A Simple Space

William Bricken

December 1986

Contents

Abstract

The lines in which we write mathematical symbols impose constraints upon mathematical thinking. A simple space of representation is proposed that does not enforce the linear concepts of associativity, commutativity, duplicity of representation, and binary scope. The properties of this simple space are discussed in the foundational case when the space is empty and in the self-referential case when the space contains only representations of itself. Concepts that evolve from this discussion include representational incompleteness, functional spaces, boundary objects, representational unity of object and process, and two kinds of void. The implications of a representational space without linear properties are explored for propositional calculus. A graph notation is proposed as a simplification of the traditional linear notation for logic.

I wish to discuss the space in which we record symbols. The central idea is that mathematical operations are more clearly expressed in a space which imposes fewer constraints upon the tokens it contains. Tokens placed in lines encourage sequential metaphors, while tokens spread over a plane encourage parallel metaphors. A *simple space* of representation frees objects within it from irrelevent syntactic constraints.

My primary motivation for writing this paper is to attempt to elucidate a different way of thinking about mathematical symbolism. This approach has been personally helpful, both in my understanding of mathematical systems and in my implementation of parallel inference engines for Artificial Intelligence applications. After some context setting in Section 1, I will construct, from scratch, a simple space of representation for mathematical expressions (Section 2). I will then show how this simple space can be used to simplify propositional logic (Section 3). My main concern is with *redundancy.* The elegance of the representation of logic, for instance, is critical to the efficiency of automated theorem proving and of inference engines in expert systems. Unnecessary and redundant rules slow computation. The space I will describe is particularly powerful computationally because it supports parallelism in deduction. It is particularly powerful psychologically because it provides a clear way of thinking about representation.

1 Context

The sequential space of linear notation is highly structured. This structure imposes redundancy upon the tokens recorded in the linear space. A planar alternative is considered in the contexts of integer addition, elementary logic, and elementary set theory. The notion of a functional space is introduced.

1.1 Representational Dimension

The space in which tokens are recorded is called the *representational space*. The assumptions embedded in a representational space deeply influence the expressive power of the tokens lying in that space. By shifting our analytic attention from the symbolic figures to the representational ground, I hope to clarify the basis upon which our understanding of syntactic forms is constructed. To further this examination, I wish to restrict the discussion to elementary mathematical concepts and to a minimal set of tokens.

The most common form of recorded symbols is written language. The space in which words are recorded is one-dimensional: we string words out on an invisible line and read these words by scanning the line from left to right. Implicit conventions permit our focus of attention to roll off the end of one line and start again at the beginning of the next lower line without interruption. In this way, we organize a two-dimensional space so that it incorporates a long one-dimensional sequence. We similarly structure multiple pages in three dimensions into a yet longer string of words.

An interesting deviation from this regime is when the text includes a picture or illustration. We set such non-linear spaces aside in a frame, recognizing that an entirely different form of information is contained therein.

Blackboards, in contrast to lined paper, tend to accumulate diagrams and doodles. When given an unstructured space, we tend to fill all of its dimensions with information. *Each dimension sustains a different kind of information.* Thus, we fill three spatial dimensions with sculpture; two spatial and one time dimension with films and television; three spatial dimensions and one time dimension with actions and behaviors.

I wish to limit the scope of this discussion to tokens written on a flat surface, in two spatial dimensions. In particular, I wish to examine the mathematical ideas that arise when tokens are arranged in sequence in order to discover which of these ideas are rendered irrelevant when tokens are spread across the two dimensions available on the page.

1.2 Limitations of a Linear Notation

We are all familiar with conventional mathematical notation. Designated tokens are assigned a meaning. Some tokens represent static objects, others represent operators that relate tokens. Object-tokens and operator-tokens are strung on a line, just like words. For example, in elementary arithmetic we write $5 + 3$. That this combination of tokens identifies a specific numerical object, 8, can be represented by further stringing of tokens: $5 + 3 = 8$. In this case, we might not know which operator ($+$ or $=$) to apply first, so we invent precedence rules or add grouping tokens such as the parenthesis to keep our sequence of operations unambiguous.

The example of integer addition identifies several limitations on the scope of mathematical representation in a linear space. Specifically:

Binary Scope: Lining up tokens in a linear representational space permits us to associate by proximity only two object-tokens with each operator-token. If we wish to add three integers, $x + y + z$, we must include a duplicate of the operator-token. The meaning we wish to assign to the plus-token, however, is not restricted to the addition of only two numbers at a time. Our convention of linearity forces a focus on binary relations. One alternative to the binary scope of the linear plus-token is column addition. We stack numbers in another dimension, and apply the addition operator to the column. Another alternative is offered by the programming language LISP, which permits multiple object-tokens under a single operator-token by writing, for example, `(+ x y z)`. To get this scheme to work, we must also add a set of delimiters (parentheses) to the expression, to tell us where the addition operator stops acting on numerical tokens.

Duplication of Tokens: Binary scope requires us to write duplicate operator-tokens for what is essentially the same operation. Similarily, when the same object is operated upon more than once, as in the logical expression $a \vee a$, the object-token must be recorded twice in order to appear on each side of the operator. Duplication of tokens is an act of overt redundancy, and a violation of Occam's razor. A notation that reduces this redundancy would be more efficient computationally and more elegant conceptually.

Sequential Precedence of Operators: Recording binary operators in a line, such as in $x+y*z$, creates a problem of precedence. Which operator should act first? Operators must be explicitly grouped in the order of their intended application. To remove its inherent ambiguity, the example can be written as $x + (y * z)$, where nested operations are understood to take precedence over those in the outer context. The deeper problem is that representation in linear space forces sequential processing. We must approach operators one at a time because one-dimensional representations must be processed in a step-wise sequence. We can see all the tokens at once, from our privileged perspective outside of the page, but the implementation of the operations is intended to occur sequentially. An alternative to the sequential bottleneck is to permit parallel operations. Parallel implementations of mathematical operations are just beginning to be understood.[1] In the example of column addition, we might imagine each column being added simultaneously, with carried integers being added to the result.

Sequential Ordering of Objects: In linear notations, we are forced to nominate a first and a second object-token for each binary operation. We must then invoke a rule that permits order not to matter when the operator is commutative. For example, the commutativity of addition tells us that $x + y$ is the same as $y+x$, even though the representation of each is different. An alternative

[1]Parallelism is quite uncomfortable to our minds; we are so accustomed to the linear one-token-at-a-time regime of spoken and written language that multiple concurrent processes find a comfortable metaphor only when spread over different minds.

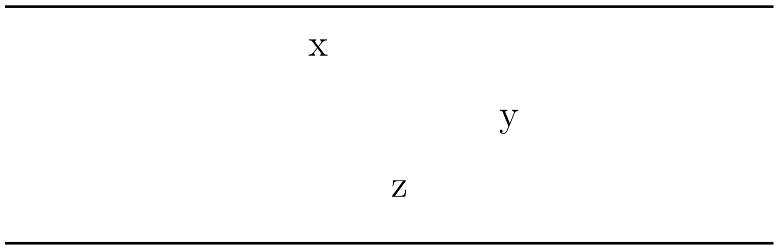


Figure 1: The Addition Plane with Three Elements

is to invoke rules only when order does matter. Linear notation creates the image that we begin with the left-most token and combine it with the right-most token using the operator-token. This regime, however, is merely an implementation detail. A parallel implementation can be imagined in which we begin with *both* object-tokens and operate upon them simultaneously.

1.3 Planar Notation

Binary scope, representational duplicity, associativity, and commutativity are artifacts of a linear notation. Some mathematical operations, such as subtraction, rely on these characteristics. Other operations, such as addition, do not. We recognize the lack of applicability of linear features by forming rules that permit rearrangement. But this method of granting permission is itself only a concession to the linear space of representation. The traditional representations of addition confuse the meaning of addition with the machinery that achieves the meaning. Imagine instead placing tokens to be added in a planar space (Figure 1). The *addition plane* does not specify ordering or grouping of tokens. The planar representation is intended to capture the idea that an operation can apply equally to all tokens in the space. Linear structure can be added later for operators that are *not* commutative or associative.

What we really want is a representation that encapsulates the intention of an operation without imposing irrelevant baggage. A space that does not require transformation rules to express our intentions will be called *simple*. A simple space supports simple forms that may be taken at face value. Acquiring an understanding of the simple space for specific operators requires the generalization of several mathematical principles. It also results in new conceptual skills.

1.4 A Simple Logical Space

I wish to restrict the discussion again, this time to the domain of elementary logic, to the common logical operators of NOT, AND, and OR and the logical constants TRUE and FALSE. Logic is an excellent system in which to develop the skills associated with non-linear representation. It is a model of rational thought processes. It is simple and intuitive. It also suffers extensively from redundancy and irrelevancy introduced by a linear space of representation. For example, the operator OR means conceptually that at least one of several alternatives is TRUE. We write the expression (a OR b OR c OR d), meaning that the entire expression is TRUE when one of the variables is TRUE. The location of the true variable is irrelevant, yet sequential processing forces an unnatural implementation: we look first at a, then, if necessary, at b, then at c, and so on. What we should do is something like `(parallel-or a b c d)`, which means: examine all the options at the same time and if any one of them is TRUE then the expression is TRUE. PARALLEL-OR is difficult to achieve by linear symbol processing, yet it is easy for our visual system. When the truth-values of each object are furnished, as in `(parallel-or f f t f)`, we quickly assess the situation visually, in parallel, to determine the result.

My motivation is to identify the minimal characteristics of a space of representation for logic. A more efficient representation implies a more efficient deductive procedure. By expressing logic in a specifically tailored simple space, I hope to come to a deeper understanding of what logic is, and which artifacts of our traditional linear notation clutter logic with representational irrelevancies. This program was first carried out by G. Spencer-Brown.[2]

A linear notation imposes irrelevant restrictions on the representation and associated meaning of logical expressions. In the formalization of logical semantics, we convert our intentions into a symbol system that permits syntactic transformations. After transformation, we re-interpret the syntactic result semantically. In this process, it is desirable to avoid syntactic systems that add more than what is intended. Traditionally logical OR, for example, is associative, commutative, and idempotent, with the constant FALSE as a right and left identity. Instead of accepting this linear definition, I will suggest that the operator OR is insensitive to order, grouping, duplicity, and identities. Further, it is the *representational space* itself, and not the operator, which must be intolerant of irrelevancies. Representing logical OR in a simple logical space simplifies the representation of logic and the process of deduction. The construction of a simple logical space is described in Section 3.2.

The characteristics of a simple logical space follow:

- **Multiple Scope:** The simple logical space supports the existence of any number of tokens, including none.

- **Intolerance of Duplicity:** Each token in the space is unique. Duplicates of tokens are not supported.

- **Intolerance of Ordering:** The space will not support an order between tokens. Tokens cannot be sequenced. All are addressed simultaneously, in parallel.

- **Intolerance of Grouping:** The space will not support groups of tokens within it. All tokens are acted upon equally and concurrently.

[2]G. Spencer-Brown, *Laws of Form*, George Allen and Unwin Ltd., London: 1969.

The process of moving from a linear to a simple space of representation is one of shifting perspective. Traditionally, a linear space is assumed as the basis of representation. The shift of perspective is to see the space of representation as not supporting sequence. Sequencing rules are secondary, to be added later when specific operators require them.

1.5 Set Notation

The image that emerges from the specification of a simple logical space is a representational space in which unique tokens float, unconnected and unstrung. The only communal property they share is that they jointly occupy the same space. The mathematical concept of a *set* achieves these objectives. Sets can contain an arbitrary number of members and they do not support duplicity or ordering.

To represent a set, the traditional notation places a boundary around the tokens that represent the set members, $\{\texttt{a, b, c, d}\}$. Implicitly, a set with duplicates[3] such as $\{\texttt{a, a, a}\}$, is transcribed into a set with no duplicates, $\{\texttt{a}\}$. As an example, the set operation of UNION might be expressed as

$$\{\texttt{a, b, c}\} \cup \{\texttt{c, d}\} = \{\texttt{a, b, c, d}\}$$

Here, the token `c` occurs twice on the left-side of the equality because of physically separate references. The result on the right-hand-side coalesces the two references into one.

A difficulty with linear set notation is the prevalence of the comma-token, intended as a reminder that the members are not really in order, they are just written that way. Another difficulty is that the set operator, UNION in the example, is external to the set itself, in a different kind of representational space.

The need to include comma-tokens might be alleviated by using both dimensions of the page. Tokens can float in this non-linear space, not grouped or related in any way other than by their common membership in the same space. To support object-tokens we can set aside a *simple set-space* within which the member-tokens are represented. This idea is illustrated in Figure 2.

An alternative to recording operator-tokens in a separate space is to eliminate the need for an operator-token altogether. Basically this is achieved by assigning the space which contains tokens the power to operate on them. The space itself can be attributed a functionality.

1.6 Functional Spaces

A *functional space* operates upon the object-tokens within it. The boundary-token that differentiates the interior of the space from the exterior can serve to identify the function of the space within. In the set example, the curly brackets can be seen as containing an active space which maintains the UNION of its

[3]This is actually a contradiction in terms. If it contains duplicates, it is a *bag*, not a set.

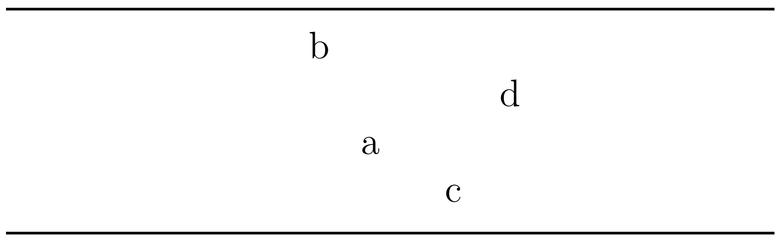


Figure 2: Members in a Simple Set-Space

contents. Boundaries partition the implementation of the function of a space from the result returned by that operation. The boundary and its contents can be interpreted as an object-token representing the result of the operation of a functional space. The tokens embedded within a functional space refer to the pre-implementation details, the initialization of the function of the space.

Consider an example of the simple addition space expressed in LISP notation: outside, we can ignore the detail of the numerical tokens within the space. The bounded addition space represents the numerical object *sum*. The boundary itself hides the details of the implementation and of the arguments of the space. Viewing the addition space enclosed by the boundary-token, we see the integers to be added embedded as object-tokens in the space. Tokens in the interior of a functional space record the input conditions of that space. The space itself converts these inputs into a result.

I have suggested that spaces themselves can operate. This change of notational perspective provides a significant advantage: recording the problem in an operational space is sufficient to generate the answer. In the addition example, it is like entering the numbers into a calculator that is pre-set to add. When the entry process is finished, the calculator shows the result. In the set union example, placing the tokens from different sets into a common simple set-space yields the union. The space itself will simply not support the entry of a duplicate token.

The function of a space and the type of tokens within that space are mutually dependent. The type of object-tokens within a space constrains the type of operations a space can embody. In the addition example, tokens of type *integer* are summed by the addition space. If the space were a logical AND space, integer-tokens would not resolve into a single token representing the sum. Thus, the kind of token and the kind of operation are intertwined; the function of a space and the type of its tokens are mutually definable.

Functional spaces introduce the powerful representational idea of object-operator equivalence. Bounded expressions can represent both an object and an operation performed on contained objects. In the example of set union, the curly brackets specify both a set object and the operation of the UNION of its contents. In the addition example, bounded integers can represent both the sum object and the operation of adding the contents of the addition space. This idea is developed from first principles in the following section.

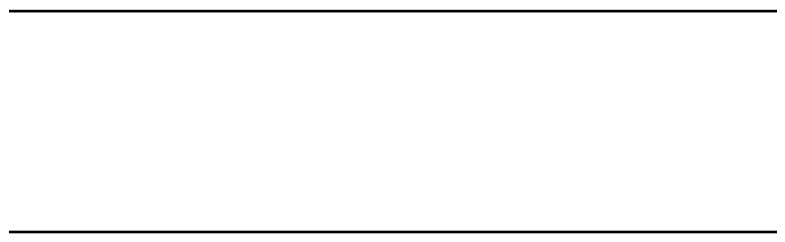


Figure 3: An Empty Simple Space

2 Foundation

The simplest space, which is empty, is described from an intuitive perspective. The *simple token* is a representation of the boundary of the simple empty space. Two rules are constructed to describe the desirable characteristics of simple tokens. When the simple token is interpreted extrinsically as an object, duplicity is not permitted. When it is interpreted intrinsically as a process, regression is not permitted.

2.1 The Empty Simple Space

I wish to limit the kind of tokens we permit into the simple space, in order to limit the kind of operations we might expect of it. The task then is to choose an appropriately simple type of token to put into the simple space. Since the simplest space is empty, I will first examine this space-without-tokens. The introduction of tokens into the empty space cancels its emptiness. Figure 3 delineates an empty simple space.

The simple space that is devoid of tokens is special in that it is difficult to locate without its boundaries. We are forced to recognize the need for a boundary, if only to contain nothing. The boundary is also handy for separating other types of space that might occupy the same page out of convenience.

What we see as the empty space depends on our point of view. If we focus on the inside, we see nothing. If we focus on the outside, we see the boundary that frames nothing. The interior of an *empty* simple space does not support its own identification. We must view it from the outside in order to identify that it exists at all.

In set theory, the notation for the empty set, curly brackets { }, is a frame around nothing.[4] In the case of addition, an empty additive space, ( ) represents zero. In general, we can concentrate on the *notation*, which is the frame, from an external perspective, or we can concentrate on the *concept*, which is nothing, from a perspective internal to the frame.

[4]Kauffman and Varela, Form Dynamics, *Journal of Social and Biological Structures*, 1980, **3**, p. 172

The key to understanding the simple space is to understand our interaction with it. I have suggested that the description of the space differs depending upon where we focus. Moving from the details of the inside to the result on the outside means that our perspective determines whether we see the contents of the space or the result of the function of the space. *We initiate functionality by shifting perspectives.*

What then is the functionality of an empty simple space? What happens when we shift our perspective from the inside to the outside of an empty simple space? We perceive the boundary. The fundamental function of the empty space is to make us aware of its existence. We move from nothing to a boundary which delineates nothing from everything else. The boundary is the objective aspect of the empty simple space. The representation of this boundary is the most *simple token*. It is discussed in Section 2.3.

Indirectly, nothingness can be defined by its container. As an analogy, we can talk about being out of coffee because we have an empty cup. Without the cup, we lose the context to make sense of the idea of being out of coffee. Similarly, without the boundary, we cannot make sense of an empty space.

2.2 The Two Voids

I am drawing attention to the fact that there are two kinds of emptiness. We can observe a *bounded emptiness*, such as an empty cup or an empty space, because the boundary provides a place from which we can observe. We can safely stand outside, without destroying the emptiness, and point to the emptiness within, and observe the nothing. *Total emptiness*, however, does not support our observation of it. If we were able to observe it, we would be placing our focus in it, rendering it non-empty and no longer what we wish it to be. It is no longer total emptiness solely because our attention fills it.

There are two kinds of void. The Absolute Void is simply not available to contemplation. It is Emptiness. The Relative Void is indirectly accessable, at least to thought. It is Empty Space.[5] We can use the Relative Void as a space of representation by putting tokens in it. It is impossible, however, to place tokens in an Absolute Void. The only route from the absolute to the relative is through non-symbolic conscious choice.

This concept, that there are two voids, forms the basis of a more general form of mathematics. The distinction between empty and emptiness is all that is necessary to construct traditional mathematics. The empty simple space is the foundation upon which a deeper understanding of the process of creating symbols can be built. In its essence, the conversion from emptiness to empty space is the process of picking up a piece of paper with the intent to record tokens. It is an acknowledgement of a sentience that is creating symbols. As a starting point, the genesis of an empty space from emptiness incorporates the mathematician as the fundamental mathematical construct. Next, I will attempt to demonstrate that this distinction is indeed informative.

[5]The word *emptiness* is a noun, it refers to the Absolute Void, a state with no properties. The word *empty* is an adjective, it identifies the only property of the Relative Void.

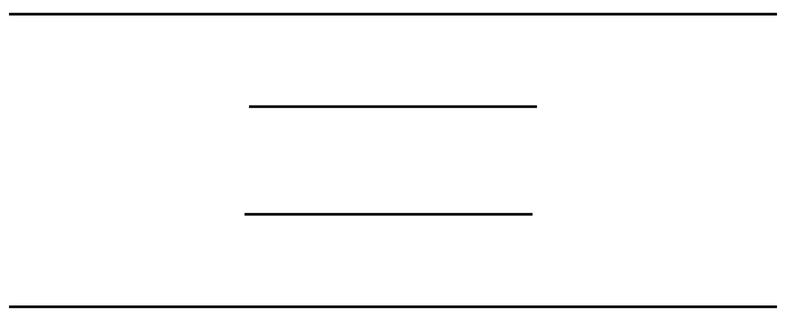


Figure 4: A Simple Space that Contains a Simple Token

2.3 The Simple Token

The *simple token* is the representation of the boundary of an empty simple space. The space is functional, a process; the boundary is declarative, an object. The simple token represents, or objectifies, the single process of the empty simple space, which is to bring its existence to our attention. The process of moving our focus of attention from the inside, where this is nothing, to the outside, where we see an empty container, constructs the simple token in the form of a boundary. The boundary-token is the representation of the container as seen from the outside.

When we shift perspectives, we discover a boundary. We can represent that discovery by drawing the simple boundary token. What is discoverable when we place the boundary token into the empty space (Figure 4)?

By construction, the boundary token represents an act of our own volition, that of shifting perspectives. By placing a boundary token into the empty space, we examine, symbolically, our shift of perspective. Self-observation is at the core of simple symbols. After a brief diversion into linear notations for boundaries, I will discuss the symbolic consequences of simple self-observation.

2.4 Delimiting Tokens

We can adopt a convention that permits limited planar representations in a single dimension. The *parenthesis* permits a planar representational space to be encapsulated in a line. Parentheses, and other delimiting tokens such as brackets, braces, and quotation marks, are by construction unusual. They come in *pairs*; it takes two of them to express the elementary idea of containment, or bounding. Parentheses put non-linear boundaries on strings of linear tokens. There are two of them because the parenthesis actually exists in two dimensions, around token strings. This means that a parenthesis can be used to express the boundary we see when we frame an empty space. Parentheses relativize the void. Therefore, I will use an empty parenthesis, `( )`, to represent the boundary between nothing, on the inside, and our observation of nothing, from the outside. Parentheses used in this manner will be called *parens*.

2.5 Extension

The *empty parens*, `( )`, indicates a choice of perspective. It represents our initialization of symbolic activity in the act of setting aside a representational space on the inside of its boundary. We can place this simple token into the empty simple space, forming a *double parens*: `(( ))`. Once having taken this step, other possible configurations arise. For instance, we might place yet another simple token in the simple space, forming the expression `(( )( ))`. Or we might insert many tokens, forming longer expressions, such as `(( )( )( )( )( ))`.

It is at this point that we must interpret these complex expressions, in order to avoid the confusion of an unbounded proliferation of forms. One possible interpretation is to say that we are *counting*. The number of simple tokens in a space defines an integer. Although this is valuable (we invent integer arithmetic), it is prematurely complex. I wish, instead, to return to an initial objective for the simple space, that it does not support duplicity. Two identical tokens should mean the same as one of them. Since we have only one kind of token, all duplicates are necessarily the same. We cannot tell the difference between copies of the simple token without adding structure to the simple space. If we were to add ordering structure, for instance, we could talk about the first token and the second token and so on. But this I wish to avoid, since the intent of the simple space is to minimize supported structures.

Like duplicate members of a set, duplicate simple tokens are the same as a single simple token. Symbolically,

$$\texttt{(( )( ))} \Longrightarrow \texttt{(( ))}$$

The benefit of not entertaining duplicity is that we are forced back to understanding what it means to fill a space with a representation of our choice of perspective: `(( ))`.

2.6 Intension

In this simple symbolic world, another complexity arises. By the same process of insertion of a simple token into an empty space, `(( ))`, we can insert the complex token of the double parens, `((( )))`. Similar to the case of duplicity, we must furnish an interpretation in order to avoid the confusion of unbounded self-observation.[6] What is the meaning of expressions such as `(((((( ))))))`? Like the unboundedness of counting upwards, there is an unboundedness of referring inward. Not only must we resolve the infinite extension, to remain simple we must resolve the infinite introspection.

G. Spencer-Brown's resolution to unbounded self-observation is this: an empty parens indicates an empty representational space; a double parens withdraws this indication and returns to non-representation. The symbolic value of the double parens is non-existence. Symbolically,

$$\texttt{(( ))} \Longrightarrow$$

[6]Such confusion occurs in the infinite regress of the homunculus.

The process of bounding is invertable. An empty parens is created by bounding emptiness. A double parens is created by bounding this bound. The effect of the double boundary is to return to the original unbounded emptiness. Thus, the double parens represents shifting perspective and then shifting back.

2.7 The Leap of Faith

Symbolizing simple self-reference has the effect of cancelling the original arrangement to enter the symbolic arena. This insight may require a leap of faith. I will attempt one other explanation of this result and then leave the subject, treating the equivalence between self-observation and non-representation as an axiom.

Spencer-Brown's *calling* is what I have referred to as intolerance of duplicity. This is easy to understand when we treat the empty parens, `( )`, as an object-token. Identical objects add nothing to our knowledge; therefore we need only one token to refer to them/it. Symbolically

$$(\ )\ (\ ) \Longrightarrow (\ )$$

We do not need two tokens to represent the one idea of creating a space for symbols.

Spencer-Brown's *crossing* I have referred to as self-observation. This is easiest to understand not as an object but as a process. The process of *indicating* our symbolic intention by creating a space is itself symbolized by the empty parens. The process of *referring to* our symbolic creation is symbolized by the double parens. Just as indication of nothing denies that nothing, reference to the indication of nothing denies that reference. We return to the non-symbolic source, not represented and not on the page. Symbolically,

$$((\ )) \Longrightarrow$$

3 Application

The simple space and the simple token, when interpreted as logical structures, simplify both the concepts and the representation of traditional logic by removing the irrelevant features imposed by a linear notation. Elementary logic can be seen as unnecessarily complex; its foundation can be traced back to the two forms of void and to the one act of shifting perspective. The introduction of variable-tokens permits a generalized description of the rules governing simple spaces. A network representation for logic is proposed.

3.1 Mathematical Notation

I have mixed cognitive psychology with mathematics. We do not need to interpret the empty parens as a shift of perspective and a double parens as a self-observation. Purely mathematically, the simple space can be defined as *idempotent*, which means that duplication does not change its contents. Functionally, the double parens can be defined as the replication of an invertable operation. The application of the second operator cancels the effect of the first.

In traditional mathematical notation, let **F** be the function of the simple space. Let **m** be the boundary-object indicated by the empty parens. The constant **m** is generated by applying **F** to no arguments. Symbolically,

$$F[\ \ ] = m$$

By composition,

$$F[m] = F[F[\ \ ]]$$

Neither of these functional expressions have an expressable value. **F**, then, can be seen as an *existence operator.* Odd compositions of **F** express the *creation operation*; even compositions express the *destruction operation.*

3.2 Logic as an Interpretation

A natural interpretation of parens expressions as propositional logic is readily developed from an intuitive basis. The rather abstract ideas of emptiness, empty space, and self-observation form a simple basis for formal symbolic logic.

We have seen that the empty parens can be viewed from two perspectives, one of which is represented. Let us choose to associate representational existence with truth. If it exists, it is TRUE. Let the empty parens, ( ), as an object-token, be the representation of the truth-value TRUE. This choice then prescribes the rest of the interpretation of logic as parens expressions. For instance, if it does not exist, it is FALSE.

The direct path to understanding the non-representation of the truth-value FALSE is to see it as a dual perspective. Falsity is the alternative perspective to truth. The dual of an empty parens is identified by shifting from its outside to its inside. The concept FALSE, then, is represented by the contents of the empty parens. Since there are no tokens on the inside of ( ), FALSE has no direct representation. This *representational incompleteness* is a source of representational power: only one of two opposite concepts need be indicated by a token. The non-representation of FALSE is a direct consequence of associating the representation of TRUE with symbolic existence.

Seeing the polarity of the two truth-values as alternative perspectives is the same as specifying the functional interpretation of the empty parens. When interpreted for logic, the boundary-token *negates* its contents in the same way that truth negates falsity. Symbolically,

$$\text{TRUE} = \neg\text{FALSE}$$

Expressed as parens expressions,

$$(\ \ ) = (\ \ )$$

where the parens on the left is an object, and the parens on the right is a process.

The parens has an interpretation both as an object and as an operator. It is both the constant TRUE and the operator NOT. Since we may need to refer to the concept of FALSE by a label, we can use the negating power of the parens to achieve this indirectly. An empty parens is TRUE; a double parens is the negation of truth. That is, the double parens is FALSE. Symbolically,

$$\text{FALSE} = \neg\text{TRUE}$$

In parens,

$$= \texttt{(( ))}$$

The double parens can also be interpreted as the composition of two negations:

$$\neg\neg\text{FALSE} = \text{FALSE}$$

Next, we associate a logical operator with the operation of the simple space itself. To see which operator, consider the expression `(( ) (( )))`. The outer boundary contains a simple space which is host to two expressions, `( )` and `(( ))`. We know the logical meaning of each of the two expressions. The first is TRUE and the second is FALSE. We also know that the FALSE expression vanishes, leaving only the TRUE expression in the space. Since we wish to construct an interpretation for the simple space that is not confusing, we will want to maintain the value, or meaning, of expressions when they are simplified. So we ask, what logical operator combines TRUE and FALSE to yield TRUE. The answer is the operator OR. The simple space operates as OR when the tokens within it are interpreted as logical values. In the example `(( ) (( )))`, the outer boundary negates the disjunction of the two inner expressions. One possible way to interpret this expression in logical tokens, then, is

$$\neg(\text{TRUE} \vee \text{FALSE})$$

The logical value of such an expression is FALSE. This is supported by the observation that the parens expression vanishes when simplified:

$$\texttt{(( ) (( )))} \Longrightarrow \texttt{(( )} \qquad \texttt{)} \Longrightarrow$$

Thus, the parens represents a generalized logical operator. It is applied to all of its contents, which may be of any number. When there are no contents, the 0-ary operation generates the constant TRUE. When there is a single argument, the parens represents NOT. When there are two or more arguments, the parens represents a generalized NOR with arbitrary arity.

The relations between expressions composed of parens and the connectives of logic are summarized in Table 1. The capital letters are variables which represent arbitrary expressions. This table illustrates that the map between traditional logical notation and parens expressions is *many-to-one*. Thus the parens notation simplifies the representation of logical expressions. Where a

TRUE	$\Longrightarrow$	`( )`
FALSE	$\Longrightarrow$	`(( ))`
$A \vee B$	$\Longrightarrow$	`A B`
$\neg A$	$\Longrightarrow$	`(A)`
$A \wedge B$	$\Longrightarrow$	`((A)(B))`
$A \rightarrow B$	$\Longrightarrow$	`(A) B`
IF A THEN B ELSE C	$\Longrightarrow$	`(((A) B) (A C))`
$A \equiv B$	$\Longrightarrow$	`(((A) B)((B) A))`

Table 1: The Map from Propositional Logic to Parens Expressions

minimum of three logical tokens are necessary (say, FALSE, NOT and OR), only one bounding token is necessary. The concept FALSE is absorbed into the simple logical space, while negation and disjunction are joined into the single concept of an arbitrary arity NOR.

As an example of the representational condensation that parens provides for logic, consider the expression

`( (( )) (( )( )) )`

There are many ways this expression can be interpreted for logic. We see the outer mark negating two inner expressions. The inner expressions are joined by OR, the function of the simple space. The first expression in the linear representation is the token for FALSE, but it is also the expression for NOT TRUE. The second expression is the negation of two TRUEs joined by OR. Thus the whole expression stands for

$$\neg\neg\text{TRUE} \vee \neg\text{TRUE} \vee \text{TRUE}$$

But it also stands for

$$\neg\text{FALSE} \vee \neg\text{TRUE} \vee \text{TRUE}$$

And it also stands for

$$\text{TRUE} \wedge (\text{TRUE} \vee \text{TRUE})$$

This latter reading is obtained by regarding the outer-most parens structure as that of the operator AND: `(( )( ))`. We can insert the non-represented logical concept FALSE at every space in the example. These spaces are virtually everywhere. Thus, another interpretation is

$$\neg\neg\neg\text{FALSE} \vee (\text{FALSE} \wedge \text{FALSE})$$

The parens expression, then, reduces many logical expressions to the same form. The key to the multiple interpretation of parens expressions is that the parens is both object and operator. The key to the representational simplicity is that the simple space is both object-without-token and operator-without-token. The parens language is a canonical reduction of logical notation. It reduces redundancy by reducing the possible forms of a logical expression. The simple space has provided us with a powerful tool, it simplifies a fundamental mathematical system, that of propositional logic. Another way of viewing this gain is to say that propositional logic is unnecessarily complex.

The linear space in which we have grown accustomed to writing symbols has lead us to establish systems of mathematical representation on a baroque foundation. By teaching the linear notation of propositional logic to students, we teach them the skills of convoluted thinking. They will be correct, but their processes will be awkward. The analogy is attempting to learn multiplication using Roman numericals.

I have specified two rules that simplify parens expressions, intolerance of external duplicity and intolerance of internal regression. These two mathematical skills are sufficient for thinking about logical expressions without variables. They provide a simple foundation for clear formal thought. The introduction of variables permits these tools to be abstracted.

3.3 Variables

A variable is a token that refers to a set of expressions. Two natural collections of parens expressions are those that vanish when simplified, and those that do not. Some examples that vanish are[7]

```
    (( ))
(( (( )) ))
  (( )( ))
```

Some that do not vanish are

```
     ( )
   ((( )))
((( ))(( )))
```

Rather than listing all the expressions that vanish, it is convenient to refer to them all by the variable `N`, for non-existent. It is relatively easy to see that expressions that do not vanish all simplify to an empty parens; I will label this set `M`.

Another use of variables is to refer to arbitrary expressions, those that we do not yet know their classification. Arbitrary expressions might belong in either set `N` or `M`. Several of these variables are handy, I will use the capital letters `{A, B, C,...}`. A special case of arbitrary variables are those that represent

[7]It may be informative to transcribe these expressions into a logical interpretation.

ground expressions, either the empty parens or non-existence. I will use the small letters {`a, b, c,...`} to represent arbitrary variables that are represent no more than one parens. Finally, variables can be used to represent arbitrary objects, not necessarily parens expressions. This leads to the construction of domain theories, which I will not address.

3.4 Equations

Now we can express the characteristics of the simple space algebraically, using equations that incorporate variables. For instance, our desire to eliminate duplicates might be written as

$$\texttt{A A = A}$$

where `A` stands for any parens expression. Any expression which shows up twice in a simple space should be eliminated. The irrelevancy of the double parens can be written as

$$\texttt{((A))} \Longrightarrow \texttt{A}$$

Any expression which is surrounded by double parens can be freed from the unnecessary inversions.

Earlier, I suggested that we have permission to insert tokens into a simple space in order to study them. This can be symbolized as

$$\texttt{A ( )} = \texttt{A (A)}$$

Here we recognize that we thought of some arbitrary expression `A`, external to the space we wish to study it in, and inserted it into an empty simple space.

Other rules of the simple space come easily. For instance,

$$\texttt{A ( )} = \texttt{( )}$$

We know that `A` is either in `N` or in `M`. If it is in `N`, then when `A` is simplified, the rule will read

$$\texttt{( )} = \texttt{( )}$$

since `A` has vanished. If `A` is in `M`, then the rule will read

$$\texttt{( ) ( )} = \texttt{( )}$$

which is a special case of intolerance of duplicity. In both cases the rule is correct.

Variables point out the inadequacy of the linear bounding language of parens. Specifically, consider the expression `(A (A))`. Because the token `A` exists in two different spaces, we are forced to duplicate it in this linear expression. We must reinstitute a planar representation to avoid this kind of duplicity.

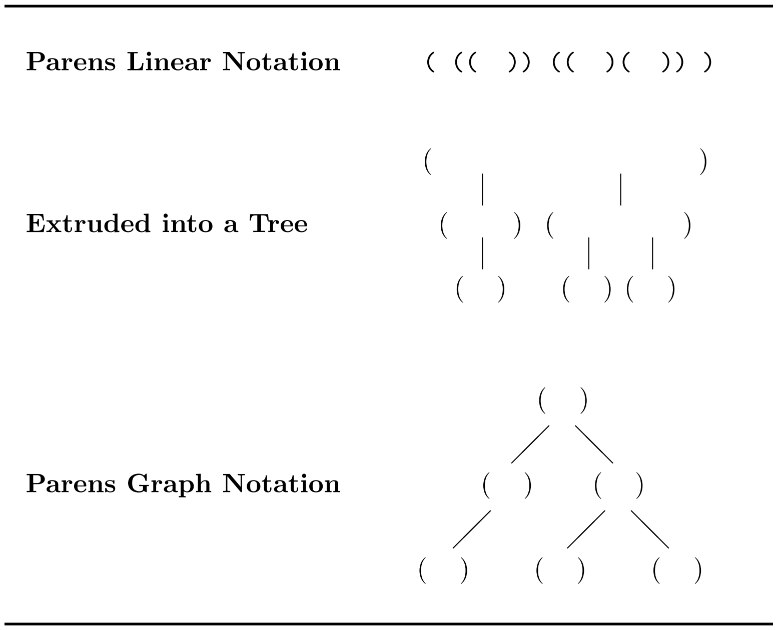


Figure 5: The Construction of a Directed Graph from a Parens Expression

3.5 Graph Notation

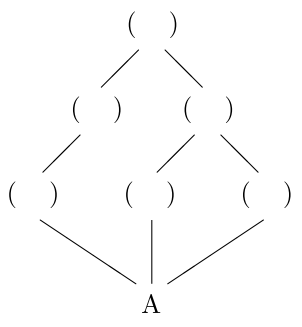
Graph notation is an alternative representation that eliminates the redundancies of linear notations. In the graph notation, each node represents a token. Multiple references to the same token are achieved by links, or pointers, to a unique object. Each link indicates that the token below is contained within the boundary-token above. (Thus, the graph is directed.) Graph notation relies on the containment exhibited by parens operators.

As an example, the expression `((( ))(( )( )))` is pictured in Figure 5. This graph is a *tree*. Its various interpretations for logic are discussed in Section 3.2. The figure illustrates how the outer-most parens forms the top-node of the graph. Levels of the graph correspond to successively deeper nested spaces in the parens expression.

The introduction of variables can change a tree into a network. Consider the graph in Figure 6. In this example, the variable `a` has been inserted into the deepest spaces of the previous example, forming the expression `(((A))((A)(A)))`.[8] The graph representation expresses the same configuration using only one variable token `a`. With graph notation, we can picture the expression `(A (A))` without duplicate references to the variable `A`. This network is presented in Figure 7.

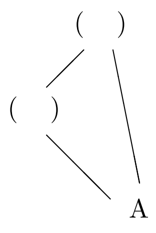
Rather than explore the power of graphic notation for complex expressions, I wish to conclude by returning to the original inadequacies of linear notations,

[8]This may be transcribed into logic as $\neg(\neg\neg a \vee (a \wedge a))$.



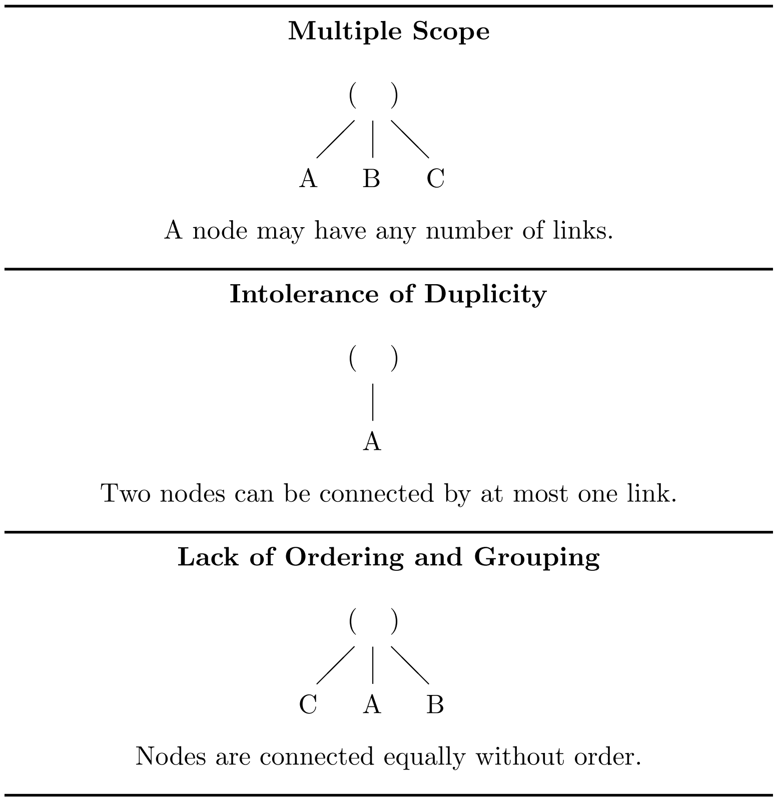
The expression `( ((A)) ((A)(A)) )` as a network.

Figure 6: A Graph Representation of an Expression with Variables



Multiple links indicate multiple references to tokens.

Figure 7: A Graph Representation of the Expression `(A (A))`



Multiple Scope

A node may have any number of links.

Intolerance of Duplicity

Two nodes can be connected by at most one link.

Lack of Ordering and Grouping

Nodes are connected equally without order.

Figure 8: Graph Notation Alleviates Linear Irrelevancies

expressing each of these in graph notation. Figure 8 contains the graph notation for multiple scope, intolerence of duplicity, and intolerance of ordering and grouping relations in the simple space.

4 Summary

In this paper, I have suggested that it is not a good idea to confine mathematical notation to lines on a page. The benefits of a notation supported by a simple, non-structured space include the elimination of binary scope, duplicity of representation, commutativity and associativity. When such a notation is interpreted as propositional logic, the foundations of traditional symbolic representation can be seen to be overly elaborate. Several notational innovations, such as the simple space, the simple token, functional spaces, parens notation and graph notation are suggested as steps toward the clarification of symbolic conventions.

The key concepts proposed in this paper follow:

- Non-linear representational spaces free elementary mathematical systems from linear irrelevancies.
- The empty simple space and the simple token provide a foundational model for formal symbol systems. They refer to the two types of void.
- The maintenance of simplicity requires constraints both on the duplicity of tokens and on the replication of operations.
- Functional spaces and boundary-tokens permit the representational collapse of the object-process dichotomy.
- Representational incompleteness permits notational elegance.
- Graph notation permits non-duplicity of representation.
- Elementary logic can benefit from a shift in perspective.

To foreshadow the potential utility of parens and graphic notation interpreted as logical expressions, consider the construction of computer programs that perform inference. Such programs are the heart of expert system technology, and are fundamental to the field of Artificial Intelligence. An inference engine based in parens notation can represent deductive tasks more efficiently and reach deductive conclusions in less steps than inference engines based in traditional techniques.[9] Linear Losp is a parens-based engine that uses list processing as its implementation paradigm. An inference engine based in graph notation performs parallel deduction over a network. Parallel Losp implements this facility using a message-passing paradigm in which each node in the network is a small computing system. Both versions of Losp demonstrate the efficiency and elegance of computation in a simple space.

[9]The formalization of these simplified processes is presented in W. Bricken, A Deductive Mathematics for Efficient Reasoning, unpublished.