Automata and Languages

1 Languages, etc.

An *alphabet* is a set of symbols. We won't define the term *symbol*; it's a primitive term. The most intuitive way to think of symbols to to make them synonymous with *characters*—but they could be other things. Traditionally Σ is used to stand for an alphabet, so we could say, for example, $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. *alphabet*

A *string* over an alphabet is a sequence of symbols of the alphabet. We use $\Sigma*$ to stand for the set of all strings over an alphabet (the idea is that $*$ means "zero or more things drawn from..."). Since symbols may be repeated and since strings may be arbitrarily long, $\Sigma*$ is necessarily an infinite set, unless Σ itself is empty. You are familiar with such concepts as string concatenation, reversal, length, substring, and the empty string. (We don't want to call the empty string $\emptyset$, because a string is a sequence, not a set; instead we call the empty string ε. (If ε happens to be a symbol in the alphabet, we're in trouble.)) *string*

A *language* over an alphabet is a set of strings over that alphabet; alternately, a language over Σ is any subset of $\Sigma*$. This may sound like an odd way to define what a language is since this means a language can be pretty much anything. It's used for talking about the mathematical concept of sets of strings, but the definition works for natural languages, too. For example, the string "*Je voudrais du café*" is in the language French. It's just harder to define such languages formally. (To stretch things further, this definition works for spoken natural languages as well as for written; the "alphabet" in that case is the set of phonetic symbols rather than written symbols. Sign languages fit the definition, too.) *language*

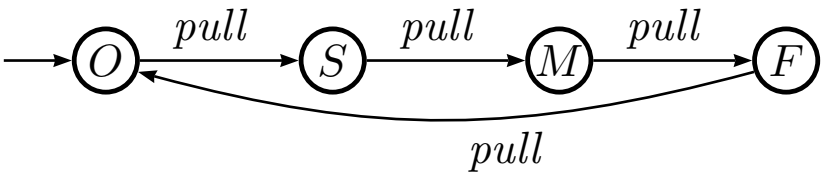
2 Finite state machines

An *automaton* is any model of a machine that works on its own (i.e., it works *autonomously* or *automatically*). The idea is that the power supply or whatever causes it to act is not included in the model. *automaton*

The simplest kind of automaton is a *finite state machine* (FSM). It's a machine that has a finite number of possible states, and it transitions among them based on its input. Its input consists in

symbols from an alphabet, given one at at time. Real-world examples—or, at least, analogies—include parking meters and ceiling fans.

There is a standard way to draw FSMs. If we assume the fan's states to be $\{O, S, M, F\}$ for off, slow, medium, and fast, respectively, then:

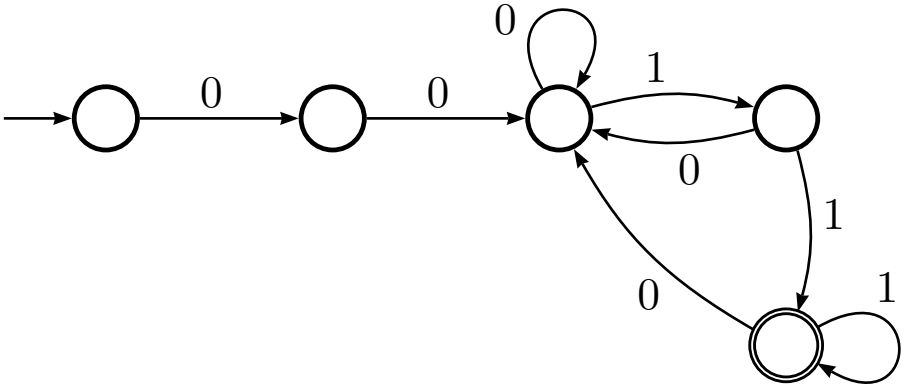


Formally, a FSM is a set of states Q together with an alphabet Σ and a transition function $\delta : Q \times \Sigma \to Q$. The transition function takes a (current) state and an input symbol and returns a new state. Typically there is a designated state $q_0 \in Q$ which is called the *start* or *initial state*. In diagrams, it is identified by an arrow from nowhere pointing to it; it's usually placed on the leftmost part of the diagram. Most FSM also have at least one *final state*, which is identified by a double circle.

initial state

final state

FSMs are used to specify (or implement, depending on how you look at it) a language. Take the alphabet $\Sigma = \{0, 1\}$ and the language of strings that begin with 00 and end with 11, such as 0011, 00011, 00111, 0001001010110011, etc.



The way to read this is that the left "branch" shows the entry into the system—a string must begin with two 0's. Then transitions happen among the three states on the right, but in order to get to the lower state, we must have come in on a path where the last two inputs were 1's. If we are in that state when the input ends, we accept the string as being in the language. For this reason the final state is also called the *accept state*.

This picture does not take into account what happens if erroneous input—something that forms a string which is not in the language—is given. Either we must allow δ to be an incomplete function (there are some state, input pairs for which there is not next state), or we say that there is also an extra state, the "reject" state, to which all otherwise undefined pairs transition. We leave this reject state out of the diagram to reduce clutter.

Because of this perspective, that an FSM accepts or rejects a string as being in a language, FSMs are also called *deterministic finite accepter*s (DFAs). (The abbreviation DFA is also sometimes interpreted as standing for "deterministic finite automaton.")

deterministic finite accepter

3 Regular expressions

A *regular expression* is a certain way of specifying a language. Since a language is a set, it's important to remember that a regular expression is a description of a set. The rules for writing regular expressions, given an alphabet Σ, are that a regular expression is one of

regular expression

- $\emptyset$, which denotes the set $\emptyset$, that is, the set/language of no strings.
- ε, which denotes the set $\{\varepsilon\}$, that is, the set/language containing only the empty string.
- $\mathtt{a}$, where $\mathtt{a} \in \Sigma$, which denotes the set $\{\mathtt{a}\}$, that is, the set/language containing only the string with only the symbol $\mathtt{a}$.
- $r \mid s$, where r and s are regular expressions, which denotes the set/language $r \cup s$ (remember that r and s each denote a set).
- rs, where r and s are regular expressions, which denotes the set/language of strings each composed of a string from r concatenated with a string from s; formally, $\{RS \mid R \in r \text{ and } S \in s\}$
- $r*$, where r is a regular expression, denoting the set of strings which are composed of the concatenation of zero or more strings, each from r; formally, $\{R_0 R_1 \dots R_n \mid n \in \mathbb{W} \text{ and } \forall\, i, 0 \leq i \leq n, R_i \in r\}$.

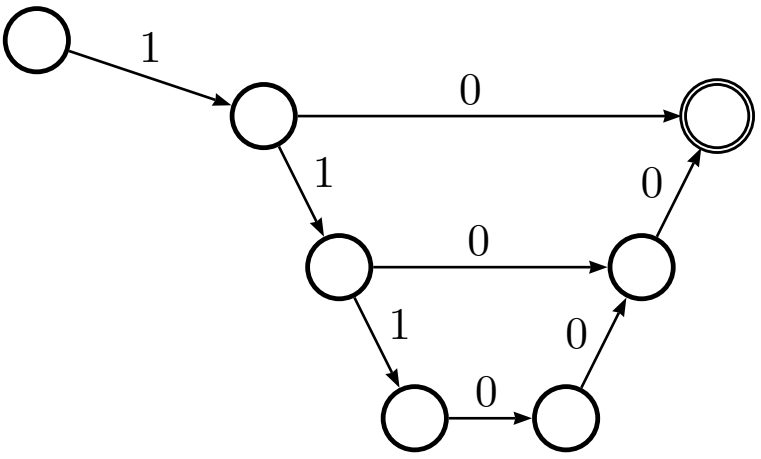
For example, the language in the previous section is described by the regular expression $00(0|1)*11$. (Note that parentheses is also legal, used for grouping; There are also a variety of extensions to regular expressions that are popularly used to make them more concise. For example, $r+$ means "one ore more occurrences of..."; thus $r+$ is equivalent to $rr*$. These are notationally convenient but do not add to the "power" of regular expressions, which we discuss below.)

It turns out that the set of languages that can be described using FSMs is the same as the set of languages that can be described by regular expressions (in this sense, regular expressions and FSMs are equivalent). The proof of this is a bit long; we'll demonstrate this intuitively by giving an example of a language that neither can handle.

Consider the language of strings that begin with a certain number of 1's followed by the same number of 0's: $\{10, 1100, 111000, 11110000, \dots\}$.

The regular expression $1 * 0*$ won't work because it also allows strings like 110. The pseudo-regular expression $1^n 0^n$, taken to mean for all n, n instances of 1 followed by n instances of 0, would work, but it is not a regular expression. In short, regular expressions have no way of counting; they also have no mechanism for recursion.

The following FSM attempts to accept this language:



However, this works only for series of 1's and 0's no longer than 3. To handle arbitrarily long strings in this language, the FSM itself would to be arbitrarily big—in which case it would no longer be a *finite* state machine.

In the practice of specifying programming languages, FSMs/regular expressions are powerful enough to describe the "language" of legitimate tokens in the language (where a *token* is a keyword, identifier, literal, punctuation, or comment), but not for specifying the entire programming language.

(In the C family of languages, block comments cannot be nested. The Java compiler, for example, will reject

```
/*   I'm commenting this code out

    /* add 1 to i. */
    i++;
*/
```

Why? If you can't figure it out, ask Tyler.)

4 Context-free grammars

Instead, we can specify our language, like this:

$$\begin{array}{rcl} S & \rightarrow & \varepsilon \\ & | & 1\ S\ 0 \end{array}$$

This means, "a string in our language can be either an empty string, or it can be the sequence formed by 1 followed by another string in our language followed by 0."

We saw this notation some weeks ago in your tree assignment. What we've used to specify the language is a *context-free grammar*. Formally, a context-free grammar is a set of *terminal symbols* T, a set of *non-terminal symbols* NT, and a set of *productions* P where each production has the form $X \rightarrow S$ where $X \in NT$ and $S \in (T \cup NT)*$, that is, the left-hand side is a non-terminal and the right-hand side is a string of terminals and non-terminals. (The two productions $X \rightarrow S_1$ and $X \rightarrow S_2$ may be abbreviated to $X \rightarrow S_1 \mid S_2$.) Since the productions can be recursive, it can handle languages that require some sort of counting. Thus context-free grammars are more powerful than regular expressions, and they are used to specify programming languages.

context-free grammar

The particular notation used by computer scientists is called *Bakus-Naur Form*, after John Bakus and Peter Naur, and it is similar to (and expressively equivalent to) a notation called *Chomsky Normal Form*, named after Noam Chomsky, used by linguists. Context-free grammars can capture most of the grammatical phenomena in natural languages, for example

Bakus-Naur Form

$$\begin{array}{rcl} T & = & \{\texttt{a}, \texttt{any}, \texttt{every}, \texttt{loves}, \texttt{man}, \texttt{seeks}, \texttt{the}, \texttt{unicorm}, \texttt{woman}\} \\ NT & = & \{\mathit{NounPhrase}, \mathit{Noun}, \mathit{Quantifier}, \mathit{Predicate}, \mathit{TransitiveVerb}, \mathit{Sentence}\} \end{array}$$

$$\begin{array}{rcl} \mathit{Sentence} & \rightarrow & \mathit{NounPhrase}\ \mathit{Predicate} \\ \mathit{NounPhrase} & \rightarrow & \mathit{Quantifier}\ \mathit{Noun} \\ \mathit{Predicate} & \rightarrow & \mathit{TransitiveVerb}\ \mathit{NounPhrase} \\ \mathit{Noun} & \rightarrow & \texttt{man} \mid \texttt{unicorm} \mid \texttt{woman} \\ \mathit{Quantifier} & \rightarrow & \texttt{a} \mid \texttt{any} \mid \texttt{every} \mid \texttt{the} \\ \mathit{Verb} & \rightarrow & \texttt{loves} \mid \texttt{seeks} \end{array}$$