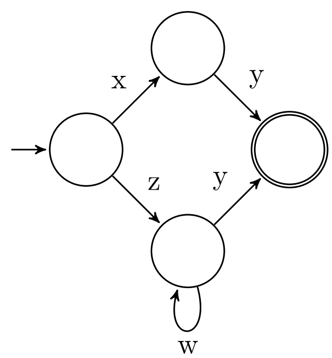
What to Declare? Let's say we have this automaton.



This corresponds to the language:

We can give a context-free grammar (more later) for regular expressions. It contains characters, which directly match parts of the input string, and meta-characters, which combine regular expressions.

R	:=	`char`
	|	LPAREN R RPAREN | RR
	|	R STAR | R PLUS
	|	R BAR R
	|	LBRACKET `char` MINUS `char` RBRACKET
	|	LBRACKET CARET `char` RBRACKET | etc.

(The last two lines and R^+ are syntactic sugar; you can write any regular expression without them, although it might be unwieldy.)

The meaning of these regular expressions is as follows:

`char`	match character `char`
R_1R_2	concatenation: accept R_1 followed by R_2
R^*	Kleene star: accept 0 or more instances of R
R^+	accept 1 or more instances of R
$R_1 \| R_2$	union: accept either R_1 or R_2
[$\texttt{c}_1$-$\texttt{c}_2$]	accept characters between $\texttt{c}_1$ and $\texttt{c}_2$
[ˆ`c`]	accept all characters except `c`

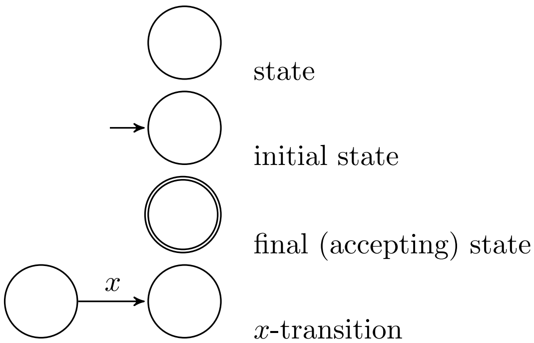
We will implement regular expressions using deterministic and non-deterministic finite automata.

Deterministic Finite Automata

DFAs can be used to implement regular expression *recognizers*; they say if an input belongs to a “language” (or not).

Sample question: “Is this string a valid JavaScript identifier or not?” Yes: `n00b`; no: `87x`.

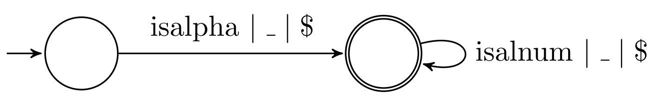
Recall the DFA diagram from before. We can now identify each of its parts:



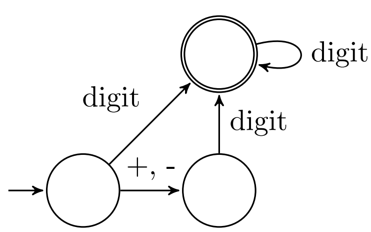
A finite automaton *accepts* a string if it reads the entire string, ending in an accepting state. It *rejects* the string of: 1) the automaton is not in an accepting state after the end of the string; or 2) the automaton gets stuck while reading the string and has no outgoing transition corresponding to an input. The set of strings accepted by a finite automaton constitutes its *language*.

Some strings to try out: `zwwwy`, `zw`, `zx`.

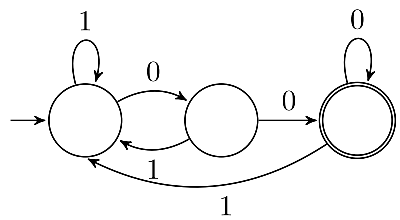
Some more Finite Automata. JavaScript identifiers:



Integer Literals:



Another example (what is it?)

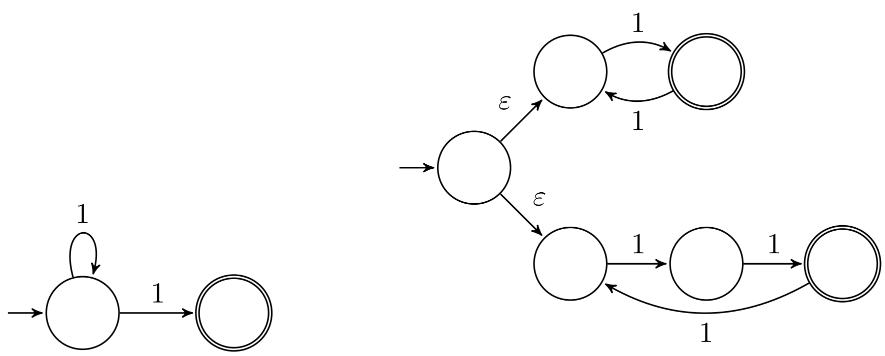


Nondeterministic Finite Automata

So far, we've talked about DFAs, where each character tells the DFA exactly which transition to take. In a *nondeterministic finite automaton*, the automaton may have a choice about which transition to take (e.g. among two transitions). If any of the choices leads to a final state at the end of the string, the NFA accepts the word.

Note: "finite" here refers to the internal memory of the finite automata; these automata only remember their current state.

Two NFA examples.



Executing finite automata

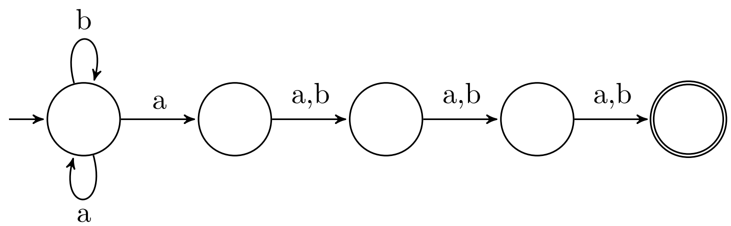
Recall that we were developing a domain specific language for text processing. We are interested in executing our languages, so we'll see how to execute finite automata next.

For a DFA: store the current state, change states upon input. (Use a transition table or a switch statement.)

For an NFA: not completely obvious: how do we know which choice to take? We need to deal with ε transitions and cases where multiple transitions are allowed.

NFA solution: view current state of the NFA as a set of states.

NFA execution example. Let's see how to run this NFA, which recognizes $(a|b)^*a(a|b)^3$.



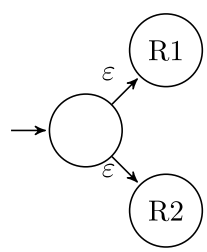
Sample executions: `aaaa`; `abaaa`.

Reminder: NFA accepts if it *can* reach a final state.

Expressive Power

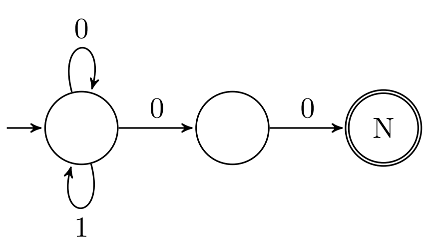
Same for NFAs and DFAs, because we can convert NFAs into DFAs. Both of these automata accept so-called "regular languages" (think "regular expressions").

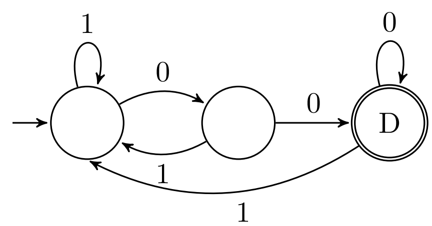
Advantage of NFAs. Allow composition of automata.



Advantage of DFAs. Easier and faster to implement: since they are deterministic, only track one state at a time.

NFAs versus DFAs.





Efficiency

We had a question about efficiency of regular expressions. They're fast. DFAs can recognize word x in time $O(|x|)$, while NFAs take time proportional to $|x|$ multiplied by the size of the NFA's transition graph. However, converting an NFA to a DFA takes time and may cause exponential blowup in the number of states (as in the example on the previous page.)

You'd probably use a DFA for a parser, because you just pay the conversion cost once, while for a domain-specific language which handles user-specified regular expressions, you'd probably use an NFA. It's a question of amortizing the NFA to DFA compilation cost.