St. Francis Xavier University
Department of Computer Science

CSCI 356: Theory of Computing
Lecture 1: Regular Languages
Fall 2023

1 Regex and Regular Expressions

If you frequently use a Unix-based system with a terminal, you may be familiar with utilities such as `grep`, which searches an input text file for lines that match a specified format. For example, on your computer, you can search the dictionary file (`/usr/share/dict/words`) for all words that contain `theory`:

```
taylor@SmithBook:~> grep theory /usr/share/dict/words
countertheory
theory
theoryless
theorymonger
```

But, to be fair, doing something like that is a bit overkill when you could just open the file in a text editor and use the Find tool to search for the word “theory”. Where `grep` really shines is when you need to search for text matching a *pattern*, like so:

```
taylor@SmithBook:~> grep ^u.*ity$ /usr/share/dict/words
ubiquity
ultimity
ultrafilterability
...
usability
utility
utterability
uxoriality
```

In this example, we searched for all words in `/usr/share/dict/words` that began with a `u` and ended with `ity`, such as `university`. The `ubiquity` of this pattern in the English language is evident:

```
taylor@SmithBook:~> grep ^u.*ity$ /usr/share/dict/words | wc -l
     235
```

Utilities like `grep` use patterns to perform fast searches in text files, and the sequence of symbols that makes up such a pattern is known as *regex* or, formally, a *regular expression.*

1.1 Definition

To define regular expressions, let’s think about the types of things we can match. For example, as a base case, we might want to be able to match nothing—this can be represented by a nonsensical regex like `a^`, which attempts to match a symbol `a` that occurs before the start of a line. We might also want to match an empty line (which is distinct from matching nothing!), which can be done with the regex `^$`.

Let’s now actually attempt to match something more meaningful. The smallest nonempty thing we can match is a single symbol, which can be matched by a regex consisting of the symbol itself; say, `a`. From this, we can build up more complicated regexes by joining together smaller ones. For instance, we can match two symbols by joining them together with a special "union" symbol; say, `(a | b)`, which matches lines that contain either an `a`, or a `b`, or both. We can also concatenate two regexes by simply putting them together—the regex `ab` matches an `a` immediately followed by a `b`. Lastly, it would be nice to incorporate some kind of repetition mechanism to match something never, once, or many times. This can be done using a special "star" symbol such as that in the trivial regex `.*`, which matches zero or more occurrences (represented by the star, `*`) of any symbol (represented by the dot, `.`).

Now, since we're in a mathematically oriented course, we should properly formalize each of these match types. Fortunately, we can bring over notions from mathematics to correspond to each match type. Matching nothing can be denoted by an empty set symbol, $\emptyset$. Likewise, matching an empty line is like matching a set that contains one element which has zero length—let's denote this zero-length element by the symbol ϵ. To denote the union of two regular expressions, we could use $\cup$, but since we're dealing with regular expressions and (strictly speaking) not sets, we'll instead use the symbol $+$. Concatenation is straightforward; we'll simply write the regular expressions side-by-side. Finally, we can keep the star symbol as it is.

Taking this all together, we arrive at a formal definition for regular expressions.

Definition 1 (Regular expression)**.** Let Σ be an alphabet. The class of regular expressions is defined inductively as follows:

1. $\boldsymbol{r} = \emptyset$ is a regular expression;
2. $\boldsymbol{r} = \epsilon$ is a regular expression;
3. For each $a \in \Sigma$, $\boldsymbol{r} = a$ is a regular expression;
4. For regular expressions $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, $\boldsymbol{r} = \boldsymbol{r}_1 + \boldsymbol{r}_2$ is a regular expression;
5. For regular expressions $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, $\boldsymbol{r} = \boldsymbol{r}_1\boldsymbol{r}_2$ is a regular expression; and
6. For a regular expression $\boldsymbol{r}$, $\boldsymbol{r}^*$ is a regular expression.

Note that our earlier regex examples used symbols like `^`, `.`, and `$`, when Definition 1 didn't define any of those symbols. This is because regexes and regular expressions aren't exactly the same thing. In fact, our definition of a regular expression is the purely theoretic definition, meant simply to give us the bare minimum needed to match simple patterns. It is therefore different from a practical regex implementation, where we can use special symbols to indicate the start or end of a word, match any symbol instead of one specific symbol, and make back-references, among other things. Appropriately, the literature sometimes refers to these practical regex implementations as *extended regular expressions*, and this is what you'll encounter on most computers. In the context of this lecture, though, when we refer to a regular expression, we will be following Definition 1.

1.2 Words and Languages

Another way in which regular expressions stand apart is in the terminology we use to refer to what we're matching. Since we aren't using a terminal to write our theoretical regular expressions, we likewise aren't matching lines in a text file. Instead, we will borrow some terminology from linguistics, which happens to be the field from which much of early theoretical computer science developed!

- The symbols we use, like $\{\mathtt{a}, \mathtt{b}\}$ or $\{\mathtt{0}, \mathtt{1}\}$, come from an *alphabet*. Often, we represent an alphabet by the symbol Σ.
- Sequences of symbols are called *words* or *strings*. For example, consider the English lowercase alphabet $\{\mathtt{a}, \mathtt{b}, \dots, \mathtt{z}\}$. Some words we can create with this alphabet are `cat`, `computer`, and `pneumonoultramicro-scopicsilicovolcanoconiosis`. We often use lowercase variables like w, x, y, or z to denote words, and we use the symbol ϵ to denote the special zero-length *empty word*.
- Sets of words are called *languages*. Much like with plain sets, we can either list the words in a language explicitly, or we can describe a language in terms of some property or properties of each word therein. For example, over the English lowercase alphabet, the language of words with three consecutive double letters is $\{\mathtt{bookkeep}, \mathtt{bookkeeper}, \mathtt{bookkeepers}, \mathtt{bookkeeping}\}$. Also much like sets, languages can be

either finite or infinite. We often use uppercase variables like L to denote languages, and we use the symbol $\emptyset$ to denote the special *empty language* containing no words.

1.3 Language of a Regular Expression

Every regular expression *represents* a language, and we denote the language represented by a regular expression $\boldsymbol{r}$ by $L(\boldsymbol{r})$. Note that each of the six basic regular expressions correspond to their own language. If $\boldsymbol{r} = \emptyset$, then $L(\boldsymbol{r}) = \emptyset$. Likewise, if $\boldsymbol{r} = \epsilon$, then $L(\boldsymbol{r}) = \{\epsilon\}$, and if $\boldsymbol{r} = a$, then $L(\boldsymbol{r}) = \{a\}$. The remaining three regular expressions correspond exactly to the union, concatenation, or repetition of their constituent languages. We will denote the class of languages represented by some regular expression by RE.

Often, figuring out the language represented by a given regular expression is as simple as reading through the regular expression and breaking it into its constituent components.

Example 2. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the language $L_{\text{odda}} = \{w \mid w \text{ contains an odd number of } \mathtt{a}\text{s}\}$. This language is represented by the regular expression $\boldsymbol{r}_{\text{odda}} = \mathtt{b}^*(\mathtt{ab}^*\mathtt{ab}^*)^*\mathtt{ab}^*$. The first component of $\boldsymbol{r}$, $\mathtt{b}^*$, recognizes zero or more leading bs. The middle component, $(\mathtt{ab}^*\mathtt{ab}^*)^*$, recognizes zero or more pairs of as, where each a is followed by zero or more bs. The last component, $\mathtt{ab}^*$, recognizes an additional a to ensure the total number of as is odd, followed by zero or more bs.

Note that regular expressions need not be unique; to illustrate, the same language in Example 2 is recognized by the regular expression $\boldsymbol{r}'_{\text{odda}} = \mathtt{b}^*\mathtt{ab}^*(\mathtt{ab}^*\mathtt{ab}^*)^*$.

Working in reverse, we can take a regular expression and determine the language it represents through a straightforward substitution process.

Example 3. Consider the regular expression $\boldsymbol{r} = (\mathtt{a} + \mathtt{b})^*\mathtt{b}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. We can "decompose" the language represented by $\boldsymbol{r}$ in the following way:

$$\begin{aligned}
L(\boldsymbol{r}) &= L((\mathtt{a} + \mathtt{b})^*\mathtt{b}) \\
&= L(\mathtt{a} + \mathtt{b})^* L(\mathtt{b}) && \text{(breaking apart concatenation)} \\
&= (L(\mathtt{a}) \cup L(\mathtt{b}))^* L(\mathtt{b}) && \text{(rewriting as union of languages)} \\
&= (\{\mathtt{a}\} \cup \{\mathtt{b}\})^*\{\mathtt{b}\} && \text{(rewriting as single-symbol languages)} \\
&= \{\mathtt{a}, \mathtt{b}\}^*\{\mathtt{b}\}. && \text{(rewriting as union of symbols)}
\end{aligned}$$

Therefore, $\boldsymbol{r}$ represents the language $L = \{w \mid w \text{ ends with } \mathtt{b}\}$.

The empty word ϵ and the empty language $\emptyset$ operate a little differently than other words and languages in regular expressions.

- For the empty word ϵ and any regular expression $\boldsymbol{r}$, we have that $\boldsymbol{r}\epsilon = \boldsymbol{r}$ but $\boldsymbol{r} + \epsilon \neq \boldsymbol{r}$ in general.
- For the empty word ϵ, we have that $\epsilon^* = \{\epsilon\}$.
- For the empty language $\emptyset$ and any regular expression $\boldsymbol{r}$, we have that $\boldsymbol{r} + \emptyset = \boldsymbol{r}$ but $\boldsymbol{r}\emptyset = \emptyset$.
- For the empty language $\emptyset$, we have that $\emptyset^* = \{\epsilon\}$.

Just like how mathematics has an order of operations, regular expressions abide by their own order of precedence. The star is always applied first, followed by concatenation, and then union. If we want to, we can modify the order in which operations are applied by adding parentheses to a regular expression, and this does not affect the language represented by that regular expression.

Example 4. Consider the regular expressions $\boldsymbol{r}_1 = \mathtt{0} + \mathtt{1}^*\mathtt{10} + \mathtt{1}^*$ and $\boldsymbol{r}_2 = (\mathtt{0} + \mathtt{1})^*\mathtt{1}(\mathtt{0} + \mathtt{1})^*$. Clearly, the only visual difference between $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ is the addition of parentheses. However, the languages represented by $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ are quite different:

- The expression $\boldsymbol{r}_1$ represents the language containing (i) the word 0, (ii) all words consisting of at least one 1 with one 0 at the end, and (iii) all words consisting of zero or more 1s.
- The expression $\boldsymbol{r}_2$ represents the language consisting of all words that contain at least one 1.

We may additionally define some shorthand notation to make our regular expressions look nicer, though strictly speaking, this notation is not "official". Recall that the star symbol matches zero or more occurrences of whatever it's associated with. If we wanted to match one or more occurrences, we could write $\boldsymbol{r}^+ = \boldsymbol{r}\boldsymbol{r}^* = \boldsymbol{r}^*\boldsymbol{r}$, and this is referred to as the "plus" symbol. Similarly, we can use exponents to denote iterated concatenation; that is, $\boldsymbol{r}^k$ denotes $\boldsymbol{r}$ concatenated with itself k times.

2 Regular Languages

Recall that, if we're given some sets, we can apply all sorts of operations to produce new sets. Of course, the operations we're most familiar with from set theory are those of union, intersection, complement, and difference. Since languages behave like sets, we can similarly apply operations to languages to produce new languages, just like we saw with our regular expressions. Indeed, the three operations we introduced in the previous section are so important that we give them a special name: the *regular operations.*

Definition 5 (Regular operations)**.** Let L, L_1, and L_2 be languages. The three regular operations are defined as follows:

- Union: $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$;
- Concatenation: $L_1L_2 = \{wv \mid w \in L_1 \text{ and } v \in L_2\}$; and
- Kleene star: $L^* = \bigcup_{i \geq 0} L^i$, where $L^0 = \{\epsilon\}$, $L^1 = L$, and $L^i = \{wv \mid w \in L^{i-1} \text{ and } v \in L\}$.

The union operation, naturally, works in exactly the same way for languages as it does for sets. The concatenation operation takes two words and "connects" the end of the first word to the beginning of the second word. Lastly, the Kleene star operation—or, as we previously referred to it, the star operation—is simply repeated concatenation of all words with all other words in some language.

Note that, since the Kleene star allows us to take zero copies of a word, the empty word ϵ is always included in the resulting language.

Example 6. Let $L_1 = \{\mathtt{a}, \mathtt{b}\}$ and $L_2 = \{\mathtt{d}, \mathtt{e}\}$. Then $L_1 \cup L_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}$, $L_1L_2 = \{\mathtt{ad}, \mathtt{ae}, \mathtt{bd}, \mathtt{be}\}$, $L_1^* = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \mathtt{aab}, \dots\}$, and $L_2^* = \{\epsilon, \mathtt{d}, \mathtt{e}, \mathtt{dd}, \mathtt{de}, \mathtt{ed}, \mathtt{ee}, \mathtt{ddd}, \mathtt{dde}, \dots\}$.

So, what makes these particular operations so special, and why do we refer to them as the "regular" operations? As it turns out, taking just these three operations is sufficient to allow us to define the smallest class of languages that is interesting enough to study[1]: the *regular languages.*

Definition 7 (Regular languages—language-theoretic def'n)**.** Let Σ be an alphabet. The class of regular languages is defined inductively as follows:

1. The empty language, $\emptyset$, is regular.
2. For each $a \in \Sigma$, the language $\{a\}$ is regular.
3. If L_1 and L_2 are regular, then $L_1 \cup L_2$ is regular.
4. If L_1 and L_2 are regular, then L_1L_2 is regular.
5. If L_1 is regular, then L_1^* is regular.

If we compare Definitions 1 and 7, we see some strong similarities. It seems that all languages that are regular can also be represented by a regular expression, and vice versa! We'll revisit this connection later, so keep it in mind.

[1]There is a smaller class called the class of *finite languages.* However, it's not too interesting: it consists only of languages with a finite number of words. Introducing the Kleene star allows us to produce infinite-size languages.

For now, you might be asking yourself: why do we call these expressions and operations and languages "regular"? Stephen Kleene[2] introduced the notion of a regular language in the 1950s, but his justification for the terminology was basically that he couldn't come up with any better name:

"We would welcome any suggestions as to a more descriptive term."
— Stephen Kleene, *Representation of Events in Nerve Nets and Finite Automata*
RAND Corporation Research Memorandum RM-704, 1951.

This, in turn, brings to mind Phil Karlton's famous quote:

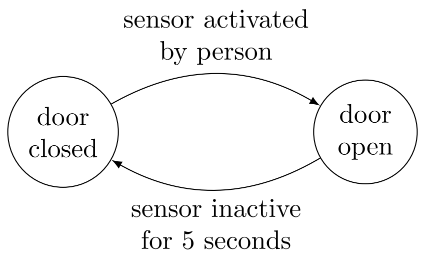
"There are only two hard things in Computer Science: cache invalidation and naming things."

3 Finite Automata

Theoretical computer science can be divided into two very broad categories: everything to do with formal languages, and everything to do with abstract machines. Now that we've established the fundamentals, let's change the course of our study away from languages and toward machines.

Some might argue that the entire point of studying computer science is to determine exactly what computers are capable of. After all, humans created computers so that we could pass off boring or repetitive work onto a machine and give our brains a break! However, considering a full computer in the very beginning of our studies is kind of like learning to swim by jumping into the deep end of a pool. In order to learn without getting overwhelmed, we will begin by considering a very simple model of computation that gives us just enough power to actually perform an elementary computation.

If you've ever used a vending machine, or waited in a car at a traffic light, or walked through an automatic door, then you're already familiar with the notion of a *finite automaton*. Consider, for example, how an automatic door works:



The door transitions between two states—closed and open—depending on what the sensor is reporting. The states (circles) represent the door's current status, and the transitions (arrows) correspond to an input given to the door. Note that the door has no way of knowing or remembering that it's closed or open apart from being in a state; it responds solely based on the input it receives from the sensor. This is a finite automaton: an automaton in the sense that it's a machine that performs an action based on predetermined conditions or instructions, and finite in the sense that there's a finite number of possible states the machine can be in at a given time.

3.1 Definition

We can use finite automata to model simple computations that take some input word and don't require memory or storage. In a computation, the states of the finite automaton correspond to our current step of the computation. For example, did we just begin the computation, or are we midway through reading some input, or something else? The transitions of the finite automaton take us between states, depending on the label of the transition. If we have, say, a binary word as the input to our finite automaton, then we can transition to a different state depending on whether the next symbol in the word is a `0` or a `1`.

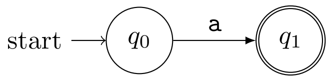
Formally speaking, a finite automaton is just a 5-tuple.

[2]This is the same Kleene for whom the "Kleene star" was named! His name will come up again later.

Definition 8 (Finite automaton)**.** A finite automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of *states*;
- Σ is an *alphabet*;
- $\delta \colon Q \times \Sigma \to Q$ is the *transition function*;
- $q_0 \in Q$ is the *initial* or *start state*; and
- $F \subseteq Q$ is the set of *final* or *accepting states.*

We're already familiar with states and alphabets, and we know a little bit about transitions from our example. The transition function δ is the mathematical formalization of the arrows in our diagram. Given an ordered pair of state and symbol being read, the transition function tells us which state to go to next. For example, if we had a very simple finite automaton like



then the single transition would be represented by the function $\delta(q_0, \texttt{a}) = q_1$. If a given finite automaton has a large number of transitions, then we can represent each transition concisely in a table format rather than writing each transition out individually.

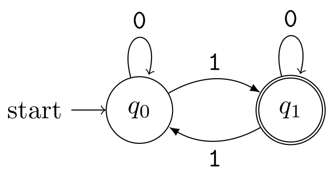
Note that, since we're dealing with a transition *function*, any pair of state and symbol can map to *at most* one state. This condition ensures that we always make the same transition on the same state/symbol pair.[3]

You may have noticed that the states in our very simple finite automaton had some special flair added to them. The state q_0 has an arrow labelled "start" pointing to it, and the state q_1 has two circles instead of one. This is how we denote initial and final states in our diagram. Initial states have an incoming transition arrow pointing at the state, while final states are double-circled. We typically have just one initial state in a finite automaton, but it's possible to have more than one. On the other hand, we can have as many or as few final states as we want.

Example 9. Consider the finite automaton $\mathcal{M}_1 = (Q, \Sigma, \delta, q_0, F)$ where $Q = \{q_0, q_1\}$, $\Sigma = \{\texttt{0}, \texttt{1}\}$, q_0 is the initial state, $F = \{q_1\}$, and δ is defined as follows:

	0	1
q_0	q_0	q_1
q_1	q_1	q_0

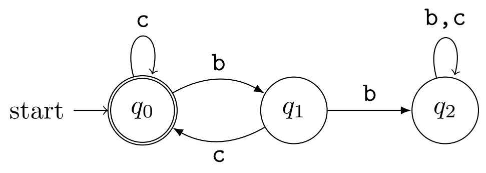
We can draw this finite automaton diagrammatically:



This finite automaton checks whether a binary word has odd parity; that is, whether it contains an odd number of 1s.

[3]Note that transition functions don't always have to behave in this way—just those that map to the state set Q. We'll soon see what happens if we don't enforce such a strict condition on our transition function, but for now, our finite automata will operate in this way.

Example 10. Consider the following diagram of a finite automaton:



This finite automaton checks whether every occurrence of `b` in an input word is immediately followed by an occurrence of `c`.

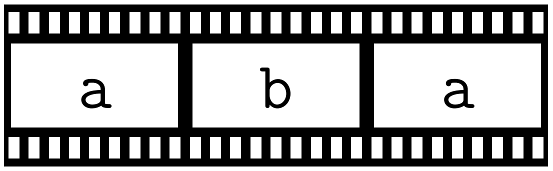
Based on this diagram, we can establish that $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{\texttt{b}, \texttt{c}\}$, q_0 is the initial state, $F = \{q_0\}$, and δ is defined as follows:

	`b`	`c`
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

3.2 Computations and Accepting Computations

Now that we know how to define a finite automaton, what can we do with it? Observe that, in our definition, we took care to specify the alphabet Σ. This alphabet gives us information about the kinds of *input words* we can give to a finite automaton. Giving an input word to a finite automaton is much like typing `input()` in a Python program or `scanf()` in a C program; it gives the computer something to read and work with.

When a finite automaton is given an input word, we can imagine the word is written on a reel of film where each symbol in the word has its own frame.



Now, imagine the finite automaton is a film projector, but the rewind button is broken. When we play the film reel starting at the first frame, the projector can only show one frame at a time, and once it moves to the next frame it can never return to the previous one. This is essentially how a finite automaton processes its input: starting with the first symbol of the input word, the finite automaton reads the symbol, transitions to a state, and then moves to the next symbol.

Once the finite automaton reaches the end of its input word and has no more symbols left to read, it must make a decision. Its decision depends entirely on the state it finds itself in at the moment it reaches the end of the word. If the finite automaton is in a final state and it has no more symbols left to read, then it *accepts* the word. Otherwise, the finite automaton must be in a non-final state, and it therefore *rejects* the word.

Going one step further, we can precisely define what it means for a finite automaton to accept an input word by introducing the notion of an *accepting computation*. An accepting computation is akin to a set of steps showing us every state a finite automaton enters from the moment it starts reading its input word to the moment it accepts its input. We don't need anything new to define this; we already have all the machinery we need.

Definition 11 (Accepting computation of a finite automaton)**.** Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton, and let $w = w_0 w_1 \dots w_{n-1}$ be an input word of length n where $w_0, w_1, \dots w_{n-1} \in \Sigma$. The finite automaton $\mathcal{M}$ accepts the input word w if there exists a sequence of states $r_0, r_1, \dots, r_n \in Q$ satisfying the following conditions:

1. $r_0 = q_0$;
2. $\delta(r_i, w_i) = r_{i+1}$ for all $0 \leq i \leq (n-1)$; and
3. $r_n \in F$.

In other words, the computation of a finite automaton must satisfy three conditions in order to be considered an accepting computation: it must start in the initial state, every subsequent state must be reachable by the transition function in one computation step after reading one symbol, and it must end in the final state.

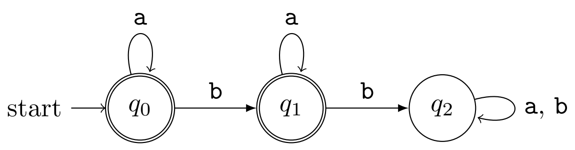
3.3 Language of a Finite Automaton

The set of all input words that a finite automaton $\mathcal{M}$ accepts is called the *language* of the finite automaton, denoted $L(\mathcal{M})$, and it's just like any other language: it consists of words over some alphabet Σ. If a finite automaton $\mathcal{M}$ accepts (or *recognizes*[4]) a language A, then $L(\mathcal{M}) = A$. Note that, although a finite automaton can accept possibly many input words, it can only recognize *one* language.

Example 12. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the language

$$L_{|w|_{\mathtt{b}} \leq 1} = \{w \mid w \text{ contains at most one occurrence of the symbol } \mathtt{b}\}.$$

This language is recognized by the following automaton:



If the input word w contains zero `b`s, then the finite automaton will remain in the final state q_0. Likewise, if w contains one `b`, then the finite automaton will enter and remain in the final state q_1. Only if w contains two or more `b`s does the finite automaton enter the state q_2, where it becomes "stuck" and can no longer accept the input word.

Example 13. A finite automaton with no final states cannot accept any words, but it is still able to recognize one language: the empty language $\emptyset$. This is because the language of input words accepted by the finite automaton is empty!

As a matter of notation, we will refer to the class of languages recognized by *some* finite automaton by the abbreviation DFA. (What does the D mean? We'll find out in the next section…)

Finally, recall our previous definition of a regular language. Since we're focused on machines now instead of languages, it would be nice to have an analogous definition that applies to finite automata. Fortunately, such a definition is simple to formulate.

Definition 14 (Regular languages—automata-theoretic def'n)**.** If some finite automaton $\mathcal{M}$ recognizes a language L, then L is regular.

That's it! Wait, that's it? Indeed—any language recognized by a finite automaton is by definition regular. While this definition may seem a bit unsatisfying when compared to our much more detailed previous definition of regular languages, we'll soon see how languages, regular expressions, and finite automata tie together in a more rigorous way.

[4]For clarity's sake, here the word "accept" will be reserved for input words given to a finite automaton, while the word "recognize" will be used to refer to the language of a finite automaton. Both words essentially mean the same thing: the finite automaton has given us a positive answer. Unfortunately, many authors and textbooks use these words interchangeably.