It is generally useful to be able to define functions in terms of themselves. For example, given computable functions \( k, l, \) and \( m, \) the fixed-point lemma tells us that there is a partial computable function \( f \) satisfying the following equation for every \( y:\)

\[
f(y) \simeq \begin{cases} k(y) & \text{if } l(y) = 0 \\ f(m(y)) & \text{otherwise.} \end{cases}
\]

Again, more specifically, \( f \) is obtained by letting

\[
g(x, y) \simeq \begin{cases} k(y) & \text{if } l(y) = 0 \\ \varphi_x(m(y)) & \text{otherwise} \end{cases}
\]

and then using the fixed-point lemma to find an index \( e \) such that \( \varphi_e(y) = g(e, y). \)

For a concrete example, the “greatest common divisor” function \( \gcd(u, v) \) can be defined by

\[
\gcd(u, v) \simeq \begin{cases} v & \text{if } 0 = 0 \\ \gcd(\operatorname{mod}(v, u), u) & \text{otherwise} \end{cases}
\]

where \( \operatorname{mod}(v, u) \) denotes the remainder of dividing \( v \) by \( u. \) An appeal to the fixed-point lemma shows that \( \gcd \text{ is partial computable.} \) (In fact, this can be put in the format above, letting \( y \) code the pair \( \langle u, v \rangle. \)) A subsequent induction on \( u \) then shows that, in fact, \( \gcd \text{ is total.} \)

Of course, one can cook up self-referential definitions that are much fancier than the examples just discussed. Most programming languages support definitions of functions in terms of themselves, one way or another. Note that this is a little bit less dramatic than being able to define a function in terms of an index for an algorithm computing the functions, which is what, in full generality, the fixed-point theorem lets you do.