CS263–Spring 2008
Topic 1: The Lambda Calculus
Section 2.1: Combinatory Arithmetic

Dana S. Scott
Hillman University Professor (Emeritus)
School of Computer Science
Carnegie Mellon University
====
Visiting Professor EECS
Visiting Scientist
Logic & Methodology Program
University of California, Berkeley

Last edited 29 January 2008

A Quick Review

The combinators J, S and K

To begin with, we single out three ***combinators***, 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[x1][x2][x3]..[xn]`, affects a ***transformation***. To give some kind of exact ***"meaning"*** to the combinators we use ***replacement rules***.

```
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 combinitor **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.

```
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 **ToC[{x, y, z}, A]** is written as **λ x λ y λ 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]
```

```
S[S[K[S]][S[K[S[K[A]]]][B]]][C][x][y] //. crules
```

```
A[B[x][y]][C[x][y]]
```

```
ToC[{x, y, z}, A[B[y]][C[x][z]]]
```

```
S[K[S[S[K[S]][S[K[K]][S[K[A]][B]]]]]][S[K[K]][C]]
```

```
S[K[S[S[K[S]][S[K[K]][S[K[A]][B]]]]]][S[K[K]][C]][x][y][z] //.
 crules
```

```
A[B[y]][C[x][z]]
```

Self-application and fixed points

```
comb = ToC[{x}, F[x[x]]]
```

```
S[K[F]][S[J][J]]
```

```
test = comb[comb]
```

```
S[K[F]][S[J][J]][S[K[F]][S[J][J]]]
```

```
Do[test = test /. crules, {12}];
test
```

```
F[F[F[F[S[K[F]][S[J][J]][S[K[F]][S[J][J]]]]]]]
```

This calculation shows that

Every function has a fixed point!

This means that given **F**, we can find a **P** such that **P ⇒ 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 ***will*** stop.

This also shows that the notion of *function*

embodied in combinators is *not* the

same as is familiar in mathematical usage.

Here is the general ***fixed-point combinator*** :

```
Y = ToC[{f}, ToC[{x}, f[x[x]]][ToC[{x}, f[x[x]]]]]
```

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

```
test = Y[F]
S[S[S[K[S]][K]][K[S[J][J]]]][S[S[K[S]][K]][K[S[J][J]]]][F]
```

```
test = test /. crules
F[S[K[F]][S[J][J]][S[K[F]][S[J][J]]]]
```

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.

```
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[J]]][K];
```

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

Zero and its successors

Let's start at the beginning.

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

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

That looks familiar. And, after reduction:

```
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[J]]
```

```
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[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]
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]][K[J]]]]]]]]]]]
```

```
cnum[11][f] //. crules
S[K[f]][S[K[f]][S[K[f]][S[K[f]][
  S[K[f]][S[K[f]][S[K[f]][S[K[f]][S[K[f]][S[K[f]][S[K[f]][J]]]]]]]]]]]
```

```
cnum[11][f][x] //. crules
f[f[f[f[f[f[f[f[f[f[f[x]]]]]]]]]]]
```

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]][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

`plus[cnum[n]][cnum[m]][f][x]` and **`cnum[n + m][f][x]`**,

if **`n`** and **`m`** are (standard) integers.

So, **`plus`** indeed works like ***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 · m`** times.

Now, try **`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[f[x]]]]]]]]
```

```
power[cnum[3]][cnum[2]][f][x] //. crules
cnum[3^2][f][x] //. crules
f[f[f[f[f[f[f[f[f[x]]]]]]]]]
f[f[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[K]][S[K[S]][S[K[S[J]]][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:

```
pair[a][b] //. crules
S[S[J][K[a]]][K[b]]

pair[a][b][left] //. crules

pair[a][b][right] //. crules
a

b
```

Defining predecessors

The idea now is ***to start*** with a pair **⟨0, 0⟩**.

Then use a ***shift operation*** **⟨p, q⟩ → ⟨p + 1, p⟩**.

Then, ***iterating*** the shift **n**-times on the start pair leaves us with **⟨n, n – 1⟩**.

```
shift = (ToC[{p}, pair[succ[p[left]]][p[left]]] //. crules)
pred = (ToC[{n}, n[shift][pair[zero][zero]][right]] //. crules)
S[S[K[S[S[K[S]][S[K[K]][S[K[S]][S[K[S[J]]][K]]]]][K[K]]]][
   S[K[S[S[K[S]][K]]]][S[J][K[K]]]]][S[J][K[K]]]

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

Here is the test.

```
pred[cnum[10]][f][x] //. crules
cnum[9][f][x] //. crules
f[f[f[f[f[f[f[f[f[x]]]]]]]]]

f[f[f[f[f[f[f[f[f[x]]]]]]]]]
```

Here is a check of ***equality of numerals***.

```
(pred[cnum[10]] //. crules) == (cnum[9] //. crules)
True
```

Testing numerals

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

```
shift1 = ToC[{p}, pair[p[right]][right]] //. crules
zeroQ = ToC[{n}, n[shift1][pair[left][right]][left]] //. crules
```

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

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

```
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`**?

Aother 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`**.

```
equalQ = ToC[{n, m}, pair[zeroQ[m[pred][n]]][right][zeroQ[n[pred][m]]]]
```

```
S[S[K[S]][S[S[K[S]][
      S[K[S[K[S[S[K[S]][S[K[K]][S[K[S]][S[K[S[J]]][K]]]]]][K[K]]]]]][
       S[K[S[K[S[S[S[J][K[S[S[K[S[S[K[S]][S[K[K]][S[K[S]][
                                S[K[S[J]]][K]]]]]][K[K]]]][S[J][K[K[J]]]]]][
                     K[K[J]]]]]][K[S[S[J][K[K]]][K[K[J]]]]]][K[K]]]]]][
        S[K[S[S[J][K[pred]]]]][K]]]]][K[K[K[J]]]]]][
 S[K[S[K[S[S[S[J][K[S[S[K[S[S[K[S]][S[K[K]][S[K[S]][S[K[S[J]]][K]]]]]][
                    K[K]]]][S[J][K[K[J]]]]]][K[K[J]]]]]][
         K[S[S[J][K[K]]][K[K[J]]]]]][K[K]]]]]][S[J][K[pred]]]]
```

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

```
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 gererated as follows:

There are given ***starting functions:***

$\texttt{null[i]} == 0$
$\texttt{succ[i]} == i + 1$
$\texttt{proj}^n_i[x_1, x_2, x_3, .., x_n] == x_i \quad \texttt{provided } i \le n$

New functions can be obtained from old functions by ***composition:***

$h[x_1, x_2, x_3, .., x_n] ==$
$\quad g[f_1[x_1, x_2, x_3, .., x_n], f_2[x_1, x_2, x_3, .., x_n], .., f_m[x_1, x_2, x_3, .., x_n]]$

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

$h[0, x_1, x_2, x_3, .., x_n] == f[x_1, x_2, x_3, .., x_n]$
$h[i + 1, x_1, x_2, x_3, .., x_n] ==$
$\quad g[i, h[i, x_1, x_2, x_3, .., x_n], x_1, x_2, x_3, .., x_n]$

Simulation by combinators

The starting functions are ***easy***.

We just have to define **null** as **K[zero]**.

We already have **succ**.

The various $\texttt{proj}^n_i$ are defined by ***variable elimation***.

Composition — even for many variables — is also defined by ***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[succ[n]][x] == G[n][H[n][x]][x]
```

Clearly, it is sufficient to solve:

```
H[n][x] == pair[F[x]][G[pred[n]][H[pred[n]][x]][x]][zeroQ[n]]
```

Thus, it is sufficient to solve:

```
H == ToC[{n, x}, pair[F[x]][G[pred[n]][H[pred[n]][x]][x]][zeroQ[n]]]
```

So, make this definition:

```
rec = ToC[{h, n, x}, pair[F[x]][G[pred[n]][h[pred[n]][x]][x]][zeroQ[n]]]
```

```
S[S[K[S]][S[K[S[K[S]]]]][
    S[K[S[K[S[S[K[S[S[K[S]][S[K[K]][S[K[S]][S[K[S[J]]][K]]]]][K[K]]]][
           F]]]]]][S[S[K[S]][S[K[S[K[S]]]]][
          S[K[S[S[K[S]][S[K[K]][S[K[G]][S[S[S[J][K[S[S[K[S[S[K[S]][
                                   S[K[K]][S[K[S]][S[K[S[J]]][K]]]]][K[K]]]][
                              S[K[S[S[K[S]][K]]]][S[J][K[K]]]]][S[J][K[K]]]]]][
                        K[S[S[J][K[K[J]]]][K[K[J]]]]]][K[K[J]]]]]]]]][
           S[S[K[S]][K]][K[S[S[S[J][K[S[S[K[S[S[K[S]][S[K[K]][
                                 S[K[S]][S[K[S[J]]][K]]]]][K[K]]]][
                           S[K[S[S[K[S]][K]]]][S[J][K[K]]]]][S[J][K[K]]]]]][K[
                      S[S[J][K[K[J]]]][K[K[J]]]]]][K[K[J]]]]]]]]][K[K[J]]]]]]][
 K[S[K[K]][S[S[S[J][K[S[S[K[S[S[K[S]][S[K[K]][S[K[S]][S[K[S[J]]][K]]]]][
                     K[K]]]][S[J][K[K[J]]]]]][K[K[J]]]]]]][
      K[S[S[J][K[K]]][K[K[J]]]]]][K[K]]]]]
```

Hence, it is sufficient to solve:

```
H == 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.

```
Y[rec][cnum[5]][x] //. crules
```

```
G[S[S[K[S]][K]][S[S[K[S]][K]][S[S[K[S]][K]][S[S[K[S]][K]][K[J]]]]]][
  G[S[S[K[S]][K]][S[S[K[S]][K]][S[S[K[S]][K]][K[J]]]]][
    G[S[S[K[S]][K]][S[S[K[S]][K]][K[J]]]][
      G[S[S[K[S]][K]][K[J]]][G[K[J]][F[x]][x]][x]][x]][x]][x]
```

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

■ Addition and multiplication reconsidered

As we recall, Church's definitions were ***"structural"*** or ***"conceptual"*** — which made them easy to understand:

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

```
n[f][m[f][x]]
```

```
n[m[f]][x]
```

But, stop to think: addition is ***iterated succesion*** and multiplication is just ***iterated addition.*** So consider these definitions:

```
sum = ToC[{n, m}, n[succ][m]]
plus
```

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

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

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

```
sum[cnum[7]][cnum[4]][f][x] //. crules
```

```
f[f[f[f[f[f[f[f[f[f[f[x]]]]]]]]]]]
```

```
(sum[cnum[7]][cnum[4]] //. crules) == cnum[11]
```

```
True
```

OK. Let's try out multiplication.

```
prod = ToC[{n, m}, n[sum[m]][zero]]
times
```

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

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

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

```
prod[cnum[7]][cnum[2]][f][x] //. crules
```

```
f[f[f[f[f[f[f[f[f[f[f[f[f[f[x]]]]]]]]]]]]]]
```

```
(prod[cnum[7]][cnum[2]] //. crules) == cnum[14]
```

```
True
```

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

```
(prod[cnum[7]][cnum[2]] //. crules) == (times[cnum[7]][cnum[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]][K[J]]]]]]]]]]]]]]] ==
 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]][K[J]]]]]]]]]]][
  S[S[K[S]][K]][S[S[K[S]][K]][K[J]]]]
```

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

```
S[K[f]][
  S[K[f]][S[K[f]][S[K[f]][S[K[f]][S[K[f]][S[K[f]][S[K[f]][S[K[f]][S[K[f]][
           S[K[f]][S[K[f]][S[K[f]][S[K[f]][J]]]]]]]]]]]]]] ==
 S[K[S[K[f]][S[K[f]][J]]]]][S[K[S[K[f]][S[K[f]][J]]]]][
   S[K[S[K[f]][S[K[f]][J]]]]][
    S[K[S[K[f]][S[K[f]][J]]]]][S[K[S[K[f]][S[K[f]][J]]]]][
      S[K[S[K[f]][S[K[f]][J]]]]][S[K[S[K[f]][S[K[f]][J]]]]][J]]]]]]]
```

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

```
True
```

■ 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 is 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], .., f[n - 1] *are defined* !

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

G[least y : F[x_1, x_2, x_3, .., 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] == zero.

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 ≥ n such that f[y] == zero.

This operator has a quick "***procursive"*** definition.

```
M[f][n] == pair[n][M[f][succ[n]]][zeroQ[f[n]]]
```

First get this combinator:

```
H = ToC[{m, f, n}, pair[n][m[f][succ[n]]][zeroQ[f[n]]]]
```

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

We then have the desired operator when we find an **M** such that: **M ⇒ H[M].**

This works, because the ***desired answer*** — given **F** — is **M[F][0]**.