Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Review of Regular Languages for Introduction to Compilers

David L. Dill
Department of Computer Science
Stanford University

Outline

1. Formal Language Theory
2. Alphabets, Strings, and Languages
3. Regular Languages
4. Deterministic Finite Automata
   - Solving Problems on DFAs
5. Nondeterministic Finite Automata
   - Equivalence of NFAs and DFAs
6. Regular expressions
   - From Regular Expressions to Finite Automata
7. Minimizing DFAs

Formal Language Theory
Alphabets, Strings, and Languages
Regular Languages
Deterministic Finite Automata
Solving Problems on DFAs
Nondeterministic Finite Automata
Equivalence of NFAs and DFAs
Regular expressions
From Regular Expressions to Finite Automata
Minimizing DFAs

Section

Formal Language Theory

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Formal Language Theory

Automata and complexity theory is concerned with properties of *formal languages*.

In formal language, automata, and complexity theory, a *language* is just a set of strings.

Almost all of the interesting problems have to do with *infinite* formal languages.

Sets of strings are universal because something can be represented in a computer iff it can be written as a string (e.g., integers, floating point numbers, graph structures, etc.).

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Section

Alphabets, Strings, and Languages

Basic concepts

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Definition (Alphabet)

An *alphabet* is a non-empty finite set. The members of the alphabet are called *symbols*.
(In CS143, we will frequently call the members of an alphabet *characters*.)

Examples: $\{0, 1\}$, ASCII, or any other character set.

The capital Greek sigma (Σ) is often used to represent an alphabet.

Strings

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Definition (String)

A *string* is a finite sequence of symbols from some alphabet.

Let $[i \ldots j]$ the set of all integers between i and j, inclusive.

I.e., a string of length n is a function $[0 \ldots n-1] \to \Sigma$. The domain of the function is the set of *positions* in the string.

The ith symbol in x is usually written x_i, although, since it is a function, it can be written $x(i)$ (as I do later in this lecture).

(Sometimes, we'll consider strings to have positions in $[1 \ldots n]$, for convenience.)

The length of a string x is written as $|x|$.

Strings, cont.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Examples:

- ϵ – the empty string (the same for every alphabet).
  (Leaving a blank space for the empty string is confusing, so we use the Greek letter "epsilon").
  ϵ is not a symbol! It is the string with no symbols; the string of zero length.
- 000, 01101 are strings over the binary alphabet $\{0, 1\}$.

Notation: The set of all strings of all lengths over an alphabet Σ is written Σ^*.

Notation: Generally, we will use lower-case letters early in the alphabet for individual symbols, *e.g.*, $a, b, c, \ldots \in \Sigma$ and lower-case letters late in the alphabet for strings, *e.g.*, $x, y, z, w, u, v \in \Sigma^*$. The ith symbol in a string x will often be written as x_i, though.

Concatenation of Strings

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Definition (Concatenation of Strings)

The *concatenation* of strings x and y over alphabet Σ is the string formed by following x by y.

It is written $x \cdot y$, or (usually) xy.

xy is a function from $[0 \ldots |x| + |y| - 1] \to \Sigma$ where $xy(i) = x(i)$ when $0 \leq i < |x|$, and $xy(i) = y(i - |x|)$ when $|x| \leq i < |x| + |y|$.

Examples:

- $abc \cdot def = abcdef$
- $\epsilon \cdot abc = abc$

Properties of Concatenation

Concatenation has some properties of multiplication. We'll use these routinely:

$(x \cdot y) \cdot z = x \cdot (y \cdot z)$ (associativity).

$\epsilon \cdot x = x \cdot \epsilon = x$ (ϵ is the identity).

Another property: $|x \cdot y| = |x| + |y|$.

Notation: As with multiplication, sometimes the $\cdot$ is omitted.
$x \cdot y = xy$.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Languages

Definition

A *language* over Σ is a subset of Σ^*.

Note: Of course, this omits almost everything that we intuitively think is important about a language, such as meaning. But this definition nevertheless leads to incredibly useful and important results.

Examples:

- $\emptyset$ (the empty language)
- $\{\epsilon\}$ (the language consisting of a single empty string).
- The set of all strings with the same number of a's as b's.
- The set of all prime numbers, written as binary strings.
- The set of all first-order logic formulas.
- The set of all first-order logic theorems.
- The set of all input strings for which a given Boolean Java function returns "true."

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Section

Regular Languages

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Regular languages

The *regular languages* are simple and useful class of formal languages.

Regular languages can be represented in a computer in many ways. The regular languages are infinite, and almost all interesting problems on them *are computationally solvable*.

Regular languages are of great practical as well as theoretical importance.

- They are widely used in applications (compilers, pattern matching, specification and formal verification of systems).
- Finite automata are the basis for more sophisticated representations (automata on infinite strings, tree automata, timed automata).

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Section

Deterministic Finite Automata

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Deterministic Finite Automata

A language is *regular* if it's recognized by a DFA.

A deterministic finite automaton (DFA) is a mathematical model of a machine that reads strings and says "yes" or "no" for each string.

("Automata" is the plural of "automaton.")

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

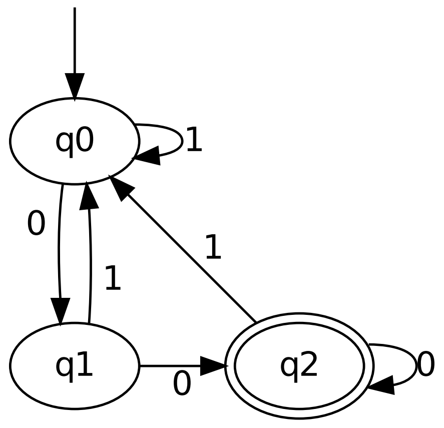
Minimizing DFAs

DFA Example

A DFA follows an input string from a start state to the end of the string. If it stops in an "accept state" (marked with a double oval), it "accepts" the string

A DFA represents a language: the set of strings which it accepts. This is especially interesting when the language is finite.

The languages of this automaton is the set of all strings over the alphabet $\{0, 1\}$ that end with 00:



Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

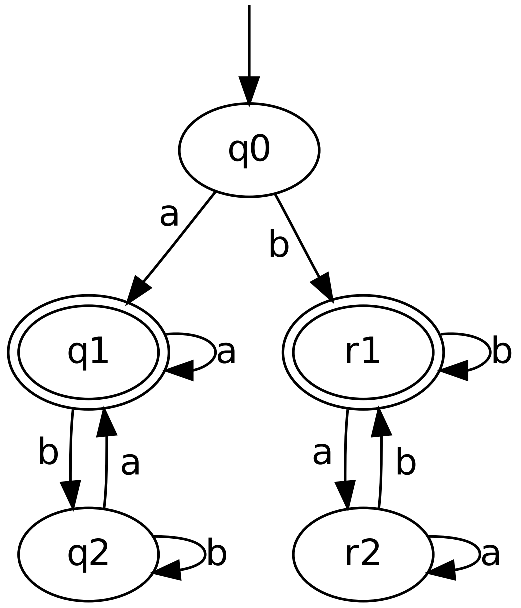
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Another DFA

Here is a DFA for $L = \{\text{All strings that start and end with the same symbol}\}$ for $\Sigma = \{a, b\}$.



Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Deterministic finite automata

Now, the mathematical definition:

Definition (Deterministic Finite Automaton)

A DFA consists of:

- Q, a finite set of *states*.
- Σ, an alphabet.
- $\delta\colon Q \times \Sigma \to Q$, the *transition function* .
- $q_0 \in Q$ is the *start state*.
- $F \subseteq Q$ is the set of *accept states*.

Note: δ must be *total* function (defined for every (q, a)).

Sometimes, a DFA is written as $(Q, \Sigma, \delta, q_0, F)$ so we know what all the parts are named.

Language of a DFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Definition (Run of a DFA on Input)

A *run* of a DFA, D, on an input string $w = w_1 w_2 \ldots w_n$ (where $w_i \in \Sigma$) is a sequence of states $r = r_0 r_1 \ldots r_n$ (so it is a sequence of length $n + 1$), where

- $r_0 = q_0$
- $r_{i+1} = \delta(r_i, w_{i+1})$ for all $0 \leq i < n$

Definition (Accepting Run of a DFA on Input)

An *accepting run* is a run whose final state is an accept state: $r_n \in F$.

Another way to write this is that $r_{|w|} \in F$.

Language of a DFA, cont.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Definition

D *recognizes* (or *accepts*) w if it has an accepting run on w.

Definition (Language of a DFA)

The language of a DFA, D, (written $L(D)$) is the set of strings accepted by D.
Formally, $L(D) = \{w \in \Sigma^* \mid D \text{ accepts } w\}$

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Subsection

Solving Problems on DFAs

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

The Membership Problem with DFAs

The most basic problem for formal languages is the *membership problem*:

Given a DFA D and a string x, is $x \in L(D)$?

This is very easy for a computer to solve, just like we did it by hand.

1. Start with q_0
2. Read each symbol from input string, go to the next state as directed by the δ function
3. After last symbol, check whether the state is in F.

An Algorithm for Deciding Emptiness

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

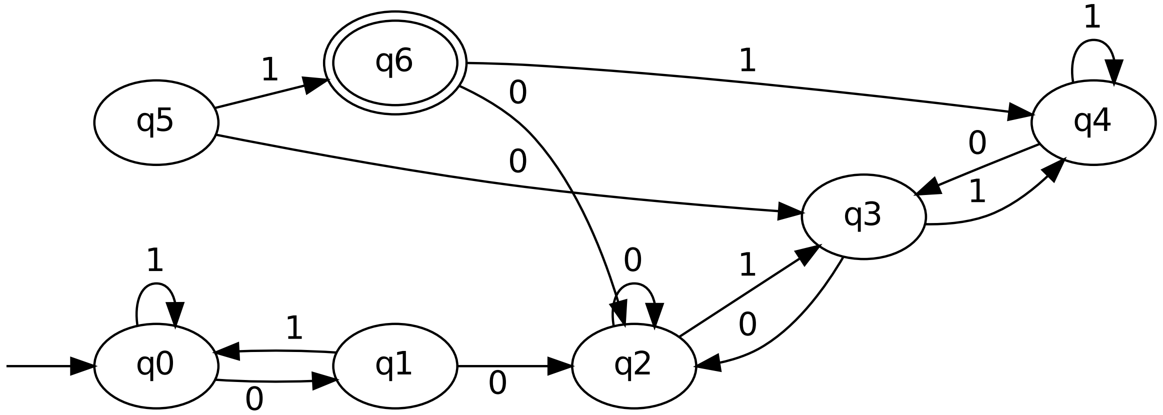
Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Is the language of this DFA empty?



Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

The Emptiness Problem on DFAs

A DFA defines a directed graph. The vertices are the states Q, and there is an edge from q to q' if $q' = \delta(q, a)$ for some $a \in \Sigma$.

Definition (Reachability in a Directed Graph)

When there is a path from q_0 to q_k, q_k is said to be *reachable* from q_0.

Checking for emptiness of $L(D)$

A DFA accepts some string if there is an accept state that is reachable from q_0. We can find a particular string in $L(D)$ by concatenating the symbols labeling the arrows along this path.

Computing Graph Reachability

Graph reachability can be computed by *depth-first* or *breadth-first* search.

1. Start with q_0.
2. Systematically explore successors of each state.
3. Mark states that have been explored so you don't do them repeatedly.
4. Stop when all states are marked.

Depth-first search always explores the successors of the most recently explored state before doing other states.

Breadth-first search always searches states that are closer to q_0 before searching other states.

States can be searched in other orders, as well.

Both of these search methods are very fast. Their run time is proportional to the number of edges in the graph of the DFA.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Summary

- *Strings* are finite sequences of *symbols* from a finite alphabet, Σ.
- A *formal language* is a set of strings.
- Deterministic finite automata (DFAs) are mathematical machines that define languages.
- If $L(D)$ is the language of a DFA, there are simple algorithms to decide whether $x \in L(D)$ for some string x and to decide whether $L(D) \neq \emptyset$.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Section

Nondeterministic Finite Automata

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

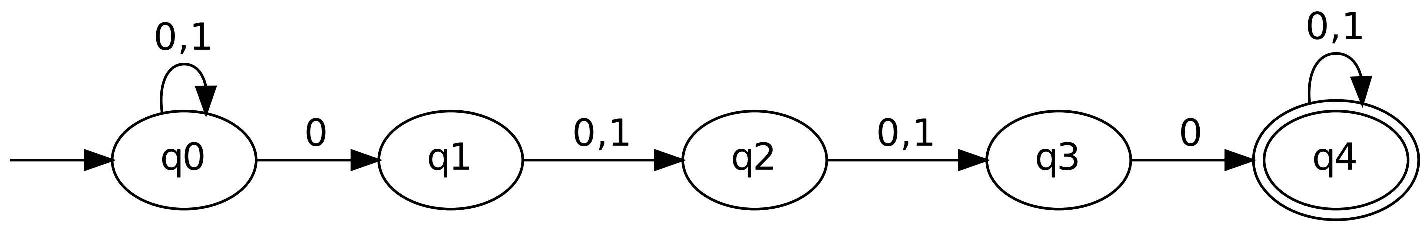
From Regular Expressions to Finite Automata

Minimizing DFAs

Nondeterministic Finite Automata

Nondeterministic finite automata (NFAs) generalize DFAs by allowing a *choice* of *possible next states* on each input symbol.

This is an NFA because q_0 can go to q_0 *or* q_1 on input 0.



An NFA can have *many* or *no* runs on an input string.

If *any* accepting run exists for input w, the NFA accepts w.

What runs are there of “111001011”?

Is there an accepting run?

ϵ Transition

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

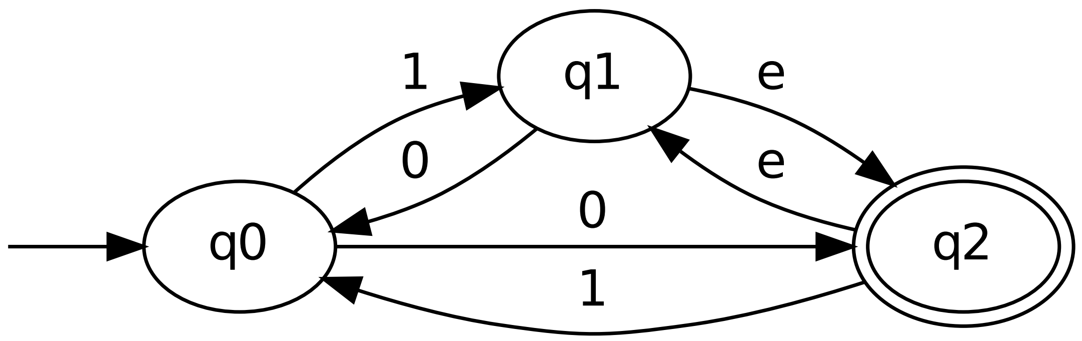
Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

NFAs can also have ϵ transitions, which change state without reading an input symbol.



We can insert ϵ symbols at arbitrary places in an input string to see what some of the runs are:

“$011 = 0\epsilon\epsilon11\epsilon$,” has the run $q_0q_2q_1q_2q_0q_1q_2$, so it is accepted.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

NFA Definition

An NFA is the same as a DFA except for the δ.

Definition (Nondeterministic Finite Automaton)

An NFA consists of:

- Q, a finite set of *states*.
- Σ, an alphabet.
- $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to \mathscr{P}(Q)$
- $q_0 \in Q$ is the *start state*.
- $F \subseteq Q$ is the set of *accept states*.

In a DFA, $\delta \colon Q \times \Sigma \to Q$. For example, in a DFA, we might have $\delta(q_1, 0) = q_1$ – it returns a unique state.

In an NFA, $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to \mathscr{P}(Q)$. For example, in an NFA, we might have $\delta(q_5, 0) = \{q_6, q_7\}$ or $\delta(q_5, 1) = \emptyset$.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

NFA Acceptance

To deal with ϵ transitions, we consider all the ways of composing an input string w as a concatenation of symbols *and* ϵ*'s*. There are infinitely many ways to do this, since any number of ϵ's can be added before and after each symbol.

Definition (Run of an NFA)

r is a *run* of an NFA, N, on input w if there exists $m \geq |w|$ and $w_i \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq m$ such that $w = w_1 w_2 \ldots w_m$ and

- $r_0 = q_0$
- $r_{i+1} \in \delta(r_i, w_{i+1})$ for all $0 \leq i < m$

Definition

An *accepting run* is one whose last state is in F.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

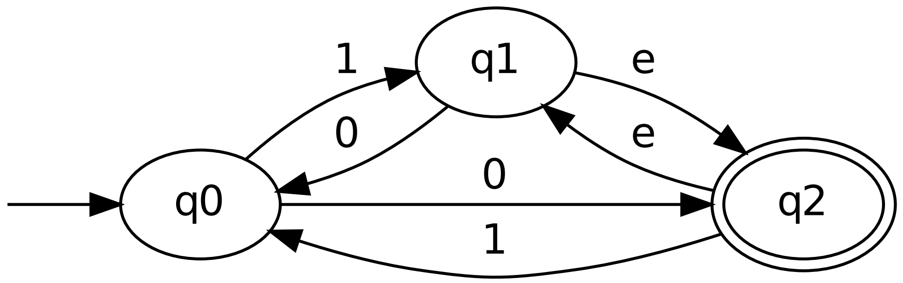
Minimizing DFAs

Nondeterminism

If N is an NFA,

$L(N) = \{w \in \Sigma^* \mid$ *there exists* an accepting run of N on $w\}$

Note: NFA accepts if there is *at least one* accepting run (even if there are many non-accepting runs).



On input "0", this automaton has an accepting run (e.g., q_0, q_2) so it accepts, even though it has many non-accepting runs (q_0, q_2, q_1).

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Subsection

Equivalence of NFAs and DFAs

NFAs vs. DFAs

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Amazingly, an NFA is no more powerful than a DFA (although it can sometimes be a lot smaller).

(With other kinds of automata, the nondeterministic versions are sometimes more powerful than the deterministic ones.)

Theorem

For every NFA N, there is a DFA D such that $L(N) = L(D)$.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

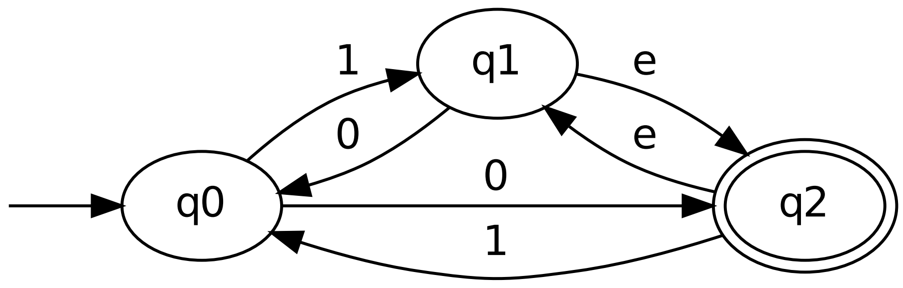
From Regular Expressions to Finite Automata

Minimizing DFAs

Equivalence of NFAs and DFAs

The theorem is based on a transformation called the *powerset construction* or *subset construction*. Each state of the DFA is a *subset of the NFA states*. Before doing the math, here's an example that illustrates the basic ideas.

NFA



Equivalence of NFAs and DFAs

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

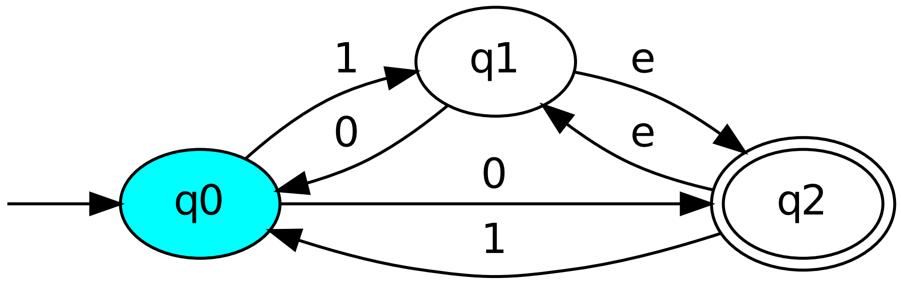
Regular expressions

From Regular Expressions to Finite Automata

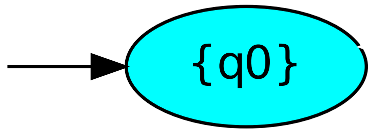
Minimizing DFAs

Start at q0.

NFA



DFA



Equivalence of NFAs and DFAs

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

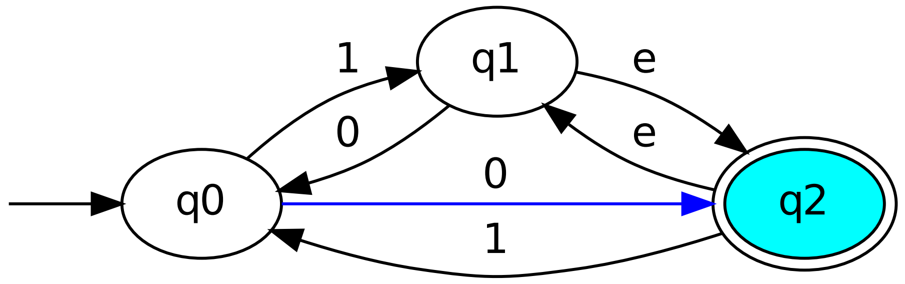
Regular expressions

From Regular Expressions to Finite Automata

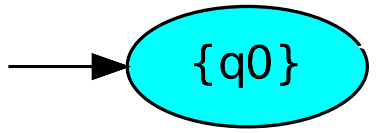
Minimizing DFAs

Following 0 from q0 to q2.

NFA



DFA



Equivalence of NFAs and DFAs

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

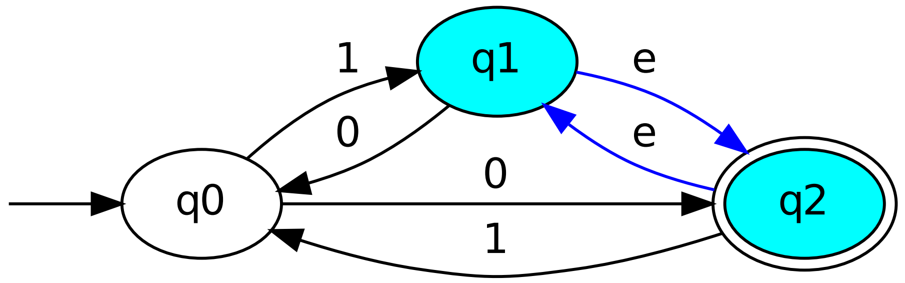
Regular expressions

From Regular Expressions to Finite Automata

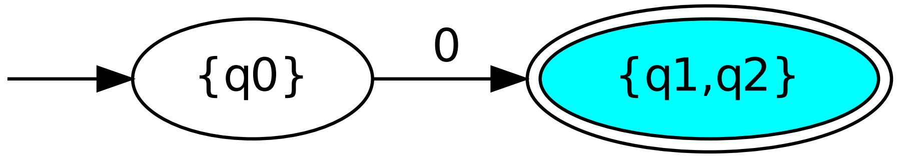
Minimizing DFAs

Following ϵ's from q2 to q1 and back.

NFA



DFA



Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

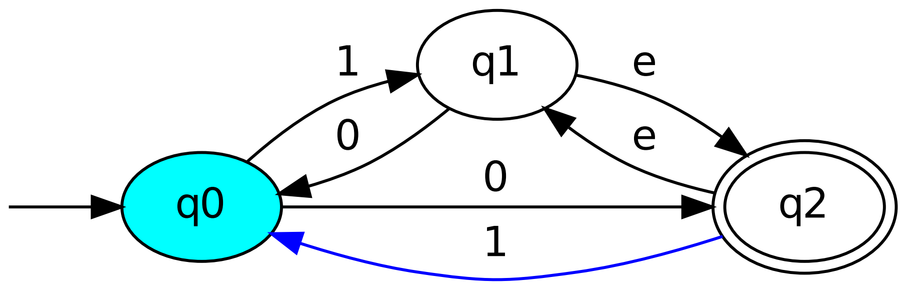
From Regular Expressions to Finite Automata

Minimizing DFAs

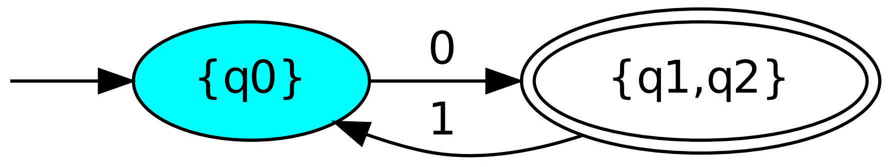
Equivalence of NFAs and DFAs

Following 1 from q1/q2 to q0.

NFA



DFA



Equivalence of NFAs and DFAs

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

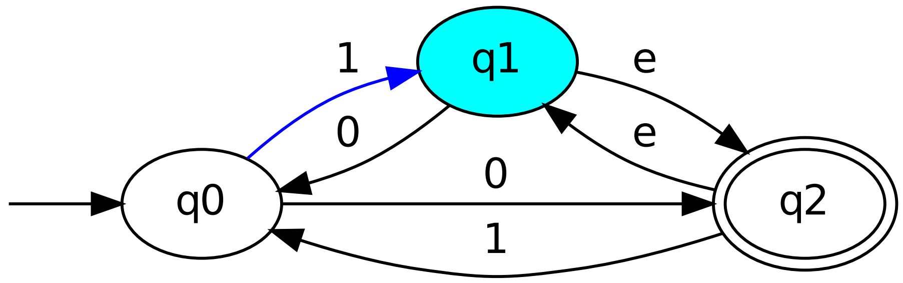
Regular expressions

From Regular Expressions to Finite Automata

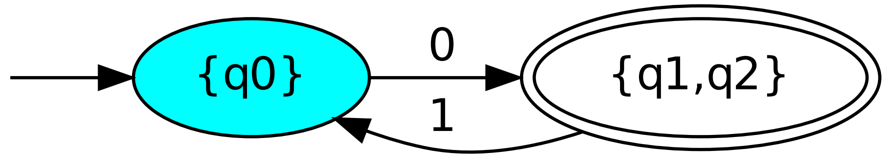
Minimizing DFAs

Follow 1 from q0 to q1.

NFA



DFA



Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

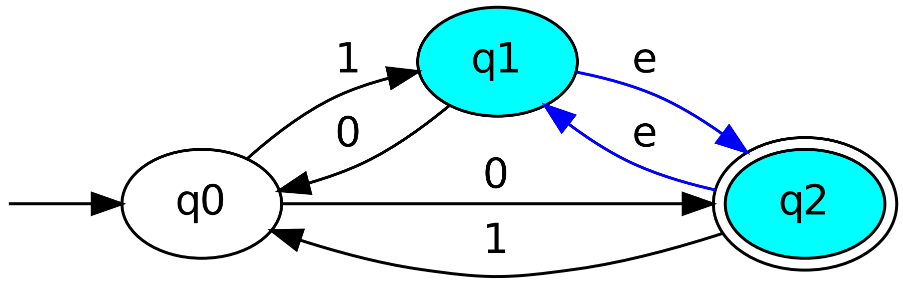
From Regular Expressions to Finite Automata

Minimizing DFAs

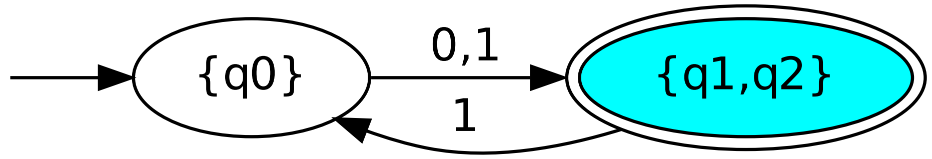
Equivalence of NFAs and DFAs

Follow ϵ's from q1 to q2 and back.

NFA



DFA



Equivalence of NFAs and DFAs

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

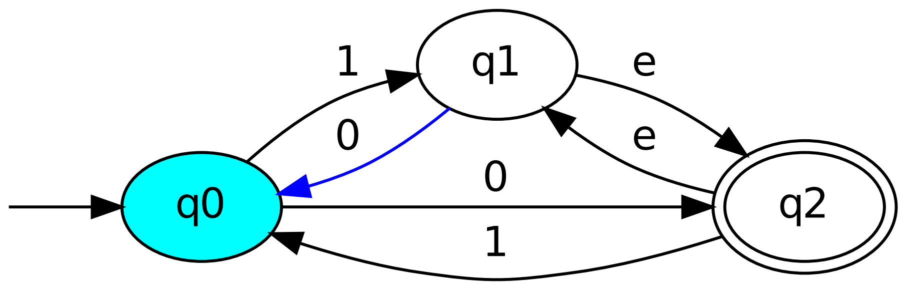
Regular expressions

From Regular Expressions to Finite Automata

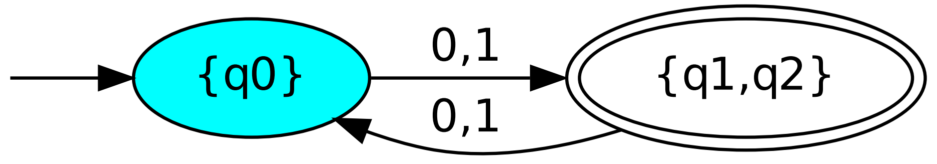
Minimizing DFAs

Follow 0 from q1/q2 to q0.

NFA



DFA



ϵ-closure

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Converting from NFAs to DFAs requires a special step to deal with ϵ transitions.

The ϵ-*closure* of a set of states S, $EC(S)$, is the set of all states reachable from states in S by zero or more ϵ transitions, including the states in S itself.

$EC(S) = \{q' \in Q \mid \exists q \in$
S there is a path of 0 or more ϵ transitions from q to q'.$\}$

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

ϵ-closure Example

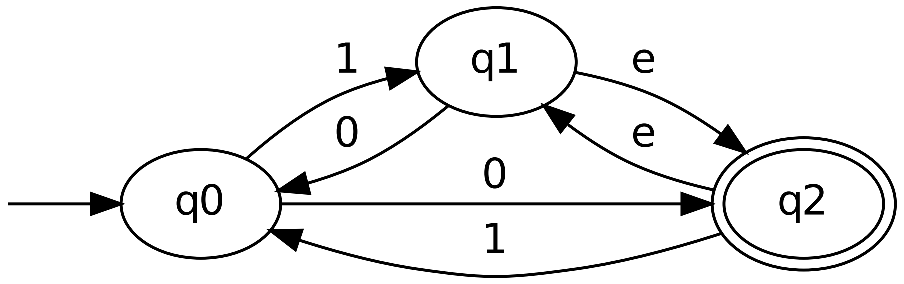
In the example NFA, earlier:

$$EC(\{q_0\}) = \{q_0\}$$

$$EC(\{q_0, q_1\}) = \{q_0, q_1, q_2\}$$

$$EC(\{q_1\}) = \{q_1, q_2\}$$

$$EC(\{q_1, q_2\}) = \{q_1, q_2\}$$



Powerset Construction Example

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

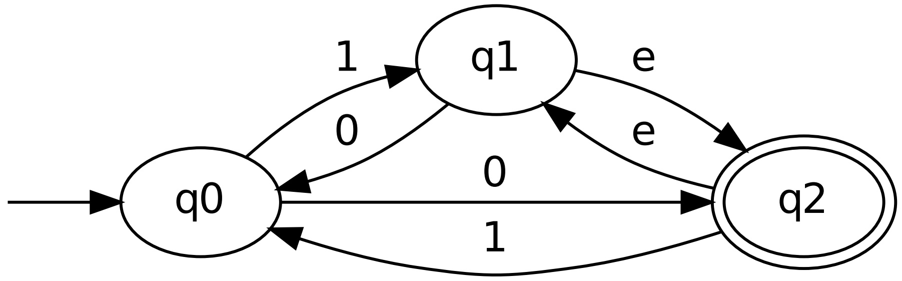
Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

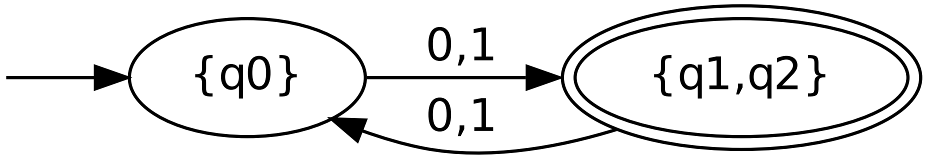
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs



converts to the DFA:



6 states from the powerset were omitted because they are not

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Converting an NFA to an DFA

Theorem

For every NFA, N, there exists an DFA D such that $L(D) = L(N)$.

Given $N = (Q, \Sigma, q_0, \delta_N, F_N)$, we can define

$D = (\mathscr{P}(Q), \Sigma, EC(q_0), \delta_D, F_D)$ as follows:

$F_D = \{S \subseteq Q \mid S \cap F_N \neq \emptyset\}$

$\delta_D(S, a) = EC(\bigcup_{s \in S} \delta_N(s, a))$

Note: δ_D has no ϵ transitions!

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Correctness of Powerset Construction

As with the product construction, the key to the proof of correctness of the powerset construction is to prove a stronger theorem about the runs.

Let w be an arbitrary string in Σ^*. We need to relate runs in the "determinized" automaton, D, to runs in the original NFA, N.

We need to prove that, if R is the run of D on w, then

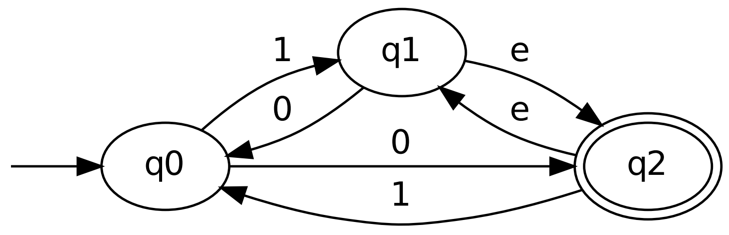
$R_i = \{r_j \mid r \text{ is a run of } N \text{ on } w\}$

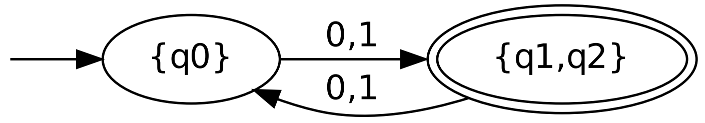
(j is the run up to the ith symbol of w. $j \geq i$ because w has ϵ symbols inserted.)

In other words, each state along the run R is a collection of all the states at the same position in w on all the runs of N on w.

This can be proved by induction on the length of strings. With sufficient patience, it can be written out as an equational proof – but I won't do that.

How the Powerset Construction Proof Works





$w = 000$, runs in N: $q_0, q_2, q_1, q_0, q_2, \ldots$ with ϵ transitions between q_2 and q_1.

Equivalent run in D: $\{q_0\}, \{q_1, q_2\}, \{q_0\}, \{q_1, q_2\}$

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Section

Regular expressions

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Regular Expression Introduction

Regular expressions are a linear textual notation for regular languages (the same languages accepted by DFAs and NFAs).

Regular expressions are widely used:

- In defining the “lexical structure” (words) of programming languages. There are many tools that transform regular expressions to programs that break input into individual words.
- Pattern matching in text processing. AWK and Perl were early languages

Example: Strings over the alphabet $\Sigma = \{a, b\}$ that start and end with the same symbol: $a \cup b \cup (a(a \cup b)^* a) \cup (b(a \cup b)^* b)$.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Informal Summary of Regular Expressions

Regular expressions represent languages (sets of strings)

- $\emptyset$ represents $\emptyset$
- ϵ represents $\{\epsilon\}$
- **a** represents $\{a\}$
- $R_1 \cup R_2$ represents the union of R_1 and R_2
- $R_1 \cdot R_2$ (or just $R_1 R_2$) concatenates strings from R_1 with strings from R_2.
- R_1^* represents a concatenation of 0 or more strings from R_1. (R_1^0 is $\{\epsilon\}$).

Regular Expression Definition

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Regular expressions are defined relative to some alphabet Σ.

This is a recursive definition on the structure of regular expressions.

The structure of regular expressions is the basis for other recursive definitions and induction proofs. Every recursive definition and proof has to handle these six cases:

- $\emptyset$ is a regular expression.
- ϵ is a regular expression.
- $\mathbf{a}$ is a regular expression, if $a \in \Sigma$.
- $R_1 \cup R_2$ is a regular expression if R_1 and R_2 are.
- $R_1 \cdot R_2$ is a regular expression if R_1 and R_2 are (also written as $R_1 R_2$).
- R_1^* is a regular expression if R_1 is.

We'll parenthesize if necessary. Often $\cdot$ is omitted (like multiplication), so concatenation looks like ab.

Precedence: $a \cup b \cdot c^* = a \cup (b \cdot (c^*))$.

Language of a Regular Expression

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

If R is regular expression, the language of R (written as $L(R)$) is defined recursively on the structure of regular expressions.

- $L(\emptyset) = \emptyset$
- $L(\epsilon) = \{\epsilon\}$
- $L(\mathbf{a}) = \{a\}$
- $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
- $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$
- $L(R^*) = (L(R))^*$

Regular Expression Examples

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

All of these are over the alphabet $\Sigma = \{0, 1\}$.

What is 0^*1^*? All strings in which there is never a "0" after a "1".

All strings ending in "11"? $(0 \cup 1)^*11$

What is $(0^*1^*)^*$? Any string of 0's and 1's. This can also be written $(0 \cup 1)^*$

With $\Sigma = \{0, 1\}$ How do we write "at least one 0"?
$(0 \cup 1)^*0(0 \cup 1)^*$

More Regular Expression Examples

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

With $\Sigma = \{0, 1\}$, how do we write "at least one 0 and at least one 1"?

$(0 \cup 1)^*(01 \cup 10)(0 \cup 1)^*$

$L^+ = L^1 \cup L^2 \cup \ldots$ How do we write it using existing operators?

LL^*

L? means "an optional L". How do we write it?

$L \cup \epsilon$

Language of a Regular Expression

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

The *meaning* of a regular expression R is its language, $L(R)$. This is defined recursively, based on the structure of regular expressions as defined in the previous slide.

- $L(\emptyset) = \emptyset$
- $L(\epsilon) = \{\epsilon\}$
- $L(\mathbf{a}) = \{a\}$
- $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
- $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$
- $L(R^*) = (L(R))^*$

We defined $L_1 \cdot L_2$ and L^* in the previous lecture using set theory.

Formal Language Theory
Alphabets, Strings, and Languages
Regular Languages
Deterministic Finite Automata
Solving Problems on DFAs
Nondeterministic Finite Automata
Equivalence of NFAs and DFAs
Regular expressions
From Regular Expressions to Finite Automata
Minimizing DFAs

Subsection

From Regular Expressions to Finite Automata

From Regular Expressions to Finite Automata

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Theorem

For every regular expression R, there is an NFA N such that $L(N) = L(R)$.

Like many results in automata theory, the heart of the proof is a *construction* – defining an automaton.

The definition is recursive, based on the structure of regular expressions.

For operators that construct a larger regular expression from smaller regular expressions, (e.g., R^*), there are specified ways of combining NFAs for the smaller regular expressions to make an NFA for the larger expression.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

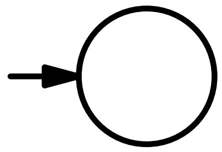
Regular expressions

From Regular Expressions to Finite Automata

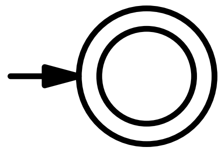
Minimizing DFAs

Base Cases

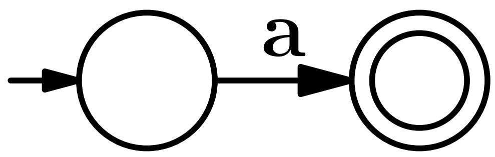
$R = \emptyset$:



$R = \epsilon$:



$R = \mathbf{a}$:



Inductive Case: Union

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

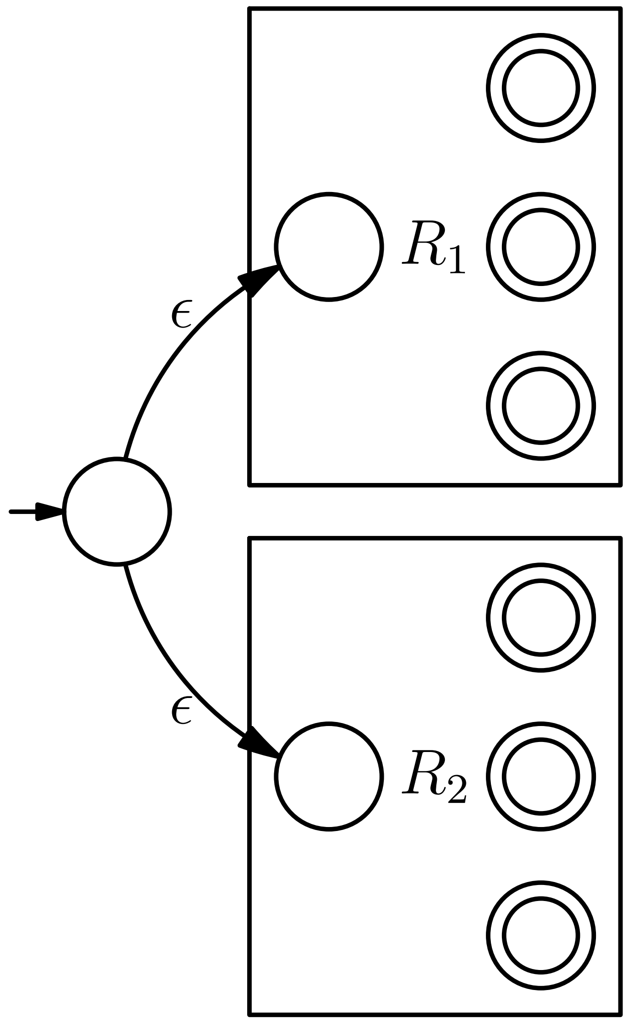
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

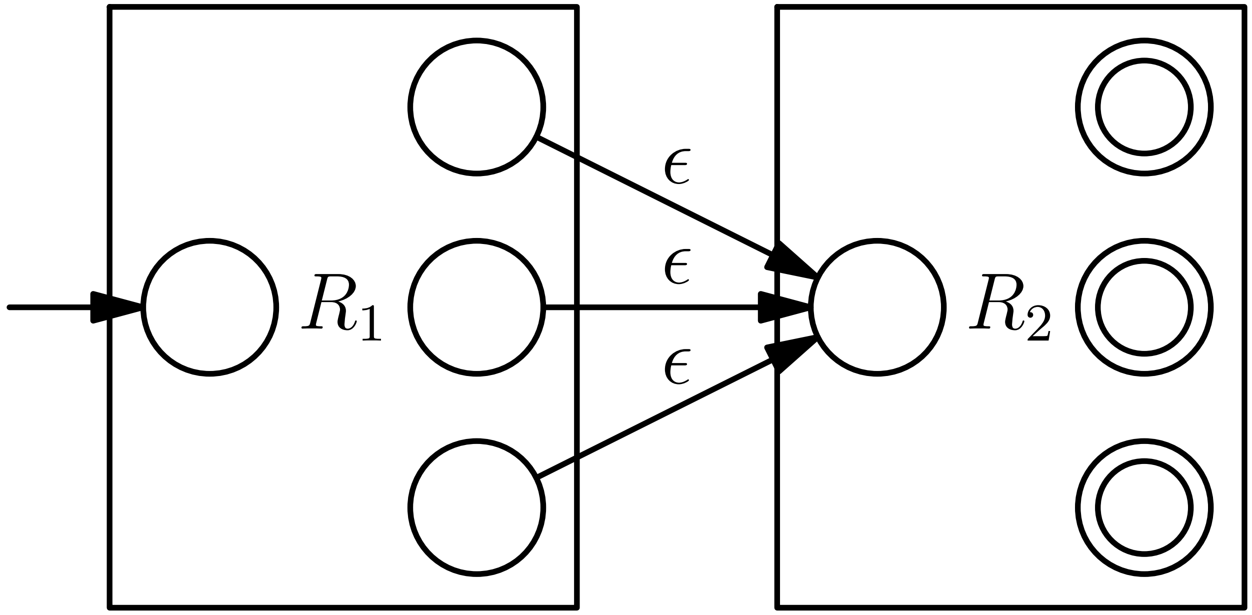
(Boxes contain arbitrary finite automata.)

$R = R_1 \cup R_2$:



Inductive case: concatenation

$R = R_1 \cdot R_2$:



Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Inductive case: Kleene closure

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

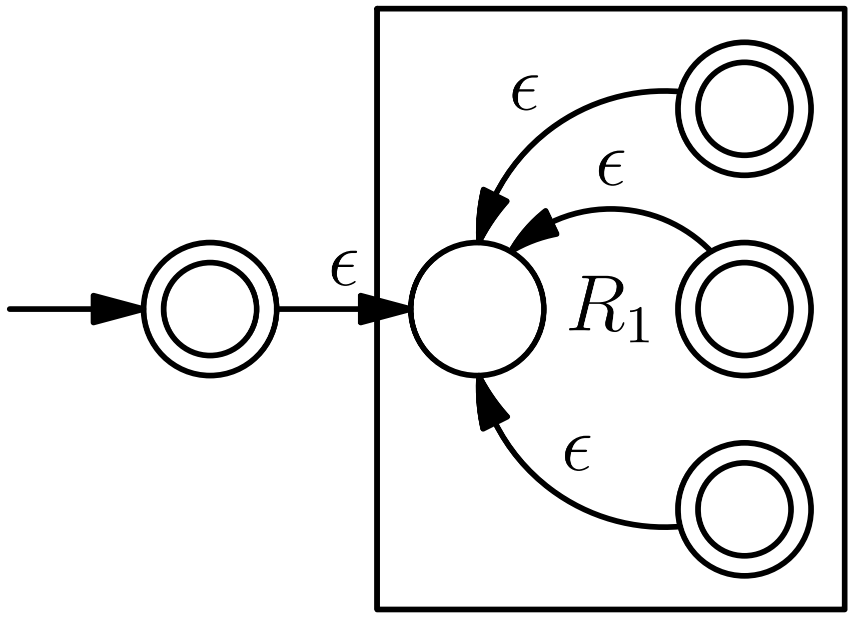
Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

$R = R_1^*$:



Example: regular expression to NFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Example: $\mathbf{0} \cup (\mathbf{11})^*\mathbf{0}$

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

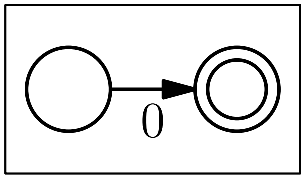
Regular expressions

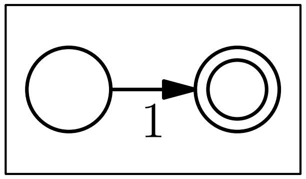
From Regular Expressions to Finite Automata

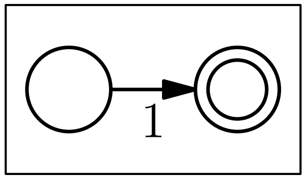
Minimizing DFAs

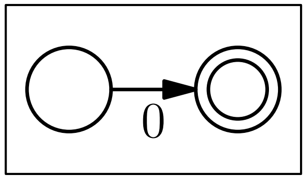
Example: regular expression to NFA

Example: $\mathbf{0} \cup (\mathbf{11})^*\mathbf{0}$









Example: regular expression to NFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

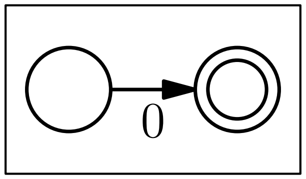
Equivalence of NFAs and DFAs

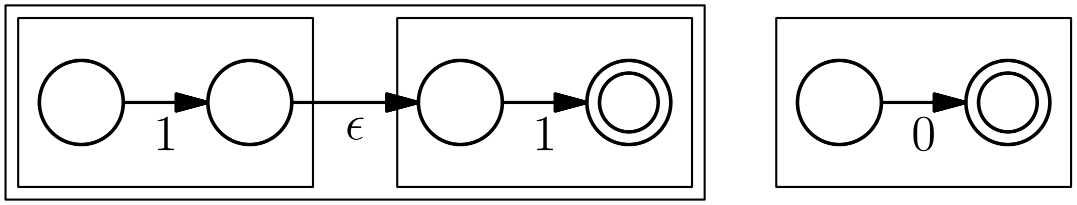
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Example: $\mathbf{0} \cup (\mathbf{11})^{*}\mathbf{0}$





Example: regular expression to NFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

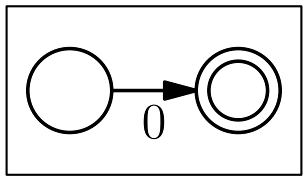
Equivalence of NFAs and DFAs

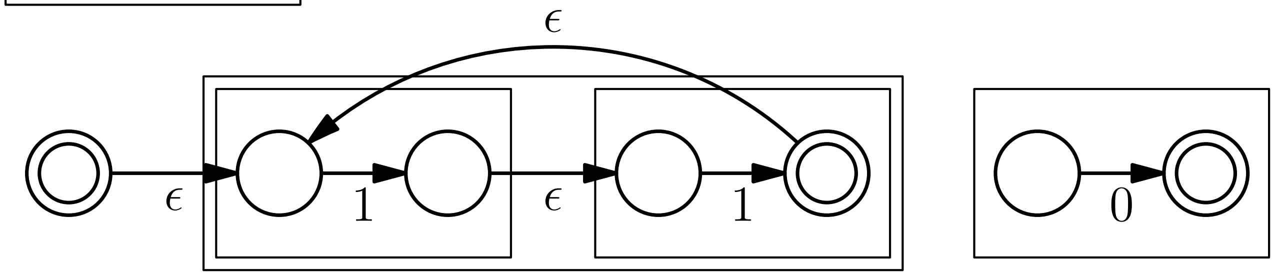
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Example: $\mathbf{0} \cup (\mathbf{11})^* \mathbf{0}$





Example: regular expression to NFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

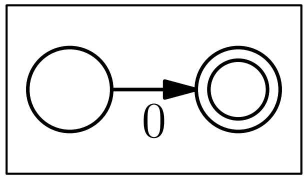
Equivalence of NFAs and DFAs

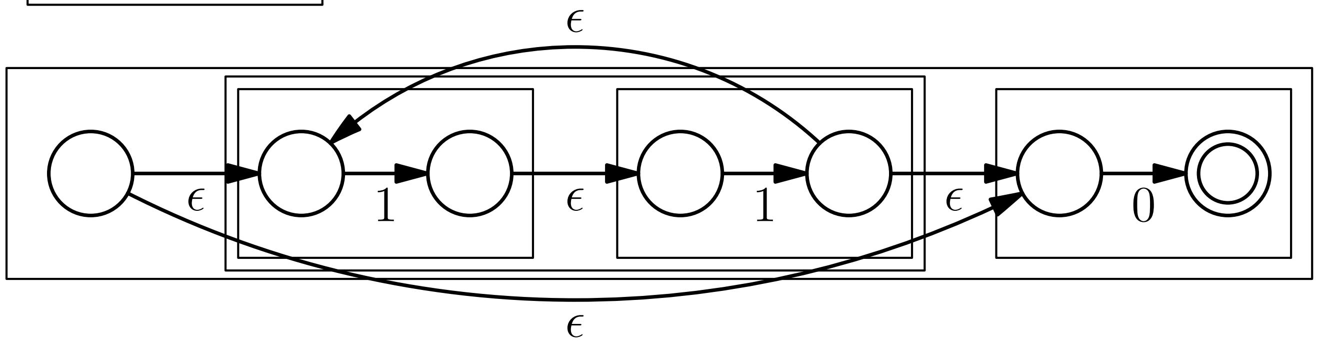
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Example: $\mathbf{0} \cup (\mathbf{11})^*\mathbf{0}$





Example: regular expression to NFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

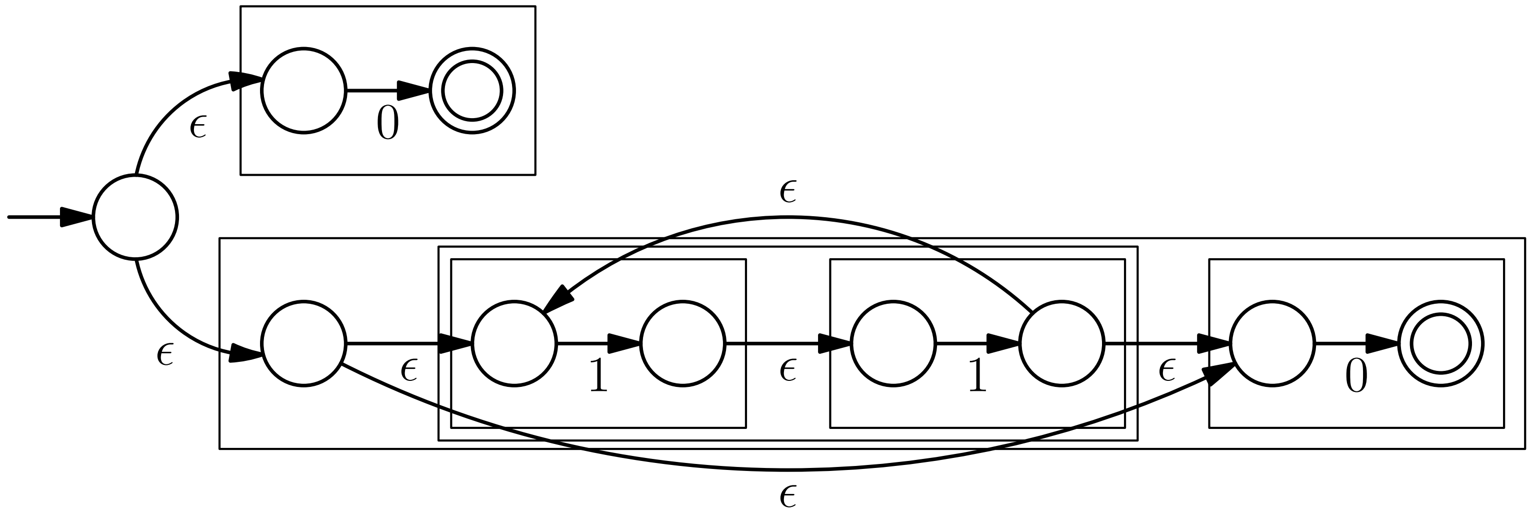
Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Example: $\mathbf{0} \cup (\mathbf{11})^*\mathbf{0}$



Proof of Correctness of Regular Expression to NFAs

Theorem

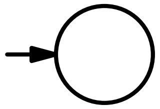
Given a regular expression R, let N be the NFA resulting from the above construction. Then $L(N) = L(R)$.

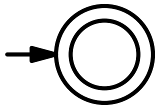
proof (sketch).

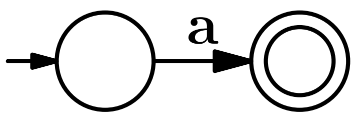
The proof is by induction on the structure of regular expressions, proving that $L(R) = L(N)$ for each case (I gave the arguments informally for each case during the description of the construction).

At all points below, N_i represents the NFA constructed from R_i.

Base cases: For each base case, it is obvious that $L(N) = L(R)$.







Proof of Correctness of Regular Expressions to NFAs, cont.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

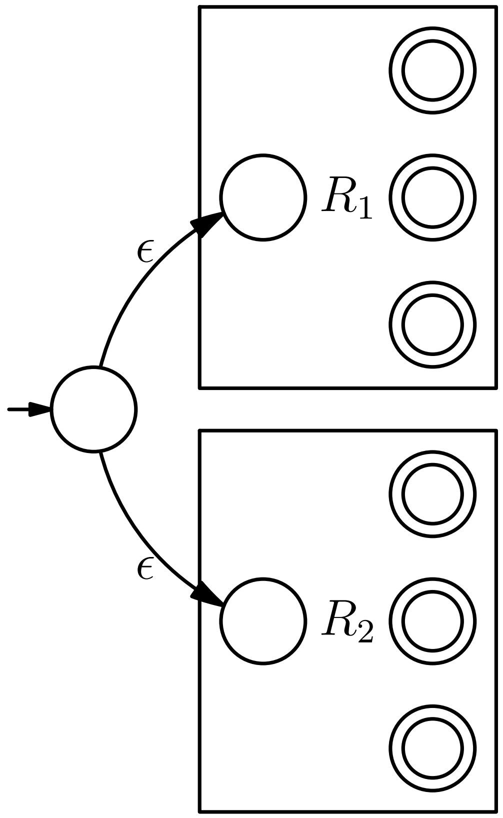
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Induction steps: In each induction step, we may assume the induction hypothesis that $L(N_i) = L(R_i)$.

Case: $R_1 \cup R_2$: $x \in L(R_1 \cup R_2)$ iff $x \in L(R_1)$ or $x \in L(R_2)$ iff x is accepted by the N_1 or N_2 iff it is accepted by N.



Proof of Correctness of Regular Expressions to NFAs, cont.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

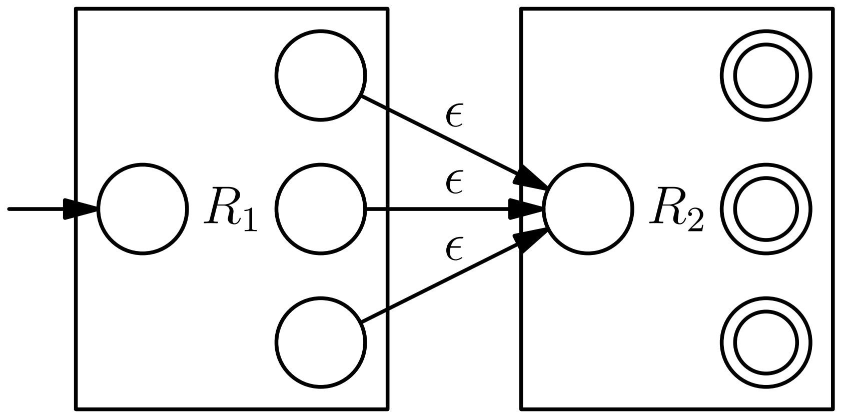
Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Case: $R_1 \cdot R_2$: $x \in L(R_1 \cdot R_2)$ iff $x = yz$ where $y \in L(R_1)$ and $z \in L(R_2)$ iff $x \in L(N_1)L(N_2)$ iff $x \in L(N)$.



Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

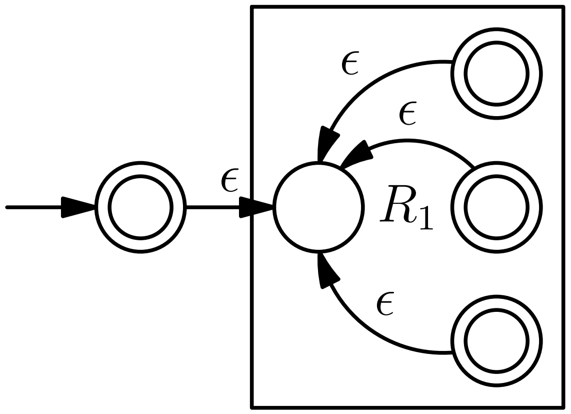
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Proof of Correctness of Regular Expressions to NFAs, cont.

Case: R_1^*: Proving this case is complicated. It uses the definition $L^* = \bigcup_{i=0}^{\infty} L^i$, and has its own nested induction on i. I'm going to skip the details.



Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Section

Minimizing DFAs

Minimum DFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

One of the coolest things about regular languages is that every language has a *unique* minimum-state DFA that accepts it.

Theorem $L_1 = L_2$ iff the minimum-state DFA accepting L_1 is the same as the minimum-state DFA accepting L_2.

This gives a nice decision procedure for equivalence of regular expressions or NFAs:

1. Convert each to a DFA.
2. Minimize the DFAs.
3. See if they are the same.

An Equivalence Relation on Strings

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Let L be a language over Σ^* and x and y be strings. $x \sim_L y$ if $xz \in L \leftrightarrow yz \in L$ for all $z \in \Sigma^*$.

In this case x and y are said to be *indistinguishable*.

When $x \not\sim_L y$, they are *distinguishable*, and there is some z such that $xz \in L$ and $yz \notin L$ (or *vice versa*). That z is called a *distinguishing input* for x and y.

$\sim_L$ is an equivalence relation. [*At this point, any of you should be able to prove this easily.*]

Example: Suppose $L = 0(10)^*$.

- Then $x \not\sim_L y$ if $x = 0$ and $y = 01$. If $z \in \epsilon$, the $xz = 0 \in L$ and $yz = 01 \notin L$.
- Then $x \sim_L y$ if $x = 0$ and $y = 010$.
- Then $x \sim_L y$ if $x = 1$ and $y = 011$ (for all z, $xz \notin L$ and $yz \notin L$.

Example Language

Example: For $L = 0(10)^*$, there are three equivalence classes:

- $[\epsilon] = \{\epsilon, 01, 0101, \ldots\}$. These strings are in L if concatenated with $0(10)^*$, not in L otherwise.
- $[0] = \{0, 010, 01010, \ldots\}$. These strings remain in L when concatenated with $(10)^*$.
- $[1] = \{1, 00, 10, 11, 011, \ldots\}$. These strings are not in L, no matter what suffix is appended.

From string equivalence to a DFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

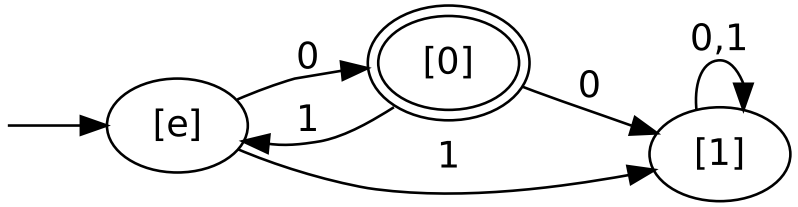
From Regular Expressions to Finite Automata

Minimizing DFAs

If $\sim_L$ has a finite number of equivalence classes, there is a DFA where strings go to the same state iff they are equivalent.

Define the DFA $D = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{[x] \mid x \in \Sigma^*\}$
- $\delta([x], a) = [xa]$ for all $x \in \Sigma^*$ and $a \in \Sigma$
- $q_0 = [\epsilon]$.
- $F = \{[x] \mid x \in L\}$.



Minimum-state DFA

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

Regular expressions

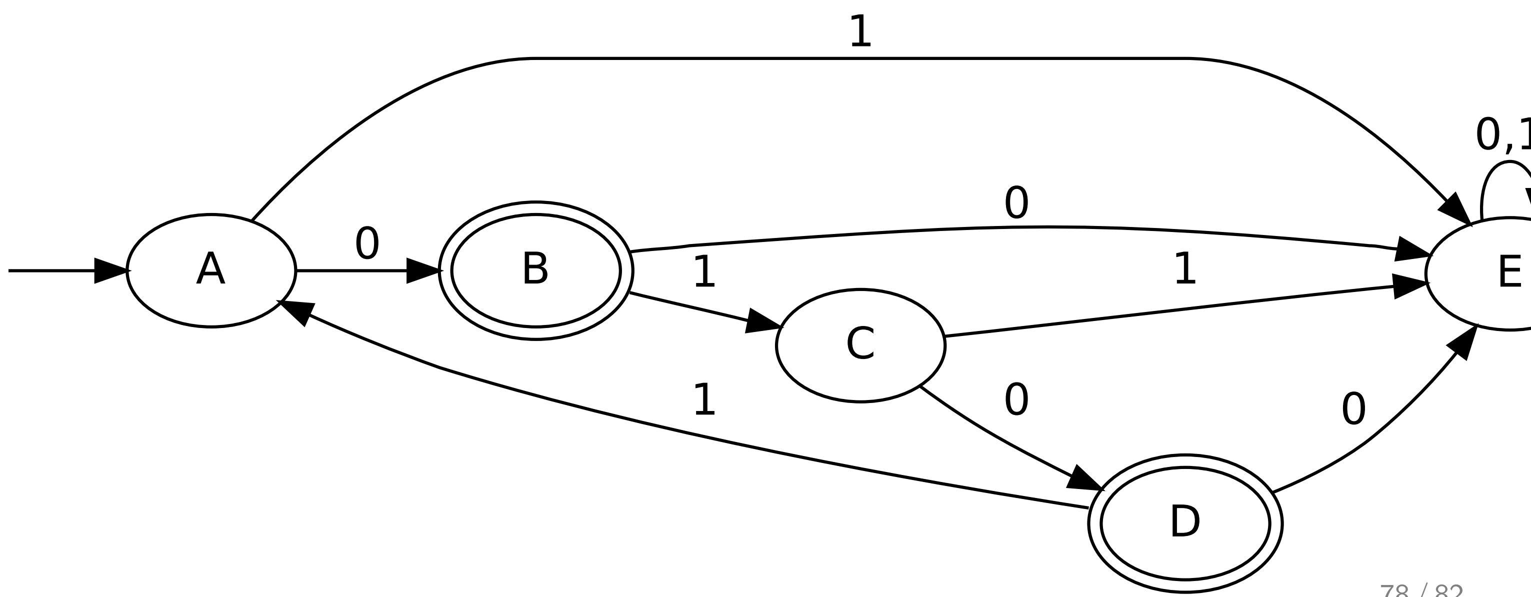
From Regular Expressions to Finite Automata

Minimizing DFAs

The DFA defined by $\sim_L$ is the *minimum-state DFA* for L.

There are many DFAs for any language that are *not* minimum-state.

Here is one:



Minimizing a DFA

Problem: Given a DFA, find a minimum DFA (in polynomial time).

Wrong approach: Try to find pairs of equivalent states.

Right approach: Start assuming everything is equivalent. Make them inequivalent only when forced to.

1. Eliminate all states that are not reachable from q_0.
2. Mark states p and q as "not equivalent" only if one is final and the other is not.
3. Whenever states p and q are not yet marked as "not equivalent" and $\delta(p, a)$ and $\delta(q, a)$ are marked as "not equivalent" for some $a \in \Sigma$, mark p and q as "not equivalent."
4. Repeat the previous step until no more pairs of states need to be marked as "not equivalent."
5. Merge sets of "equivalent" states into individual states of the resulting automaton (see example).

Correctness of the DFA minimization procedure

This definition uses the concept of a run starting from an arbitrary state instead of q_0.

Def: A string $z \in \Sigma$ *distinguishes* states p and q in a DFA if the run from p on z ends in an accept state and the run from q on z ends in a non-accept state or *vice versa.*

Theorem: States p and q are marked as “not equivalent” iff there is a string that distinguishes them.

I'm going to skip the proof, but it is by induction on the length of distinguishing strings and/or the number of applications of rules 2 and 3 on the previous slide.

Table filling algorithm

Build a 2D triangular Boolean array to represent whether each state is equivalent.

Mark more and more states as inequivalent until no more are marked.

B	x			
C		x		
D	x		x	
E	x	x	x	x
	A	B	C	D

This terminates because there are finitely many table entries.

$A = C$, $B = D$, E.

Review of Regular Languages for Introduction to Compilers

Formal Language Theory

Alphabets, Strings, and Languages

Regular Languages

Deterministic Finite Automata

Solving Problems on DFAs

Nondeterministic Finite Automata

Equivalence of NFAs and DFAs

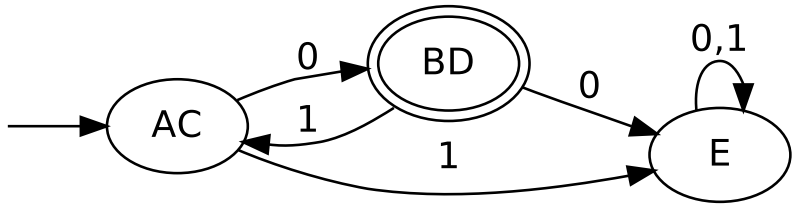
Regular expressions

From Regular Expressions to Finite Automata

Minimizing DFAs

Minimization example

To minimize, merge equivalent states.



Detail: Also should delete states that are not reachable from q_0.

The resulting DFA is the unique minimum-state automaton.

It is the same as every other minimum-state DFA, except for state naming.