One usefulness of the "set of triples" definition of a DFA is that it extends naturally to define a nondeterministic finite automaton (NFA). We can recap it in a general way:

The formal definition of a finite automaton is a 5-tuple (i.e., an object) \( N = (Q, \Sigma, \delta, s, F) \) where:

- \( Q \) is a finite set of states
- \( \Sigma \) is the input alphabet
- \( s \), a member of \( Q \), is the start state (also called \( q_0 \))
- \( F \), a subset of \( Q \), is the set of accepting states (also called final states)
- \( \delta \) is a finite set of instructions (also called transitions) of the form \((p, c, q)\) where \( p, q \in Q \) and \( c \in \Sigma \).

The machine is deterministic (a DFA) if \( (\forall p \in Q)(\forall c \in \Sigma)(\exists! q \in Q) : (p, c, q) \in \delta \). Else it is "properly" nondeterministic (an NFA).

So DFA is a special case of an NFA. When we have a DFA \( M \), we can regard \( \delta \) as a function from \( Q \times \Sigma \) to \( Q \). With an NFA, we could regard \( \delta \) as a function from \( Q \times \Sigma \) to \( 2^Q \), which is the set of all subsets of \( Q \) and called the power set of \( Q \). But in most cases I prefer to think of \( \delta \) as a set of instructions.

**NFAs with \( \epsilon \)-transitions**

The NFA can be augmented by allowing it to move from a state \( p \) to a state \( q \) without changing a character. The instruction is then written as \((p, \epsilon, q)\). Then we have

\[ \delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q. \]

This is confusing because \( \epsilon \) is a string, not a character in \( \Sigma \). We will shortly see the point of this extension. The Sipser text, on which Debray's notes are based, makes this the standard definition of NFA, while other sources call it an NFA with \( \epsilon \)-transitions (\( \text{NFA}_\epsilon \)). The extension does not affect the formal definition of the language of the NFA, which I give in my own terms as follows:
Say that $N$ can process a string $x$ from state $p$ to state $q$ if there is a sequence of instructions

$$(p, c_1, q_1)(q_1, c_2, q_2)(q_2, c_3, q_3) \cdots (q_{m-2}, c_{m-1}, q_{m-1})(q_{m-1}, c_m, q)$$

such that $c_1c_2 \cdots c_m = x$. Then we write $x \in L_{pq}$ (with $N$ understood). Now formally define:

$$L(N) = \cup_{f \in F} L_{sf}.$$ 

If $N$ has only one accepting state $f$ (a design goal we can meet for NFAs but often not for DFAs) then the language is just $L_{sf}$. We will find the $L_{pq}$ concept especially handy with "GNFAs" later on.

$\begin{align*}
x &= \$DD \\
N:
\quad &\rightarrow D \\
\quad &\rightarrow x \in L_{s,dead} \text{ but } dead \text{ is not accepting} \\
\quad &\rightarrow so \ x \text{ is not in the language.}
\end{align*}$

Without the dead state and arc to it, the NFA $N$ on input $x = \$DD$ would "crash" in state $s$. Even though $s$ is an accepting state (and even though this would count as legal termination by a Turing machine), not all of $x$ would be processed, so it does not count in the FA's language. With the dead state present, $x$ gets processed to $dead$, but $dead \notin F$ so $x \notin L(N)$ still.