Like the universal deterministic finite automaton simulator in Theorem 4.1, the universal Turing machine $U$ in Theorem 4.11 may initially seem counterintuitive — how can any automaton simulate the operation of another automaton that has more states than it itself has, or a larger input alphabet, or a more intricate transition table? But recall that $U$ treats the machine $M$ that it is simulating purely as a datum, an input value to be processed. The fact that the datum is sometimes very large and complicated doesn’t pose any special difficulty for the automaton that processes it, even if the input is longer than the description of $U$ itself would be. You yourself have probably written at least one program that can process inputs that are larger than the source code for the program itself.

To put it another way: The universal Turing machine $U$ is analogous to an interpreter that is both written in Scheme and capable of interpreting Scheme programs. Such an interpreter has a fixed size, but that doesn’t keep it from successfully interpreting Scheme programs that exceed that fixed size. Similarly, any C compiler is capable of compiling C programs larger than itself. There’s no paradox here. Such a compiler does not have to “know” anything about the applications it compiles. It needs to “know” only about C and about the processor that will execute the instructions it generates. So everything it needs can be written into in a program that has a fixed size.

It may have occurred to some of you to wonder: Can the universal Turing machine $U$ simulate itself? The answer is yes. On input $\langle U, \langle M, w \rangle \rangle$, $U$ will simulate $U$ on input $\langle M, w \rangle$ as it in turn simulates $M$ on input $w$. If $M$ ever enters its accept state, then the simulated $U$ will enter its accept state, in which case the $U$ that is simulating $U$ will enter its accept state. Feel free to add more layers of simulation if you like.

The “important early role” that the universal Turing machine played in the development of stored-program computers is that it convinced military authorities and manufacturers of business machines that stored-program computers were conceptually possible and would not even be particularly difficult to design and build. Before Turing’s work, essentially all business machines were single-purpose devices, or at best very limited in their repertoire (e.g., adding machines could also do subtraction). The idea of using a single physical device both to sort data values and to perform arithmetic operations on them (for example) was completely revolutionary.

Cantor was not the first to notice that there is a bijection between the positive integers and a proper subset of the positive integers. In Dialogues concerning Two New Sciences, published in 1638, Galileo Galilei pointed out that there is a bijection between the positive integers and their squares. But Galileo considered this a paradox because he was unable to reconcile it with the intuition that “the whole is greater than the part.” He concluded that, when considering infinities, it was nonsensical to try to compare them as less, equal, or greater.
Cantor’s original idea was to create new definitions of less, equal, and greater for infinite sets around the existence or nonexistence of bijections and surjections. Two sets, even two infinite sets, are equal in cardinality by definition if there is a bijection from one to the other. The cardinality of set $A$ is less than the cardinality of set $B$ if there is no surjective function from $A$ to $B$. Under Cantor’s definitions, we must indeed abandon some instances of the generalization that “the whole is greater than the part” and acknowledge that the set of positive integers and the set of squares of positive integers are equal in cardinality even though the latter is a proper subset of the former. But this concession does not lead to any inconsistency. On the contrary, it opens up the possibility for exactly the kind of theory of infinite sets that Galileo (prematurely) gave up on.

Diagonalization proofs are proofs by contradiction. Cantor developed the method in order to prove that the cardinality of one infinite set $A$ is less than the cardinality of another infinite set $B$, by assuming that there is a bijection $f : A \rightarrow B$ and then constructing a member $y$ of $B$ such that $f(x) \neq y$ for any member $x$ of $A$. If $f$ were actually a bijection, there could be no such $y$. That’s the contradiction. If this reasoning holds for every function $f : A \rightarrow B$, then no such function is surjective, so the cardinality of $A$ must be strictly less than the cardinality of $B$. Cantor initially proved this for the sets $\mathcal{N}$ and $\mathcal{R}$, using essentially the argument used in Sipser’s proof of Theorem 4.17 (pages 205–206).

The importance of the definition of uncountability, Definition 4.14, is that it simplifies the statement and proof of a lot of theorems about cardinality, because of the following lemma, which Sipser doesn’t state formally, but which is an obvious consequence of Definition 4.14: The cardinality of any countable set is strictly less than the cardinality of any uncountable set, so there is no bijection between any countable set and any uncountable set. Sipser uses this lemma tacitly in his proof of Corollary 4.18: The set of Turing machines is countable, but the set of all languages (on any fixed alphabet $\Sigma$) is uncountable, so there must be languages that aren’t recognized by any Turing machine.

The diagonalization proof of Theorem 4.11 (pages 207–209) isn’t about cardinality, but it is again a proof by contradiction. Assuming the existence of a decider $H$ for the language $A_{TM}$ enables us to construct a string $\langle D, \langle D \rangle \rangle$ that $H$ can demonstrably neither accept nor reject, so that it cannot be a decider after all.

In the first paragraph of the proof of Theorem 4.22, Sipser skips a step that some readers may need to see explicitly: Assume that $A$ is decidable, and let $M$ be deterministic single-tape Turing machine that is a decider for $A$. Then $M$ recognizes $A$, and we can define another deterministic single-tape Turing machine $M'$ that rejects any input that $M$ accepts and accepts any input that $M'$ rejects: $M'$ is exactly like $M$ except that its final states are exactly the non-final states of $M$. Then $M'$ recognizes $\overline{A}$. So both $A$ and $\overline{A}$ are Turing-recognizable.
The statement of Problem 4.31 (page 212) is slightly inaccurate. The first sentence should read

Say that a variable $A$ in a CFG $G$ is usable if it appears in some derivation of some string $w \in L(G)$.

(Sipser includes this correction on the Errata page on the Web.)

As Sipser notes on his Errata page, the algorithm for the Turing machine that recognizes $\overline{E_{TM}}$ in the solution to Exercise 4.5 needs a preliminary step, before the simulation of $M$ begins: “Check whether $M$ is a valid description of a Turing machine; if not, accept.”