing**.
- Regular languages are also heavily used in text search & replace.

Regular Grammars & Languages — Definitions

A **regular grammar** is a grammar, each of whose productions looks like one of the following.

$A \to \varepsilon$ $\quad$ $A \to b$ $\quad$ $A \to bC$

That is, the left-hand side of each production is a single nonterminal, while the right-hand side is one of:

- the empty string
- a single terminal, or
- a single terminal followed by a single nonterminal—*which may be the same as the left-hand side*.

A **regular language** is a language that is generated by some regular grammar.

Regular Grammars & Languages — Examples [1/3]

Here is an example of a regular grammar.

$S \rightarrow \varepsilon$
$S \rightarrow t$
$S \rightarrow xB$
$B \rightarrow yS$

Q. What language does this grammar generate?

A. The set of all strings that consist of zero or more concatenated copies of *xy*, followed by an optional *t*.

$\{\varepsilon, xy, xyxy, xyxyxy, \ldots, t, xyt, xyxyt, xyxyxyt, \ldots\}$

So this language is a regular language.

Regular Grammars & Languages — Examples [2/3]

Here is another grammar. This is *not* a regular grammar.

$S \rightarrow A$
$S \rightarrow At$
$A \rightarrow Axy$
$A \rightarrow \varepsilon$

Q. What language does this grammar generate?

A. The set of all strings that consist of zero or more concatenated copies of *xy*, followed by an optional *t*.

$\{\varepsilon, xy, xyxy, xyxyxy, \ldots, t, xyt, xyxyt, xyxyxyt, \ldots\}$

Q. Is this a regular language?

A. Yes! Because it is generated by a regular grammar: the one on the previous slide.

Regular Grammars & Languages — Examples [3/3]

There are languages that are not regular.

For example, this grammar …

$S \rightarrow aSa$

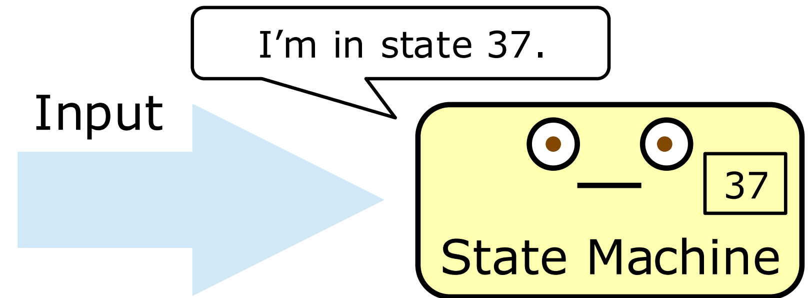
$S \rightarrow b$

… generates the following language.

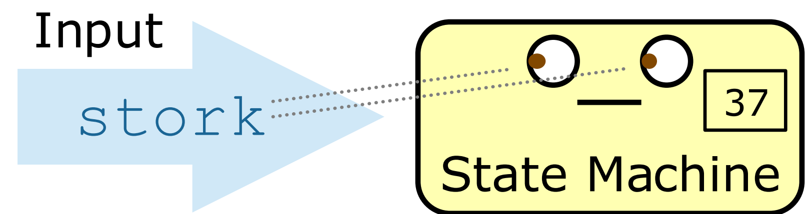
$\{b, aba, aabaa, aaabaaa, \ldots\} = \{ a^k b a^k \mid k \geq 0 \}$

There is *no* regular grammar that generates this language. It is not a regular language. (I am not saying this is obvious; but it is true. Proving that a language is not regular is beyond the scope of this class.)

A **state machine** is a kind of theoretical construct. It is always in some **state**.

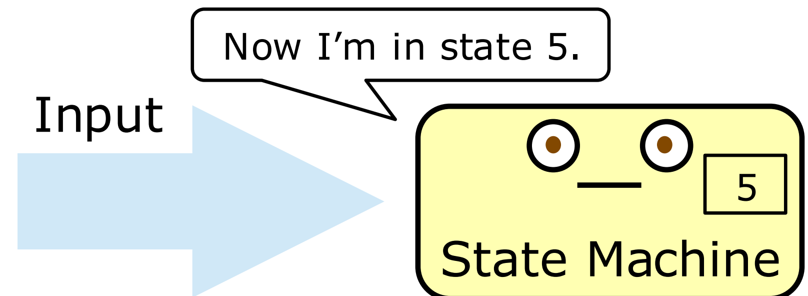


Repeatedly, it looks at its input, and, depending on what it sees and what state it is in, it **transitions** to a new state—which may or may not be the same as the old state.



Transitioning …

It may also do other things when transitioning.



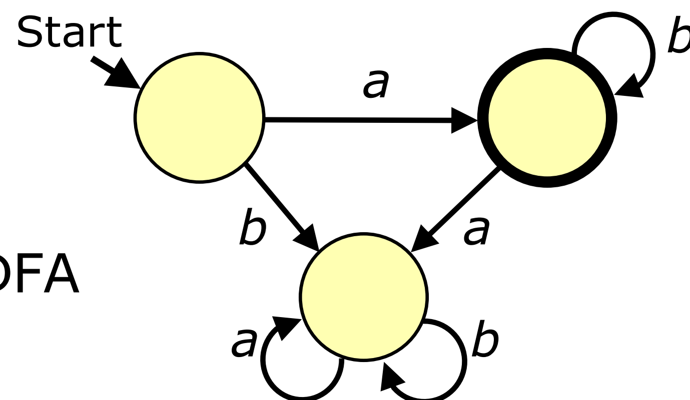
A **deterministic finite automaton** (Latin plural “**automata**”), or **DFA**, is a kind of state machine that forms a recognizer for regular languages.

A DFA consists of a finite collection of **states** and **transitions** between these states.

- One state is the **start state**.
- Some states may be **accepting states**.
- Each transition begins at a state, ends at a state, and is associated with a character in the alphabet—that is, some terminal symbol.
- For each character, each state has *exactly one* transition leaving it that is associated with that character.

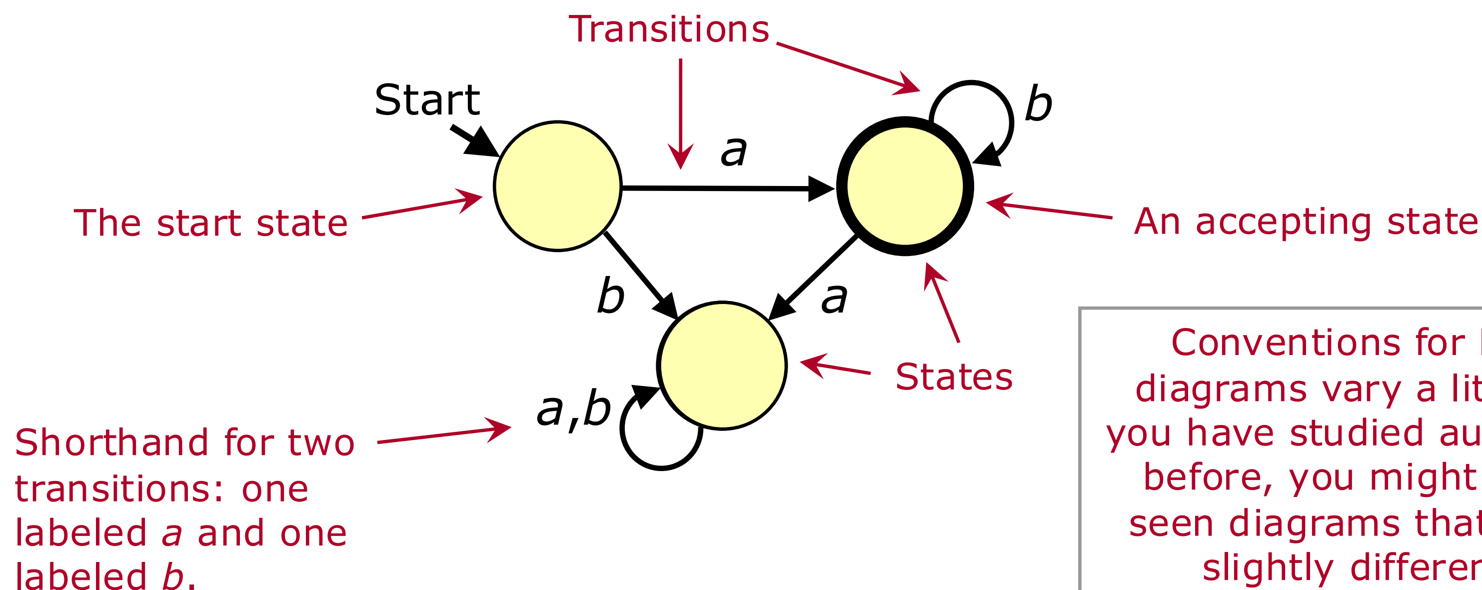
Here is a **diagram** of a 3-state DFA with alphabet $\{a, b\}$.

On the next slide we make the idea of a DFA diagram more precise.



To make a diagram of a DFA, we draw a circle (or other enclosed shape) for each state. For each transition, we draw an arrow from the state it begins at to the state it ends at, labeling the arrow with the transition's character.

We make accepting states bold, and we draw an arrow labeled "Start" to the start state.



Conventions for DFA diagrams vary a little. If you have studied automata before, you might have seen diagrams that were slightly different.

Finite Automata — Basics [4/5]

We use a DFA as a recognizer as follows.

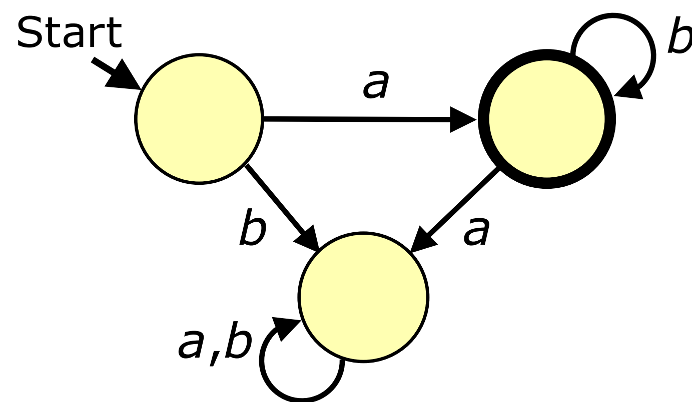
- We are always in one of the states, beginning with the start state.
- We proceed in steps. At each, we read a character from the input and follow the transition beginning at the current state and labeled with the character that was read. Where this ends is our new state.
- If, when we reach the end of the input, we are in an accepting state, then we **accept** the input; otherwise we do not accept.

The set of all inputs that are accepted is the language **recognized** by the DFA.

Q. What language does this DFA recognize?

A. The set of all strings that consist of an *a* followed by zero or more *b*’s.

{*a*, *ab*, *abb*, *abbb*, *abbbb*, *abbbbb*, …}

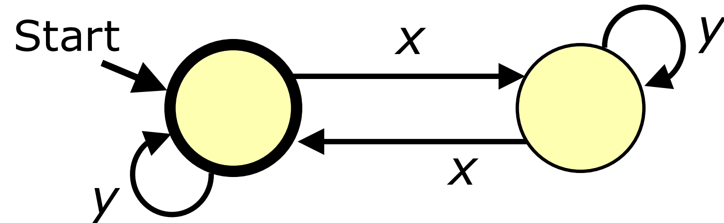


Fact. The languages that are recognized by DFAs are precisely the regular languages.

That is:

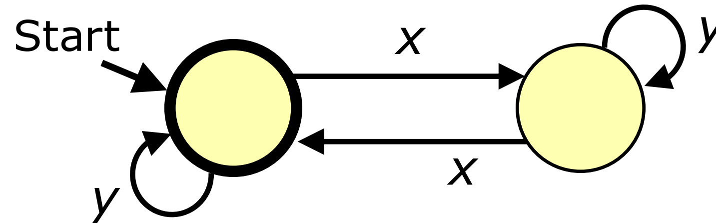
- For each DFA, the language it recognizes is a regular language.
- For each regular language, there is a DFA that recognizes it.

We will see the state-machine idea in code form later in the semester when we write code to do lexical analysis.



Exercises

1. What language is recognized by the DFA whose diagram is shown?
2. Draw the diagram of a DFA that recognizes the language consisting of all strings of 0s and 1s that contain at least one character.



Answers

1. What language is recognized by the DFA whose diagram is shown?

The language containing all strings of *x* and *y* characters that have an even number of *x* characters (including the empty string!).

2. Draw the diagram of a DFA that recognizes the language consisting of all strings of 0s and 1s that contain at least one character. *Hint. Only 2 states are required.*