On pp. 181-2 of our textbook, *Discrete Mathematics etc*., 4th ed., by Kenneth Rosen, the treatment of the Halting Problem is not so clear as I would like. The following treatment may be easier to understand. Some strings of characters will be processed as potential programs much as compilers do it; when a string P is presented to a compiler it produces a compiled program that the computer can run, or else an error-message to the effect that string P is not valid source-program text. Other strings will be treated as inputs to programs; when a running compiled program reads such a string I it produces an output (or error message) after running for a finite time, or else runs forever (or until stopped by some external agency).

Can a program H(P, I) be written to accept any two strings and then, in a finite time, tell whether the program compiled from string P ultimately halts (perhaps with an error-message) or runs forever after reading input string I ? Alan Turing showed why no such program H can exist. This conclusion saddens us because we often wish to know whether a program P(I) will ever halt; and we can't be satisfied by just running it until it halts or until we lose patience and stop it perhaps prematurely. What we desire is some kind of computation by way of proof that assures us in advance that P will not run forever after reading I in. But Turing showed why, unless P is designed with such a proof in mind, no routine way exists to decide whether P(I) halts.

Suppose, for the sake of argument, that such a program H(P, I) did exist; we shall try to infer a contradiction that will prove no such H exists. The text of H would look something like this:

```
string function  H(string P,  string I):
     begin
          Text of a program that reads string  P  and checks its validity as a program,
          then reads string  I  and computes whether  P(I)  would halt.
          If   ( P  is a valid program and  P(I)  would never halt ),
                    return( "Program text runs forever with Input text")
               else return( "Program is invalid or Input text lets it halt") ;
     end program  H .
```

From H a program K(P) could be constructed by copying the italicized text from H after the substitution of "*P*" for "*I*" and adding a few other changes. The text of K would look like this:

```
string function  K(string P):
     begin
          Text of a program that reads string  P  and checks its validity as a program,
          then reads string  P and computes whether  P(P)  would halt.
          If   ( P  is a valid program and  P(P)  would never halt ),
                    return( " Program  K(K)  halts.")
               else while true do { run forever } ;
     end program  K .
```

Let's run K(K) , submitting the text of K as input to K . What would happen? If H(K, K) would have put out "Program is invalid or Input text lets it halt" then K(K) would run forever; but if H(K, K) would have put out "Program text runs forever with Input text" then K(K) would return " Program K(K) halts." Either way, program H(K, K) would have put out the wrong message, so no such H(P, I) can predict correctly for *every* program P whether it halts.

Why is our textbook's treatment of the Halting Problem not so clear as I would like? Our textbook does not distinguish clearly between a program H(P, I) that can examine the source-text of P and another program h(P, I) that can invoke (call) P(…) but cannot examine its text. For example, h(P, I) could look like this:

```
string function  h(string P,  string I):
     begin
          programpointer := CompiledCode(P) ;
          reset StopWatch to zero and restart it ;
          If ( programpointer ≠ NULL ),
             { (*programpointer)(I) ;    /* … runs program P(I) */
               If ( StopWatch = Infinity ),
                   return( "Program text runs forever with Input text");};
          return( "Program is invalid or Input text lets it halt") ;
     end program  h .
```

Of course h(P, I) is a silly program because it fails to halt whenever P(I) fails to halt, but we have to wait forever to find out. A similar cloud darkens our textbook's version of program K(P) on p. 182. For all we know its source-text could look like this:

```
string function  k(string P):
     begin
          If ( H(P,P) = "Program text runs forever with Input text" ),
                    return( " Program  k(k)  halts.")
               else while true do { run forever } ;
     end program  k .
```

What would happen when k(k) ran with the text of k as input? If H(k, k) put out "Program is invalid or Input text lets it halt" then k(k) would never halt, but if H(k, k) put out "Program text runs forever with Input text" then k(k) would return " Program k(k) halts." Either way, if H(k, k) halted the message it put out would disagree with what k(k) actually did, thus undermining our confidence in the correctness of H . But this may be an unfair test of H .

When the text of k is read by H(k, k) it sees that k invokes H(k, k) but does not see the text of H . Unable to see that text, H cannot be expected to figure out what H(k, k) does and therefore cannot be expected to deduce correctly what k(k) does. It is unfair to condemn H for failing to do what we desired but wouldn't let it do.

Can this unfair test can be put right by granting H(P, I) a license to read its own source-text? How to do so is not clear. Trying to embed the source-text of H within H as a compile-time constant runs the risk of infinite regression like the picture, FIGURE 1 on our textbook's p. 203, that contains within itself a picture of itself. Putting into H's source-text a statement that reads the file containing H's source-text runs the risk of reading that file infinitely often when H is run. Whatever is done to avoid these infinitudes amounts to a constraint upon the way H(P, I) may be written. Then all that running k(k) can prove is that, burdened by such a constraint, H cannot do what we desire correctly. This is far feebler than Turing's insight. He saw why no conceivable program H(P, I) that always halts can reveal always correctly whether P(I) halts.