Revisit to the "Y combinator," and visit to the "recursion theorem"

Break this down into:

1. how to get a fixed-point operator,

2. how to use it to get "recursion" (or other self-reference).

---

Consider any program transformation (program) $g$.

(Our "transformations" amount to "compilations" that are total.)

(Speaking in terms of *programs* (rather than the partial functions they compute)
seems to concretize and simplify things.)

In general, for $z$ a program, we do not expect $g(z) \equiv z$.

But, for *some* $z$, $g(z) \equiv z$ does hold!

Moreover, there is a fixed program transformation Y that *gives* such $z$.

From *what*? From $g$.

I.e., for $z = \mathrm{Y}(g)$, $g(z) \equiv z$; i.e., $g(\mathrm{Y}(g)) \equiv \mathrm{Y}(g)$.

I.e., $\mathrm{Y}(g)$ is a "semantic fixed point" for $g$.

This is the "fixed-point version" of the recursion theorem.

---

The clever recipe for such a transformation Y:

$$\mathrm{Y}\ g \ \equiv\ \lambda s(g(s\ s))\ \lambda s(g(s\ s)),$$

where, to save parentheses, juxtaposition indicates application.

This is *literally* the transformation we used for lambda calculus,
but we can easily implement the recipe for *any* common programming language.

---

How use this to get "recursion" (or other self-reference)?

For each *try* $z$ at self-reference,
the result is a fixed transformation of $z$ to *some* program $g(z)$.
(The image of "chasing one's tail" comes to mind.)

When the try is the semantic fixed point $\mathrm{Y}(g)$, the reference(s)
will indeed be to a program $\mathrm{Y}(g)$ that is *equivalent to* the result $g(\mathrm{Y}(g))$.
("*Catching* one's tail (sort of)" comes to mind.)

This is the "self-reference version" of the recursion theorem.