Combinators
and
Combinatorial Logic

Russell's Paradox was discovered by the mathematican Bertrand Russell at the end of the Nineteenth Century. It is based on $\mathcal{S}$, the set of all sets that are *not* elements of themselves. For example, we might let $\mathcal{Big}$ be the set of all sets that have at least 10 members. We can find lots of sets with at least 10 members, so $\mathcal{Big} \in \mathcal{Big}$ and $\mathcal{Big} \notin \mathcal{S}$. On the other hand, let $\mathcal{N}$ be the set of integers {0,1,2,3,...}. $\mathcal{N}$ is not itself an integer, so $\mathcal{N} \notin \mathcal{N}$ and $\mathcal{N} \in \mathcal{S}$. So some sets are in $\mathcal{S}$ and some are not. The paradox comes when we ask if $\mathcal{S}$ is an element of itself; we get a contradiction either way.

Some people dismissed Russell's paradox as a stupid word game, but to people interested in the foundations of Mathematics it was profoundly troubling. Our set S can't exist because its existence leads to a contradiction, yet it was defined in the same way as many other mathematical objects. How can we determine if anything in mathematics makes any sense? This was one of the great research questions of the early Twentieth Century. There were numerous attempts to move the foundations of mathematics away from Set Theory, which seemed especially problematic.

One of these attempts was called *Combinatory Logic*, or the logic of combinators, by Moses Schonfinkle. This work led to the development of the lambda-calculus, which in turn formed the basis for the Scheme language.

A combinatory term is any of the following:

- a variable, taken from an infinite list of possible variables
- a combinator
- $(E_1\ E_2)$, an application of the combinatory term E_1 to the combinatory term E_2. We take application to be left associative, so $(E_1\ E_2\ E_3)$ is shorthand for $((E_1\ E_2)\ E_3)$ and $(E_1\ E_2\ E_3\ E_4)$ is shorthand for $(((E_1\ E_2)\ E_3)\ E_4\ )$

A *combinator* is either an element of a finite list of primitive functions, or a new combinator C defined by an expression such as

$$(C\ x_1\ \ldots\ x_n) = E$$

where E is a combinatory term all of whose variables are in the set $\{x_1,x_2,\ldots,x_n\}$

Note that there is no lambda expression in combinatory logic -- there is no way to make a new function on the fly or to have a function return a new function. If we wanted to use Scheme to represent combinators, every combinator would be represented by a lambda-expression with no free variables. Combinatorial logic does have ways to write things that have the same effect as lambda expressions.

Here are some standard combinator definitions.

- The Identity Combinator (**I** x) = x
- The Constant Combinator (**K** x y) = x. Remember our convention that application associates from the left. So another way to read this definition is ((**K** x) y) = x. This says that (**K** x) is a function that, given any argument y, always returns x.
- The Generalized Application Combinator

  (S f g x) = (f x (g x))

Note that the combinatory expression (**S K K**) has the same effect as the combinator **I**: For any x ((**S K K**) x) = (**S K K** x) = ((**K** x) (K x)) = x, since ((**K** x) y) = x for any y. We say that (**S K K**) and **I** are *functionally equivalent*.

The *Composition Combinator* B is defined by (B f g x) = (f (g x)).
The following calculation shows that we can write B in terms of just S and K:

(S(K S) K f g x) = ((K S) f (K f) g x) by the definition of S
= (S (K f) g x) by the definition of K
= ((K f) x (g x)) by the definition of S
= (f (g x)) by the definition of K
= (B f g x)

So B = S (K S) K

The *Diagonalizing Combinator* W is defined by (W f x) = (f x x). We can write W in terms of S and K:

(S S (S K) f x) = ((S f) (S K f) x) by the definition of S
= (S f (S K f) x) by left associativity
= (f x (S K f x)) by the definition of S
= (f x ((K x) (f x)) by the definition of S
= (f x x) by the definition of K
= (W f x)

So W = S S (S K)

It is a remarkable fact that everything in the lambda-calculus can be expressed in terms of the two combinators S and K. Since the lambda calculus is Turing Complete (it can simulate a Turing Machine), this means that all programs can be written as combinations of S and K.

Rather than dwelling on the details of how combinatory logic works, we will focus on just one specific combinator called "Y" that we can use to explain recursion. And rather than expressing Y in terms of S and K, we are going to write it in Scheme, which is much easier to understand