A Quick Review

The combiners $J$, $S$ and $K$

To begin with, we single out three combinator{s}, from which we will generate all others.

Initially, by a combination we understand either

- a $J$, an $S$ or a $K$, forming the basic constants, or
- a letter set aside to be a variable, or
- a compound expression of the form $A[B]$, where

$A$ and $B$ are previously obtained combinations,

A combination without variables is also called a combinator. Intuitively, a combinator is some kind of function $F$ which
when applied to arguments, as in $F[x_1][x_2][x_3]\ldots[x_n]$, affects a transformation. To give some kind of exact "meaning" to the combinator we use replacement rules.

```mathematica
crules = {J[x_] -> x, S[x_][y_][z_] -> x[z][y[z]], K[x_][y_] -> x};
```

We have to do some examples, however, to see what these rules accomplish in giving meaning to all combinations.

Note: As an aid to memory, we might nickname the basic combinators as follows:

- $J$ is the Joker;
- $S$ is the Slider; and
- $K$ is the Killer.

Warning: The combinator $J$ is usually written as $I$.

But Mathematica has a special role for $I$ which does not concern the current discussion.

### Functional abstraction

Given a list of variables and a combination, we create a combinator by removing variables one at a time, starting with the right-most variable.

```mathematica
ToC[vars_, comb_] := Fold[rm, comb, Reverse[vars]];

rm[v_, v_] := J;
rm[f_[v_], v_] /; FreeQ[f, v] := f;
rm[h_, v_] /; FreeQ[h, v] := K[h];
rm[f_[g_], v_] := S[rm[f, v]][rm[g, v]];
```

Warning: In Mathematica, FreeQ means "to be free of". Do not confuse this with "free and bound variables".

```
?FreeQ
```

FreeQ[expr, form] yields True if no subexpression in expr matches form, and yields False otherwise. FreeQ[expr, form, levelspec] tests only those parts of expr on levels specified by levelspec.

Note: In traditional notation $\text{ToC}[\{x, y, z\}, A]$ is written as $\lambda x \lambda y \lambda z. A$.

Some examples.

```
ToC[{x}, A[B[x]][C[x]]]
S[S[K[A]][B]][C]
ToC[{x}, A[B[C[x]]][D[x]]]
S[S[K[A]][S[K[B]][C]]][J]
ToC[{x, y}, A[B[x][y]][C[x][y]]]
S[S[K[S]][S[K[S[K[A]]]][B]]][C]
```
Self-application and fixed points

\begin{align*}
&\text{comb} = \text{ToC}[[x], F[x[x]]] \\
&\text{test} = \text{comb}[\text{comb}] \\
&\text{Do}[\text{test} = \text{test} /. \text{crules}, (12)];
\end{align*}

This calculation shows that

Every function has a fixed point!

This means that given \( F \), we can find a \( P \) such that \( P \Rightarrow F[P] \) by the crules.

And, moreover, we see that

The reduction of a combinator need not stop!

The problem here is trying to know when reductions \textit{will} stop.

This also shows that the notion of function embodied in combinators is \textit{not} the same as is familiar in mathematical usage.

Here is the general \textit{fixed-point combinator}:

\begin{align*}
&\text{Y} = \text{ToC}[[f], \text{ToC}[[x], f[x[x]]]]\text{ToC}[[x], f[x[x]]]] \\
&\text{S}[[S[S[K[S]]][K]], [[K[S][J][J]]][S[S[K[S]][K]]][K[S][J][J]]] \\
&\text{S}[[S[S[K[S]][K]]][K[S][J][J]]][S[S[K[S]][K]][K[S][J][J]]]
\end{align*}
Doing Arithmetic

The Church numerals

- **Some definitions**

Surprisingly enough, one can do *integer arithmetic* with combinators.

Here are the basic definitions proposed by Alonzo Church.

```plaintext
zero = K[J];
succ = S[S[K[S]]][K];
plus = S[K[S]][S[K[S[K[S]]]][S[K[K]]]]; 
times = S[K[S]][K];
power = S[K[S][S[J]]][K];
```

OK. Very tidy. *But what do they really mean?*

- **Zero and its successors**

Let's start at the beginning.

```plaintext
num = zero
K[J]
```

```plaintext
test = num[f][x]
K[J][f][x]
```

That looks familiar. And, after reduction:

```plaintext
test = test //. crules
x
```

So! The meaning of `zero[f][x]` is to *cancel* the `f`.

What about *successors*?
num = succ[num]
S[S[K[S]][K]][K]

test = num[f][x]
S[S[K[S]][K]][K][J][f][x]

test = test // . crules
f[x]

Let’s try a longer one.

num = succ[succ[succ[succ[succ[zero]]]]];
test = num[f][x];
test = test // . crules
f[f[f[f[f[f[x]]]]]]

It looks like the meaning of the 6th successor of zero iterates the f six times.

How can we prove a general theorem? This calculation might help.

succ[n][f][x] // . crules
f[n[f][x]]

So, we know that zero[f] iterates f no times;
if, n[f] iterates f n-times, then succ[n][f] iterates f (n + 1)-times;
hence, the nth Church numeral iterates a function n-times.

Numerating the numerals

Mathematica supports integer arithmetic.

So, let us try a recursive definition from the Mathematica integers to the Church numerals.

cnum[0] := zero;
cnum[n_] := succ[cnum[n - 1]];

cnum[11]
Looks good!

- **Doing addition**

  First, a small test.

  ```
  cnum[4] //. crules
  S[S[K][S][K]][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]][K][J]]
  
  test = plus[cnum[2]][cnum[2]] //. crules
  S[S[K][S]][S[K][K]][S[S[K][S]][K]][S[S[K][S]][K]][K][J])][S[S[K][S]][K]][S[S[K][S]][K]][K][J]])]
  ```

  Ouch! *The answers are not the same!* We need some tests.

  The general situation will be discussed later.

  ```
  test[f][x] //. crules
  f[f[f[f[x]]]]
  
  plus[n][m][f][x] //. crules
  n[f][m[f][x]]
  ```

  Ah, that is beginning to make sense: first iterate \(f\) for \(m\) times, then pile on \(f\) iterated \(n\) times.

  We can see now that, for Church numerals, we are alway going the have the *same results* in reducing

  \[
  \text{plus}[\text{cnum}[n]][\text{cnum}[m]][f][x] \quad \text{and} \quad \text{cnum}[n+m][f][x],
  \]

  if \(n\) and \(m\) are (standard) integers.

  So, \textbf{plus} indeed works like \textit{addition} on Church numerals.

  Here is a test:

  ```
  plus[cnum[2]][cnum[3]][f][x] //. crules
  cnum[2 + 3][f][x] //. crules
  f[f[f[f[f[x]]]]]
  f[f[f[f[f[x]]]]]
  ```

- **Doing multiplication and exponentiation**

  We try out at once the general pattern.

  ```
  times[n][m][f][x] //. crules
  n[m[f]][x]
  ```

  The part \(m[f]\) replicates \(f\) for \(m\) times; then the \(n[m[f]]\) replicates that \(n\) times.

  Altogether, then, we get an iteration \(n \cdot m\) times.

  Now, try \textit{power}.  

power[n][m] //. crules
m[n]

This is somewhat abstract, as the numbers are operating on numbers.

Here the n-fold iterator is itself iterated m times.

That produces an iterator of size n^m.

Some call this higher-order programming.

Here are some tests.

power[cnum[2]][cnum[3]][f][x] //. crules
cnum[2^3][f][x] //. crules
\[ f[f[f[f[f[f[f[f[x]]]]]]]]
\]
\[ f[f[f[f[f[f[f[x]]]]]]]]

power[cnum[3]][cnum[2]][f][x] //. crules
cnum[3^2][f][x] //. crules
\[ f[f[f[f[f[f[f[x]]]]]]]]
\[ f[f[f[f[f[f[f[x]]]]]]]]

A problem

Challenge: Find the combinator for pred.

Conjecture: There is no very short one.

Higher-order iteration

Creating structure

First we need to simulate pairs of objects by combinators, so we can then compute two values at the same time.

pair = ToC[{x, y, z}, z[x][y]]
S[S[K[S]][S[K][S]][S[K][S]][S[K][S][J][K][K]]][K][K]

left = ToC[{x, y}, x]
right = ToC[{x, y}, y]
K
K[J]
These new names may seem redundant.
But it does not hurt to have extra names to remind you what the combinators are meant to do.
Later, we may want to call them true and false!

Here is a test:

```plaintext
pair[a][b] //. crules
pair[a][b][left] //. crules
pair[a][b][right] //. crules
a
b
```

### Defining predecessors

The idea now is to start with a pair \(\langle 0, 0 \rangle\).

Then use a shift operation \(\langle p, q \rangle \rightarrow \langle p + 1, p \rangle\).

Then, iterating the shift \(n\)-times on the start pair leaves us with \(\langle n, n - 1 \rangle\).

```plaintext
shift = (ToC[[p], pair[succ[p][left]][p][left]]) //. crules
pred = (ToC[[n], n[shift][pair[zero][zero]][right]] //. crules)

S[S[K[S][K[S]][S][K[K]]][S[K[S]][S[K][S][J]][K]]][K][K][K][K][K][K][K][K][S][K[S][K][S][K[S]][S][K][S][K][S][K][S][J][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K][K]}
Testing numerals

We can use the same idea employed for predecessor to define a combinator that tests a numerable for being zero.

```plaintext
shift1 = ToC[(p), pair[p[ right]]][ right]] // crules
zeroQ = ToC[(n), n[shift1][ pair[left][ right]][ left]] // crules
   K[K]][J]]
S[S[S][J][K[S[K[S[S[K[S]]][S[K[K]]][S[K[S]]][S[K[S>J]]][K]]]][K]][K]]][K]]]
   S[S][J][K[K]][J]]][K[K]][J]]][K[K]][J]]][K[K]][J]]]]
pair[a][b][zeroQ[zero]] // crules
pair[a][b][zeroQ[cnum[12]]] // crules
a
b
```

In other words, a combination `pair[a][b][zeroQ[n]]` means

*if the numeral n is zero, choose a, else choose b.*

Do you see now why I might want to use the names `true` and `false`?

Another problem

**Problem:** Find a combination `pair[a][b][equalQ[n][m]]` which means

*if the numeral n is equal to the numerable m, choose a, else choose b.*

Equality

The idea is to *subtract* each of two numbers from each other to see if both answers are `zero`.

```plaintext
equalQ = ToC[(n, m), pair[zeroQ[m[pred][n]][right][zeroQ[n[pred][m]]]][S[S[K[S]][S[S[K[S]]]
```

Here is the test. Note that even numerals of a moderate size may take a long time to give the answer.

```plaintext
Timing[pair[a][b][equalQ[cnum[3]][cnum[3]]] // crules]
{0.029853, a}
```
More general recursion

- The big problem

Can combinators be used to define all recursive functions more generally? And what will this mean about undecidability of questions involving combinators?

- Primitive recursive functions

Using a temporary notation for functions of several variables of integers in the ordinary sense, the primitive recursive functions are generated as follows:

There are given starting functions:

- \( \text{null}[i] = 0 \)
- \( \text{succ}[i] = i + 1 \)
- \( \text{proj}^i [x_1, x_2, x_3, \ldots, x_n] = x_i \) provided \( i \leq n \)

New functions can be obtained from old functions by composition:

- \( h[x_1, x_2, x_3, \ldots, x_n] = g[f_1[x_1, x_2, x_3, \ldots, x_n], f_2[x_1, x_2, x_3, \ldots, x_n], \ldots, f_n[x_1, x_2, x_3, \ldots, x_n]] \)

New functions can be obtained from old function by primitive recursion:

- \( h[0, x_1, x_2, x_3, \ldots, x_n] = f[x_1, x_2, x_3, \ldots, x_n] \)
- \( h[i + 1, x_1, x_2, x_3, \ldots, x_n] = g[i, h[i, x_1, x_2, x_3, \ldots, x_n], x_1, x_2, x_3, \ldots, x_n] \)

- Simulation by combinators

The starting functions are easy.

We just have to define \( \text{null} \) as \( K[\text{zero}] \).

We already have \( \text{succ} \).

The various \( \text{proj}^i \) are defined by \( \text{variable elimination} \).

Composition — even for many variables — is also defined by \( \text{variable elimination} \).

Finally, primitive recursion takes a little more thought.

Let’s try this special case, where \( F \) and \( G \) are given, and \( H \) is to be found:

- \( H[0][x] = F[x] \)
- \( H[\text{succ}[n]][x] = G[n][H[n][x]][x] \)
Clearly, it is sufficient to solve:

\[ H[n][x] = \text{pair}[F[x]][G[pred[n]]][H[pred[n]][x]][x][\text{zeroQ}[n]] \]

Thus, it is sufficient to solve:

\[ H = \text{ToC}([n, x], \text{pair}[F[x]][G[pred[n]]][H[pred[n]][x]][x][\text{zeroQ}[n]]) \]

So, make this definition:

\[
\text{rec} = \text{ToC}([h, n, x], \text{pair}[F[x]][G[pred[n]]][h[pred[n]][x]][x][\text{zeroQ}[n]])
\]

Hence, it is sufficient to solve:

\[ H = \text{rec}[H] \]

But, we know we can do this by the fixed-point combinator.

Therefore, all primitive recursive functions can be defined (= simulated) by combinators.

Here is a test.

\[
\text{Y}[\text{rec}][\text{cnum}[5]][x] // \text{. crules}
\]

Warning! Do not try to reduce \text{Y}[\text{rec}] by itself! (Why?)

### Addition and multiplication reconsidered

As we recall, Church's definitions were "structural" or "conceptual" — which made them easy to understand:
But, stop to think: addition is *iterated succession* and multiplication is just *iterated addition*. So consider these definitions:

```
let sum = ToC[{n, m, n[succ][m]}
  S[J][K[S[S[K][S]][K]]]
  S[K[S]][S[K[S]][S[K][K]]]

let prod = ToC[{n, m, n[sum][zero]}
  S[S[K][S]][S[S[K][S]][K]][K[S][S][S[K][S]][K]]]
  S[K][S][K]

let plus = sum[n][m][f][x] // . crules
let times[n][m][f][x] // . crules
let n[f][m[f][x]]

(n[7][4][f][x] // . crules

(sum[7][4][f][x]) // . crules) == n[11]

True
```

Okay. Let's try out multiplication.

```
let prod = prod[n][m][f][x] // . crules

(prod[7][2][f][x]) // . crules) == n[14]

True
```

But the two methods give different answers when not applied to arguments.

```
(prod[7][2] // . crules) == (times[7][2] // . crules)

S[S[K][S]][K][S[S[K][S]][K]]
  S[S[K][S]][K][S[S[K][S]][K]][S[S[K][S]][K]]
  S[S[K][S]][K][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]]

S[S[K][S]][K][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]]
  S[S[K][S]][K][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]]
  S[S[K][S]][K][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]]

S[S[K][S]][K][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]][S[S[K][S]][K]]
```

Ha! That is shorter than Church's! And it works well:

```
S[[K][K][K]]
```

Ah. This time Church wins hands down. But the new definition does work.

```
S[[K][K][K]]
```
Partial recursive functions

In his fundamental work on *Recursive Function Theory*, S.C. Kleene added to the schemes for defining the primitive recursive functions the *minimalization scheme*, which provides a version of *search*:

> Given a function $f[n]$, search for the *least integer* $n$ such that $f[n] = 0$.

Combining this with the other schemes gives the *partial recursive functions*.

**Warning!** In finding the *least integer* $n$ such that $f[n] = 0$,

be sure *all* the previous values $f[0], f[1], f[2], \ldots, f[n-1]$ are defined!

Moreover, Kleene showed that only *one search* is necessary: that is, it is sufficient to compute functions:

$$G[\text{least } y : F[x_1, x_2, x_3, \ldots, x_n, y] = 0]$$

where $F$ and $G$ are given *primitive recursive* functions (which are always well defined and not partial).

Kleene's *Normal Form Theorem* can perhaps best understood by showing that partial recursive functions are the same as those computed by *Turing Machine Programs*.

So, how can we program in combinators to search for

the *least integer* $n$ such that $f[n] = 0$?

**Doing the search**

It would be nice if we could at once *define* an operator $M[f]$ with the meaning that its value is

the *least* $y$ such that $f[y] = 0$.

But it is perhaps a little hard to see directly.

A slightly *easier* question (though at first it might seem *harder*) is to define $M[f][n]$ meaning

the *least* $y \geq n$ such that $f[y] = 0$.

This operator has a quick "*procrusive" definition.
\[ M[f][n] = \text{pair}[n][M[f][\text{succ}[n]]][\text{zeroO}[f[n]]] \]

First get this combinator:

\[ H = \text{ToC}[(m, f, n), \text{pair}[n][m[f][\text{succ}[n]]][\text{zeroO}[f[n]]]] \]

We then have the desired operator when we find an \( M \) such that: \( M \Rightarrow H[M] \).

This works, because the \textit{desired answer} — given \( F \) — is \( M[F][0] \).