Boundary Number Systems -- James Imaginary

William Bricken
January 2001

The James Imaginary

Introductory Comments

The quintessential imaginary number is `i`, the square root of minus one.

```
i = sqrt[-1]
```

`i` is the solution to the quadratic equation

```
i^2 = -1
```

Expressed as a self-referential equation,

```
i = -1/i = - (i^-1)
```

The imaginariness of `i` comes from the composition of two inverse operations, subtraction and division. When the quadratic is equated to positive rather than negative unity, `i` represents a standard unity:

```
i^2 = 1

i = {-1, 1}
```

In the self-referential equation, removal of the additive inverse expresses the same result:

```
i = + (i^-1) = 1/i
```

When the self-referential equation does not implicate the reciprocal of `i`, `i` becomes equal to minus `i`, a role traditionally reserved for zero.

```
i = - (i^+1) = -i
```

Thus it appears that both the additive and the multiplicative inverses are required to identify the imaginary unity.

The Boolean analog to the numerical `i` is the "square root of NOT" [Shoup], N. What Boolean value, when composed with itself, is equal to the negation of itself?

```
N op N = not N
```

Self-referentially

```
N = N and not N

N = ((N)((N))) = ((N) N)
```

with the solution (the Kauffman-Varela imaginary)

```
N = not N

N = (N)
```

The Boolean imaginary oscillates with a cycle of two. The numerical `i` has a cycle of four:

```
i^0 = 1
i^1 = i
i^2 = -1
i^3 = -i
i^4 = 1
```

Strictly, this cycle is defined through successive multiplications, `i`, we might say, is the multiplicative imaginary. Addition does not shift i through imaginary and real numerical domains. Thus a complex number can be expressed as a sum of a real and an imaginary component, with zero acting in its usual multiplicative role to orthogonalize the complex in either domain:

```
1*1 + 0*i = 1
0*1 + 1*i = i
```

`i` is, in fact, a complex imaginary, a numerical composition of a simpler imaginary, the additive imaginary, which we will label using `J`:

```
J = -J
```

`J` is not equal to zero, it is imaginary. We are restricted not to divide each side of the above equation by 2 since the operation of division undermines the imaginary property of `J`.

What are the characteristics of this new imaginary? We will relate it to `i`, showing that `i` is a particular combination of two `J`s; we will relate it to standard numerical operations, showing that

```
J = ln -1
```

Accepting the above as a definition, we see that

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

That is,

```
i^2 = e^J

i = e^(J/2)
```

```
J = ln i^2 = 2 ln i
```

Some properties of `J` are proved below. The most interesting and fundamental of these is that `J` does not equal `0`, however it is its own additive inverse.

```
J = -J
```

That is,

```
J + J = 0
```

In the additive domain, J has a cycle of two:

```
J +  0 = J
J +  J = 0
J + -0 = J
J + -J = 0
```

We can now see that `i` is composed of two `J` cycles:

```
i^0   = (e^(J/2))^0 = e^0         = 1
i^1   = (e^(J/2))^1 = e^(J^2) = i
i^2   = (e^(J/2))^2 = e^J          = -1
i^3   = (e^(J/2))^3 = e^(3J/2) = e^(J + (J/2)) = e^J * e^(J/2) = -1 i
i^4   = (e^(J/2))^4 = e^2J = e^0 = 1
```

J, Ln(-1)

Logarithms are defined for positive numbers only, since `ln 0 = -infinity`. Euler, in 1751, defined logarithms of negative numbers as belonging to the complex domain. The exact relationship is given by *Euler's equation*:

```
e^ib =     cos b + i*sin b

  ib = ln (cos b + i*sin b)
```

When `b = PI` we get

```
iPI = ln (-1 + i*0) = ln -1
```

The meaning of logarithms of negative numbers was widely discussed in the eighteenth century. However, Euler's result seemed to resolve the questions: logs of negative numbers were complex numbers.

The James imaginary, `J`, also addresses the logarithm of a negative number, but without introducing complex numbers. When the angle `b` in Euler's equation rotates through `360` degrees, or `2PI` radians, it returns to its origin. A rotation of `PI` radians, `180` degrees, exactly reverses the direction of the complex vector. Since `sin 180 = 0`, there is no i-imaginary component to this rotation, thus *no reference to* `i` *is necessary* in this case. `J` represents this specific rotation. Let

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

A logarithm can be partitioned into a real and a `J`-imaginary part, the imaginary part carrying the impact of a negative number on a logarithm:

```
ln -n = ln(n*-1) = ln n + ln -1 = ln n + J
```

Demonstration:

```
ln -5 = ln (5*-1) = ln 5 + ln -1 = (ln 5) + J
```

In boundary notation:

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

Some properties of `J` are proved below using the same axioms as non-imaginaries. The most interesting and fundamental of these is that `J` does not equal `0`, however it is its own additive inverse.

```
J = -J
```

That is,

```
J + J = 0
```

Illegal Transforms

Here is a simple demonstration of the generation of `J` from standard transforms:

```
0 = ln 1 = ln(-1*-1) = ln-1 + ln-1 = J + J = 0
```

Compare this to a similar transformation of the imaginary `i`:

```
1 = sqrt 1 = sqrt(-1*-1) = sqrt-1 * sqrt-1 = i*i = -1
```

Conventionally, we put a restriction on splitting `1` into `-1` squared. There is no particular logic to this other than if we allow it, then we can generate contradiction. Somehow, our conceptualization of the imaginary `i` does not work as smoothly as it should.

The imaginary `J` manages this potential contradiction without restriction. For example:

```
ln(sqrt 1) = ln 1^(1/2) = (1/2)*ln 1 = (1/2)*ln(-1*-1)

           = (1/2)*(J+J) = 0
```

Inverting the `ln` function by raising `e` to the power of the result (i.e. `0`) restores the correct answer of `1`.

Due to the self-inverse property of `J`, care must be taken in using `J`, since the normal algebraic operations do not remain consistent. For example,

```
J + J = 2J = 0
```

The problem is

`2J = 0` does not imply `J = 0/2 = 0`

In general, `J` cannot be partitioned, or divided in pieces, as can the non-imaginary numbers. `J` is an additive concept, with non-standard behavior for multiplication. Basically, `J` acts as a parity mechanism. All even counts of `J` reduce to zero. For division, `J` will stand in relation to any denominator (such as `J/5`). All numerators reduce either to zero (in the case of an even numerator) or to one (in the case of an odd numerator).

J Theorems

Definition

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

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

Independence

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

Imaginary Cancellation

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

Own Inverse

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

```
J = <J>
```

J abstract

(converts all <>-forms into `J`-forms)

```
 (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]])
```

J invert

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

Proofs:

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

[<(A)>] =     [<(A)>][ () ]                     involution
        = [ ([<(A)>][ () ]) ]                   involution
        = [<([ (A) ][ () ])>]                   promote
        = [ ([ (A) ][<()>]) ]                   promote
        =       A   [<()>]                      involution


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

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


J = <J>

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


A (J [A]) = void

      A  ([<()>][A])                            substitute
      A <([ () ][A])>                           promote
      A <(      [A])>                           involution
      A <        A  >                           involution
      void                                      inversion


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

      (A [J])                                   lhs
      (A [<J>])                                 J inverse
      <(A [J])>                                 promote
```

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), and in the definition of a root since A^(1/n) = A^(n^-1). All occurrences of the generalized inverse can be converted to J forms:

Operation	*Interpretation*	*J form*
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]))
log base A	logA B	([[A]]<[[B]]>) = ([[A]] (J [[[B]]]))

The exchange of <>-forms for J-forms mimics process/object confounding. Converting a container, <>, into an object, J, simplifies pattern matching but renders the form more difficult to read.

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 can offer a short cut for reduction. The primary substitution is -1 = (J). Some other *examples:*

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

```
A^(-1) = 1/A       (([[A]] [<()>])) =?= (<[A]>)
                   (([[A]]   J   ))                        substitute
                   ( <[A]>        )                        J abstract


1/(1/A) = (A^-1)^-1 = A

             (<[   (   <[A]> )  ]>)
             (<[   ((J [[A]]))  ]>)                        J abstract
             ((J [[((J [[A]]))]] ))                        J abstract
             ((J     J [[A]]     ))                        involution
             ((        [[A]]     ))                        J cancel
                         A                                 involution


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

             ([a ()][a <()>])
             ([a ()][a (J)])                               substitute
             ([a ()][a]) ([a ()][(J)])                     distribution
             ([a ()][a]) ([a ()]  J  )                     involution
             ([a][a]) ([()][a]) ([a] J) ([()] J)           distribution
             ([a][a])      a    ([a] J) (     J)           involution
             ([a][a])      a     <a>    (     J)           J abstract
             ([a][a])                   (     J)           inversion
             (([[a]][2])) (J)                              cardinality

             a^2 - 1                                       interpret
```

Conventional algebra is naturally much more efficient than using boundaries and `J`. With boundary numbers, we are working closer to the foundations of computation. That is, with fewer types of steps and with more steps taken, BN resembles a RISC architecture for numerical computation.

Dot as -1

A notational tool helps keep track of when the imaginary `J` is used. Whenever the value `-1` is converted to `J`, call it `·`.

```
J = [·]

· = (J) = -1

i = ·^(1/2) = ·^(2^·)


- A = A*·                       Ae^J

1/A = A^·                       A^e^J
```

```
        n^(1/A) = n^A^·                n^A^e^J
```

Some computations using •:

```
    a^· + b^· = (a+b)*(ab)^·

    (([[a]][·])) (([[b]][·])) =?= ([a b][ (([[([a][b])]][·])) ])

      ([a b] ([[a][b]][·]) )                                   rhs
      ([a b] ([[a]][·]) ([[b]][·]) )                           distribution
      ([a] ([[a]][·]) ([[b]][·]) ) ([b] ([[a]][·]) ([[b]][·]) ) distribution
      (               ([[b]][·]) ) (    ([[a]][·])            ) J abstract


    (a^·)^· = a

          (([[a]][·]))^·                                       hybrid
      (([[(([[a]][·]))]][·]))                                  substitute
      ((    [[a]][·]    [·]))                                  involution
      ((    [[a]]          ))                                  J cancel
              a                                                involution
```

Base-free

In going through imaginary logarithmic space and then returning, the base can be arbitrary. We demonstrate this:

```
    Let J' = logb -1

        J' = logb -1 = (ln -1)/(ln b) = ([J]<[[b]]>)

        b^J' = -1 = b^(logb -1)

        b^J' = b^([J]<[[b]]>)                          hybrid
               (([[b]] [([J] <[[b]]>)]))               substitute
               (([[b]]   [J] <[[b]]>  ))               involution
               ((        [J]         ))                inversion
               (          J          )                 involution
               e^J                                     interpret
               -1                                      interpret
```

We have demonstrated independence of base:

```
        b^(logb -1) = e^(ln -1)
```

Since the choice of base is arbitrary (given that it is consistent throughout a form), we can abstract it:

J = log# -1 where # is any number.

At times it may be advantageous to chose the base to be the same as the number being inverted, Below, A is both the form being operated upon, and the base of transformations; that is

J = logA -1 A^J = -1

Inverse operation	*Representations*		
- A	A*(-1)	A*·	= A*(A^J) = A^(J+1)
1/A	A^(-1)	A^·	= A^A^J
A^(1/A)	A^A^(-1)	A^A^·	= A^A^A^J

J Self-interaction

We have seen some rules which involve reduction using J. For example:

A (J [A]) = void *J abstract*

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

J permits transformation of inverse operations through its inversion ambiguity, i.e.:

J = <J>

J also interacts with itself:

J log J

J [J] = [J]

Proof:

J [J] =	[(J [J])]	involution
	[<J>]	J abstract
	[J]	J invert

J Parity

The relationship between J and cardinality is non-standard. Let n be an integer:

Parity Theorems

([J][n]) = void n even
([J][n]) = J n odd

```
J ([J][n]) = J                      n even
           = void                   n odd

J ([J]<[n]>) = ([J]<[n]>)           n even
             = void                 n odd
```

The last parity theorem illustrates the unusual effect of the `J`-imaginary on cardinality. Interpreting the theorem yields:

```
J + J/n = 0                     n odd
```

which is to say

```
J + J/1 = J + J/3 = J + J/5 = ...
```

while

```
J/1 =/= J/3 =/= J/5 =/= ...
```

Additionally,

```
J + J/2 = J/2
J + J/4 = J/4
J + J/6 = J/6
```

while

```
J/2 =/= J/4 =/= J/6 =/= ...
```

Generalized `J` parity

```
([J] [m] ) ([J] [n] ) = ([J][m n])

            = void                  m+n even  (m,n same parity)
            = J                     m+n odd  (m,n different parity)

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

            = void                  (m*n + 1) even  (both odd)
            = ([J]<[n]>)            (m*n + 1) odd  (either even)

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

            = void                  m+n even
            = ([J]<[m][n]>)         m+n odd
```

Proof:

```
J ([J]<[n]>) = (   [J][()]) ([J]<[n]>   )            cardinality
               (   [J][()      (<[n]>)])             distribution
```

```
(   [J] [([([()][n])()]   <[n]>)])        distribution
(   [J]   [      n  ()]   <[n]>  )        involution
([([J]    [      n  ()])] <[n]>  )        involution
([J..n+1..J]<[n]>)                        hybrid
```

```
if n is odd,
      ([J..n+1..J]<[n]>) = ([ ]<[n]>) = void

if n is even,
      ([J..n+1..J]<[n]>) = ([J]<[n]>)
```

Demonstrations:

```
J + J/3 = J + J/7              J/3 =/= J/7

      J ([J]<[3]>) =?= J ([J]<[7]>)

          J       ([J]<[3]>)                     lhs
      (   [J][()]) ([J]<[3]>)                    cardinality
      (   [J][ ()   (<[3]>)])                    distribution
      (   [J][([4]   <[3]>)])                    distribution
      (   [J]   [4]   <[3]>   )                  involution
      ([([J]   [4])]<[3]>   )                    involution
      ([            ]<[3]>   )                   J cancel
      void                                       dominion
```

The same steps reduce `J/7` to `void`.

```
J/3 + J/7 = 0

      ([J]<[3]>)([J]<[7]>) =?= void

      ([J]<[3]>)([J]<[7]>)                       lhs
      (   [J] [3 7]    <[3][7]>)                 distribution
      (   [J] [ 10]    <[3][7]>)                 addition
      ([([J] [ 10])] <[3][7]>)                   involution
      ([             ] <[3][7]>)                 J cancel
      void                                       dominion
```

Whether or not the unusual relationship between `J` and cardinality is of computational advantage (with infinite series, for example) is unexplored territory.

Algebra of **J**

Consider how J behaves when undergoing algebraic transformation:

Operation	*Boundary form*	*Value*
J + J	J J	0
J - J	J <J>	0
J * J	([J] [J])	J^2
1 / J	(<[J]>)	1/J
J / J	([J] <[J]>)	1
J ^ n	(([[J]] [n]))	J^n
J ^ J	(([[J]] [J]))	J^J
J ^ 1/J	(([[J]]<[J]>))	J^(1/J)
e ^ J	(J)	-1
ln J	[J]	ln J

Whether or not some of these forms reduce further is an open question.

Multiplicative Forms

A * 0	([ ][A])
A * 1	([()][A]) = A
A * e	(() [A])
A * -1	(J [A])
e * 0	([ ]())
e * 1	([()]()) = (())
e * e	(() ())
e * -1	(J ())

Cyclic Forms

If we list successive cardinalities of `J`, we see that it's value oscillates.

```
void = JJ = JJJJ = ...                    Period 2
```

Period 2 sequences:

```
    ->   J   ->       ->   J   ->       ->...         J cancel
( ) ->  (J)  -> ( ) ->  (J)  -> ( ) ->...         exponent
 1  ->  -1   ->  1  ->   -1  ->  1  ->...         interpret

 A  ->  J A  ->  A  ->  J A  ->  A  ->...         J cancel in context
(A) -> (J A) -> (A) -> (J A) -> (A) ->...         exponent
e^A ->  -e^A -> e^A ->  -e^A -> e^A ->...         interpret
```

If we combine `1/2 J` at each step, the period is 4:

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

Period 4 sequences:

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

( ) -> (([J]<[2]>)) -> (J) -> (J ([J]<[2]>)) -> ( ) ->...

 1  ->      i       -> -1  ->      -i        ->  1..->...
```

Incrementing by `J/2` generates the period 4 oscillation of `i`. However, the above `J/2` sequence is also degenerate, since

```
J ([J]<[2]>) = 3J/2 = (J+J+J)/2 = J/2
```

That is,

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

The difference in interpretation between `i` and `-i` depends upon whether or not the inverse canceling effect of `J` is applied or not.

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

From generalized J parity, if we increment each step by `1/n J`, the period is apparently `n`. This will always degenerate into a period 2 sequence:

```
0J/n -> 1J/n -> 2J/n -> 3J/n ->...-> nJ/n ->      ->...
  0  ->  J/n ->   0  ->  J/n ->...->  0   -> J/n ->...
```

The relationship between J and cardinality is unusual in that the standard arithmetic operations are not consistent.

Demonstration:

```
J = 1*J       = (2/2)*J            = (2*J)/2            = 0/2       = 0

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


J = 1*J       = (3/3)*J            = (3*J)/3            = J/3

J = ([()][J]) = ([([3]<[3]>)][J]) = ([([J][3])]<[3]>) = ([J]<[3]>)
```

The problem here is that J cannot be carved into pieces. That is, J supports reciprocals but no other numerators except 1. This difficulty for the system could be addressed by a prohibition:

```
(n/n)*J =/= (n*J)/n
```

Alternatively (and more in line with boundary math techniques), we can define multiplication by J as canceling numerators, forcing a result that is either the void or a reciprocal.

J and i

First we determine the form of i:

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

                   (([[<()>]] [(<[2]>)]))          substitute
                   (([   J    ] [(<[2]>)]))          substitute
                   (([   J    ]    <[2]>   ))          involution
                   (([   J    ](J [[2]]) ))          J abstract

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

i is the multiplicative imaginary, with a phase of four {1,i,-1,-i}. J is the additive imaginary, with a phase of two {0,J}. The imaginary i is the answer to the question:

```
x times x = -1
```

The imaginary J answers the question:

```
x plus x = 0
```

Comparing the forms of `i` and `J`:

```
J = ln -1                     i = (-1)^1/2

J = -J                        i = -1/i

J + J = 0                     i + 1/i = 0

J = (-1)*(J^1)                i = (-1)*(i^-1)

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

`J` is imaginary because it is its own inverse. `i` is also imaginary, it is its own reciprocal inverse. From this perspective, `J` is a simpler, more elementary, imaginary than `i`.

Note that the boundary representation of `i` contains `J` within it.   We can evaluate the boundary form of the definition of `i` by using `J`:

```
i + 1/i = 0

           i                (<[i]>)          transcribe
([  (     [i] [i]       ) ()] <[i]>)          compound distrib
([        <()>            ()] <[i]>)          lemma
([                          ] <[i]>)          inversion
            void                              dominion


      lemma      ([i][i]) = <()>

                 (     [i]     [i]     )
                 (([[      i      ]][2]))      cardinality
                 (([[(([J]<[2]>))]][2]))      substitute
                 ((    [J]<[2]>    [2]))      involution
                 ((    [J]            ))      inversion
                 (      J              )      involution
                     <()>                     substitute
```

The exact relationship between `J` and `i` reflects the inverse canceling effect of `J`:

Conventional notation	Boundary form
`J*i = J/i`	`([J][i]) = ([J]<[i]>)`

This is easy to prove:

```
J*i*i = -J = J
```

A void-based boundary proof of the same relationship follows, using the void-equivalent form:

```
([J][i]) <([J]<[i]>)> = void

([J][i])  ([J]  <   [i]   >  )              J invert
([J][i])  ([J][(<   [i]   >)])              involution
([J][i         (<   [i]   >)])              distribution
([J][  ([([i][i])()]<[i]>)  ])              distribute compound
([J][  ([  <()>  ()]<[i]>)  ])              lemma
([J][  ([          ]<[i]>)  ])              inversion
([J][                       ])              dominion
           void                             dominion
```

The equations for `i` and `J` expressed in terms of each other:

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

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

Proof:

```
i = (([      J      ] <[2]>)           given
  = (([ ([[i]][2]) ] <[2]>))           substitute
  = ((   [[i]][2]    <[2]>))           involution
  = ((   [[i]]            ))           inversion
  =        i                           involution
```

The form of `i` leads to the interesting interpretation:

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

Squaring both sides:

```
i^2 = (e^(J/2))^2 = e^(J/2 + J/2) = e^J = -1
```

This can be derived directly:

```
i^2 = -1          (([[i]] [2]))  =  <()>
                [((([[i]] [2])))] = [<()>]      ln both sides
                   ([[i]] [2])   =   J          involution/substitute
```

Reading this form yields a consistent interpretation:

```
J = 2 ln i

e^J = e^(2 ln i) = (e^ln i)*(e^ln i) = i*i = -1
```

Another common transformation of `i` is:

```
            i = (1+i)/(1-i)

                  ([() i]<[()<i>]>)

                  ([()]<[()<i>]>) ([i]<[()<i>]>)
                  (    <[()<i>]>) ([i]<[()<i>]>)
                  ([([()<i>])([i][()<i>])] <[()<i>][()<i>]>)
                  ([  ()<i>  ([i][()<i>])] <[()<i>][()<i>]>)
                  ([  ()<i>  ([i][()<i>])] <([2][[()<i>]])>)
%%%
```

Complex Numbers

The `J` form of i-complex numbers is:

```
a+ib = a ([b][        i       ])
       a ([b][(([J](J [[2]])))])          substitute
       a ([b]  ([J](J [[2]]))  )          involution
```

In contrast to i-complex numbers, the imaginary part of J-imaginary numbers is quite limited. Let the representation of a J-imaginary be similar to a complex number:

```
a+Jb = a ([J][b])
```

Essentially, `b` can take on only two integer values, `0` or `1`.

When b is an even integer,

```
a + Jb = a
```

When b is an odd integer,

```
a + Jb = a + J
```

The sign of `b` is irrelevant, since it can be removed by J invert. The only fractional values which `b` can express are reciprocals.

However, the comparison between `i` and `J` is faulty since the i-imaginary part, `ib`, is a multiplicative component, characteristic of `i` but not of `J`. The appropriate J-complex number form is additive, and simpler than an i-complex number:

```
a + kJ          k in {0,1}
```

In boundary notation, the `k` multiplier is simply the existence or absence of `J`:

```
a  J
```

Euler's Formula

Euler's formula provides a cyclic, phase-oriented interpretation of i-imaginary numbers:

```
e^(a+ib) = (e^a)*(cos (b + 2kPI) + i*sin (b + 2kPI))
```

Powers of complex numbers are interpreted in the complex plane, with the angle of rotation defined by the ratio of real and imaginary components. This is how PI, as a measure of rotation in radians, becomes associated with every complex number. At 0 and 180 degree rotations, the imaginary component is zero. Since `k` can be any integer, the complex power function is a one-to-many mapping.

Setting the real component to zero yields:

```
e^(a+ib) = (e^a)*(cos (b + 2kPI) + i*sin (b + 2kPI))

e^(  ib) = (e^0)*(cos (b + 2kPI) + i*sin (b + 2kPI))

      e^ib = cos (b + 2kPI) + i*sin (b + 2kPI)
```

Additionally setting the angle of rotation to 180 degrees (`PI`) leads to the simplified Euler equation. Ignoring the cyclic component, we get

```
e^ib = cos b + i*sin b

e^iPI = -1 + i*0

e^iPI = -1
```

This leads directly to J:

```
e^iPI = e^J
```

Let us reintroduce the cyclic component to the reduced equation:

```
J = i(PI + 2kPI) = iPI(2k + 1)            k is an integer
```

This results implies that `J` has an infinite set of values:

```
J = ([      i      ][        PI        ][2k 1])          hybrid
    ([(([J]<[2]>))][([J] ([J] <[2]>))][2k 1])          substitute
    (  ([J]<[2]>)    [J] ([J] <[2]>)  [2k 1])          involution
    (    J           [J]              [2k 1])          J log J
    (                [J]              [2k 1])          distribute/J

      J = J*(2k+1)
```

Recalling J parity, we see that this result is both necessary and consistent.

```
([J][n]) = void                    n even
([J][n]) = J                       n odd
```

However, since J is not partitionable, the cyclic component of Euler's formula is eliminated, replaced by J cycles of period 2.

Logarithms

In general, the logarithm of a complex number is given by

```
ln (a+ib) = ln |z| + i*angle z
```

where

```
|z| = (a^2 + b^2)^(1/2)

angle z = arctan(b/a)
```

J-imaginary numbers remove some of the complexities of working with complex numbers. Specifically, using Euler's formula we can express PI in terms of J:

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

       J = i*PI
```

In the simple conventional case of the exp-log inverse relation:

```
e^ln z = z + 2kiPI
```

Substituting

```
e^ln z = z + 2kJ = z + k*0 = z
```

The self-canceling property of J removes the cyclic component, since all cycles return only to zero. The analogous Euler's formula for J-complex numbers is:

```
e^(a + J) = (e^a)*(e^J) = -e^a
```

This form does not introduce i-complexity even though it uses J-imaginary numbers which can be expressed as i-imaginaries. This is simply because J does not permit rotational partitions. A J rotation is either 0 or 180 degrees. The J-imaginary logarithm is

```
ln(a+J) = [a J]
```

Transcendental Functions

Transcendental functions are those that are not algebraic. They include the trigonometric, exponential, logarithmic, and inverse trigonometric functions. (Algebraic functions involve the operators {+,-,*,/,^,root}.) When the exponential and the logarithmic base is set to the natural log base e, the J mechanisms address transcendental functions.

e, the natural logarithm base

```
 (())  = e^(e^0) = e^1 = e

<(())>  = -e = (J (()))

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

Since no rules reduce `(())` to any other form, `e` is additively incommensurable with other forms. That is, `e` is transcendental. The logarithm function coverts the transcendental `e` into the integer `1`:

```
[(())] = ()                    ln e = 1
```

ln, the natural logarithm

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

[[[]]]          ln(J <[]>) = ln J * ln inf
                             ([[J]] [<[]>])

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

P I

```
J = i*PI

PI = J/i = ([J]<[       i        ]>)           hybrid
           ([J]<[(([J](J [[2]])))]>)           substitute
           ([J]<  ([J](J [[2]]))  >)           involution
           ([J]   ([J](J [[2]]))   )           J invert

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

Interpreting:

```
PI = ([J] ([J] (J [[2]])))
     ([J] ([J] <[2]>))                         J abstract
     ([J]    J/2     )                         hybrid
     ([J]  [(J/2)]   )                         involution
     J * e^(J/2)                               interpret

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

Here is a different construction of `PI`:

```
PI = -1*i*i*PI

   = (J)*i*J                                      hybrid

     ( [(J)] [(([J] (J [[2]])))] [J] )            substitute
     (   J     ([J] (J [[2]]))   [J] )            involution
     (         ([J] (J [[2]]))   [J] )            J log J

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

Another construction:

```
PI = 2i ln i

   = ([2] [(([J]<[2]>))] [[(([J]<[2]>))]])        substitute
     ([2]   ([J]<[2]>)       [J]<[2]>    )        involution
     (      ([J]<[2]>)       [J]         )        inversion
     (      ([J] (J [[2]]))  [J]         )        J abstract

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

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

```
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]>)

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

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

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

cos x = (b [(d) (<d>)])
      = (b [(d) ((J [d]))])
```

Expanding:

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

cos x = ([(([x] ([J] (J [[2]])))) ((J [x] ([J] (J [[2]])))] (J [[2]]))
```

sin x = (e^ix - e^-ix)/2i

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

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

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

Substituting and simplifying:

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

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

sin x = ([((([J] b)[x])) <(<(([J] b)[x])>)>] b <([J] b)>)

      let c = (b [J])

sin x = ([((c [x]))<(<(c [x])>)>] b <c>)

      let d = (c [x])

sin x = (b <c> [(d)<(    <d>   )>])
        (b <c> [(d)<(  (J [d]))>])                J abstract
        (b <c> [(d) (J (J [d])) ])                J abstract
        (b  c  [(d) (J (J [d])) ])                J  invert
```

Expanding:

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

([(([x] ([J](J [[2]])))) (J (J [x]([J] (J [[2]]))))] ([J](J [[2]])) (J [[2]]))
```

e^ix = cos x + i sin x

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

      (b [(d)(<d>)]) ([((([J]<[2]>))][(b c [(d)<(<d>)>])])   substitute
      (b [(d)(<d>)]) (  ([J]<[2]>)    b c [(d)<(<d>)>]  )   involution
      (b [(d)(<d>)]) (      c         b c [(d)<(<d>)>]  )   substitute
      (b [(d)(<d>)]) (     <c>        b c [(d)<(<d>)>]  )   J invert
      (b [(d)(<d>)]) (                b   [(d)<(<d>)>]  )   inversion
      (b [(d)(<d>)(d)<(<d>)>])                              distribution
      (b [(d)     (d)       ])                              inversion
      (b [([(d)][2])]  )                                    cardinality
      (b     d  [2]    )                                    involution
      (<[2]> d  [2]    )                                    substitute
      (  d             )                                    inversion
      (([x] c        ))                                     substitute
      (([x] ([J]  b  )))                                    substitute
      (([x] ([J]<[2]>)))                                    substitute
```

An Open Question

It is an open question whether or not the following structures composed of transcendental forms are themselves transcendental:

```
      e^e        (([[ e  ]][ e  ]))                         hybrid
                 (([[(())]][(())]))                         substitute
                 ((          ()  ))                         involution
```

This reduction is not a rigorous proof, but we can see that `e^e` is incommensurable with other number forms, and therefore likely to be transcendental.

The following three forms, `1` raised to a irrational value, are known to have non-unitary values in the complex plane.

```
      1^(sqrt 2)  (([[()]] [       sqrt2       ]))
                  (([[()]] [(([[2]] [(<[2]>)]))]))
                  (([    ]   ([[2]]   <[2]>  )  ))

      1^e         (([[()]] [(())]))
                  (([    ]   ()  ))


      1^PI        (([[()]] [           PI          ]))
                  (([[()]] [([J] ([J] (J [[2]])))]))
                  (([    ]   [J] ([J] (J [[2]]))  ))
```

It is not known whether or not the following two forms reduce further.

```
PI^e      (([[        PI        ]] [ e  ]))
          (([[([J] ([J]<[[2]]>))]] [(())]))        substitute
          (([  [J] ([J]<[[2]]>)  ]   ()  ))        involution


PI^PI     (([[       PI         ]][       PI         ]))
          (([[([J] ([J]<[[2]]>))]][([J] ([J]<[[2]]>))]))    substitute
          (([  [J] ([J]<[[2]]>)  ]  [J] ([J]<[[2]]>)  ))    involution
```

Axioms of Infinity

The interaction between infinity and form requires new theorems. These absorption rules define when an infinite form absorbs, or renders void, other forms sharing the same space. Negative Dominion cannot be proved, it is an axiom. The other reduction rules for infinity are derived from the Dominion axiom.

Negative Infinity

```
A [] = []                    dominion, any A

([]) = void                  involution of negative infinity

[[]] = J <[]>                log of negative infinity
```

Positive Infinity

```
A <[]> = <[]>                positive dominion, A not in {[],J}

(<[]>) = <[]>                infinite exponent

[<[]>] = <[]>                infinite log
```

Proofs:

```
[[]] = J <[]>            loglog 0

     [      []    ]
     [<    <[]>  >]                         inverse cancel
     [<(J [ [] ])>]                         J abstract
     [ (J [<[]>]) ]                         inverse promote
        J [<[]>]                            involution
```

The implication from this result is that

```
[[]] = J <[]> =/= <[]>
```

J does not absorb into positive infinity. In using J, we are introducing a calculus of infinities which is not completely degenerate. That is, we can distinguish J-imaginaries in the presence of a positive (but not a negative) infinity. Negative dominion of J is consistent with the Dominion axiom:

```
J [] = []
```

Theorems

```
<[]><[]> = <[]>            positive infinity

[[]] <[]> = [[]]           infinite absorption
```

Proofs:

```
<[]><[]>
<[]  []>                   inverse collect
<[]    >                   dominion


[[]] <[]>
J <[]> <[]>                loglog 0
J <[]>                     positive infinity
[[]]                       loglog 0
```

Void Transformations

Examining the void cases of the axioms sheds light on the arithmetic of James boundaries. We begin by noticing that the two unit forms are empty, they contain void:

```
( )          e^0 = 1

[ ]          ln 0 = -inf
```

We see that the void theorems will include operations on infinity.

Void Reduction Rules

```
Involution       ([ ]) = [( )] = void

Distribution     (A [ ]) (A [ ]) = (A [   ])

Inversion          < > = void
```

Void Algebraic Operations

Operation	Interpretation	Form	Reduced Form
Addition	`0+0`		
Multiplication	`0*0`	`( [ ]   [ ])`	`void`
Power	`0^0`	`(([[ ]]  [ ]))`	`( )`
Subtraction	`0-0`	`<     >`	`void`
Division	`0/0`	`( [ ] < [ ] >)`	`void`
Root	`A^(1/0)`	`(([[A]]< [ ] >))`	`<[]>`
	`0^(1/B)`	`(([[ ]]< [B] >))`	`void`
	`0^(1/0)`	`(([[ ]]< [ ] >))`	`void`
Logarithm	`logB 0`	`([[B]]<[[ ]]>)`	`[ ]`
	`log0 B`	`([[ ]]<[[B]]>)`	`void`
	`log0 0`	`([[ ]]<[[ ]]>)`	`void`

The Dominion theorem,

```
(A  [] ) = void

(A <[]>) = (<[]>) = <[]>            A not in {J,[]}
```

specifies the behavior of positive and negative infinity, and plays a central role in the reduction of these infinite forms. All reduced forms are as would be expected, including a proof that

```
0^0 = 1
```

which in conventional systems is taken as a definition. Also

```
logB 0 = ln 0
```

since the log of `0` is the same regardless of base.

Reduction proofs:

```
0^0         (([[]] []))
            (         )                              dominion


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


0^(1/0)     (([[]]  < [] >))
            (([[]]        ))                         infinite absorption
                 void                                involution


0^(1/B)     ([[B]]< [[]]  >)
            ([[B]]< J <[]>>)                         loglog 0
            ([[B]]<<J><[]>>)                         J inverse
            ([[B]]<<J  []>>)                         inverse collect
            ([[B]]  J  []  )                         inverse cancel
            (          []  )                         dominion
                 void                                involution

log0 0      ([[ ]]<[[ ]]>)
            (J [ ]<[[ ]]>)                           loglog 0
               void                                  dominion
```

Infinities and Contradiction

The James calculus has a natural representation of infinity, <[]> which can be used computationally. However, the use of infinity leads to contradictions, just as it does in standard approaches. Fortunately, these contradictions can be eliminated by restricting specific transformation rules.

Division by Zero

An initial question concerns the form of division by zero. It is clear that

```
A/0 = inf         ([A]<[]>)
                  (   <[]>)         positive dominion
                      <[]>          inf
```

But what is 0 divided by 0?

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

The question revolves around whether or not `+inf` inverts `-inf`. Or does Dominion apply instead? Here are the possibilities:

`[] <[]> = void`	inversion
`[] <[]> =  []`	negative dominion
`[] <[]> = <[]>`	positive dominion

When infinities collide, we have contradictory results, and must therefore enact a restriction. This choice interacts with the computational use of infinity. Specifically

`0/0 = ([]<[]>) = (     ) = 1`	inversion
`0/0 = ([]<[]>) = ( [] ) = 0`	negative dominion
`0/0 = ([]<[]>) = (<[]>) = <[]> = inf`	positive dominion

It is appealing that the choices are the three most fundamental numerical concepts, `{0,1,inf}`. However, we must choose between them so that we can freely use infinities during computation. To resolve this contradiction, a somewhat arbitrary restriction must be placed on transformations involving infinities. The exact choice depends on both syntax (i.e. which yields the most consistent results) and semantics (i.e. what is natural for the exp-log interpretation). The relation y= ln x approaches negative infinity very rapidly, and positive infinity very slowly. As well, the boundary representation initially confounds the concepts of negative and infinite. Therefore, we will assume

Negative dominion takes precedence.

Thus,

```
0/0 = 0
```

Further clarification of the above inconsistencies is an open problem.

Inconsistent Forms

`0/0`	`([]<[]>)`
`inf/inf`	`([[]]<[[]]>)`
`0*inf`	`([][<[]>]) = <([][[]])>`
`inf - inf`	`<[]> <<[]>> = <[]> []`

Infinite Powers

Here we can see that `0` or `1` raised to any power will not change that base.

```
1^inf = (([[1 ]][inf ]))                                hybrid
        (([[()]][<[]>]))                                substitute
        (([    ][<[]>]))                                involution
        (               )                               dominion
               1                                        interpret

0^inf = (([[]] [<[]>]))
        (([[]]  <[]> ))                                 inf
        ((J <[]><[]> ))                                 loglog 0
        ((J <[]  []> ))                                 inverse collect
        ((J <[]    > ))                                 dominion
        (([[]]       ))                                 loglog 0
          void                                          involution
            0                                           interpret
```

Some other results:

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


0*inf =  ([][<[]>])
         void                                           negative dominion
          0                                             interpret
```

Infinity and **J**

We can explore the behavior of infinity in imaginary contexts.

```
inf + J = <[]> J
          <[]><J>                                       J inverse
          <[]  J>                                       inverse collect
          <[]   >                                       dominion
            inf                                         interpret

inf * J = ([<[]>] [J])
          ([  []  ] [J])                                J invert
          (J <[]> [J])                                  loglog 0
          (  <[]> [J])                                  J log J
```

```
inf / J = ([<[]>]<[J]>)
          ( <[]> <[J]>)              inf
          ( <[]   [J]>)              inverse collect
          ( <[]      >)              dominion
            <[]>                     inf

J / inf = ([J]<[<[]>]>)
          ([J]< <[]> >)              inf
          ([J]   []   )              inverse cancel
          void                       dominion

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

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

Some of these transformations result in a structural relationship between `J` and `inf`:

```
inf * J =  (<[]> [J] )

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

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

Understanding this behavior is an open question.

Here is a confirmation that even cardinality is associated with `J=0`. It is reliant on negative infinity:

```
([J][n]) = void              n even

 ([J][n])     =  ([])                involution
[([J][n])]    = [([])]               ln both sides
  [J][n]      =   []                 involution
  [J][n]<[n]> =   [] <[n]>           both sides
  [J]         =   [] <[n]>           inversion
  [J]         =   []                 dominion
 ([J])        =  ([])                exp both sides
   J          =                      involution
```

Given the constraint equation, the only value for `J` which will fulfill it is `J=0`.

Imaginary Logarithmic Bases

Since the form of a logarithm is defined, we can explore the values of forbidden log bases:

```
log1 A = ([[A]]<[[()]]>)
         ([[A]]<[    ]>)          involution
         (     <[    ]>)          dominion
               <[    ]>           inf
             inf                  interpret


log0 A = ([[A]]<[[  ]]>)
         ([[A]]<<[  ]>>)          substitute
         ([[A]]  [  ]  )          inverse cancel
            void                  dominion
             0                    interpret


loginf A = ([[A]]<[[<[]>]]>)
           ([[A]]<<[  ]>>)        substitute
           ([[A]]  [  ]  )        inverse cancel
            void                  dominion
             0                    interpret

log-1 A = ([[A]]<[[<()>]]>)
          ([[A]]<[  J   ]>)       substitute
          ln A / ln J             interpret
```

Infinite Series

e is most often defined in terms of an infinite series. Several relevant series follow.

Although we cannot directly substitute infinity into a conventional formula, we are free to do so with boundary forms, since infinity is simply another form which follows the same rules.

```
lim[n->inf] (ln n)/n = 0

      ([[n]] <[n]>) =?= void        as n->inf        hybrid

            ([[<[]>]] <[<[]>]>)                      substitute inf
            (  <[]>   < <[]> >)                      inf
            (  <[]>      []   )                      inverse cancel
            void                                     negative dominion
```

```
lim[n->inf] n^(1/n) = 1

      (([[n]][1/n])) =?= ()         as n->inf          hybrid

            (([[<[]>]][(<[<[]>]>)]))                   substitute inf
            (([[<[]>]]  <[<[]>]>  ))                   involution
            ((  <[]>    < <[]> >  ))                   inf
            ((  <[]>       []     ))                   inverse cancel
            (                      )                   dominion


lim[n->inf] x^(1/n) = 1             x>0

      (([[x]][1/n])) =?= ()         as n->inf          hybrid

            (([[x]][(<[<[]>]>)]))                      substitute inf
            (([[x]]  <[<[]>]>  ))                      involution
            (([[x]]  < <[]> >  ))                      inf
            (([[x]]     []     ))                      inverse cancel
            (                   )                      dominion
```

Comparing the reduction of the above two limit forms, we can readily see why the base `x` is irrelevant (i.e. it is dominated in any event).

```
lim[n->inf] (1 + x/n)^n = e^x

      (([[1 x/n]][n])) =?= (x)      as n->inf          hybrid

            (([[() ([x]<[ n  ]>)]][ n  ]))             substitute
            (([[() ([x]<[<[]>]>)]][<[]>]))             substitute inf
            (([[() ([x]< <[]> >)]] <[]> ))             inf
            (([[() ([x]   []   )]] <[]> ))             inverse cancel
            (([[()              ]] <[]> ))             dominion
            (([                  ] <[]> ))             involution
            (                            )             dominion
                           1                           interpret
```

This result is in error, indicating that the inconsistencies in working with infinity are still present and still an unresolved problem.

Here are some infinite sums relevant to `e`.

```
e^x = SUM[n=0->inf] (x^n)/n!

e^-x = SUM[n=0->inf] (-1)^n (x^n)/n!

ln[1+x] = SUM[n=0->inf] (-1)^n (x^n)/n

ln[1-x] = SUM[n=0->inf] -(x^n)/n
```

Working with James calculus and infinite series is an open problem.

Differentiation

Let `'A'` be `dA`. The rules of differentiation in the James calculus expressed as are:

Name	Derivative	Interpretation
constant	'c' = void	dc = 0
x/dx	'x' = ()	dx = 1
exponent	'(A)' = (A ['A'])	de^A = e^A dA
logarithm	'[A]' = (<[A]>['A'])	dln A = 1/A dA
inverse	'<A>' = <'A'>	d-A = -dA
space	'A B' = 'A' 'B'	d(A+B) = dA + dB

These rules are syntactically very regular, and thus useful for algorithmic computation.

Proof of the Chain Rule of Differential Calculus

d(AB) = B dA + A dB	
'([A][B])'	
([A][B]['[A][B]'])	exponent
([A][B]['[A]''[B]'])	space
([A][B][(<[A]>['A']) (<[B]>['B'])])	logarithm
([A][B][(<[A]>['A'])]) ([A][B][(<[B]>['B'])])	distribution
([A][B] <[A]>['A']) ([A][B] <[B]>['B'])	involution
([B]['A']) ([A]['B'])	inversion

Using James differentiation, *example 1*:

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]))]) = ae^(ax)
```

Example 2:

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 <()>]))]) = nx^(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'>
```