The construction of \( \text{SELF} \) in Lemma 6.1 (page 246), though mind-bending, is actually simple enough that building corresponding examples in real-world programming languages — an expression, statement, or program that computes and prints out its own source text — is a popular meme/pastime among programmers. Here’s an example that works in Racket:

\[
((\lambda (p) (\text{write} (\text{list} p (\text{list} (\text{quote} \text{quote}) p))))
(\text{quote} (\lambda (p) (\text{write} (\text{list} p (\text{list} \text{quote} \text{quote}) p)))))
\]

You can find vast collections of such programs on the Web; unfortunately, most of them have undocumented preconditions or depend on implementation-dependent quirks of the compiler or interpreter.

My own favorite is from R\(^5\)RS Scheme. Under that (now nearly obsolete) standard, the empty string was a syntactically correct Scheme program! When you ran it, it (of course) did not do anything, which is behaviorally equivalent to printing out an empty string.

These programs are called quines, in honor of the logician Willard Van Orman Quine. In a 1961 lecture called “The Ways of Paradox,” Quine offered this example of a sentence that asserts its own falsity:

‘Yields a falsehood when appended to its own quotation’ yields a falsehood when appended to its own quotation

This is probably the direct inspiration for Sipser’s English prose example (page 248).

However, as Quine knew, his clever formulation was just an application of a much earlier result. The Recursion Theorem in computability theory (Theorem 6.3, pages 248–249) derives from a similar proposition in the theory of recursive functions that was first proven by Stephen Kleene in 1938.

In the proof of Theorem 6.3, Sipser’s instruction to “redesign \( q \) so that \( P_{\langle BT \rangle} \) writes its output following any string preexisting on the tape” seems a little too confident: How do we know that \( q \) can be “redesigned,” or rather redefined, in that way?

To justify this step, go back to the description of the Turing machine \( Q \) in the proof of Lemma 6.1, which computes the original function \( q \). \( Q \) starts by constructing a Turing machine \( P_w \) that depends on the input \( w \). In the description of \( P_w \), change the instruction “1. Erase input.” to read

1a. Move the tape head to the right, without changing the contents, until the tape head is on a blank square.

1b. Print a separator symbol, \#, and move one square rightwards.

After this revision, the instructions describe a different Turing machine \( Q' \) that computes a different function \( q' \), one that takes any string \( w \) as input and outputs \( w\#\langle P_w \rangle \), which is what Sipser’s construction requires.

The separator symbol is my idea. It enables \( B \) to split its input easily and unambiguously into \( w \) and \( \langle BT \rangle \). \( B \) can erase the separator either before or after applying
If the point of Sipser’s proof of Theorem 6.5 still seems obscure, I can spell it out a little more: $H$ cannot be a decider for $A_{TM}$ because $H$ gives the wrong answer when given the input $\langle B, w \rangle$ — if $B$ actually accepts $w$, then $H$ must have determined (incorrectly) that $B$ rejects $w$, and vice versa.