Spencer-Brown Numbers

William Bricken

January 2001 with minor edits and formatting changes, September 2012

Spencer-Brown published *Laws of Form* in 1969, the first and seminal text on iconic mathematics. In it he demonstrated an application of iconic techniques to logic, developing a void-based boundary logic. In private correspondence he has shared his unpublished version of integers based on iconic techniques. This system provided the initial example of how to think about numerical computation in void-based and boundary terms.

Spencer-Brown Arithmetic (Parenthesis Version)

In Spencer-Brown arithmetic, each number is both an *object* and an *operator*. Operations are very easy, however the representation of integers is clumsy.

Integers	***Stroke arithmetic in a container***
0	( )
1	(())
2	(()())
3	(()()())

Reduction Rules

Two transformations of form provide all computational, standardization, and evaluation techniques.

((A)) = A	**Involution**
(()()) A = ((A)(A))	**Distribution**

From a group theory perspective, commutativity and associativity are assumed to be implicit in the form. From an iconic perspective, commutativity and associativity are irrelevant concepts.

Operations

In iconic systems, an operation consists of forming a new configuration. All computation is expressed as a reduction of the new configuration.

Addition is placing each form to be added in a container and then placing the entire collection in a container.

addition	A+B	((A)(B))

Multiplication is placing forms together in the outermost space.

multiplication	A*B	A B

Power (exponentiation) is placing the base in a container, then placing that and the exponent into another container.

power	A^B	`((A) B )`

The containers themselves do not have an interpretation in conventional numerics. They maintain structural relations between spaces; the structures do have meaning as both numerical objects and as operations. Alternatively, sharing space can be interpreted as addition, while double-bounding becomes multiplication. This is the strategy of both Kauffman numbers and James numbers; only Spencer-Brown's representational approach is presented here.

Confounding Objects and Operations

The form of an integer is also the form of addition. This is characteristic of iconic systems which confound *containment as a function* with *container as an object*. That is

```
0 = ( )       = 0
1 = (())      =  +0
2 = (()())    = 0+0
3 = (()()()) = 0+0+0
```

Spencer-Brown integers count the cardinality of nothings added together. An integer is the cardinality of partitions of an empty set. This is a link between Conway (set-based) systems and other iconic systems.

Since we can also interpret a form functionally, a "number" is the cardinality of an application of the distribution rule.

`(( )( )( )) A`	`3*A`	*implicitly*
`((A)(A)(A))`	`A+A+A`	*explicitly*

Addition is placing the forms to be added in separate spaces, two levels deep. Multiplication is placing forms in the same space at the zero, or outermost, level. The capital letter `A` means than any form (not just numbers) can participate in these transformations. The absence of a form, the *void*, is excluded from this system.

By interpreting an empty container, `( )`, as zero rather than as a unit, a binary system is created with bases `( )` and `(( ))`. This system maps to propositional logic and to finite set theory. The idempotent equation that converts integers into logic is:

`( )( ) = ( )`	**Call**	***idempotency***

Computation

Compound forms are constructed via algebraic substitution. These forms reduce using the two reduction rules. In essence, the container around a number-object cancels with a void container inside a number/operator, producing a new number form through the involution transform.

Examples (square brackets are solely for highlighting, they are identical to parentheses):

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

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


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

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


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

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

Notice that during reduction, the forms do not have an interpretation, they are in a sense notational jottings. In another sense though, the numbers themselves arrange into new structural configurations during an operation. The new structures are unstable, they reduce by the given rules into forms that are stable. Stable forms represent numbers.

Void Transforms

Consider the transformation rules applied to nothing, to the void.

`(( ))  =` **void involution**

`(()()) = (()())` **void distribution**

The first rule tells us that the implicit background value, the value of the void, is 1, not 0. By defining space to be multiplicative, space takes on the value of the multiplicative unit, such that

```
1*A = A

1*void = void
```

The second rule says that integers are stable whenever they are indicating the cardinality of void distribution. In meta-language,

```
(()()()) void = ((void)(void)(void))
```

This meta-language is fundamentally *incorrect*, since the void cannot be replicated. More accurately:

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

Syntactic Inconsistency

Combining forms in space results in multiplication. However, this does not apply to forms combined in the space inside a container. Consider a hybrid reading of the number three:

```
(()()())  ≠  ( 1*1*1 )
```

Basically, the outer container has no interpretation as an operation, while reading the inner space as multiplication undermines the integrity of the system. Algebraically, the space inside the outer container should still be multiplicative, just like any space in this system. That is to say

```
((a)(b))              a+b

((a)(b))              contain[(a) times (b)]
```

Thus, Spencer-Brown numbers have an *ambiguity*: it is unclear when (in what space) multiplication applies. Consider multiplying one times one:

```
1*1                (()) (())
```

Void involution would convert this result to the undefined void as shown below, so we must prohibit void involution.

```
   (()) (())
(( (()) (()) ))                       involution
(    0 + 0    )                       hybrid
(() (()())    )                       distribution
(0 +   2      )                       hybrid
(   (()())    )                       add 0
()() (  (  )  )                       distribution
 0 * 0 * 1       = 0                  interpret
```

This problem also shows up when evaluating the power relationship. Consider multiplying by zero:

```
0*A                ( ) A
```

We need a new rule to be able to reduce this to 0. Spencer-Brown suggests that the configuration is an instruction to multiply, and multiplication takes place by distribution. Therefore, `A` must distribute into `( )`. Since there is nothing to distribute into, the `A` is absorbed by the void.

```
( ) A = ( )
```

It is generally agreed that this is not a strong argument, although it is very similar to Conway's handling of void partitions when testing for a number. Still, the problem of form ambiguity recurs for the representation of powers. Revisiting the example:

```
2^3 = 8           (()()) ^ (()()()) = (()()()()()()()())

            [[ (()()) ] (()()()) ]                    power
            [   ()()    (()()()) ]                    involution
```

The problem is in the next distribution step:

```
            [( (()()) (()()) (()()) )]                distribute ()() into 3
```

Forms are independent in space, therefore it is a choice to distribute one or two unit objects. The system should allow distribution of any set of forms that are distinct in space. Trying this, we see that it results in an inconsistent form:

```
            [( ) ( (()) (()) (()) )]                  distribute () into 3
            [ 0         1+1+1     ]                   hybrid
```

We might then attempt to reduce this form by further distribution:

```
            [( ) ( (((()) (()))) )]                   distribute 1*1 into 1
            [( ) (   (()) (())   )]                   involution
            [( ) (    ((()))     )]                   distribute 1 into 1
            [( )       ((()))     ]                   involution
            [( )         ()       ]                   involution
                        2                             interpret
```

Interpreting the inner forms as numbers is illegal, so this particular problem can be defined away. However, stepping back prior to the distribution, we must introduce a restriction that blocks generating the illegal form. This is necessary for representing the algebra, but it effectively undermines the simplicity of the original approach.

Spencer-Brown uses a colon for bracketing:

```
            [ : ()()   (()()()) ]                     involution
```

This means to distribute *en masse*. The same distinction is needed to differentiate between several power forms.

Spencer-Brown form	***Conventional semantics***
`((a)())`	`a + 0`
`((a)():)`	`a ^ 0`
`(()())`	`2`
`(()():) = (((()))():)`	`1 ^ 0`

The ambiguity in effect introduces a new axiom, called Dominion, which is explored in the James algebra. Unusually, the colon is a spatial operator, converting its container into an absorber, similar to the way that infinity works in conventional number systems.