Contents:

Introduction

Numbers are often taken to be be sets or constructions out of sets. Thus the number five can be taken to be the set of all those sets that have 5 members: the set of fingers on my left hand, the set of fingers on my right hand, the set of working days in the week, and so on. In this scheme the number zero is the set of all those sets that have no members, it is the set that has just the null-set as its sole member. On another scheme the number zero is the null-set, and any other natural number is the set of its predecessors. On both schemes natural numbers are possible answers to the question "how many" or more precisely, "how many things".

But natural numbers can equally well be taken to be answers to the question "how many times" or "how often". This is how they are treated in Church's lambda calculus. They are functions which take a unary function as argument and return a unary function as value. If the argument function is the squaring function, and the number in question is three, then the returned function computes the square of the square of the square. So numbers are second order functions. But since they are unary functions, they can be applied to themselves at least in the untyped lambda calculus. So numbers are also third order functions, and so forth - they are simply higher order functions.

Numbers must be distinguished from numerals or more generally, numeric expressions. The numerals "7", "seven", "sept", "VII" and also the expressions "3+4" and "10-3" are all different, and yet they denote the same natural number. The so-called Church numerals of the lambda calculus are expressions which denote higher order functions of the kind described above. The remainder of this note shows how Church numerals can also be expressed in Joy.

Church numerals

In Joy a Church numeral will have to be a program which expects on the stack a quoted program, say [P], and below that whatever parameters are needed to execute the quoted program. The numeral will indicate how often [P] is to be executed: not at all, once, twice, thrice, or whatever.

First, the Church numeral C0, which is to execute the quoted program [P] zero times. Clearly this is achieved by just popping off the quoted program.

The other Church numerals will have to be constructed from C0 together with Csucc, a successor function for Church numerals. This function Csucc will take as parameters a quoted Church numeral, say Cn, and below that whatever quotation is to be executed repeatedly. So a program of the form ... [P] [Cn] Csucc should execute [P] exactly (n+1) times. This can be done by executing [P] once, and then again n times. So Csucc should first execute a copy of [P] and then allow Cn to execute [P] n times.

Here are the definitions of C0 and Csucc:

```
DEFINE
       C0  ==  pop;
       Csucc  ==  [dup [i] dip] dip i.
```

Church numerals do not denote numbers as data values but numbers as repetitions of execution of a program. So they need a program [P] to be repeated zero or more times. Preferably the output should bear some resemblance to what is being tested. A good choice is the ordinary successor function succ for ordinary numbers, to be called zero or more times with the ordinary number 0 on top of the stack.

```
DEFINE
       Peano  ==  0 [succ].
```

Here are the obvious tests:

```
Peano C0.
 0
   Peano [C0] Csucc.
 1
   Peano [[C0] Csucc] Csucc.
 2
   Peano [[[C0] Csucc] Csucc] Csucc.
 3
```

It is convenient to introduce definitions for the first few positive Church numerals. The tests follow.

```
DEFINE
       C1  ==  [C0] Csucc;
       C2  ==  [C1] Csucc;
       C3  ==  [C2] Csucc.


Peano C0.
 0
   Peano C1.
 1
   Peano C2.
 2
   Peano C3.
 3
```

Instead of using the natural numbers, with 0 and [succ], for tests of the Church numerals, we can also use many others data structures. Here are two examples:

```
DEFINE
       Doublings  ==  1 [2 *];
       Lists  ==  [] [[i] swoncat].


Doublings C0.
 1
   Doublings C1.
 2
   Doublings C2.
 4
   Doublings C3.
 8


Lists C0.
 []
   Lists C1.
 [i]
   Lists C2.
 [i i]
   Lists C3.
 [i i i]
```

Arithmetic operators for Church numerals

Church numerals in the lambda calculus or in Joy can be added, multiplied and exponentiated just like ordinary numerals. The three operators Cadd, Cmul and Cpow all take two quoted Church numerals [Cm] and [Cn] as parameters. Of course below those two quotations they expect the quoted function that is to be repeated, and below that the initial value.

To add two numbers m and n, we can take the n-th successor of m. This is also what the definition of Cadd below will do. It will transform the quotation [Cm] into another quotation by repeatedly constructing the successor quotation, where the number of repetitions is given by [Cn] which is called by the first i. The reculting quotion is then called by the final i.

To multiply two numbers m and n, we can start with 0 and add m exactly n times. To exponentiate, to compute m to the n-th power, we start with 1 and multiply by m exactly n times. This is also what the definitions of Cmul and Cpow do. They are more complex than the definition for Cadd for two reasons: the starting values C0 and C1 have to be specified, and also the operations Cadd and Cmul which they use are binary. The starting values are quoted, as [C0] and [C1] respectively, and they are squeezed below the [Cm] and [Cn]. At the same time [Cm] is transformed into either [[[Cm] Cadd] cons] or [[[Cm] Cmul] cons] respectively. This is then repeatedly executed, where the number of repetitions is given by [Cn] which is calle by the first i. The resulting quotation is then called by the final i.

Here follow the definitions and two forms of tests.

```
DEFINE
       Cadd  ==  [ [[Csucc] cons] ] dip i i;
       Cmul  ==  [ [[C0]] dip  [Cadd] cons [cons] cons ] dip i i;
       Cpow  ==  [ [[C1]] dip  [Cmul] cons [cons] cons ] dip i i.


Peano [C2] [C3] Cadd.
 5
   Peano [C2] [C3] Cmul.
 6
```

```
    Peano [C2] [C3] Cpow.
 8


Doublings [C2] [C3] Cadd.
 32
   Doublings [C2] [C3] Cmul.
 64
   Doublings [C2] [C3] Cpow.
 256
```

The operators can of course be used in complex expressions, as in these examples. The first just computes a value, and the second compares the same value with the identical value obtained from ordinary arithmetic.

```
Peano [[[[[C3] Csucc] [C2] Cadd] [C3] Cmul] Csucc] [C2] Cadd .
 21
   Peano [[[[[C3] Csucc] [C2] Cadd] [C3] Cmul] Csucc] [C2] Cadd
               3   succ    2   +      3   *     succ    2   +      = .
 true
```

As the second example shows, the value computed using Church arithmetic is the same as the one using ordinary arithmetic. However, the example also shows up how complex the expressions using Church arithmetic become when compared with ordinary expressions. This is because the Church operators need quotations as parameters, and hence the Church expression contains many square brackets whereas the ordinary expression requires none. So it seems that the quotation brackets are performing somewhat the same role as the ordinary grouping parentheses are performing in infix notation.

It is of some interest to see whether this is an inherent problem with expressing Church arithmetic in Joy, or whether there is a way of expressing Church arithmetic in Joy that retains the usual bracket free notation of Joy as for ordinary arithmetic.

Quotation numerals and Quotation operators

In Joy it is possible to define quotations and also quotation producing operators on quotations. Such quotations can then be executed by the i combinator or any other combinator. For an analogue of Church arithmetic the quotations will have to be the quoted Church numerals. These will be called quotation numerals. The quotation operators, corresponding to the Church operators, will take quotations off the stack and leave a new quotation on top of the stack.

The remainder of this section defines quotation numerals and quotation operators in the same sequence as in the two preceding sections.

First, the quotation numeral Q0 and the quotation operator Qsucc:

```
DEFINE
       Q0  ==  [pop];
       Qsucc  ==  [ [dup [i] dip] dip i ]  cons.


Peano Q0 i.
 0
   Peano Q0 Qsucc i.
 1
   Peano Q0 Qsucc Qsucc i.
 2


DEFINE
       Q1  ==  Q0 Qsucc;
       Q2  ==  Q1 Qsucc;
       Q3  ==  Q2 Qsucc.


Peano Q0 i.
 0
   Peano Q1 i.
 1
   Peano Q2 i.
 2
   Peano Q3 i.
 3
```

Since Church numerals denote (higher order) functions, they cannot be written - they do not have a finite representation. However, this is not true of the quotation numerals as just defined - these can be written since they are just quoted programs to compute such functions. Here are the first four quotation numerals:

```
Q0.
 [pop]
   Q1.
 [[pop] [dup [i] dip] dip i]
   Q2.
 [[[pop] [dup [i] dip] dip i] [dup [i] dip] dip i]
   Q3.
 [[[[pop] [dup [i] dip] dip i] [dup [i] dip] dip i] [dup [i] dip] dip i]
```

The quotation operators for addition, multiplication and exponentiation are just like the quotation operator Qsucc: they expect quotations as parameters and produce quotations as results. Their definitions below are a simple adaptation of the corresponding Church operators. A few tests also follow.

```
DEFINE
       Qadd  ==
         [ [[Qsucc i] cons] swap i i ]  cons cons;
       Qmul  ==
         [ [[Q0 i]] dip [Qadd i] cons [cons] cons] dip i i ] cons cons;
       Qpow  ==
         [ [[[Q1 i]] dip [Qmul i] cons [cons] cons] dip i i ] cons cons.



   Peano Q2 Q3 Qadd i .
 5
   Peano Q2 Q3 Qmul i .
 6
   Peano Q2 Q3 Qpow i .
 8
```

Again, because these three quotation operators take quotations as parameters and produce quotations, the result quotations can always be written without being called:

```
Q2 Q3 Qadd .
 [[[[pop] [dup [i] dip] dip i] [dup [i] dip] dip i] [[[[pop] [dup [i] dip] dip i] [dup [i] dip] dip i] [dup [i] dip] dip i] [[Qsucc i] cons] swap i i]
   Q2 Q3 Qmul .
 [[[[pop] [dup [i] dip] dip i] [dup [i] dip] dip i] [[[[pop] [dup [i] dip] dip i] [dup [i] dip] dip i] [dup [i] dip] dip i] [[[Q0 i]] dip [Qadd i] cons [cons] cons] dip i i]
   Q2 Q3 Qpow .
 [[[[pop] [dup [i] dip] dip i] [dup [i] dip] dip i] [[[[pop] [dup [i] dip] dip i] [dup [i] dip] dip i] [dup [i] dip] dip i] [[[Q1 i]] dip [Qmul i] cons [cons] cons] dip i i]
```

Finally, here is the complex expression from the end of the previous section now written in quotation notation. As can be seen, there are no quotation brackets visible, and as the second example shows, the whole structure of the expression is identical to the ordinary postfix expression for ordinary arithmetic.

```
Peano      Q3 Qsucc  Q2 Qadd  Q3 Qmul Qsucc Q2 Qadd i.
 21
   Peano      Q3 Qsucc  Q2 Qadd  Q3 Qmul Qsucc Q2 Qadd i
               3  succ   2  +     3  *    succ  2  +      = .
 true
```

Church truth values

Whereas Church numerals can be taken to be answers to the question "how often", Church truth values can be taken to be answers to the question "which one", when there is a choice of two. From an operational point of view, Church numerals represent looping (for-loops), and Church numerals represent branching. Whereas CHurch numerals take as a parameter one function, Church truth values take two parameters of which one will be ignored and the other evaluated. In Joy these two parameters will have to be quotations on the top of the stack. As for Church numerals, Church truth values do not denote data values, but choices between executing one of two quoted programs to be executed.

Here are the definitions of the two Church truth values, also two pairs of functions, and of course some tests.

```
    DEFINE
        Ctrue  ==  pop i;
        Cfalse  ==  popd i.
```

```
DEFINE
       Boole  ==  ["Yes, yes"] ["No, no"];
       Comparison  ==  2 3 [<] [>].


Boole Ctrue.
 "Yes, yes"
   Boole Cfalse.
 "No, no"
   Comparison Ctrue.
 true
   Comparison Cfalse.
 false
```

The negation operator for Church truth values will take as its parameter a quotation which, when executed, will expect two quotations on the stack. In analogy with the definition of Church negation in the lambda calculus, the Cnot operator for Church truthvalues in Joy might be defined as

```
DEFINE
       Cnot  ==  [ [Cfalse] [Ctrue] ] dip i.
```

However, because Joy is stack-based, a much simpler definition results if the Cnot operator simply swaps the two quotation parameters below the Church truth value, as follows.

```
DEFINE
       Cnot  ==  swapd i.


Boole [Ctrue] Cnot.
 "No, no"
   Boole [Cfalse] Cnot.
 "Yes, yes"
   Comparison [[Ctrue] Cnot] Cnot.
 true
   Comparison [[Cfalse] Cnot] Cnot.
 false
```

The definition of the two binary operators on Church truth values look quite different from the binary operators on Church numerals. This is because Church truth values take two quotation parameters, whereas Church numerals only take one. Apart from that, the definitions of the disjunction and conjunction operators on truth values are actually simpler than those for Church numerals.

Both operators expect two quotations denoting Church truth values. Both may need to be executed to determine the result of the binary operation. In the case of disjunction, Cor, if the evaluation of the top quotation yields Ctrue, then the other quotation will be discarded and the quotation below that will be executed. That will have to be the quotation [Ctrue], which will have to be placed there before Cor can execute the top quotation. On the other hand, if the evaluation of the top quotation yields Cfalse, then the added quotation [Ctrue] will be discarded and the second quotation will be executed whose result then fully determines the result yielded by Cor. Since the second quotation is only executed when its result is needed, this definition of Cor is what is sometimes called a "short circuit" implementation.

The definition of Cand is entirely analogous. The four tests for Cor and for Cand also follow.

```
DEFINE
       Cor  ==  [[Ctrue]] dipd i;
       Cand  ==  [[Cfalse]] dip i.


Boole [Ctrue ] [Ctrue ] Cor.
 "Yes, yes"
   Boole [Ctrue ] [Cfalse] Cor.
 "Yes, yes"
   Boole [Cfalse] [Ctrue ] Cor.
 "Yes, yes"
   Boole [Cfalse] [Cfalse] Cor.
 "No, no"

   Boole [Ctrue ] [Ctrue ] Cand.
 "Yes, yes"
   Boole [Ctrue ] [Cfalse] Cand.
 "No, no"
   Boole [Cfalse] [Ctrue ] Cand.
 "No, no"
   Boole [Cfalse] [Cfalse] Cand.
 "No, no"
```

Church predicates, Church conditionals and Church lists

The operators for Church logic are not so useful when they can only operate on Church truth value constants Ctrue and Cfalse. But they become more useful when there are predicates whose evaluation yields Church truth values.

Below is the definition of just one such predicate, the unary predicate Ceq0 which tests whether its parameter on the top of the stack is equal to zero, or better: whether it is C0. In the case of equality Ceq0 has to behave like Ctrue, otherwise it has to behave like Cfalse. To perform the test, it has to execute its parameter, which is a quotation [Cn] of a Church numeral or a quotation which eventually yields such a numeral. Below that parameter has to be a quotation which might not be executed at all if n=0, but might be executed many times. In the first case, when n=0, the Church numeral C0 has already been executed, and the redundant quotation has already been popped, but now Ctrue has to be executed. In the other cases, when n is positive, the Church numeral will have executed a a quotation several times, but now Cfalse has to be executed.

The definition below will do just that. First, below the quotation [Cn] it inserts two other quotations, [Ctrue] and [pop [Cfalse]]. The second of these may be executed zero or more times. The first time, if ever, it will replace the [Ctrue] by [Cfalse], and any further time it will replace the [Cfalse] it has placed there last time by a new [Cfalse]. After these two quotations have been inserted below the quoted Church numeral [Cn], the first i executes [Cn], resulting in either the original [Ctrue] or a [Cfalse] to become the top element on the stack. Finally, the second i executes that, as required.

After the definitions there are some obvious tests, and the last two also use the equality predicate in expressions using Cor and Cand.

```
DEFINE
       Ceq0  == [ [Ctrue] [pop [Cfalse]] ] dip i i.


Boole [C0] Ceq0 .
 "Yes, yes"
   Boole [C1] Ceq0 .
 "No, no"
   Boole [C2] Ceq0 .
 "No, no"
   Boole [[C0] Ceq0] [[[C2] Csucc] Ceq0] Cor.
 "Yes, yes"
   Boole [[C0] Ceq0] [[[C2] Csucc] Ceq0] Cand.
 "No, no"
```

The Church truth values expect two quotations on the top of the stack, and depending on which truth value it is, one of the quotations will be discarded and the other will be executed. So they implement a simple version of branching. A more useful one would expect a third quotation on the stack that is executed to determine which of the other two is to be executed. The third quotation, the if-part, could be above of below the other two, the then-part and the else-part. For efficient execution the if-part should be on top, to be executed by the combinator i. On the other hand, for analogy with the ordinary combinator ifte, it is best if the if-part is the third element on the stack. In the following definition of a Church combinator Cifte, the if-part is moved to the top by rolling down the other two parts. Following that, the if-part is executed by the i combinator.

Several examples also follow. The later examples, which use Church conditionals nested within Church conditionals, are written over several lines for ease of reading.

```
DEFINE
       Cifte  ==  rolldown i.


Peano  [[C0] Ceq0]  [C2]  [C3]  Cifte.
 2
   Peano  [[C1] Ceq0]  [C2]  [C3]  Cifte.
 3
   Peano  [[[C0] Ceq0] Cnot]  [C2]  [C3]  Cifte.
 3
   Peano  [[[C1] Ceq0] Cnot]  [C2]  [C3]  Cifte.
 2


Peano [Ctrue ]
              [[Ctrue ] [C0] [C1] Cifte]
              [[Ctrue ] [C2] [C3] Cifte]
              Cifte.
 0
   Peano [Ctrue ]
              [[Cfalse] [C0] [C1] Cifte]
              [[Cfalse] [C2] [C3] Cifte]
              Cifte.
```

```
1
  Peano [Cfalse]
              [[Ctrue ] [C0] [C1] Cifte]
              [[Ctrue ] [C2] [C3] Cifte]
              Cifte.
2
  Peano [Cfalse]
              [[Cfalse] [C0] [C1] Cifte]
              [[Cfalse] [C2] [C3] Cifte]
              Cifte.
3
```

Church truth values are actually very similar to the two operations which select one or the other from a pair of objects. The "dotted pair" constructor cons in Lisp and Scheme builds a pair of objects from its two parameters. The two operators car and cdr select the first or the second from such a pair. The last sentence just about captures the semantics of the constructor cons and the two operators. Any implementation must capture this semantic condition, but otherwise any further detail of the implementation is immaterial.

The following define a constructor Ccons and two operators Ccar and Ccdr. The latter use the Church truth values Ctrue and Cfalse to effect the selection from a pair constructed by Ccons. Ccons of course expects two parameters on the top of the stack, but since there is no requirement that the two be wrapped into one parcel, it does not do so. Instead it just pushes the sligtly curious quotation [i]. This will be executed by the call to the i combinator in Ccar and Ccdr, and that will result in either Ctrue or Cfalse being executed, depending on which one caused the execution of [i].

Note that Ccons is more general than the cons operator on Joy which requires its topmost parameter to be a list.

The last four examples are again written over several lines for ease of reading. They are the four list-counterparts of the four conditionals of the previous examples.

```
DEFINE
       Ccons  ==  [i];
       Ccar  ==  [Ctrue] swap i;
       Ccdr  ==  [Cfalse] swap i.


Peano  [C0] [C1] Ccons Ccar.
0
  Peano  [C0] [C1] Ccons Ccdr.
1


Peano [[C0] [C1] Ccons Ccar]
              [[C2] [C3] Ccons Ccar]
              Ccons Ccar.
0
  Peano [[C0] [C1] Ccons Ccdr]
              [[C2] [C3] Ccons Ccdr]
              Ccons Ccar.
1
  Peano [[C0] [C1] Ccons Ccar]
              [[C2] [C3] Ccons Ccar]
              Ccons Ccdr.
2
  Peano [[C0] [C1] Ccons Ccdr]
              [[C2] [C3] Ccons Ccdr]
              Ccons Ccdr.
3
```

Miscellaneous

As may be seen, complex expressions using any of the operators of the preceding two sections will require the same annoying bracketing as was encountered in the sections on Church arithmetic. However, in analogy with the quotation numerals of section 4 it would be straightforward to define quotation truth values and operators and also quotation predicates, conditionals and lists.

The definitions of Church concepts in Joy, like the corresponding concepts in the lambda calculus, are mainly of theoretical interest. So far none of the definitions have been found useful in practical programming in Joy. Accordingly none of the definitions have been included in any of the libraries. However, the Joy source for this note, without the text, is available on: jp-church.joy

Church numerals and related matters are treated in many of the more theoretical books dealing with functional languages based on the lambda calculus. There is also a vast literature available online, a web search for "Church numeral" will find many excellent expositions.

Brent Kerby's elegant and efficient alternative Church numerals

Shortly after I put the preceding sections on the web page for Joy, on 9-Oct-2002 Brent Kerby sent a note to the "concatenative" mailing list. The remainder of this section is the first part of his note.

```
About the construction of Csucc (which adds one to a church number),

   Csucc ==  [dup [i] dip] dip i.

  this can be simplified a bit:

   Csucc ==  [dup dip] dip i.

  because "dup [i] dip" is the same as "dup dip"; by the way, the "dup dip"
  combinator is an interesting one; I call it "run", since it in effect
  executes a program without destroying the program.

  ----------
  Anyhow, onto the next construction:

   Cadd ==  [ [[Csucc] cons] ] dip i i;

  Here's an alternative:

   Cadd == [sip] dip i

  It works like this:

     [p] [m] [n] [sip] dip i
  == [p] [m] sip [n] i
  == [p] m [p] [n] i
  == [p] m [p] n

  ------------
  Next,

   Cmul ==  [ [[C0]] dip  [Cadd] cons [cons] cons ] dip i i;

  An alternative:

   Cmul == [cons] dip i

  It works like this:

     [p] [m] [n] [cons] dip i
  == [p] [m] cons [n] i
  == [[p] m] n

  -----------------
  Finally,

   Cpow ==  [ [[C1]] dip  [Cmul] cons [cons] cons ] dip i i.

  And, an alternative:

   Cpow == [[cons] cons] dip i i

  Working like this:

     [p] [m] [n] [[cons] cons] dip i i
  == [p] [m] [cons] cons n i
  == [p] [[m] cons] n i
  == [[[[[p] m] m] m] ...] i
```

```
Unfortunately, in this system of church numbers, the Cpow is not a true
combinator, only a pseudo-combinator (unlike Csucc, Cadd, and Cmul, which
are true combinators).
```

Testing Brent Kerby's Church numerals

```
# The original definitions, commented away, are given for comparison.
   # Brent Kerby's new definitions over-ride the old ones.

   DEFINE

   #  Csucc ==  [dup [i] dip] dip i;
      Csucc ==  [dup dip] dip i;

   #  Cadd ==  [ [[Csucc] cons] ] dip i i;
      Cadd ==  [sip] dip i;               # but sip needs to be defined:
                                            sip ==  dupd dip;

   #  Cmul ==  [ [[C0]] dip  [Cadd] cons [cons] cons ] dip i i;
      Cmul ==  [cons] dip i;

   #  Cpow ==  [ [[C1]] dip  [Cmul] cons [cons] cons ] dip i i;
      Cpow ==  [[cons] cons] dip i i.


   # Now the old tests using the new definitions:

   Peano [C2] [C3] Cadd.
5
   Peano [C2] [C3] Cmul.
6
   Peano [C2] [C3] Cpow.
8

   Doublings [C2] [C3] Cadd.
32
   Doublings [C2] [C3] Cmul.
64
   Doublings [C2] [C3] Cpow.
256

   Peano [[[[C3] Csucc] [C2] Cadd] [C3] Cmul] Csucc] [C2] Cadd .
21
   Peano [[[[C3] Csucc] [C2] Cadd] [C3] Cmul] Csucc] [C2] Cadd
               3   succ    2   +      3   *     succ    2   +      = .
true
```

Another kind of Church numeral, also by Brent Kerby

This section is verbatim the second part of his note.

```
By the way, there are several other ways of doing church numbers in Joy. The
current way assumes,

   [p] Cn  ==  p p ... p

But alternatively we could do:

   [p] Rn  ==  p p ... p [p]

The only difference here is that we save the original program "p". The
principal advantage of this approach is that then Rn are reversible
combinators (note, Cn were all irreversible combinators since they destroy
the original program). We'd have,

   R0 ==
   R1 == run
   R2 == run run
   R3 == run run run
   ...

This is quite an elegant setup, seeing that simply "run" acts as a successor
 function, and C0 is simply the empty program. Also, the negative church
numbers then are

   R-1 == unrun
   R-2 == unrun unrun
   R-3 == unrun unrun unrun
   ...

Note the reverse of any church number then is simply its negative. Also, to
add church numbers is simply to concatenate them. Of course, "cat" itself is
irreversible, so we might use "rat":

   [b] [a] rat ==  [b a] [a]

This is a suitable addition function for the Rn church numbers; notice it
preserves one of the parameters, like a reversible addition function should.
Also, unlike Cadd, it does not execute the newly formed program (doing so
would be irreversible, as "i" is irreversible).

Here's how to multiply these Rn church numbers:

   Rmul == swap cons untack

It works, for example, like this:

    [2] [3] swap cons untack
== [[3] 2] untack
== [3 3 [3]] untack
== [3 3] [3]
== [6] [3]

So that's nice, except that this is not really reversible, since "cons" is
not reversible (assuming we have transparent quotation; if we have opaque
quotation then "cons" is reversible, but the multiplication then doesn't
work because "untack" fails):

   [a] [dup] cons    == [[a] dup]  ==  [[a] [a]]
   [[a] [a]] [] cons == [[a] [a]]

See how cons can give the same result from different inputs? This means it
is irreversible.

So can Rmul be defined from reversible primitives? If so, we would expect to
fail in the case of multiplying by zero, since that is irreversible. Also,
having a reversible Rmul would give us the ability to do division, and the
ability to construct fractional church numbers. But do fractional church
numbers make sense? For example, the church number for 1/2 would have the
rule:
```

```
  [a a] R1/2  == a [a a]

In a system with transparent quotation, it seems like R1/2 would not be
straightforward to implement, perhaps impossible, since it would require the
system to try to coerce an arbitrary "[p]" into the form "[a a]", by using
reductions and expansions inside the quotation. But maybe there is some
reasonable way to do it ...

- Brent
```

Yet another kind of Church numeral, again by Brent Kerby

Also on 9-Oct-2002 Brent wrote another note, which is reproduced her in full:

```
[ a propos my question: ]
  > Could you clarify why you say that your Cpow is not a true combinator?

  Well, it's pretty simple, really. When we try to write a "reduction rule"
  for Cpow, the best we can get is this:

    [p] [m] [n] Cpow  ==  [p] [[m] cons] n i

  Note the appearances of "cons" and "i" on the right hand side. This is in
  essence what makes it a pseudo-combinator. A true combinator has a reduction
  rule without such things on its right hand side, for example:

          [a] dup   == [a] [a]
      [b] [a] swap  == [a] [b]
      [p] [n] Csucc == p [p] n
  [p] [m] [n] Cadd  == [p] m [p] n
  [p] [m] [n] Cmul  == [[p] m] n

  See how in all of these the right hand side consists solely of variables,
  mixed up through concatenation and quotation. That is why they are called
  true combinators, because they possess a reduction rule of this kind.

  For the curious, there is some interest in a system that contains only true
  combinators, no pseudo-combinators. Such a system would have to get rid of
  quotation, since the quotation of any combinator is a pseudo-combinator
  (e.g., "[dup]" or "[swap]" is a pseudo-combinator, since it possesses no
  reduction rule). This idea hasn't been fully explored, but you might like to
  look back on the post
  http://groups.yahoo.com/group/concatenative/message/1051 and other posts
  thereabouts. The idea is that a workable base is {nil, zap, dup, i, unit,
  take, cat}:

           nil  == []
       [a] zap  ==
       [a] dup  == [a] [a]
       [a] i    == a
       [a] unit == [[a]]
   [b] [a] take == [a [b]]
   [b] [a] cat  == [b a]

  I suspect that all true combinators can be constructed form these, using
  concatenation alone (no quotations allowed), and there is very probably a
  smaller base that will work. Here's a few examples of some constructions in
  this system (a searcher reveals that these are the minimal constructions):

    dip  == take i
    swap == unit take i
    flip == unit take take i  (i.e., a combinator that reverses the top 3
  items)
    dupd == unit take i unit dup cat take i
         == unit take i dup unit take take i
    sip  == unit take i unit dup cat take i take i
         == unit take i dup unit take take i take i

  More simply, we could have

    dupd == swap dup flip
    sip  == dupd dip

  [ a propos another question from me: ]
  > I do remember that in one reference (I cannot remember which, alas)
  > it was pointed out that exponentiation could be implemented
  > as applying the exponent n to the base m, but that was
  > considered "cheating" [sic]. I did not understand that
  > either, but maybe you are talking about the same thing.

  Yes, in the classical combinatory logic you can exponentiate Church numbers
  simply by applying them (although it is the base that is applied to the
  exponent, not the other way around). We can do the same thing in Joy if we
  reform the Church numbers once more (I'll call them Zn, as that is what the
  Church numbers are standardly called):

    [x] [p] Zn  ==  [[[[x] p] p] ...] p

  For example,

    [x] [p] Z0  == x
    [x] [p] Z1  == [x] p
    [x] [p] Z2  == [[x] p] p

  Note these Church numbers take two parameters as opposed to the one
  parametered Cn and Rn. Anyhow, we can define the operators then in this way:

       [x] [p] [n] Zsucc ==  [[x] [p] n] p
   [x] [p] [n] [m] Zadd  ==  [[x] [p] n] [p] m
   [x] [p] [n] [m] Zmul  ==  [x] [[p] n] m
   [x] [p] [n] [m] Zpow  ==  [x] [p] [n] m

  Or by construction,

    Zsucc  == [B] S
    Zadd   == [B] N
    Zmul   == B
    Zpow   == i

  where B, S, and N are the classical combinators

      [c] [b] [a] B == [[c] b] a
      [c] [b] [a] S == [[c] b] [c] a
  [d] [c] [b] [a] N == [[d] c] [[d] b] a

  See how neat that works, with Zpow being simply "i".

  Anyhow, these Zn seem to be the true analogue of the classical Church
  numbers, although that's not to say that they are necessarily the most
  appropriate in a concatenative system. But, Zn do have use; for example,

    dip2 == [dip] Z2
```

```
  dip3 == [dip] Z3
  dip4 == [dip] Z4
  ...

See,

   [p] dip3
== [p] [dip] Z3
== [[[p] dip] dip] dip

That seems pretty handy.
```

GitHub