Sipser says that “we almost never give formal descriptions of Turing machines because they are too big” (page 167). This is not quite correct. Automata theorists manage the design and implementation of large, complex Turing machines by modularizing them, just as a programming team manages the design and implementation of a large, complex application program.

A more complete and detailed presentation of Turing machines would begin with the formal definition, present a few small Turing machines like Examples 3.7 and 3.9, but then develop a library of general higher-order operations for tweaking arbitrary Turing machines to give them some useful property (such as erasing the tape before transitioning to \( q_{\text{accept}} \) or \( q_{\text{reject}} \)) and for combining arbitrary Turing machines to enable them to perform two or more computations in succession, to use the result of one Turing-machine computation to determine which of two other Turing-machine computations to initiate, and so on.

By giving names to these higher-order operations and settling on a precise syntax for using them, we could quickly develop a special-purpose programming language in which the directions for constructing large, complex Turing machines from small and simple ones can be concisely expressed. With a modest library of small and simple Turing machines to use as starting points and another modest library of higher-order operations (for sequencing, conditional execution, and iteration, perhaps), it is straightforward to produce completely formal specifications of Turing machines that have thousands or millions of states. The specifications look like programs in this special-purpose programming language.

Sipser doesn’t do this because devising and implementing the higher-order operations, proving their correctness, and defining a syntax for the programming language would require him to address a lot of fiddling little details that are not interesting to most readers and are, from a mathematician’s point of view, unimportant. However, you should remind yourself, from time to time, that when Sipser asserts the existence of a Turing machine that implements some algorithm that he describes in a mixture of English prose and pseudocode, you are expected to believe that that Turing machine conforms precisely to Definition 3.3, even if displaying its state diagram would require a sheet of paper the size of MacEachron Field.

It is possible for a pushdown automaton or even a nondeterministic finite automaton to loop forever without consuming any input and hence without ever conclusively accepting or rejecting that input. We got away with ignoring that possibility because, for regular and context-free languages, it turns out that there is no difference between recognizability and decidability. If we don’t want to work with a nondeterministic finite automaton that can loop forever, we can always find another nondeterministic finite automaton that recognizes the same language but doesn’t loop, and similarly for pushdown automata.

For Turing machines, however, the distinction is important, because there are some languages that are recognizable by Turing machines, but only by Turing machines that are capable of looping behavior. Those languages are not Turing-decidable (by
Definition 3.6). We’ll eventually encounter many such languages, beginning with one called \( A_{TM} \) (in Theorem 4.11, page 202).