Exotic Number Systems

History of Integers

Number systems evolve in abstraction and expressibility.

Assyrian tokens `IIIIII + IIIIIIIIIII = IIIIIIIIIIIIIIIII`

very easy to add, hard to multiply, very hard to read

Roman numerals `VI + XI = XVII`

easy to add, hard to multiply, moderate to read

Arabic decimal numbers `6 + 11 = 17`

hard to add and to multiply, easy to read

Binary numbers `110 + 1011 = 10001`

easy to add and to multiply, moderate to read

Boundary numbers `((*)*) + (((*))*)* = ((*)*)(((*))*)*`

very easy to add and to multiply, hard to read

Types of numbers

existential	(*indicative*)	exists or not	
categorical	(*nominal*)	share some property	+ attribute
ranking	(*ordinal*)	put in order	+ less than
discrete	(*interval*)	relate to integers	+ equal steps
comparative	(*ratio*)	relate to fractions	+ parts and zero
continuous	(*real*)	relate to infinite	+ compactness
complex	(*imaginary*)	relate to model	+ i, other unit bases

The Numerical/Measurement Hierarchy

i	*Imaginary*	(real1, real2)	relation of reals
delta	*Real*	dense packing between rationals	relation of rationals
divide	*Rational*	integer1 / integer2	relation of integers
equal	*Interval*	ordinal1 - ordinal2 = constant	relation of ordinals
less-than	*Ordinal*	nominal1 less-than nominal2	relation of nominals
member-of	*Nominal*	indicative1 member-of indicative2	relation of indicatives
<void>	*Indicative*	( )	relation of existence

Integers as Sets

	1	2	3	4	5
Cardinality:	{}	{}{}	{}{}{}	{}{}{}{}	{}{}{}{}{}
Ordinality:	{}	{{}}	{{{}}}	{{{{}}}}	{{{{{}}}}}
Uniqueness:	{}	{{}}	{{},{{}}}	{{},{{}},{{},{{}}}}	

Some Varieties of Numbers

Conway numbers (surreals) provide a single coherent framework for defining all types of numbers, and provide ways to manipulate infinite forms.

Spencer-Brown arithmetic is a boundary representation in which each form is both a numerical object and an operator.

Kauffman arithmetic uses a boundary form of place notation to provide a more efficient computational representation while maintaining operations which are both parallel and insensitive to the magnitude of the number.

The James Calculus uses three boundaries to shift the representation of numbers between exponential and logarithmic forms. This mechanism generalizes the ideas of cardinality and inverse operations, and constructs a new imaginary which imparts phase structure on numbers.

Conway Numbers (Surreal Numbers)

A ***number*** is a partitioned set of prior numbers, `{L|G}`,
such that no member of `L` is greater than or equal to any member of `G`.

The set `L` contains *lesser* numbers, while the set `G` contains *greater* numbers.

Let `xL` be an arbitrary member of `L`, and `xG` be an arbitrary member of `G`.

```
x = {xL|xG}        such that no xL >= any xG

                             i.e. every xL < every xG
```

Two Conway numbers are ***ordered***

```
x >= y       when        no xG <= y and no yL >= x

                         i.e. every xG > y and every yL < x
```

Two Conway numbers are ***strictly ordered***

```
x > y        when        x >= y and not y >= x

                   i.e.  all xG > y, all yL < x, some xL < y, some yG > x
```

Two Conway numbers are ***equal***

```
x = y        when        x >= y and y >= x

                   i.e.  all xG > y, all xL < y, all yL < x, all yG > x
```

Integers from Ordinals

"Before we have any numbers, we have a certain set of numbers, namely *the empty set*, `{}`."
-- John H. Conway

Base: `{ | }` empty partitions of the empty set

Generator: every partition of the set of prior numbers

The base of this system is the *act of partitioning*, not the empty set. Partitioning creates the first distinction, which serves as sufficient structure to build all numerical computation.

The conventional names of numbers can be assigned to Conway numbers:

Is `{ | }` a number? `{ | } = 0`

`every xL < every xG?` yes since there are no `xL`

Is `{ | } >= { | }` is `0 >= 0`?

`every xG > 0 and every yL < 0?` yes since there are no `xG` or `yL`

`By symmetry y >= x, thus 0 = 0`

Building from Zero

`0` is a Conway number, making the set of numbers currently known = `{0}`. This generates three new number partitions:

`{0| }` `{ |0}` `{0|0}`

`{0|0}` is not a number, since there is an `xL >= xG`, namely `xL = 0`

`{0| }` is a number, call it `1`

`{ |0}` is a number, call it `-1`

What is the ordering of these new numbers? For illustration, we'll test `0` against `-1`:

Ordered:

Is `{ | } >= { |0}`? i.e. is `0 >= -1`? `x=0, y=-1`

`every xG > -1 and every yL < 0?` yes since there are no `xG` or `yL`

Thus `0 >= -1.`

Strictly ordered:

Is `{ | } > { |0}`? i.e. `not-1 >= 0`?

`every -1G > 0 and every -1L < 0?` no, since `-1G = 0`

Thus `0 > -1`. Similarly (tests omitted) `1 > 0.`

Building from One

Now, the current set of prior numbers = `{-1,0,1}`, with a strict ordering, `1 > 0 > -1`.

Three prior numbers generate 8 (2^3, the powerset) possible sets to partition. The definition of a number constrains the forms generated from these sets to 21 new number forms:

`{-1|0} {-1|0,1} {-1|1} {0|1} {-1,0|1} { |R} {L| }`

where `R` and `L` stand for any of the eight sets in the powerset of prior numbers.

Conway numbers have multiple representations (just like another representation of 7 is 3+4). A closer analogy would be to have the number three written in different languages (three, trois, drei,...). For example:

```
0 = { | } = {-1| } = { |1} = {-1|1}
```

In general: *the smallest* `xG` *defines* `G`*, the largest* `xL` *defines* `L`*.*

This is easy to see since the tests for numbership and ordering compare small `xG`s and large `xL`s.

The new numbers are:

```
{1| } = 2        { |-1} = -2        {0|1} = 1/2        {-1|0} = -1/2
```

This gives a hint about how to think about Conway representations: the new number is "in between" the largest `xL` and the smallest `xG`. When one side of the partition is void, a new integer is formed.

Number Form Rules

A contribution of Conway numbers is that they incorporate all types of numbers in one consistent system. Given a number `{a,b,c,...|d,e,f,...}`, the interpretation of that form is the simplest conventional number which is strictly greater than `max[a,b,c,...]` and strictly less than `min[d,e,f,...]`. In general:

If there's any number that fits, then use the simplest number that fits.

`x` is an ***ordinal*** number when

```
x = {L| }

{n| } = n+1
```

`x` is a ***negative integer*** when

```
x = { |G}

{ |-n} = -(n+1)
```

`x` is a ***fraction*** when

```
{n|n+1} = n + 1/2

{0|1/2^(n-1)} = 1/2^n

{p/2^n | (p+1)/2^n} = (2p+1)/2^(n+1)
```

x is a ***real*** number when

```
x = {x - 1/n|x + 1/n}   for n > 0
```

Infinities and Infinitessimals

Conway numbers allow computation with a diversity of infinities and infinitessimals. ***Infinite*** numbers are generated when an infinity of ordinals is included in xL:

```
w = {0,1,2,...| }        w is infinite
```

Unlike conventional numbers, operations on varieties of infinite numbers are defined:

```
w + 1   = {0,1,2,...,w| }

w - 1   = {0,1,2,...|w}

w/2     = {0,1,2,...|w,w-1,w-2,...}

w^(1/2) = {0,1,2,...|w,w/2,w/4,w/8,...}
```

Conway Operators

For a representation to be useful, it must be accompanied with a set of transformation rules. Here is how the standard numerical operations are defined recursively for Conway numbers:

Addition

```
Base:              0 + 0 = { | }
Generator:         x + y = {xL+y, x+yL | xG+y, x+yG}
```

Example: `2 + (-1) = {1| } + { |0}`

```
xL+y = 1 + (-1)    = 0         this sum is computed recursively
x+yL = 2 + void    = void
xG+y = void + (-1) = void
x+yG = 2 + 0       = 2         this sum is computed recursively

x + y = {0|2} = 1
```

To show that `{0|2}` is a representation of `{0| } = 1`, show equality:

```
x={0|2} =?= y={0| }

every xG > y                 2>1  true
every xL < y                 0<1  true
every yL < x                 0<1  true
every yG > x                 none true
```

Negation

```
Base:           -0 = { | }
Generator:      -x = {-xG|-xL}
```

Multiplication

```
Base:           0*0 = { | }
Generator:      x*y = {xL*y+x*yL-xL*yL, xG*y+x*yG-xG*yG |
                       xL*y+x*yG-xL*yG, xG*y+x*yL-xG*yL}
```

Multiplication recurs on each partition of each variable.

Division

`y` is a number and `x*y = 1`

```
Base:           y = {0| }
Generator:      y = {0, (1 + (xG-x)*yL/xG, (1 + (xL-x)*yG/xL |
                        (1 + (xL-x)*yL/xL, (1 + (xG-x)*yG/xG}
```

Conway Star

The form `{0|0}` is not a number. However, it can be treated as an *imaginary* number, `*`, such that

```
* + * = 0          * =/= 0
```

Star is its own inverse. Generally,

```
n + * = {n|n}  for any n

n + * = {0+*,1+*,...(n-1)+*|0+*,1+*,...(n-1)+*}
```

Consider `{0|*}`, which is less than or equal to `{0|1}, {0|1/2},{0|1/4},...`

`{0|*}` is a positive number which is smaller than all other positive numbers, call it `d+`.

`{*|0}` is a negative number which is larger than all other negative numbers, call it `d-`.

```
{d+|d-} = {d+|0} = {0|d-} = {0|0} = *

d+ + * = {0,*|0}

d- + * = {0|0,*}

{0|d+} = d+ + d+ + *
```

Spencer-Brown Numbers

Spencer-Brown Arithmetic (Parenthesis Version)

In Spencer-Brown arithmetic, each number is both an object and an operator.

Integers: (Stroke arithmetic in a container)

```
0          (  )
1          (())
2          (()())
3          (()()())
```

Operations:

```
a+b        ((a)(b))
a*b          a  b
a^b        ((a) b )
```

Reduction Rules (implicit commutativity and associativity):

```
((a)) = a                    Involution

(()()) a = ((a)(a))          Distribution
```

Examples: (Brackets are for highlighting, they are identical to parentheses.)

```
2+3 = 5          (()()) + (()()()) =?= (()()()()())

      [[ (()()) ][ (()()()) ]]                  sum
      [   ()()      ()()()   ]                  involution
                 5                              interpret

2*3 = 6          (()()) * (()()()) =?= (()()()()()())

      [[][]] (()()())                           product
      [[ (()()()) ][ (()()()) ]]                distribute 3 into 2
      [   ()()()      ()()()   ]                involution

2^3 = 8          (()()) ^ (()()()) =?= (()()()()()()()())

      [[ (()()) ] (()()()) ]                    power
      [   ()()    (()()()) ]                    involution
      [( (()()) (()()) (()()) )]                distribute 2 into 3
         (()()) (()()) [[][]]                   involution   (2*2*2)
         (()()) [[(()())][(()())]]              distribute 2 into 2
         (()()) [   [][]    [][]  ]             involution
      [[(()())] [(()())] [(()())] [(()())]]     distribute 2 into 4
      [  ()()     ()()     ()()     ()()  ]     involution
```

Kauffman Numbers

Kauffman Arithmetic (String Version)

Integers

```
1     *
2     **    -->  (*)
3     ***   -->  (*)*
4     ****  -->  ((*))
```

Operations

```
+     juxtapose               a + b       is    a   b
x     substitute for *        a x b       is    a   b
                                                 \_/
```

Canonical Transformations

```
Instance                   Conventional arithmetic          Algebraic

* a  =  a *                1 + a = a + 1                   a  b = b  a

**  =  (*)                 1 + 1 = 2 x 1               a a = 2a = (a)

(a)(b)  =  (a b)           2a + 2b = 2(a + b)              a)(b = a  b
```

Example 2 x (4 + 3) --> 14

```
(*)    ((*)) (*)*    =    (  ((*))(*)*  )  -->  (  ((*)*)*  )
  \______/_____/_/
```

Kauffman Arithmetic (Molecular Version)

Addition is *physical mixing*. 4 + 3

```
  *    *              (*)       (**)            ((*))
*   *      -->   (*)  *     -->        *   -->          *
  *  *  *          (*)            (*)              (*)
```

Multiplication is *chemical mixing*. 2 x 7

```
    ((*))          (((*)))
  ___/                                (((*)))
 /         (*)  -->        ((*))    -->                -->  (((*)*)*)
(*)    *   /            (*)                ((*)*)
 \_____/___/
```

James Numbers

James calculus uses three types of containers/boundaries to represent all types of numbers. Several unique numerical concepts arise from this approach. *Generalized cardinality* applies to negative and fractional counts, as well as to integer counts. The *generalized inverse* unifies subtraction, division, and roots into a single concept and operation. The *James imaginary*, **J**, embeds inverse operations into numbers with *phase* as well as magnitude. J can be used for numerical computation as an alternative to using numbers.

James Form	Interpretation
Boundary Operators	(swapping between exponential and logarithmic spaces)
(a)	e^a
[a]	ln a
<a>	inverse a (generalized)
Boundary Units	(every boundary is both an object and an operator)
( )	e^0 = 1
[ ]	ln 0 = negative infinity
< >	negative 0 = 0
Integers	(stroke arithmetic using containers)
void	0
( )	1
( ) ( )	2 ...

Since stroke representation is rather clumsy. I will use decimal numbers to abbreviate stroke numbers throughout this section.

Operations

a b	a+b	(shared space)
([a] [b])	a*b	
(([[a]][b]))	a^b	

Boundaries can be read as exponents and natural logs:

```
a*b = ([a][b]) = e^(ln a + ln b) = e^ln a * e^ln b = a*b

a^b = (([[a]][b])) = e^(e^(lnln a + ln b))
          =      = e^(e^lnln a * e^ln b)
                 = e^(ln a * b) = (e^ln a)^b = a^b
```

Reduction Rules (Axiomatic basis)

Computation is achieved through application of three reduction rules:

([a]) = [(a)] = a	*Involution*
(a [b]) (a [c]) = (a [b c])	*Distribution*
a <a> = void	*Inversion*

The Form of Numbers

Type	Standard form	James form
zero	0	void
natural	n	()()..n = ([n][()])
negative integer	-n	<()()..n> = ([n][<()>]) = <([n][()]>
rational	m/n	([m]<[n]>)
irrational	m^-n	(([[m]]<[n]>))
complex	i	(([[<()>]]<[2]>))
transcendental	e	(())

The Form of Numerical Computation

In the container representation, the relationships between numerical operations becomes overt. Essentially, any operation is applying the pair (...[...]...) to a particular part of the existing form. Addition begins with no boundaries. Like stroke arithmetic, addition (and its inverse subtraction) is putting things in the same space. Multiplication (and its inverse division) involves converting to logs with [...] and then back to powers of e with (...). Power (and its inverse root) is another application of the (...[...]...) form.

addition	A B
multiplication	([A] [B])
power	(([[A]] [B]))
subtraction	A < B >
division	([A] < [B] >)
root	(([[A]]< [B] >))

These forms are spread out to show how each operator is an (...[...]...) elaboration of the previous form:

```
addition                A    B
multiplication       ( [ ]  [ ] )
power               ( [   ]      )

subtraction             A  <  B  >
division             ( [ ]   [ ]  )
root                ( [   ]        )
```

The placement of containers reflects the properties of each operator. Both forms are free of containment for commutative addition. Both forms are enclosed for commutative multiplication. One form is enclosed for power, it is not commutative. Inversion is generic, the second form is simply inverted in all cases, creating the non-commutative inverse operations.

Logarithms

```
log base e          ln n              [n]
antilog base e      antiln n          (n)

log base b          logb n           ([[n]] <[[b]]>)

antilog base b      antilogb n      (( [n]   [[b]] ))
```

Setting the logarithmic base to `e` results in the appropriate reduction:

```
ln n = ([[n]] <[[(())]]>)
     = ([[n]] <[  ()  ]>)
     = ([[n]] <        >)
     = ([[n]]           )
     =   [n]
```

Converting between bases:

```
ln n = log10 n * ln 10

[n] = (([[n]]<[[10]]>)) times [10]        hybrid

    = ([(([[n]]<[[10]]>)][[10]])
    = (  [[n]]<[[10]]>  [[10]])
    = (  [[n]]                )
    =     [n]
```

Using the spread out form, we can see the relationship between logs and the other operations:

```
subtraction             A  <  B  >
division             ( [ ]   [ ]  )
logarithm base B       [    ] [    ]

addition                A     B
multiplication       ( [ ]   [ ]  )
antilog base B      (        [    ]  )
```

Generalized Inverse

The ***generalized inverse*** treats subtraction, division, and roots as the same operation in different contexts. Below, the spacing between characters is used for emphasis.

```
-1                       < ()  >
-B                       <  B  >
A-B                   A  <  B  >

1/2                 (    <[ 2 ]>)
1/B                 (    <[ B ]>)
A/B                 ( [A] <[ B ]>)

A^2                (([[A]] [ 2 ] ))
A^B                (([[A]] [ B ] ))
A^-B               (([[A]] [<B>] ))
A^(1/B)            (([[A]]<[ B ]>))

ln B                       [ B ]
logA B              ([[A]]<[[B]]>)
antilogA B         (( [A]  [[B]] ))
```

Some James Calculus Theorems

Name	Form	Interpretation
Cardinality	A..n..A = ([A][n])	A+..n..+A = A*n
Dominion	[ ] A = [ ] ([ ] A) = void	-inf + A = -inf e^(A + -inf) = 0
Inverse Collection	<A><B> = <A B>	(-A)+(-B) = -(A+B)
Inverse Cancellation	<<A>> = A	--A = A
Inverse Promotion	([A][<B>]) = <([A][B])>	A*-B = -(A*B)

Some examples of proof:

```
--A = A                        <<A>>
                               <<A>><A>A          inversion
                                       A          inversion

-ln(e^A) = -A = ln(e^-A)       <[(A)]>
                               <  A  >            involution
                               [(<A>)]            involution

A/A = 1                        ([A] <[A]>)
                               (         )        inversion
```

```
e^A * e^-A = 1          ([(A)][(<A>)])
                        (  A    <A>  )          involution
                        (            )          inversion
```

Generalized Cardinality

Multiple reference can be explicit (a listing) or implicit (a counting). n references to A can be abstracted to n times a single A, in both the additive and the multiplicative contexts. The form of cardinality is:

```
([A][n])
```

Adding A to itself n times is the same as multiplying A by n:

```
A ..n.. A   =  ( [A] [n])
```

Multiplying A by itself n times is the same as raising A to the power n:

```
([A]..n..[A]) = (([[A]][n]))
```

Negative cardinality cancels or suppresses positive occurrences. The form of negative cardinality is

```
([A][<n>])
```

Adding A to itself -n times is the same as multiplying A by -n, and is also the same as adding -A to itself n times:

```
A..<n>..A = ([A][<n>]) = <([A][n])> = ([<A>][n]) = <A>..n..<A>
```

Adding ln A to itself n times and then raising e to that power is the same as multiplying A by itself -n times.

```
(<[A]>..n..<[A]>) = (([<[A]>][n])) = (<([[A]][n])>)
                  = (([[A]][<n>])) = ([A]..<n>..[A])
```

Multiplying -A by itself n times is the same as raising -A to the nth power:

```
([<A>]..n..[<A>]) = (([[<A>]][n]))
```

Here is a proof that negative cardinality cancels positive cardinality:

```
([A][n]) ([A][<n>])             (n*A) + (-n*A) = 0

   ([A][n <n>])                 distribution
   ([A][ ])                     inversion
   ([ ])                        dominion
   void                         inversion
```

Fractional cardinality constructs fractions and roots. The form of fractional cardinality is:

```
([A]<[n]>)
```

Adding the fraction `A/n` to itself `n` times yields `A`. Here is a proof that fractional cardinality accumulates into a single form:

```
([A]<[n]>)..n..([A]<[n]>)          (A/n) +..n..+ (A/n) = A

    ([([A]<[n]>)][n])              cardinality
    (  [A]<[n]>  [n])              involution
    (  [A]         )               inversion
        A                          involution
```

Multiplying the fraction `n/A` by itself `1/n` times yields `1/A`:

```
([n]<[A]>)..1/n..([n]<[A]>)        (n/A)*(1/n)= 1/A

    ([([n]<[A]>)][(<[n]>)])        cardinality
    (  [n]<[A]>    <[n]>  )        involution
    (      <[A]>          )        inversion
```

James Calculus Unit Combinations

The two-unit combinations generate {0, e, inf}. The only three unit combination which does not reduce has an imaginary interpretation.

Two unit combinations

```
(<>) = () = 1                e^-0
(()) = e                     e^1
([]) = void = 0              e^(ln 0) = e^(-inf)

[<>] = [] = -inf             ln -0 = -inf
[()] = void = 0              ln e^0 = ln 1 = 0
[[]] = [] = J inf            lnln 0 = ln -inf = ln -1 + ln inf

<<>> = <> = void             --0 = 0
<()> = -1                    -e^0
<[]> = inf                   --inf = inf
```

Three unit combinations

```
<([])> = <[()]> = ([<>]) = [(<>)] = 0

(<[]>) = <[]>          e^--inf = e^inf = inf

[<()>]                 J, the James imaginary

    J = [<()>] = ln -1
```

The James Imaginary

Independence

```
[<(A)>] = A [<()>] = A J
```

Interpretation:

```
ln(-e^A) = A + ln-1 = A + J
```

Proof:

```
[<(A)>] =    [<(A)>][ () ]               add 0
        = [ ([<(A)>][ () ]) ]            involution
        = [<([ (A) ][ () ])>]            promote
        = [ ([ (A) ][<()>]) ]            promote
        =       A   [<()>]               involution
```

Imaginary Cancellation

```
[<()>] [<()>] = JJ = void
```

Interpretation:

```
J + J = 0
```

Proof:

```
[<()>][<()>] = [  ([<()>][<()>])  ]      involution
             = [<<([ () ][ () ])>>]      promote
             = [<<(              )>>]    involution
             = [  (              )  ]    cancel
             = void                      involution
```

Own Inverse

(only `0` has this property in conventional number systems)

```
J = <J>
```

Interpretation:

```
J = -J            and J =/= 0
```

Proof:

```
J = J <>                                 add 0
  = J <JJ>                               J cancel
  = J <J><J>                             collect
  = <J>                                  inversion
```

Phase

The phase of J is determined by its cardinality.

```
void = JJ = JJJJ = ...                period 2
```

Rules of J

Here are some common patterns which involve `J`.

```
J = <J>
JJ = void

 J  = [<()>]
(J) =  <()>

  A   = <(J  [A] )>                 A (J [A]) = void
 <A>  =  (J  [A] )

 (A)  = <(J   A  )>                 (A) (J A) = void
<(A)> =  (J   A  )

 [A]  = <(J [[A]])>                 [A] (J [[A]]) = void
<[A]> =  (J [[A]])

(  A  [J]) = <( A [J])>
([<A>][J]) =  ([A][J])

void = () <()> = () (J)
```

Inverse Operations as J Operations

`J` is intimately connected with the act of inversion. Its definition contains `-1`; as well, it is implicated in the definition of a reciprocal since `1/A = A^(-1)`. All occurrences of the generalized inverse can be converted to `J` forms:

subtraction	`A-B`	`A <B> = A (J [B])`
reciprocal	`1/B`	`(<[B]>) = ((J [[B]]))`
division	`A/B`	`([A]<[B]>) = ([A](J [[B]]))`
root	`A^(1/B)`	`(([[A]]<[B]>)) = (([[A]] (J [[B]])))`
negative power	`A^-B`	`(([[A]][<B>])) = (([[A]][(J [B])]))` `= ((J [B][[A]]))`
log base A	`logA B`	`([[A]]<[[B]]>) = ([[A]] (J [[[B]]]))`

J in Action

J provides an alternative technique for numerical computation. Consider the two versions of this proof:

```
(-1)*(-1) = 1        ([<()>][<()>])
                    <([ () ][<()>])>         promote
                   <<([ () ][ () ])>>        promote
                     ([ () ][ () ])          cancel
                     (            )          involution

(-1)*(-1) = 1        ([<()>][<()>])
                     (  J     J   )          J
                     (            )          J cancel
```

Finding and creating Js in a form usually offers a short cut for reduction.

```
(-1)/(-1) = 1        ([J] <[J]>) = ()        inversion


A^(-1) = 1/A       (([[A]] [<()>])) =?= (<[A]>)

                   (([[A]]   J   ))
                   ( <[A]>       )


(a+1)(a-1) = a^2 - 1

([a ()][a <()>]) = ([a ()][a (J)])
                 = ([a ()][a]) ([a ()][(J)])
                 = ([a][a]) ([()][a]) ([a] J) ([()] J)
                 = ([a][a]) (    [a]) ([a] J) (     J)
                 = ([a][a])      a    ([a] J) (     J)
                 = ([a][a])                   (     J)

                 = (([[a]][2])) (J) = a^2 - 1
```

Transcendentals and Complex Functions

```
(()) = e^(e^0) = e
```

Since no rules reduce (()) to any other form, e is incommensurable with other numbers.

i, the square root of -1

```
i = (-1)^(1/2)     (([[ -1 ]] [  1/2  ]))        hybrid

                   (([[<()>]] [(<[2]>)]))
                   (([  J   ]   <[2]>  ))
                   (([  J   ](J [[2]]) ))
```

```
i = (([J](J [[2]])))
```

This leads to the interesting interpretation:

```
i = (([J]<[2]>)) = e^(J/2)
```

Squaring:

```
i^2 = e^J = -1
```

This can be derived directly:

```
i^2 = -1            (([[i]] [2]))  =  <()>
                  [(([[i]] [2]))] = [<()>]
                    ([[i]] [2])   =   J
```

This yields an interpretation which is consistent with the derivation from i:

```
J = 2 ln i
```

The J form of complex numbers is:

```
a+bi = a ([b][i]) = a ([b][(([J](J [[2]])))])
                  = a ([b]  ([J](J [[2]]))  )
```

P i

Using Euler's formula we can find another interesting result:

```
e^(i*Pi) = -1
e^J      = -1

J = i*Pi
```

Now we can express Pi in terms of J:

```
Pi = J/i = ([J]<[i]>) = ([J]<[(([J](J [[2]])))]>)
                      = ([J]<  ([J](J [[2]]))  >)
                      = ([J]   ([J](J [[2]]))   )

Pi = ([J] ([J] (J [[2]])))
```

Interpreting:

```
Pi = ([J] ([J] (J [[2]])))
   = ([J] ([J] <[2]>))
   = ([J]    J/2     )              hybrid
   = ([J]  [(J/2)]   )
   = J * e^(J/2)

Pi = Je^(J/2)
```

Yet another relationship:

```
Pi = 2i ln i = J*i

2i ln i = ([2] [(([J]<[2]>))] [[(([J]<[2]>))]])
        = ([2]   ([J]<[2]>)       [J]<[2]>    )
        = (      ([J]<[2]>)       [J]         ) = Pi
```

Combining results from above, we get the straightforward result:

```
J*i = J/i
```

cos x = (e^ix + e^-ix)/2

Trigonometric functions can be expressed as imaginary powers of e:

```
cos x = ([(ix)(<ix>)]<[2]>)                hybrid

      where  i = (([J] <[2]>))

             ix = ([i][x]) = (([J]<[2]>)[x])

   cos x = ( [((([J]<[2]>)[x])) (<(([J]<[2]>)[x])>)] <[2]>)

         = ([((([J]<[2]>)[x]))] <[2]>) ([(<(([J]<[2]>)[x])>)] <[2]>)
         = (  (([J]<[2]>)[x])   <[2]>) (  <(([J]<[2]>)[x])>   <[2]>)

   let c = <[2]> = (J [[2]])

         cos x = ((([J] c)[x]) c) (<(([J] c)[x])> c)

   let d = [x] (c [J])

         cos x = ((d) c) (<(d)> c)
               = ((d) c) ((J d) c)

               = (c   (d))   (c   (J d)  )
               = (c [((d))]) (c [((J d))])
               = (c [((d))((J d))])
```

J in Standard Notation

i is the additive imaginary. J is the multiplicative imaginary.

```
J + J = 0

J = ln -1

e^J = e^(ln -1) = -1
```

Bricken Star, *

When the value -1 is accessed by going through the imaginary J, call it *.

```
* = -1          *^2 = 1          * = -*          * + * = 0

     - A = A times *         = Ae^J

     1/A = A^*               = A^e^J

     n^(1/A) = n^A^*         = n^A^e^J

     i = *^(1/2)             = *^2^*
```

Base-free

In going through imaginary hyperbolic space and then returning, the base is arbitrary.

```
n^J = n(logn -1) = -1
```

Thus the base can be chosen to be the same as the inverse number, i.e.:

```
     - A = A times *         = AA^J = A^(J+1)

     1/A = A^*               = A^A^J

     A^(1/A) = A^A^*         = A^A^A^J
```

* *algebra*

```
a^* + b^* = (a+b) times (ab)^*

(([[a]][*])) (([[b]][*])) = ([a b] ([[a]][*]) ([[b]][*]))
```

Infinities

This calculus has a natural representation of infinity, <[ ]> which we can use computationally:

```
[] <[]> =/= void

<[]> = (J [[]])

<[]> X = <[]>                 inf + X = inf

[<[]>] = <[]>                 ln inf = inf

(<[]>) = <[]>                 e^inf = inf        (<[]>) = 1/0

[[]] = J <[]>                 ln ln 0 = ln -inf = J inf
```

```
1^inf = (([[1 ]][inf ]))       hybrid
      = (([[()]][<[]>]))
      = (([    ][<[]>]))
      = (              ) = 1
```

Here we can see that `1` raised to any power will result in `1`. However:

```
0^inf = (([[]][<[]>]))
      = (([[]] <[]> ))
      = ((     <[]> ))
      = <[]>                          = inf

0^0 = (([[]][])) = (([])) = ()        = 1

0/0 = ([]<[]>) = ([]) = void          = 0

1/0 = ([()] <[]>) = (<[]>) = <[]>     = inf

0^(1/0) = (([[]] [([()] <[]>)]))
        = (([[]]       <[]>  )) = <[]>   = inf
```

Differentiation

The rules of differentiation in the James calculus follow. Let `'A'` be `dA/dx`.

```
'c' = void                       dc = 0

'x' = ()                         dx = 1

'(A)' = (['A']  A  )             de^A = e^A dA

'[A]' = (['A']<[A]>)             dln A = 1/A dA

'<A>' = <'A'>                    d-A = -dA

'A B' = 'A' 'B'                  d(A+B) = dA + dB
```

Proof of the Chain Rule of Differential Calculus

```
d(a*b) = b da + a db

'([a][b])'

 ([a][b]['[a][b]'])

 ([a][b] [(<[a]>['a']) (<[b]>['b'])])

 ([a][b] [(<[a]>['a'])]) ([a][b] [(<[b]>['b'])])

 ([a][b]   <[a]>['a'])   ([a][b]  <[b]> ['b']  )

 (   [b]        ['a'])   ([a]           ['b']  )
```

Using James differentiation:

```
y = e^(ax)              dy = ae^(ax)

y = (([a][x]))

dy = '(([a][x]))'
   =  ([ '([a]  [x])'        ] ([a][x]))
   =  ([(['[a]  [x]'] [a][x])] ([a][x]))
   =  (  ['[a]  [x]'] [a][x]   ([a][x]))
   =  (  ['[a]''[x]'] [a][x]   ([a][x]))
   =  (  [(['a']<[a]>)(['x']<[x]>)] [a][x] ([a][x])  )
   =  (  [(([   ]<[a]>)([() ]<[x]>)] [a][x] ([a][x])  )
   =  (  [            (     <[x]>)] [a][x] ([a][x])  )
   =  (                     <[x]>   [a][x] ([a][x])  )
   =  (                             [a]    ([a][x])  )
```

Interpreting:

```
dy =  ([a]  ([a][x])  )
dy =  ([a][(([a][x]))]) = a*e^(a*x)
```

```
y = x^n                 dy = nx^(n-1)

y = (([[x]][n]))        dy = ([n][(([[x]][n <()>]))])
                        dy = ([n]  ([[x]][n <()>])  )
```

```
dy ='(([[x]][n]))'
   = (['([[x]][n])'             ] ([[x]][n]))
   = ([(['[[x]]  [n]'] [[x]][n])] ([[x]][n]))
   = (  ['[[x]]  [n]'] [[x]][n]   ([[x]][n]))
   = (  ['[[x]]''[n]'] [[x]][n]   ([[x]][n]))

   = (  [(['[x]'       ]<[[x]]>) (['n']<[n]>)] [[x]][n] ([[x]][n]))
   = (  [([(['x']<[x]>)]<[[x]]>) (['n']<[n]>)] [[x]][n] ([[x]][n]))
   = (  [(  ['x']<[x]>  <[[x]]>) (['n']<[n]>)] [[x]][n] ([[x]][n]))
   = (  [(  [() ]<[x]>  <[[x]]>) ([   ]<[n]>)] [[x]][n] ([[x]][n]))
   = (  [(       <[x]>  <[[x]]>)             ] [[x]][n] ([[x]][n]))
   = (           <[x]>  <[[x]]>                [[x]][n] ([[x]][n]))
   = (           <[x]>                                [n] ([[x]][n]))

   = ([n] ([[x]][<()>]) ([[x]][n     ]))
   = ([n]               ([[x]][n <()>]))
```

Interpreting:

```
dy = ([n]   ([[x]][n <()>])  )
dy = ([n] [(([[x]][n <()>]))]) = n*x^(n-1)
```

The next derivation illustrates the use of J:

```
y= J   = [<()>]

dy = '[<()>]'
   = (['<()>'] <[<()>]>)
   = ([<'()'>] <[<()>]>)
   = ([<    >] <[<()>]>)
   = ([      ] <[<()>]>) = void


'<A>' = '(J [A])'
          = (['J''[A]'          ] J [A])
          = ([   (['A'](J [[A]]))] J [A])
          = (     ['A']          J    )
          =       <'A'>
```