A hierarchy of languages, logics, and mathematical theories

by Charles William Kastner, Houston, Texas, U.S.A.

We present mathematics from a foundational perspective as a hierarchy in which each tier consists of a language, a logic, and a mathematical theory. Each tier in the hierarchy subsumes all preceding tiers in the sense that its language, logic, and mathematical theory generalize all preceding languages, logics, and mathematical theories. Starting from the root tier, the mathematical theories in this hierarchy are: combinatory logic restricted to the identity I, combinatory logic, ZFC set theory, constructive type theory, and category theory. The languages of the first four tiers correspond to the languages of the Chomsky hierarchy: in combinatory logic Ix = x gives rise to a regular language; the language generated by S, K in combinatory logic is context-free; first-order logic is context-sensitive; and the typed lambda calculus of type theory is recursive enumerable. The logic of each tier can be characterized in terms of the cardinality of the set of its truth values: combinatory logic restricted to I has 0 truth values, while combinatory logic has 1, first-order logic 2, constructive type theory 3, and categeory theory ω_0. We conjecture that the cardinality of objects whose existence can be established in each tier is bounded; for example, combinatory logic is bounded in this sense by ω_0 and ZFC set theory by the least inaccessible cardinal.

We also show that classical recursion theory presents a framework for generating the above hierarchy in terms of the initial functions zero, projection, successor, as well as composition and μ-recursion, starting with the zero function I in combinatory logic

This paper begins with a theory of glossogenesis, that is, a theory of the origin of language, since this theory shows that natural language has deep connections to category theory and it was through these connections that the last tier of the hierarchy was discovered. The discussion covers implications of the hierarchy for mathematics, physics, cosmology, theology, linguistics, extraterrestrial communication, and artificial intelligence.

Glossogenesis

Studies of great apes in the wild over the last 40 years have given us a riveting portrait of what early human life may have been like.[1,2,3,4] One aspect of our history in particular has remained obscure, however, and that is how we made the transition to human language.[5,6] We present here a glossogenesis theory based on natural selection that uses the overall setting provided by wild ape studies as a social and environmental backdrop, adding one essential new ingredient: the ability to ground symbols denoting physical objects, by which we mean the ability to name things with spoken language. This new ingredient is not far-fetched, given the variety of chimpanzee and other primate vocalizations and their semantics, and particularly in light of the ability of vervet monkeys to emit different alarms for leopard, snake and eagle.[7,8,9,10,11]

The theory is based on the following ideas: 1) on the road to present-day human language, man developed a series of protolanguages of gradually increasing complexity,[12] 2) there exists within human language a well-defined hierarchy of syntactic structures, with higher level structures dependent on all lower level structures, 3) the above-mentioned hierarchy is linear, so that one can define the nth protolanguage to be the language containing the nth syntactic structure in addition to all n-1 syntactic structures above it in the hierarchy, 4) the hierarchy of syntactic structures can be deduced from the physical world, 5) the selection process at work involved competing, hostile groups speaking different protolanguages or the same protolanguage with different levels of proficiency, 6) a group speaking one protolanguage had better survival chances than a rival group speaking the same protolanguage with less proficiency or a less complex protolanguage within the same environment, everything else being equal, 7) as one approached modern human language in the hierarchy, the selection advantage for the individual with greater linguistic proficiency became increasingly important.

One key assumption of the theory is that natural selection can operate on a group level, an hypothesis that has proponents among evolutionary biologists.[13,14] It also makes two major assumptions about early human society: that humans socialized in well-defined groups and that rival groups competed with one another violently. In fact, the second assumption is really only used in the theory in the development of two syntax structures corresponding to the numerical concepts one and two, and it may be that other scenarios that selected for these concepts can be found that do not require the existence of hostile groups. In any event, there is precedent among primates for such groups. Chimpanzees in particular live in small groups inhabiting established territories with habits and technologies so distinguished from those of neighboring groups that to speak of chimpanzee *culture* is appropriate.[15] Chimpanzees are also known to raid other groups and attack their members.[16] A likely group size for early humans may have been from 50 to 150 individuals, although this size of course grew with time.[17]

For each syntax structure presented below we will do the following: 1) define what it means, 2) discuss why it requires the structures above it, 3) discuss the horizontal extension of the syntactic structure, that is, the evolution of meaning within the protolanguage, 4) apply a statement in the protolanguage to a scenario in which the survival rate of the population or individual that can utter it is enhanced, and 5) discuss the selection advantages of the new structure.

We have strictly applied the following criterion in distinguishing between the horizontal extension of a structure and vertical extension to a new structure: no protolanguage is allowed to depend on the horizontal extension of a less complex protolanguage. For example, nowhere in the theory do we presume the ability to count to three, since two is the highest structure related to number. Further counting is a horizontal extension of two.

An informal notation is adopted using the idea of an *operator*. Within each syntax structure after *symbol* an operator is introduced whose meaning defines that syntax structure. The term *operator* is used to emphasize that the new syntax acts on something to bring out something new. The operands, i.e. the things operators act on, in all cases represent symbols, which we will define presently; they are written α, β. We call an operator that operates on one symbol a unary operator, and an operator that operates on two symbols a binary operator.

Symbol

symbol is a language comprised only of symbols. A symbol is a spoken word which refers to a thing in the real world and which has meaning even when that thing is absent from the context of usage.

Horizontal Extension: We assume that over time speakers were able to agree on a symbol for any physical object, and ignore the issue of how such symbols were grounded. However, at the time when *symbol* was the only protolanguage, it is likely that new symbols would only arise if they had immediate natural selection implications for the group. The first symbols may have been alarm cries analogous to those of vervet monkeys.[18]

The horizontal extension of *symbol* happens in all subsequent protolanguages. Indeed, as the hierarchy of protolanguages is traversed, most dependent sub-protolanguages continue to expand horizontally. The process of adding *symbols* to the language must have been continually refined, so that beyond the *classification* protolanguage, one can imagine symbols being added quite readily.

Usage Scenario: "Tiger," in the presence of a tiger.

Although we don't mention them further here, all utterances given in all usage scenarios are assumed to be accompanied by gesturing and other cues such as tone and volume of voice, and so on. The complex non-verbal communications systems in primates and the copious amount of information they convey stand in stark contrast to the meager information content of early language.

Selection Advantage: Obvious.

Negation Epoch

Negation

Operator: ! is a unary operator indicating the absence of the physical object referred to by the symbol upon which it acts.

Valid statements in the *negation* language consist of all valid statements from *symbol*, which are just symbols α, β, and so on, plus statements using the operator **!,** such as $!\alpha$, $!\beta$ and so on. In general, each protolanguage includes all valid statements from the preceding protolanguage and adds the new statements made possible by its operator, including compositions of the new operator with previous operators. These compositions are not part of the horizontal extension of the protolanguage since the meanings of the component operators are known and so the composite operator is immediately meaningful.

Other Required Structures: *negation* is obviously meaningless without *symbol.*

Horizontal Extension: In the *negation* and all subsequent protolanguages, the *!* operator means no or not.

Usage Scenario: "Not tiger," when a tiger warning has already been issued, but turns out to be false.

Selection Advantage: The original warning will cause a panic, which can be exploited by other predators. Calling off the warning allows for a return to normal vigilance.

Singularity

Operator: *singular* α is a unary operator that indicates there is exactly one instance of the physical object referred to by the element of *symbol* it acts on.

Other Required Structures: *singular* is obviously meaningless without *symbol,* but it also requires *negation*, since it is predicative, meaning that it says something about a subject, and a predicate only has meaning if the concept of the truth or falsity of a predicate is available, i.e. only in *negation* and subsequent protolanguages. Without the concept of the truth or falsity of a predicate, any predicate would have to be universally applicable and therefore could not be used to characterize a subject in a meaningful way.

Given a symbol β, suppose we have a singularity operator without *negation*. In order for this operator to be well defined, that is, in order for a speaker to know when to use this operator, it must have meaning in all situations in which a speaker encounters the object referred to by β, and those situations include more than one object referred to by β. Otherwise, if we assumed that *singular* only had meaning when a single object were present, then in order for the speaker to have a criterion for when to use the operator, *symbol* would need to contain at least two kinds of *β*, one for a single instance of the object referred to by *β* and one for multiple instances of the object referred to by *β*. Clearly that could not happen: two such betas could not arise before *singular*, since their meaning requires the functionality of *singular*. In other words, it is the job of the *singular* operator to expand the symbolic world; it is not up to *symbol* to spontaneously evolve. So what happens when there is more than one object referred to by *β*? The putative *singular* operator without *negation* in that case is not defined and the speaker has no criterion for deciding whether or not to use the operator, which makes the operator meaningless.

In order for *singular* to be always properly defined, we need *negation.* Now when a speaker encounters an object or objects referred to by *β,* he or she can use the *singular* operator when there is one object, or the negation of the *singular* operator when there is more than one. To indicate that there are no objects referred to by *β*, the speaker of course can use !β.

The requirement that *negation* precede *singularity* is analogous to the requirement in classical recursion theory that the primitive recursive function α be defined before any primitive recursion predicate function can be defined. The function α(x), defined as 1 if x is 0 and otherwise as 0, fills the role of negation (see Appendix I, Table x+1).

Horizontal Extension: None.

Usage Scenario: "Single enemy," when a single enemy is present.

Selection Advantage: A single enemy can be overcome by a group of speakers. The advantage is that this weakens the rival population and lessons competition for food resources.

Duality

Operator: *dual* α is a unary [sic] operator that indicates there are exactly two instances of the physical object referred to by the element of *symbol* it acts on.

Other Required Structures: *dual* is obviously meaningless without *symbol* and it clearly also requires *singular*. The reason that *duality* is required as a separate syntax structure is that it is required by the following structure, *similarity*.

Horizontal Extension: *duality* extends horizontally as counting: first there is a symbol for three instances of an object, then four instances, and so on.

Usage Scenario: "Two enemies," when two enemies are present.

Selection Advantage: Two enemies can be overcome by a group of several speakers. The advantages of this are the same as given above for a single enemy. One can imagine that a member of the population who hears this may react differently depending on sex, age, size, and so on.

Similarity

Operator: *similar* α,β is a binary operator that indicates there is some perceived similarity between the two instances of the physical objects referred to by the elements of *symbol* it acts on. One or both of the objects referred to may be understood by pointing or by gazing; these forms of non-verbal communication are observed in other primates.[19,20,21]

Other Required Structures: *similar* is obviously meaningless without *symbol,* but it clearly also requires *duality*, since making a statement that things are alike requires that two objects be brought to mind simultaneously. It also requires *singularity,* since specific solitary instances of α and β are initially involved, but *singularity* is inherited from *duality*.

Note that a demonstrative pronoun is not required as a syntax structure here because that meaning must be supplied by pointing. Demonstrative pronouns arise as a horizontal extension of *reflexive verb*, since it is not until then that the implied reference back to the speaker makes sense.

Note also that *similar* requires *symbol,* but it does not necessarily require known symbols. This operator was undoubtedly used as new symbols were being established.

Horizontal Extension: Eventually *similar* operates in situations where neither symbol operated on refers to a specific instance of a thing.

Usage Scenario: "[this] bug like [that] bug!" when the second bug is known by everyone to be good to eat and non-biting, and the speaker knows the first bug to also be good to eat and non-biting.

Selection Advantage: Obvious.

Classification

Operator: *class_x* α is a unary operator that indicates that the thing referred to by the *symbol* it acts on has quality x, where x represents any quality or attribute.

Other Required Structures: *class_x* is obviously meaningless without *symbol.* To see why *similarity* is necessary, assume it is not necessary. If a speaker of *classification* only knows *duality*, then the only way he can classify things is by number. But that's something he can already do, as a speaker of *duality*. He must know *similarity* so that he can group things together in his mind based on their being alike in some way.

As in the case of *similar*, *class_x* requires *symbol,* but it does not necessarily require a known symbol. The object referred to may be understood by context.

Horizontal Extension: *class_x* from the beginning says what kind of thing a thing is and that includes things that can be classified by an intransitive action. In this regard it is worth noting that some modern languages such as Chinese do not clearly distinguish between adjectives and intransitive verbs.[22] At any rate, there would be no way to distinguish in *classification* between *hot* and *boiling*, for example, since all usage is predicative. Of course, in subsequent protolanguages the ability to distinguish the quality from the action related to the quality was developed.

Usage Scenario: "Good plant!" when the speaker wants to convey that a plant is good for a given ailment.[23] Here the quality referred to is the quality of being beneficial.

Selection Advantage: Obvious.

Comparison

Operator: *more_class_x* α, β is a binary operator that indicates that the physical objects referred to by the symbols upon which it acts differ from one another in terms of degree with respect to a given classification *class_x*. It means that the first operand has more of the quality denoted by *class_x* than the second operand.

Other Required Structures: *more_class_x* clearly requires *classification.* As was the case in *classification*, *more_class_x* requires *symbol,* but it does not necessarily require known symbols. The objects referred to may be understood by context.

Horizontal Extension: The quality x in *more_class_x* initially was probably only the quality of being good; in other words, initially *more_class_x* meant *better.* Gradually, more differentiating qualities were referred to, such as sharpness in the usage scenario below.

Usage Scenario: "[This] rock sharper than [that] rock."

Selection Advantage: Being able to differentiate things by degree allows speakers to store more detailed knowledge about what works and what doesn't work to their advantage.

Sequence

Operator: *first* α is a unary operator that indicates that the object referred to by the symbol upon which it acts occurs before some other object as perceived by the speaker, with respect to time or space.

Other Required Structures: *first* requires *comparison,* for the following reason. *first* as applied to space is really identical to *more_class_x*, where x is the quality of being close or near in space and the second operand is understood from context . As an example from English, consider *The apple tree is the first tree in the row of trees that extends to the south from the road*.

The reason that *sequence* is required as an independent syntax structure is that it is required by subsequent structures; otherwise, it could be considered a horizontal extension of *comparison*.

Horizontal Extension: In the beginning, *!first* was understood as *last*. Later, *sequence* extended horizontally to the ordinals *second*, *third*, and so on. The degree of extension is the same or less than the degree of extension of the cardinals, which occurs in *duality*. In addition to ranking with respect to space or time, the operator was extended to allow ranking with respect to any quality, for example the quality of being good or fast. Included in this extension is the superlative, which denotes first rank of at least three things, as in the ranking best, better, good, or fastest, faster, fast.

The speaker cannot distinguish at this point between time and space. The usage scenario below illustrates a sequence in time.

Usage Scenario: "Impala first, spear last," when the speaker has thrown a spear at an impala running across his field of vision. He aims his throw at the current position of the impala and when the spear lands there, the impala is no longer at that position.

Selection Advantage: The author remembers as a boy the first few times he tried to throw a ball at a receiver running across his field of vision: the ball always landed behind the receiver. The ability to throw ahead of the receiver so that the ball hits him as he runs is not innate, but requires a conscious, i.e. verbally based, effort, at least in order to quickly acquire the ability. This ability has obvious selection advantages for speakers who hunt using a spear or rock trying to hit a running animal. [24]

Elapsed Time

Operator: *earlier_class_x* α is a unary operator that indicates that the object referred to by the symbol upon which it acts was in a state characterized by the quality x at some point in time earlier than when the operator is used.

Other Required Structures: *earlier_class_x* clearly requires *sequence* and also *classification*. It must be an independent structure because it is the first structure to distinguish time from space, and subsequent structures require that capability.

Horizontal Extension: *earlier_class_x* has little horizontal extension. Eventually the state quality x may be dropped and the operator used adverbially.

Usage Scenario: "Earlier impala sleeping," when the speaker finds matted grass after seeing an impala from a distance and going over to investigate.

Selection Advantage: The speakers become better hunters if they are consciously aware of the habits of their prey. In the above scenario, they might come back another day and see if the impala return.

Transitive Verb

Operator: *action_x* α,β is a binary operator that indicates that the object referred to by the first symbol upon which it acts performed an action x on the object referred to by the second symbol. α or β might be understood by context.

Other Required Structures: *action_x* clearly requires *symbol*, but it also requires *elapsed time*. We will give a lengthier explanation than usual for this, since at first glance it may seem counterintuitive, especially if one considers that by the time *transitive verb* is reached considerable time may have elapsed since the introduction of *negation*, the first syntax structure which unambiguously distinguishes human language from other native primate languages.

The idea that the meaning of a verb could be established by an innovative speaker of *duality* who mimics or performs an action for an audience must be rejected. Context would already convey the meaning intended by the verb and so the verb could neither take hold nor even have a reason for being. Another reason that such a scenario does not work is that the speaker would have had to have at least spoken *sequence,* because the source of the action begins acting before the receiver begins receiving the action and a speaker must understand this before discovering the concept of subject and object (α and β). But even here we fall short, for the following fundamental reason.

The receiver of the action of the verb undergoes a change of state due to that action; for example, in the case of the verb *hit*, he goes from being untouched to touched, or unhurt to hurt, or sleeping to awake, as the case may be. Or take the example of the verb *kill*; here the receiver goes from the live state to the dead state. Any transitive verb one can think of also has this effect of changing the state of the receiver of the action: *kiss, push, trick, eat, throw, pull, tear, break*, and so on. The *elapsed time* structure provides the mechanism for expressing this change of state, via *earlier_class_x*, *!earlier_class_y,* where in the latter statement *!earlier…*means *later* and the quality y is the negation of the quality x in the former. Without this understanding of change of state, *transitive verb* cannot be discovered, just as *similarity* cannot be discovered without an understanding of *duality*.

One final reason that *action_x* requires *elapsed time*: *action_x* only makes sense as a past tense when it is first introduced. As we've mentioned, context makes usage of the transitive to describe concurrent action unworkable at this stage of the hierarchy. This is consistent with the fact that a transitive verb brings about a state change in the receiver: in order for the receiver to be in the new state, the action of the verb has to take place before the speaker speaks. So in *transitive verb*, what speakers were really saying was *kissed, pushed, tricked, ate, threw, pulled, tore,* and *broke.* Since no syntax structure before *elapsed time* distinguishes between time and space, *transitive verb* cannot be before *elapsed time* in the hierarchy.

Horizontal Extension: *action_x* instances originally had to be expressed in terms of the instrument used to perform the action, possibly accompanied by all or part of *earlier_class_x, !earlier_class_y.* So, for example, *speared* was used before *killed*.

Since *sequence* is available at this level, at some point two *action_x* statements were chained together to describe two events in order, and then three statements, and so on.

After the introduction of *cyclic time*, *transitive verb* extends to different time contexts and to verbs of emotion, etc.

Usage Scenario: "Rock cut skin," when the speaker brings up the fact that he just cut an animal skin with a rock, in order to point out a new application for a sharp rock.

Selection Advantage: The selection advantage of the above usage scenario is clearly less immediate than in the earliest structures. This reflects a trend for successive syntax structures to have selection advantages that are more and more systematic. Originally, language served to immediately save lives and was used chiefly in life and death situations. Over time, it increased survival rates by improving survival processes, so that its use was not only for life and death situations, but also for referring to situations that had long-term implications for the survival of the group.

The selection advantage of *transitive verb* is that it allows a more elaborate description of processes that work for survival, so that these can be preserved, and it facilitates the discovery of new survival processes because existing processes that work have been better described and understood.

Reflexive Verb

Operator: *action_x_on_self* α is a unary operator that indicates that the object(s) referred to by the symbol upon which it acts performed an action x on himself/herself/itself/themselves.

Other Required Structures: *action_x_on_self* clearly requires *transitive verb*. However, it cannot be a horizontal extension of *transitive verb* because it is required by the next syntactic structure in the hierarchy, *cyclic time*. This is a point worth discussing here.

We first point out that before *reflexive verb* a verbalized concept of identity is not available. In *transitive verb* there is nothing in the language that allows for a speaker to refer to himself or herself, or for a speaker to refer to another object as referring to itself. The word *I* does not exist yet, because nothing has allowed it to exist. It is possible for *action_x* to be used with the speaker understood by context to be the one who carried out the action, but that does not mean that the speaker has a concept of herself as a separate identity, nor does it mean that other speakers will interpret the fact that the speaker carried out the action as meaning that the speaker has an identity. In the speaker's mind the focus is on the action and the receiver of the action, but not on herself as the initiator of the action, because nothing in *transitive verb* allows her to refer to herself using words. If no speaker has a concept of his or her own identity, no particular thing can have an identity in a speaker's mind either, because the concept of identity does not exist.

We also point out that in order for the concept of future to emerge, it is necessary for there to be an external time marker undergoing periodic motion that serves as a time reference, such as the sun or moon. Without a time marker, there would be no way to associate an event with a time, and no way to project a certain amount into the future, because there would be no metric for such a projection. In the discussion below, we refer to two separate events at two different times. We assume the two events both occur during daylight hours and that they are separated by less than a few hours.

Now we are able to discuss the discovery of *cyclic time*. The speaker knows how to think backwards in time from *time elapsed*, so she first thinks backwards. She must remember some remarkable event in the past that she witnessed, which she can do because she speaks *reflexive verb*, and associate with that event the position of the sun, her time marker, at that time. She can make that association if the sun hurt her eyes, for example, again because she speaks *reflexive verb*. In effect, she must now transport herself into her previous self for an instant to that event and from that point in time and space think of herself in her current state, noting the current position of the sun and realizing that the position of the sun has continuously evolved from its position at the time of the event to its current position. Finally, having understood the forward-moving nature of time, it dawns on her that she can project from her current state to the future some amount, and extrapolate to get the position of the sun at that future time. Once this mapping from sun position to events in time is understood, the cyclical aspect of the sun's motion is also readily understood.

Clearly the process just described requires that the speaker understand that the person who witnessed the remarkable event and the person reflecting on that event are one and the same, that is, that she have a concept of her own identity, which as we've argued first arises in *reflexive verb*.

Horizontal Extension: *reflexive verb* extends horizontally in many ways. As argued above, a speaker's sense of identity is established here. Also, it becomes apparent that other speakers also have identities which can be addressed with language. This shift in consciousness must have generated an explosion of verbal communication. Before *reflexive verb* language was strictly declarative, the only form of discourse being declarations in sequence, which were limited by the syntax available in *transitive verb* --- repetition was the main form of linguistic interaction between speakers. In *reflexive verb* it became possible to ask for information from others, because they were thinking beings, too, with whom a convention could be established for exchanging information. Dialogue, in which speakers may explicitly refer to themselves and their interlocutors, could now take place. Furthermore, language was now a conscious act, which means it could be consciously, that is deliberately, developed. It is no exaggeration to liken this abrupt change in human communications to a phase transition in a physical system, in which a system undergoes a transition from a disordered to an ordered state due to an infinitesimal change in some physical quantity upon which the system depends.

In *reflexive verb* many new forms of expression are possible. The question words *what*, *where*, *who*, *which* and *how*, personal pronouns such as *he*, *she*, *you*, *we*, possessive markers such as *my*, *your*, *mine*, clearly all of these can now be discovered and used. Also, the demonstratives *this*, *that*, *these*, and *those* make sense because they refer indirectly to the speaker, who now has an identity. Relative clauses such as *that I threw* in *the rock that I threw* are also possible after questions and demonstratives have been introduced. After relative clauses, the definite and indefinite markers *the* and *a* too make sense.

Representative art is now possible, since speakers can be aware in some dim sense of the essence of *symbol*.

Usage Scenario: "Hit myself," when the speaker hits himself with a sharp rock and explains to others what happened.

Selection Advantage: In *reflexive verb* selection operates to a significant extent on the individual as well as the group level. Although selection on the individual level undoubtedly occurred in previous protolanguages, in *reflexive verb* interpersonal relations have a much stronger verbal component than before. This allows an intelligent speaker more opportunity to stand out in complex social situations, which results in better chances of reproductive success. On the group level, the advantage that the drastically improved communications made possible in *reflexive verb* is obvious.

Cyclic Time Epoch

We now describe the hierarchy's next ten structures, each of which we will later show to be closely analogous to a corresponding structure already described, namely the structure occurring ten steps back in the hierarchy. The selection advantages of these new structures are obvious.

Cyclic Time

In *cyclic time* speakers discover the meaning of units in time by reference to something that repeats itself over and over at regular intervals in time.

We have already discussed the mechanism for discovery of *cyclic time* while discussing *reflexive verb*, where we argued that *cyclic time* requires *reflexive verb*. From that discussion it is clear that along with *cyclic time* emerges the concept of the present as well as the future, so that speakers now understand past, present and future. Therefore, all time-related verbal markers can now occur, although not all at once since there is a dependence among them. Verbs of volition now make sense, too, because the speaker has a concept of the future and the ability to anticipate verbally, and likewise imperatives are now possible. Time units *day*, *month, year*, and fractions of *day* also arise here.

The reason *cyclic time* must exist as a separate syntax structure is that it is required by the next structure, *implication*.

Implication

implication is a syntax structure characterized by the if-then construction. We argue that *cyclic time* must be available in order for *implication* to emerge.

In order for *implication* to establish itself, there must be a belief that given a certain set of circumstances, the consequences of those circumstances are similar whenever that certain set of circumstances happens again. Such a belief can only arise if it is seen over and over as a pattern. Here is an example.

Let us call A a circumstance and B an event for the sake of simplicity; in general, A and B each could be either a circumstance or an event. Now consider the following pattern in time: AB*************AB*********AB************AB…, where the ************* indicate arbitrary events and circumstances not including A, and the three dots at the end indicate the pattern repeats itself an arbitrary number of times.

If the pattern is repeated enough, then it will be held that given A, B will follow as inevitably as it follows that if the sun is currently rising, it will later be at high noon, provided *cyclic time* is spoken.

One might argue that the mere repetition of AB in itself might be sufficient to cause one to understand that given A, B will always follow, without *cyclic time*; one could imagine for example a rat in a cage with constant lighting being conditioned to respond to a stimulus. The point to be made here in response is that we are trying to show where in the hierarchy the understanding that B follows A can be explicitly expressed with language. Let us look at the problem more closely: suppose there are many such rats in the above-mentioned controlled environment and that these rats speak *reflexive verb,* which we assume is a necessary condition for the discovery of *implication.* If *cyclic time* were not available, then the rats could not say that B *always* follows A, because this implies that they have a concept of something happening over and over, now and in the future. From *elapsed time* they can express order in time, but without *cyclic time* they don't have a notion of the present or future, since in *reflexive verb* the action of all expressions involving time starts and ends at some point in the past. In *reflexive verb* they could say B always *followed* A, which strictly refers to the past and does not involve *implication*.

There is an interesting question that arises from the rats discussion above, namely whether the notions of present and future time could have been discovered without discovering the cyclic aspect of time; the question is interesting because *implication* might be discovered with these notions alone. If A and B were both events, and B followed A at regular intervals, then the rats could use the AB cycles to discover *cyclic time* by the mechanism we've already described: A and B would be analogous to two points on the sun's trajectory. We want to consider what might happen if the intervals between A and B were aperiodic. According to our arguments above, the discovery of *cyclic time* relied on the speaker realizing that the position of the sun had moved in a predictable fashion from one point to another during the time separating a memorable past event and the present. If the only thing that can be said about the relation in time between A and B is that they are ordered, then it is not possible to predict when B will occur, given A. So even if there are events A and B corresponding to the past event and the present, respectively, and even if the speaker happens to notice the corresponding position in time of both A and B, there is no way for him to establish a rule for measuring distances in time based on the distance in time between the past event and the present. Without such a metric, the idea of projecting into future time cannot occur to him, and without an understanding of future time, the present has no meaning, being a bridge between past and future time. We conclude that understanding the cyclic aspect of time goes hand in hand with understanding present and future time and therefore is indispensable for the discovery of *implication*.

The horizontal extension of *implication* includes the question *why*, the concept of causality, and conditional modalities in verbs, as in *if you had listened to the crow, you would have known the rain was coming*. Verbs of cognition such as *to know* are also possible now for the first time.

AND

AND is a syntax structure that includes the two logical operators AND and OR. They emerge together because a AND b is the same as !(!a OR !b) and likewise a OR b is the same as !(!a AND !b). In logic, the operators negation, AND and OR form what is called an adequate set of connectives, which means that any truth function can be represented by a statement whose variables are acted upon by a combination of those connectives; some other adequate sets are {negation, OR}, {negation, AND}, and {negation, implication}.[25] The essence of the *AND* structure is that speakers now have an intuitive understanding of the concept of an adequate set of connectives.

Obviously, these operators require *implication* in order to have meaning. They could be considered to be a horizontal extension of *implication*, but once again they must be a separate structure because they are required by the next structure, *equality*. Regarding their origin, one can use arguments similar to those used for *implication* based on *cyclic time*.

Equality

equality is a syntactic structure which allows one thing to be thought of as equal or equivalent to another in some sense. Looking back, we notice that *similarity* is a structure that goes in the direction of *equality*; this is the first obvious hint of the parallels between structures to be found throughout this article.

The new syntax is given by the binary operator *equivalent_x* α, β, meaning α is equivalent to β with respect to a classification x. This operator requires *AND* as we shall now show, and therefore was not possible before now.

We recall that *more_class_x* α,β, the *comparison* operator, indicates that α has more of the quality denoted by *class_x* than β. Combining this operator with *implication* and *AND*, we get

if(!*more_class_x* α,β AND !!*more_class_x* α,β) then *equivalent_x* α,β.

In words: if α doesn't have more of the quality x than β and it doesn't have less, then it has the same amount, i.e. α is equivalent to β with respect to the quality x. This procedure for defining equality is analogous to the procedure for defining equality as a primitive recursive function $x = y$, which defines two numbers to be equal by first defining the primitive recursive absolute value $|x - y|$, and then uses a form of negation of the absolute value. The definition of the absolute value is $|x - y| = (x \dot{-} y)+(y \dot{-} x)$, where the binary operator $\dot{-}$ is defined by $x \dot{-} y = x - y$ if $x > y$, and 0 otherwise. The definition of equality is then $\alpha(|x - y|) = \alpha((x \dot{-} y)+(y \dot{-} x))$, where $\alpha(x)$ is defined as 1 if x is 0 and otherwise as 0.

A natural context in which such a thought process would occur is in the cutting and playing of two reeds as musical instruments; in this case the quality x is the length of the reeds: if they are the same length, they give the same tone, which is <u>much</u> more pleasing than if the tones are slightly different, outside of a very small tolerance. By cutting different lengths related in a certain manner consonant intervals could be generated, for which humans appear to have a universal preference.[26] This context appears so natural that the author holds that *equality* was in fact discovered this way, under the assumption that our preference for consonant intervals did not develop *after* the discovery of *equality*. Other contexts, such as the sizing of spear tips to fit stone spearheads, seem to lack the motivation for such exactness or they lack such a sensitive gauge for measuring convergence on an exact value; weighing things using a balance scale is another possibility, but that appears to be premature at this point in the hierarchy.

The horizontal extension of *equality* is extremely rich. As indicated, music starts here. The question of what something is or is not can first be asked in this structure. Abstract concepts such as love and courage can be synthesized. Epistemology and the beginnings of science, from history to Euclidean geometry to physics, are now possible.

Equivalence Class

equivalence class is a mathematical structure. Several definitions emerge with it at once: binary relation, equivalence relation, equivalence class, partition, set, subset, and element. These definitions are possible because *equality* allows new concepts to be defined using existing syntax structure.

Provisionally, we say a set is a collection of things called elements. A subset of a set S is a set containing none, some or all of the elements of S. A null set contains no elements. Given a non-null set S containing elements A, B, C, …, a binary relation on S is a set whose elements are ordered pairs (A, B) of elements of the set S. A binary relation R on a set S is an equivalence relation if the following three statements hold for all elements A, B and C of S:

1. if (A, B) is an element of R, then (B, A) is an element of R.
2. if (A, B) and (B, C) are elements of R, then (A, C) is an element of R.
3. (A, A) is an element of R.

An equivalence class is defined in terms of equivalence relation, as follows. Given a set Z with an equivalence relation defined on it, there is a theorem that says that Z can be partitioned in a unique way into subsets based on the equivalence relation.[27] These subsets are disjoint, meaning that every element in Z is in exactly one of the subsets. Each subset is called an equivalence class; the decomposition of the set into equivalence classes is called a partition. The theorem also says the converse is true; in other words, given a partition of a set, a unique equivalence relation on that set is determined.

For example, given the set Z = {1,2,3,4,5} partitioned into three equivalence classes as follows {1}, {2,4,5}, {3}, the corresponding equivalence relation is given by R = {(1,1), (2,2), (3,3), (4,4), (5,5), (2,4), (4,2), (2,5), (5,2), (4,5), (5,4) }.

Now we return to the definition of set and say a set is something we construct by the procedure indicated above: starting with a collection of things, we define an equivalence relation on the collection, and that makes our collection of things a well-defined set.

To show that the above-mentioned definitions are possible at this stage of the hierarchy, we justify them as follows. The definition of set, first in its provisional form as a collection, makes use of *sequence* to build the set one by one from its constituent elements in any order we choose; subset also uses *sequence* in the same way. The definition of null set requires *negation*. Ordered pairs in the definition of binary relation come from *sequence* and *duality*. The conditions required for an equivalence relation use syntax from *implication* and *AND*. The above-mentioned theorem makes use of *implication*, *comparison*, *AND* and *singularity* in its statement and proof. Finally, the names of these concepts, as all words, come from *symbol*.

equivalence class is required as a separate structure because the concept of set is required by the following structure, *function*.

Function

function is a mathematical structure. A function relates one set to another, a source set to a target set, by associating with each element of the source set exactly one element of the target set. The source set of a function f is called the domain of f, and the target set is called the codomain. Other names for function are mapping, transformation, operator, and map. A function f from the set A to the set B is often written

$$A \xrightarrow{f} B$$

For each element a in the domain A, f assigns an element b = f(a) in the codomain B.

A careful definition of a function f says that it is a set whose elements are ordered pairs (x, f(x)), where x ranges over the entire domain of f, subject to the condition that given an x, f(x) is a single element of the codomain of f. We are able to speak of sets and elements here because of *equivalence class* and of ordered pairs again because of *sequence* and *duality*. Also, the *subject to the condition* in the definition comes from *implication*. The word *entire* comes from *sequence*: we build these ordered pairs starting from the first and ending at the last element in the domain according to any order of our choosing. Lastly, we note that *single* comes from *singularity.*

The horizontal extension of function includes monomorphism, epimorphism, isomorphism, endomorphism, and automorphism, all of which are functions having special properties with respect to domain and codomain. For example, an endomorphism is a function for which A and B are the same set.

In physics, Kepler's laws, force, Newton's laws, and Maxwell's equations can all occur here, as well as thermodynamics, statistical mechanics, optics, and so on.

Function Composition

function composition is a structure needed by the next structure, *category*. Given sets A, B, and C, and a function ϕ from A to B and a function ω from B to C, we define the function $\zeta = \omega \circ \phi$ from A to C to be given by first applying ϕ, then ω. *function composition* is actually a function, since it takes two functions and maps them to a single function.

function composition is the glue that holds category theory together.[28] Practically anything of interest in category theory can be drawn up as a diagram with arrows between objects, where the arrows represent functions. Many interesting cases arise when there is more than one sequence of arrows available to get between two given sets in a diagram. In such cases, when more than one arrow is involved in such a sequence, functions are being composed.

In theory our speakers could still only be counting to two if there had been no horizontal extension in *duality* at all. Using A, B and C appears to violate our principle of not allowing any structure to rely on the horizontal extension of a less complex structure. We get around this problem by noticing that we can speak only of two objects in this case, so C must be either A or B.

The horizontal extension of composition involves chaining three functions together to get a new function, then four, and so on.

Category

A *category* **C** is a set of things A, B, C, D, E … called objects, and a set of function-like things called morphisms or arrows that go from object to object. A category is subject to the following two conditions:[29,30,31]

1. Composition of morphisms is defined, and is associative, which means that given morphisms ω from A to B, ϕ from B to C, and ζ from C to D, the following holds: $(\zeta \circ \phi) \circ \omega = \zeta \circ (\phi \circ \omega)$.
2. For each object A in **C** there exists an identity morphism 1_A such that, given morphisms κ from A to B and λ from C to A, we have $1_A \circ \lambda = \lambda$ and $\kappa \circ 1_A = \kappa$.

Morphism is a generalization of the concept of function. A morphism distinguishes itself from a function in that it is allowed to have no ordered pairs at all, although we still give it a domain and codomain; the name arrow reflects this generalization.

An object refers to anything at all, a fish, a sack of potatoes, a set, anything, as long as 1. and 2. above hold. Object, morphism, and category all arise together in this structure.

Categories are pervasive throughout mathematics; in fact, it is possible to describe all of mathematics using category theory, a point we will touch upon later. To give an example of a category with supreme importance in mathematics and physics, we shall define what a group is and show how that definition makes a group a category. The theory of groups is a major component of modern algebra.

First, we define a binary operation on the set G informally as an operation that takes two elements of G and maps them to an element of G; a more formal definition would require the concept of a product so that we could define a binary operation as a function from the product GxG to G. A set G with a binary operation * is a group if the following hold:

1. There is in a unity element 1 within G such that, for all g in G, 1*g = g*1 = g.
2. For all g in G, there exists an element h in G such that h*g = g*h = 1.
3. The binary operation * is associative.

A group is a category containing one object, G. We take as morphisms the elements of G with both domain and codomain G, and we define composition of morphisms to be the binary operation *. The elements of G as morphisms do not have ordered pairs associated with them.

Groups provide a natural framework for expressing symmetry. They were an absolutely indispensable tool in 20th century physics, well before they were interpreted as categories. The horizontal extension of category includes special and general relativity, quantum mechanics, quantum field theory and string theory.

Within category theory itself, an important horizontal extension of category is topos, which is used in the categorical development of set theory and logic.[32]

Functor

functor is a map between categories. If C and D are categories, a functor F from C to D is a map that assigns to each object A in C an object F(A) in D and to each morphism f:A->B in C a morphism F(f):F(A)->F(B) in D such that:

1. $F(1_A) = 1_{F(A)}$.
2. $F(\zeta\circ\phi) = F(\zeta)\circ F(\phi)$.

In other words, a functor maps objects to objects and morphisms to morphisms so that objects in the target category are related to one another by morphisms in the same way they are in the source object. In this fashion functors preserve structure. It may occur that the target category is much simpler, in which case the functor must forget some of the structure of the source category.

There are many examples of non-trivial functors.[33] For physicists the most interesting example of functors are topological quantum field theories, which relate as categories the world of general relativity on the one hand and quantum theory on the other.[34] The existence of topological quantum field theories, established by Witten in the late 1980's, is further evidence of the profound links between mathematics and physics.[35]

Natural Transformation

natural transformation is a mapping between functors. If S and T are functors from the category C to the category D, a natural transformation is a map τ: S(A) -> T(A) which assigns to each object A in C a morphism in D such that given a morphism f: A->B in C, $T(f)\tau_A = \tau_B S(f)$. In other words, the following diagram "commutes":

$$\begin{tikzcd}
S(A) \arrow[r, "\tau_A"] \arrow[d, "S(f)"'] & T(A) \arrow[d, "T(f)"] \\
S(B) \arrow[r, "\tau_B"] & T(B)
\end{tikzcd}$$

A *natural transformation* preserves structure between two functors that map between the same two categories, C and D.

category and *functor* were originally defined in order to define *natural transformation*, which itself was motivated by connections between topology and algebra.[36] The horizontal extension of *natural transformation* appears to include n-categories.[37]

It appears that as of the year 2003 we find ourselves in *natural transformation*, since no applications of natural transformations as yet have led to new physics.

Generalizations

0	Negation	Cyclic Time
1	Singularity	Implication
2	Duality	AND
3	Similarity	Equality
4	Classification	Equivalence Class
5	Comparison	Function
6	Sequence	Function Composition
7	Elapsed Time	Category
8	Transitive Verb	Functor
9	Reflexive Verb	Natural Transformation

Table I

Table I summarizes our results so far. The generalizations we now describe have in common that they account for variation with time; this is more clearly manifested in the early structures.

negation is generalized by *cyclic time*. The negation operator applied twice is the same as the identity operator, at least until intuitionistic logic is discovered. Both *negation* and *cyclic time* behave cyclically.

singularity is generalized by *implication*. In *singularity* there are two operators, *singular*, and *!singular,* which together allow exactly two propositions in the physical presence of the object referred to by any operand; one of the propositions must be true and the other false. *implication* allows propositions with variable truth values to be considered using its operands. *singularity* gives speakers an inkling of nature of the boolean constants true and false, and propositional logic; *implication* gives speakers an inkling of the nature of boolean variables and first order logic (see Appendix A).

duality is generalized by *AND*. *duality* is discovered by combining two invocations of *singular*. *AND* is discovered by combining two implications. *duality* makes possible a static picture of two things. *AND* makes possible a variable picture of two ideas. These operators extend their predecessors in an analogous way: *duality* gives meaning to the idea of a numerical successor function that leads to the natural numbers, while AND gives rise to the concept of a complete set of logical connectives, which allows for an arbitrary formula in first order logic.

similarity is generalized by *equality*. For its discovery, *similarity* requires two separate invocations of *singularity* to first isolate the operands and then the use of *duality* to relate them; likewise, *equality* requires two boolean variables from *implication* as well as *AND* for its discovery. *similarity* results from a one-time evaluation; *equality* results from a sequence of evaluations.

classification is generalized by *equivalence class*. *classification* uses *similarity* to determine how to group things. Likewise, *equivalence class* requires *equality*, without which the concept of equivalence relation is impossible; *equivalence class* determines how to group things unambiguously.

comparison is generalized by *function*. The *comparison* operator *more_class_x* selects one of two things on the basis of *classification*, thus relating one thing to another in an integral sense; *function* maps one thing to another on the basis of *equivalence class*, relating one thing to another in a structural sense.

sequence is generalized by *function composition. function composition* combines two functions to yield a new function, whereas *sequence* combines two comparisons to yield a new comparison. To explain the latter, consider the usage scenario "impala first, spear last" given above. Translated into operator notation, this reads *more_class_x* α β, *!more_class_x* β α, where α and β refer to impala and spear, respectively, and x is the quality of being near in a temporal sense to the time when the spear was thrown --- viewed from the point in space where the spear landed. The scenario could also be translated as "impala before spear," which is effectively a comparison in itself. Both *sequence* and *function composition* extend horizontally in parallel with the horizontal extension of *duality*.

elapsed time is generalized by *category*. *earlier_class_x* α represents an evolution in time of α; "earlier impala sleeping" says something about the impala at an earlier time which may or may not still hold at the time of the utterance. A category can be used to model the evolution in time of a dynamical system; in such a category the objects represent states of the system and the morphisms represent transitions between states.

The subject of a *transitive verb* takes its object from one state to another, the two states being *earlier_class_x*, !*earlier_class_y*, where *y* is !*x,* as mentioned above. So in "rock cut skin," the animal skin earlier was in the uncut state and later was in the cut state. By themselves *earlier_class_x* and !*earlier_class_y* each implicitly contain the same information, but from a different viewpoint. *functor*, which generalizes *transitive verb*, maps from one category to another category. The two categories are analogues of the initial and final states of the receiver of the transitive verb's action. In terms of structure given by morphisms between objects, these two categories contain the same information by the definition of functor.

The subject of a *reflexive verb* takes itself from one state to another. We now consider the state of the "giver" of the action of the verb as well as the state of the receiver, before and after the action takes place. The reflexive aspect of the verb equates the giver with the receiver, and therefore the transitive aspect of the verb takes the both the giver and the receiver of the action from one state to another. A *natural transformation* is a mapping from one functor to another, whereby each of the two functors act on one category and relate it to a second category. Again the two categories can be thought of as analogues of the initial and final states of the receiver of the verb's action. The functors are analogues of the transitive action of the verb, which takes the giver and receiver of the action from initial to final state. The natural transformation, which links equivalent structure within the functors, is the analogue of the reflexive aspect of the verb.

Language Evolution Timeline

Only a few protolanguages have been discovered in times of recorded history: the three protolanguages of category theory in 1945 by MacLane and Eilenberg, and the notion of equivalence class in the 19th century. Since earlier protolanguages evolved in prehistory, we must look at material culture specimens to determine approximate dates for them. We shall do that for a few protolanguages and then use those dates to put together a very rough overall timeline.

First, we look for a latest possible date for *equality*. Any musical instrument capable of producing tones requiring the notion of an exact interval can be considered to be evidence for *equality*. Flutes found in the Henan province of China dating from 8000 to 9000 years ago have this quality; one of the flutes even has a correcting hole that brings it into a better tuning.[38] An older flute from the Geissenkloesterle cave in southern Germany, dating from roughly 37,000 years ago, has been shown in a reconstructed model to play the notes C, D, F and B melodically, so that multiples of the half-step interval of the chromatic scale were clearly deliberately produced.[39,40] It is possible as well that stone tools were used as musical instruments in the manner of a xylophone at still earlier times.[41]

There are several things which would indicate competence in *cyclic time*: 1) representations of the sun at different positions in the course of a day, 2) representations of cycles of the moon, 3) seasonal behavior clearly planned for in advance, such as agriculture, and 4) structures such as the boulders of Stonehenge which show knowledge of the solstice or equinox. The earliest such evidence is that of bone carvings found in Europe from roughly 30,000 years ago that points to an awareness of the lunar cycle according to Marschack.[42] However, we know from the evidence for *equality* that *cyclic time* was spoken sometime before roughly 37,000 years ago.

reflexive verb is the first protolanguage in which symbolic art is likely to emerge. A pair of recently found engravings in ochre from the Blombos cave of South Africa, shown in Figure 1, suggests that such art may go back at least as far as 77,000 years, but this is open to interpretation since the engravings are abstract and may have no meaning.[43] The author suspects that the two engravings contain the same number of diagonal lines in one direction, eight, and so at least might be interpreted as a representation of the number eight, especially if other such engravings were found. In the spirit of Marschack, he further suspects that the two engravings represent the same thing, namely the trajectories of the moon and the sun over a one-month period with the skyline in the middle, whereby the new moon trajectory breaks the symmetry of the representation in a single segment as it approaches the sun. If the latter interpretation were correct, the engraving would correspond at least to *cyclic time*. Speculations aside, the first firm evidence of symbolic art is from the Upper Paleolithic in Europe, about 35,000 years later.[44]

sequence would be reflected in material culture as an object which required at least two steps to make, the steps being in a particular order. The Mousterian stone tool tradition, which began roughly 200,000 years ago, produced such objects.[45] In fact, the distinguishing characteristic of the Mousterian technique was precisely that it was a multistep process: first a stone core was shaped, from which stone was "peeled off" subsequently piece by piece, and then each of these individual pieces of stone was finished into a tool itself. This tradition appears early in North Africa, South Africa, Northwest Europe and Mediterranean France, perhaps independently.[46]

In order to find traces of *duality* we look for something involving two distinct aspects that were deliberately produced. We find this in the stone tool technique preceding the Mousterian called the Acheulean, which began roughly 1.5 million years ago in East Africa.[47] It differs from the Oldowan tradition that preceded it in that the finished stones had two distinct faces, as opposed to being of haphazard shape as finished stones were in the Oldowan tradition, which began roughly 2.5 million years ago.

From the above observations we establish the rough timeline shown in Figure 2, which shows the appearance of syntax structures as a function of time. Note that since we have taken conservative dates for each point in the above curve, revisions are expected to shift the points corresponding to *equality* and all protolanguages before *equality* to the left. Of course, the plot in Figure 2 does not mean to imply that there was a single group that started in *duality* whose direct descendants made it all the way to *equality*. One can easily imagine that syntax structures were independently discovered by different groups, as may be reflected in the Mousterian tool traditions just mentioned.

From Figure 2 one concludes that humans have been using language for at least 1.5 million years. Such a time span is long enough for substantial changes to have occurred in the human brain and vocal tract due to selection based on language proficiency in both speech and syntax.[48,49] In Figure 3 we speculate what the plot in Figure 2 might look like as more evidence becomes available, shifting the points corresponding to *reflexive verb*, *cycle* and *equality* to the left. Figure 4 shows a plot of hominid brain size as a function of time; it is apparent that brain size increases in a way that qualitatively matches our plot in Figure 3. In particular, there is an explosive growth period in the last 200 thousand years, which one would expect from the increased importance of language for both group survival and individuals' reproductive success within the group as higher syntax structures were employed.

In Figure 5 we have plotted, in addition to the points of Figure 3, three exponential functions fitted to the endpoints *duality* and *natural transformation*, *sequence* and *natural transformation*, and *reflexive verb* and *natural transformation*, respectively; from the plot it is clear that the latter two fits give a better approximation to the data than the first fit, and they would for any reasonable shift to the left of points corresponding to *sequence*, *reflexive verb*, *cyclic time* and *equality*. In fact the shifted points in Figure 3 were positioned so as to illustrate that the strange dependence on time that we see in Figure 2 may have to do with the population size of the speakers of a given syntax structure. One could imagine that the discovery of *reflexive verb* gave a competitive advantage over other groups and allowed for a larger population size and more complex social interactions and structures, which in turn allowed for greater selection of individuals with more language ability and so led to greater brain size, which in turn, coupled with the now increased population size, hastened the discovery of the next syntax structure, *cyclic time*, which in turn allowed for an even larger population size, and so on, in a positive feedback loop. This process may have culminated in the so-called Upper Paleolithic Revolution, a term associated with an explosion of new behaviour beginning about 35-40 thousand years ago (kya) in Europe which is reflected in innovative tool forms, sculptured stone and bone, personal decoration, evidence of activities requiring long-term planning and strategy, more structured living environments, cave art, and so on.[50] Evidence indicating that this revolution may have started at least 35,000 years earlier in Africa has begun to emerge.[51]

It is worth considering whether there are mechanisms other than increased speaker population size that might explain the time behaviour of Figure 2. One possibility is that the later syntax structures are by nature easier to discover and for that reason they occurred in more rapid succession; this appears counterintuitive and unlikely. Perhaps the above-mentioned feedback loop took place without population growth, fueled only by ever-increasing cognitive ability; this is possible, but then it would be necessary to explain why population size did *not* increase because more cognitive ability naturally leads to a higher population size, and furthermore this would weaken the feedback loop since higher population size *certainly* increases the likelihood that a new syntax structure would be discovered, all else being equal. Another possibility is that there is simply a gap in the archaeological record and that we must shift many of the higher syntax structure points hundreds of thousands of years to the left; this doesn't seem likely, since stone tools are well represented in the archaeological record going back 1.5 million years over a wide geographical area and, if higher syntax structures were reached hundreds of thousands of years earlier than the current evidence suggests, there is no easy way to explain why we have not found corresponding stone tool evidence similar to that found in Europe dating from roughly 40 kya onward.

On the other hand, there is some evidence in the archaeological record for increased population in the Upper Paleolithic in Europe.[52] More extensive evidence comes from human origins studies based on genetics, which have indicated that it is necessary to assume a low effective hominid population, on the order of 10,000, over the last 2 million years in order to account for the gene distribution in living humans.[53] They have also indicated that an explosion in effective population occurred at some time in the past, and that a likely time for such a population explosion appears to be between 50,000 and 100,000 years ago.[54] This evidence lends some weight to the conjecture that the time behavior of Figure 2 may be related to speaker population size; by the same token, Figure 2 appears to reinforce the genetic evidence for low effective population and a sudden population expansion between 50,000 and 100,000 years ago. Implicit in the above argument is the assumption that if the effective population used in the population genetics studies increases, so does the speaker population size and vice versa; the argument also only makes sense if the low effective population long-term was not due in large part to extinction and recolonization of populations, which allow for large breeding populations to leave a small effective population trace.[55]

Population size is a crucial aspect of the human origins debate centered on whether *homo sapiens* emerged as a new species, unable to breed with other hominids, within the last 200,000 years. How this article bears on that debate is a question best left to experts in paleontology, archaeology, and population genetics.[56]

Natural Selection for Language Evolution
It appears that the selection mechanisms responsible for the evolution of language gradually shifted over time. In what follows, we argue that gradually the process became less dependent on the group as a whole and more dependent on individuals. Bits and pieces of this argument have been given above; we bring them together here and add a few others. For the sake of argument, we assume that the above timeline is basically correct and that in particular a lapse of a million years or more intervened between the discovery of negation and the discovery of sequence. The aim of these remarks on natural selection is only to give a plausible viewpoint on the matter, and it goes without saying that the remarks remain speculative.

The extremely slow rate of new syntax discovery in the earliest phases of language evolution may indicate that new syntax was not discovered in an all or nothing fashion, rather that the process of discovery occurred over a period of time, perhaps spanning many generations. One can imagine for example that the full-fledged understanding of negation may have followed a long period of time when only some of the group was involved in its use and gradually individuals who did not understand the meaning of negation were weeded out. If in fact the usage scenario for the discovery of negation give above --- “no tiger!” --- is correct, then it may have slowly gained a foothold just as an alarm call for a specific predator must have slowly gained a foothold among vervet monkeys. If on the other hand one assumes that the discover occurred suddenly, one is obliged to find some rarely occurring, spectacular scenario that the entire group would experience all at once.

Another argument against a discovery of negation or singularity happening all at once comes from the psychology of early man, if one can call it that. There can have been no real reflection on the fact that one was using language until *reflexive verb*, which occurred much, much later. In the earliest stages, if the warning call was in fact where it all started and we let the vervet monkeys guide our intuition, a spoken word had the same effect in the people hearing the warning as it did in the person issuing the call: they repeated the call and acted as though they had perceived the predator themselves. It is easy to imagine that the same sort of mass repetition also occurred after *negation* had taken hold in a group. Language at this stage was part of a process, a group process that only arose in dire emergencies. In such an environment it is difficult to imagine anything like the insight of an individual at the root of the discovery of new syntax that would make the syntax suddenly comprehensible, and it is still more difficult to imagine such an insight occurring to an entire group all at once.

Gradually new syntax must have allowed for more complex social interaction and new technologies and thus led to larger population size. The challenges offered by these social and technological changes --- and perhaps also changes in habitat, climate, and so on --- would have provided problems whose solution selected for neural structures that were preadaptations for the discovery of new syntax, which led to larger population size and additional challenges, and so on in a positive feedback loop. As this feedback loop gathered steam, the role of the individual in the discovery of language must have become more and more prominent and the language ability of the individual must have become more and more of a deciding factor in reproductive success. However, even at this stage it seems likely that the selection mechanism for language was indirect in the sense that although an individual may have discovered transitive verb, for example, and that individual may have been able to explain the meaning of a verb to others, such a discovery was probably not due to a mutation of a gene that somehow can be specifically associated with the new structure. Instead, it seems more plausible that genetic changes were such that more complex neural structures were produced as preadaptions which increased the likelihood that new syntax structure would be discovered.

The glossogenesis theory presented here makes the entire selection process more plausible in that it is clear at any stage exactly what is meant by the evolution of language: it is either a vertical or a horizontal evolution. For example, horizontal evolution in *duality* meant that people learned to identify groups of things according to their number, that is, they learned to count beyond two and to count different things; vertical evolution at that point meant discovering *similarity*. At any point in the hierarchy it is not difficult to imagine scenarios in which the individual or the group can exploit the new syntax. The problem of natural selection heretofore has seemed difficult because it was never clear that there was a sequence of well-defined stages in the process of language evolution and so it was difficult to even describe what a selection mechanism for language might have selected for.

Finally, we mention group selection. There appear to be just two places in the hierarchy where rival groups come into play: in the discovery of *singularity* and *duality*. The usage scenarios given for these syntax structures were “single enemy” and “two enemies”; one might replace even these with “one baboon” and “two baboons” on the assumption that isolated baboons away from their group might have made good prey. So on the one hand one can argue that the theory presented here may not require group selection in the sense that entire groups of early humans were wiped out by other groups speaking more advanced protolanguages. On the other hand, the advantage that one group would have over another at a lower level in the hierarchy is so obvious that it would seem rash to discount such a possibility. The demise of the Neanderthals may have been due to group selection on a massive scale, for example, if in fact homo sapiens and Neanderthal were part of the same species. The last wave of human migration out of Africa starting slightly over 100k years ago may have spelled the end for many groups such as the Neanderthals because of the superior language skills of what started as a single group.

Predictions of the Glossogenesis Theory

The glossogenesis theory described here makes two main predictions, which are testable.

1. No computer simulation of glossogenesis starting from nothing more than the ability to ground symbols can arrive at the concept of one before arriving at the concept of negation, nor similarity before duality, and so on. The linearity of the theory precludes skipping steps.
2. It is possible for a simulation to start from the ability to ground symbols and from there arrive at negation, and from there singularity, and so on.

Regarding the archaeological record, the theory does predict a time order in which certain material culture specimens can occur. Unfortunately, since the possibility of new finds can never be excluded and since the material culture does not tell us whether two different specimens were made by speakers of related or unrelated languages, one cannot with certainty validate or invalidate the theory using such time order arguments. However, the archaeological record can be interpreted broadly in terms of the theory with no apparent contradictions, as we've shown above.

There is some slim evidence that there existed a single common ancestral human language of some form based on the widespread occurrence of a few common words, among them the form *tik meaning *one* or *finger*, and *pal meaning *two*.[57] The theory cannot predict that there was one such single common language, since two or more languages could in theory have emerged in isolation and arrived at *equality* independently, but if there was only one, then obviously *tik would have arisen in *singularity* and **pal* in *duality*. Evidence for common parentage of existing or reconstructed languages of course must be weighed on a statistical basis, and certainly the present theory can be useful in quantifying the likelihood that language families are related and in determining where they may have diverged in the syntax hierarchy. Other corroborating evidence for the existence of such relations would likely come from genetics studies and the archaeological record.[58]

Computer Simulations of the Glossogenesis Model

It is beyond the scope of this paper to discuss simulation details, but it is worthwhile to underscore a few key points. First of all, a simulation of this model must do two things. It must on the one hand show that it is possible to go from protolanguage to protolanguage starting from *symbol*, and the other hand it must show that a group has a better survival rate or an individual is reproductively more successful by virtue of language proficiency. The latter is simple once the former has been achieved.

A general feature of simulations of this model is that speakers must recognize patterns of behavior in response to threat and opportunity which lead to the success of the individual and the group. In effect they must also be able to generate initially a random symbol when such a pattern is recognized and use that symbol until or unless they hear two other speakers generate a different symbol in response to the same pattern, in which case that symbol must be adopted.

It is clear that a connectionist approach is appropriate here for simulating the linguistic behavior of speakers, which means that associated with every speaker in each of two groups of speakers there must be a neural network. The speakers themselves would be actors in a world simulation that starts from initial conditions and is continually updated by small time increments. The simulation might be in an abstract world or it might represent the real world; an abstract world would be preferable if machines of pure intelligence were the desired result of the simulation. In a real world simulation, during each update speakers may be hunting, foraging, running or climbing trees when avoiding predators, sleeping, grooming and so on. Other parts of the environment would include a variety of edible plants, predators, and prey. All animals would of course be simulated so that they act according to their behavior in the wild. The important thing is that a diversified environment be simulated, with a rich variety of rewards and penalties, the understanding of which allows for altered behavior through new syntax structure. A starting point for such an environment is provided by the great ape field research mentioned above.

Surveying current simulations of glossogenesis, one sees that a common approach has been to assume that certain meanings exist a priori.[59, 60] The point of the simulation is then for speakers to associate symbols with those meanings. Such an approach, although it has merits, obviously will not work as is for this model. A more appropriate approach for this model is that of simulations in which progress has been made in simulating the formation of meaning in social situations.[61, 62]

Non-bootstrapped Language Acquisition

Extensive research has been done in the area of human-animal communication, especially human-primate communication, but not exclusively.[63,64] The exploits of Washoe the chimpanzee[65], Koko the gorilla,[66] Chantek the orangutan,[67] Kanzi the bonobo[68] and others clearly indicate that all of them have a grasp of *symbol*, some of them can count,[69] and some of them have even shown some understanding of the concept of self and of time, although none has shown an understanding of *implication* or *AND*. In all cases, horizontal extension is very limited when compared with human linguistic performance. It is interesting to note that the results of the mirror test[70], which purports to establish whether a primate recognizes himself in the mirror, may depend on the rearing and training of the subject.[71]

The implications of this article on interpretations of the above-mentioned research are limited due to the fact that glossogenesis theory tries to account for a bootstrap process in the sense that new syntax structure must be discovered without outside aide, whereas animal language learning through human intervention is not a bootstrap process. So, for example, it might be possible to teach a primate *singularity* without teaching him *negation*, or *duality* without teaching him *singularity*, but glossogenesis has nothing to do with this form of syntax acquisition and so in this context is irrelevant. That said, it may be that a distinction can be made between partial and full language competence, where full competence refers to competence in the current and all preceding protolanguages. Although such a distinction may be blurry, it could be helpful in determining what methods of teaching syntax lead to the best linguistic performance of animals.

The above arguments also hold for child language development in humans, which likewise is not a bootstrap process. In particular, children with learning disabilities may be responsive to teaching methods that focus on mastering the structures in order.

Physical Universe, Life and Begin Algorithm Epochs

0	Physical Universe	Life	Algorithm	Negation	Cyclic Time
1	1 String	1 Cell	1 Process	Singularity	Implication
2	2 Strings	2 Cells	2 Processes	Duality	AND
3	Nucleus	Endosymbiosis	Integrated Parallel Procedure	Similarity	Equality
4	Atom	Neuron	Classification Procedure	Classification	Equivalence Class
5	Molecule	Linked Neurons	Compare Procedure	Comparison	Function
6	Polymer	Neural Network	Sort Procedure	Sequence	Function Composition
7	Chiral Polymer	Sensory, Motor Neural Network	Serial Input, Serial Output	Elapsed Time	Category
8	Translation	Unidirectional Neural Network	Coordinated Serial I/O	Transitive Verb	Functor
9	Replication	Bidirectional Neural Network	Symbol	Reflexive Verb	Natural Transformation

Table II

Next we will discuss the three new epochs presented in the matrix of Table II, which were discovered by extrapolating backward from the two epochs already discussed.

By way of nomenclature we will call the rows in the matrix depending on context either *rows* or *i-sequences*, the columns either *columns* or *epochs, 0-epochs* or *j-sequences*, each matrix entry an *operator*, and the matrix itself the *matrix*, or *hierarchy*.

Physical Universe Epoch

The *physical universe* epoch starts with *physical universe,* which lies in the same row as the operators *life*, *algorithm*, *negation* and *cyclic time.* As for the details of the big bang, the overall picture of an expanding universe from an initial infinite density and temperature has been well accounted for by the standard model;[72] however, at the high energies seen at the earliest times after the bang a consistent physical theory of both general relativity and quantum theory is still lacking, and the question whether the universe will eventually recollapse into a big crunch is still open. As we shall discuss below, it does appear that the causal structure imposed by special relativity first appears in *physical universe*.

From other operators in row 1 of the matrix, we take the *1-particle* operator to mean a single entity of matter, whether that turns out to be a string, a brane, or something else. Presumably the first entity existed for some finite amount of time in the Planck era, an era that lasted until 10^-43 seconds after the bang. We interpret this to mean that for the entire duration of this operator there was literally just one entity of matter.

The *2-particle* operator comes about through decay of the first entity into two entities. Its horizontal extension includes further decays.

In *nucleus* the neutron and proton are joined together, bound by the strong force; in the standard model of physics, this takes place at roughly 1 second after the bang. The *nucleus* i-sequence shows a pattern of merging two things, which is also evident in *similarity* and *equality*.

In *atom* the temperature has finally cooled to the point where electrons are bound to hydrogen nuclei, which happens around 700,000 years after the bang. The i-sequence[4] operators *atom, nerve cell, classification procedure, classification,* and *equivalence class* are each a basic unit of construction.

In *molecule* multiple atoms bind into a single unit, and in *polymer* molecules form chains with repeating structural units.

In *chiral polymer*, polymers exists in two forms, one left-handed and the other right-handed. All polymers occurring in living cells have a fixed handedness; for example, the nucleic acid DNA, a polymer with repeating nucleotide units, is always found to be right-handed, and similarly, all protein polymers, technically known as polypeptides, have a particular handedness in living cells. It is unclear whether the first chiral polymers on Earth were polypeptides or nucleic acids, or something simpler --- this is a crucial question in the origin-of-life debate, as is the actual nature of the mechanism that might have originally generated chiral polymers, and one of a number of questions of the chicken-or-egg type.[73, 74, 75] A recent study has shown that molecules of the amino acid aspartic acid are selectively absorbed on different faces of a calcite crystal according to the chirality of the aspartic acid molecules.[76] The authors suggest that this separation may have been the first step toward the construction of homochiral (left- or right-handed) polypeptides and leave open whether nucleotides may have similar absorption properties on crystals.

In *translation,* one chiral polymer is able to construct another chiral polymer. In living cells today, one can think of nucleic acids such as DNA and RNA in a certain sense as playing the role of software and polypeptides as playing the role of hardware: DNA and RNA contain code that allows for the synthesis of polypeptides, which are the physical building blocks as well as workhorses of cells. In fact, the cutting, copying and pasting operations directed by DNA have been shown to be equivalent in computational strength to a Turing machine (see Appendix H).[77] In *translation*, some chiral polymer A was able to synthesize another chiral polymer B by means of a code that mapped structural elements in A to structural elements in B. In DNA and RNA, the basis for such mappings are three-letter words from a 4 letter-alphabet defined by adenine, cytosine, guanine and thymine (thymine is replaced by uracil in RNA). Orginally, the code was probably simpler; for example, it may have consisted of one letter words from a two-letter alphabet.

In *replication*, a chiral polymer copies itself. One possibility is that the polymer B constructed by A in *translation* acquires the ability to construct A, either directly or via a chain of constructions that lead to A.

Life Epoch

There is no universally accepted definition of life, but all definitions include the concept of replication, and also metabolism, which is the extraction and use of energy from the environment via chemical processes. Since before *life* there is nothing in the hierarchy referring to metabolism, it may be that newly evolved metabolic processes were what made *life* possible.

There is reason to suppose there was a time lag between *life* and the next structure, *1-cell,* based for example on the analogy with the transition from *negation* to *singularity*, which could not have been an immediate transition. The presence of *1-cell* after *life* therefore indicates that the first life forms may have been amorphous, with no clear membrane separating individuals, similar in this respect to the cyanobacteria of today. The other possibility is that they resembled other membraneless life forms such as mycoplasmas or viroids that exist as individuals.

In *2-cells* there must have been bacteria-like organisms consisting of two cells in a symbiotic relationship, a conclusion we draw from the following structure, *endosymbiosis*.

endosymbiosis refers to the joining of two independent life forms into a new single-celled life form; unicellular eukaryotes living today show vestiges of several such unions, in the form of mitochondria and chloroplasts, for example.[78, 79] *endosymbiosis* appears to be a necessary bridge from the world of unicellular life to the world of multicellular life and the specialization of the cell.

nerve cell is a particular type of cell that contains a nucleus, an axon, and a mechanism for propagating a signal.

In *linked nerve cells* nerve cells develop synapses, which allow for the signal output of one nerve cell to become the signal input of another nerve cell.

A *neural network* is a network of interconnected nerve cells.

The author speculates that in *sensory and motor neural network*, sensory and motor networks exist but are not linked. A sophisticated example of such a motor network may be the pattern generator network associated with the flight of the locust, which works in a manner independent of sensory input. [80] Presumably the first such networks were pattern generators for swimming that essentially ran continuously unless shut on or off by means of proprioceptor or chemoreceptor cells that detected food or danger. A sensory network without a corresponding link to a motor network was perhaps some primitive version of an eye; since the organism must have reacted to the signal, there must have been a non-neuronal chemical link that led to some change in state of the organism.

In *unidirectional network*, input networks feed forward to output networks via an intermediary network. For example, the lamprey's swimming motion is accomplished by oscillating signals between muscle stretch receptor neurons and motor neurons that contract muscle along its length.[81] Most well understood neural mechanisms are simple networks of this type.[82] Advanced intermediary networks are visible in the visual cortex of mammals, for example, which have a laminar structure, with inputs coming from sensory networks and outputs leading to motor control regions of the brain.

In *bidirectional network*, inputs again lead to outputs via an intermediary network, but within that network there are closed loops between neurons that allow for feedback. From computer simulations of neural networks it is known that such networks are well suited to solve problems involving multiple constraints.[83]

Algorithm

Current knowledge of the inner workings of the brain is so scant that the following discussion of this epoch must be drawn almost completely from parallels with other epochs. One should therefore read "If our analysis is correct…" before each of the following paragraphs describing this epoch.

In *algorithm* an essential step is made in the development of a self-contained, task-specific neural computation: a non-local jump. In this structure, there is a mechanism for branching to a separate subnetwork from within a given active network for the purpose of some computation, after which the given network resumes its computation. Such a branching has recently been discovered in human brains.[84] The closed loops of *bidirectional network* make this structure possible, since without feedback the calling network could not "understand" the concept of a subnetwork.

1-process is a sequence of neural signals in one or more networks that has a unified purpose and that occurs after a non-local jump, as described above. To differentiate this operator from the previous one, we speculate that here some value is returned and used within the context of the calling environment.

In *2-processes* two separate processes run at once from the same calling environment for disparate computations.

integrated procedure is the integration of two simultaneously running processes to effect an integral computation. The horizontal extension of this operator includes massively parallel computations typical of mammals.

classification procedure is a procedure whose purpose it is to evaluate and categorize data. It is a mechanism for defining data type.

compare procedure is a procedure which allows two pieces of data to be compared according to some appropriate ordering principle. It requires *classification procedure* so that a criterion for comparison can be established.

sort procedure takes a series of data and sorts it according to a certain *compare procedure.* Similarly, in software a sorting routine typically takes a comparison routine as input in addition to the data to be sorted.

In *serial input, serial output* there are procedures capable of producing and parsing a sequence of data that has meaning. The vocalizations of many mammals must be based on such procedures. Cats, for example, have different vocalizations for warning they are about to attack, for expressing that prey has been found, and so on, each of which can be thought of as a pitch that varies in time and each of which is intelligible to other cats. Even though such vocalizations may be mostly innate, they are nonetheless neural-based and they could only come into being via the classification, compare and sort procedures given above.

In *coordinated serial I/O*, meaningful sequential input is mapped to meaningful serial output. The vervet monkey, upon hearing alarm calls, interprets them as serial input and immediately generate their own identical call.[85] The alarm call acts transitively in the sense that it invokes in the receiver the same neural mechanisms that the original stimulus for the call invokes in the caller.

In *symbol*, the procedure that maps the serial communication to serial output also refers to itself, i.e. the speaker is aware of the form of the communication in addition to its meaning.

Timeline since *Bang*

In Figure 6 we plot the *physical universe, life, algorithm, negation, and cyclic time* as a function of time, as well as *natural transformation*, our current location. The pattern resembles the time development within *negation* and *cyclic time* shown in Figure 3 and once again indicates that the pace of traversal through the matrix is quickening drastically, growing at a faster-than-exponential rate The implications of this fact will be addressed below when we discuss unresolved questions raised by this article.

There is one aspect of the Figure 6 that merits special attention. The ratio (ti – ti-1)/(ti+1 – ti), where ti is time after the big bang when the ith epoch began, gives a rough idea how much the pace of matrix traversal is changing, and would be expected to be constant for self-similar time intervals. It has values 3.3, 8.54, 284.6, and 14, showing a trend of rapid increase. The puzzling thing is the value 284.6, which measures the change from *negation* to *cyclic time* versus *algorithm* to *negation*; in a natural process such as this one would expect at least some kind of regular trend. The time we have estimated for *algorithm* is 400 million years ago, which, as opposed to all the other dates, is highly uncertain. However, even if we set this date to a time just before the earliest primates, to 75 million years ago, which is almost ridiculously late, then the values become 3.3, 50, 52, 14, still a puzzling result.

One can either attribute the above behaviour to the random nature of the process, or to the paucity of data, or assume that there may be another epoch in the model that we have missed. We found that if we inserted an arbitrary epoch and varied its starting date and the starting date of *algorithm* it was still not possible to arrive at a monotonic increase in the values. Dropping the monotonicity requirement, a typical result plausible in terms of values was 3.7, 3.9, 27, 20.3, 14 with the variable epochs at 800 million and 30 million years ago, dates that one could not just justify in any case. Our conclusion is that the data does not call for an additional epoch.

Order, Logic and Quantum Logic Epochs

In Table III we have added the first three columns: order followed by *logic* and *quantum logic*, and these complete the matrix.

0	1	2	3	4	5	6	7
Order	Logic	Quantum Logic	Physical Universe	Life	Algorithm	Negation	Cyclic Time
1	Unique Null Set	R	1 Particle	1 Cell	1 Process	Singularity	Implication
!1 = 0	Pair	R*	2 Particles	2 Cells	2 Processes	Duality	AND
!!1 = 1	Union	C	Nucleus	Endosymbiosis	Integrated Parallel Procedure	Similarity	Equality
A<1	Infinity	Cn+1 = Cn X C	Atom	Neuron	Classify Procedure	Classification	Equivalence Class
A!<0	Replacement	L^2	Molecule	Linked Neurons	Compare Procedure	Comparison	Function
< = !!<	Power Set	Spin Network	Polymer	Neural Network	Sort Procedure	Sequence	Function Composition
A<B →B>A	Set Membership: x ∈ y → y !∈ x	Ordered Spin Network	Chiral Polymer	Sensory, Motor Neural Networks	Serial Input, Serial Output	Elapsed Time	Category
A<B, B<C→ A<C	x ∈ y, y ∈ z → x ∈ z	Spin Foam	Translation	Unidirectional Neural Network	Coordinated Serial I/O	Transitive Verb	Functor
A=A	x !∈ x	Spin Network Evolution	Replication	Bidirectional Neural Network	Symbol	Reflexive Verb	Natural Transformation

Table III

Order Epoch

After the zeroth operator, which is the input to the entire matrix and will be discussed below, the next operator is 1. 1 is followed by its negation, !1, which is 0. The third operator is the negation of 0, which is 1. As will become clear in the *logic* epoch, 1 is unmistakably a totality --- after concluding our discussion of the remaining epochs it will seem appropriate in retrospect to regard it as the entire universe, with all subsequent operators giving it structure --- whereas 0 takes its meaning in reference to that totality: it contains no part of it. We shall also see it is natural to interpret 1 as truth and 0 as falsehood, and in view of this we point out that the third operator establishes the law of the excluded middle, which says that a proposition not true must be false, and vice versa.

Placing 1 and not 0 in row 1 is appropriate because the primary function of negation is to nullify, as it was originally in the *negation* protolanguage. It then follows naturally in row 3 that !!1 yields 1, since it is the nullification of a nullification.

The definition of the ordering relation <= begins next and it will not be completely defined until the end of this epoch. It cannot appear before !!1, since it does not make sense to order 1 and !1 until their relationship via negation is completely defined.

In $a < 1$ we say a is less than 1, for arbitrary a. In $a\ !< 0$, we introduce the negation of <, and say a is not less than 0, for arbitrary a. In !!< = <, the relationship between < and !< is fully defined. Placing $a < 1$ before $a\ !< 0$ establishes < as more primitive than >, which is consistent with the use of < as the set membership relationship in the *logic* epoch. The justification for introducing a variable here will become clear in the sequel.

In the next three operators the defining properties of < are given for arbitrary variables: first, its asymmetry, meaning $a<b$ is the same as $b!<a$, unless $a<b$ and $b<a$, in which case $a=b$; second, its transitivity: if $a<b$ and $b<c$, then $a<c$, for none of a, b and c equal; finally, its reflexivity: $a=a$. There are three main issues to discuss about these three operators: 1) why three or more inputs, 2) why do the operators appear in their given order, and 3) whether they presuppose first-order logic.

Regarding three inputs, one can say the following. The first operators, 1, !1, and !!1, define two entities related by a unary operation. The next three operators define two binary relations on those two entities. In the final set there is nothing to do unless we accomodate additional entities to be ordered by the binary relations. Looking forward, one can also say that at least three variables are necessary for a valid set of axioms for boolean algebra, the algebra of logic.[86]

The order given by asymmetry followed by transitivity followed by reflexivity can be justified as follows. In the order given, there is a well-defined ordering at each operator: first pairwise ordering, in which there is only a well-defined order in terms of pairs of variables, then strict ordering (e.g. the usual < on the integers), and finally partial ordering (e.g. <= on the integers). In particular, the given order of these operators is the only order in which strict ordering is defined, and as we shall see in the next epoch, the natural ordering on all sets is given by the set membership relation, which is a strict ordering. Moreover, it is clear that transitivity must follow asymmetry so that it can be verified that $a<b$ and $b<c$, namely by checking b!< a and c !< b, and reflexivity must be thought of as a condition made possible by transitivity, taking c to be equal to a --- this latter viewpoint is reasonable since reflexivity has nothing to do with ordering per se and only enters into the definition as a sort of boundary condition that results from extending asymmetry to an entire collection of objects via transitivity.

Asymmetry and transitivity appear to require logic for their formulation, but logic is not really possible until boolean algebra is defined, which will be done in the first operator of the next epoch. The way around this problem is to think of the ordering definition as more of a recipe than a definition. From this perspective, one can think of variables a, b, and c as real objects themselves as opposed to placeholders for as yet non-existent objects; in other words, these operators are not a passive filter of objects, but rather a model for constructing objects. The first step is to build a and b such that a stands in an unambiguous way in relation to b, and b in the opposite way to a. The next and final step is to build c such that b stands to c as a stands to b, and so on.

$a = a$, the final operator in this epoch, completes the definition of the ordering relation <=, just as !!0 completed the definition of the unary operator negation and <=!!< completed the definition of the binary ordering relation < for two fixed inputs. A point worth repeating is that this ordering <= just defined is a partial ordering; the word *partial* is used since it does not imply that for every pair a and b, either $a < b$ or $b < a$ or $a = b$, conditions which define a *total* ordering. A brief summary of orderings and lattice theory is given in Appendix G.

Logic Epoch

The *logic* epoch starts by defining the concept of a boolean algebra, and then defines the Zermelo-Fraenkel axioms of set theory in order to construct the universe of sets using first-order logic.[87]

In the *logic* operator we build on the notion of partial ordering to arrive at the lattice-theoretic definition of a boolean algebra. The definition requires three postulates in addition to the ones given in the *zeroth* epoch:

1) Each pair of elements a and b has a greatest lower bound i such that $i<a$, $i<b$, and $y < i$ for all y satisfying $y<a$ and $y<b$.
2) Each element a has a complement a' such that if $x < a$ and $x < a'$, then $x = 0$, and if $x > a$ and $x > a'$, then $x = 1$.
3) if $a < b'$ is false, then there is a non-zero element x such that $x < a$ and $x < b$.

The above definition can be shown to be equivalent to the more usual definition of a boolean algebra in terms of the operators + and *, where $a<b$ is equivalent to $a*b=a$ and $a+b=b$.[88] The lattice-theoretic definition subjects these two operations to the conditions

1) $a*1=a$, $a+0=a$, for all a; and
2) there exists for all a a unique a' such that $a*a'=0$, and $a+a'=1$.

These operators will have a simple interpretation in terms of regions in a two-dimensional plane: $a*b$ is the intersection of regions a and b, $a+b$ is the union of a and b, and a' is the complement of a. The above conditions then make sense if we interpret 0 as the null set, 1 as the universal set, and $<$ as set inclusion. It can be shown that the transitivity property of $<$ implies that $a*(b+c) = a*b +a*c$ and that $a+(b*c) = (a+b) * (a+c)$; this is called the distributivity property of * and +. The commutativity of + and *, defined by $a*b = b*a$ and $a+b = b+a$ is obvious from our interpretation of * and +. The associative property, defined by $(a*b)*c = a*(b*c)$ and $(a+b)+c = a+(b+c)$, can also be derived from the lattice-theoretic definition. Including these latter three properties completes the usual definition of a boolean algebra.[89]

The reason we refer here to logic is that we can also interpret * and + as the logical operators *and* and *or*, respectively, in a space of propositions, and the complement operator ' as negation; in fact, propositional logic satisfies the definition of a boolean algebra, with 1 interpreted as the proposition that is always true and 0 the proposition that is always false.[90]

In *null set* axiom there is just one set, the null or empty set. The empty set has no elements, and is usually written 0. To insure that the null set is unique, another axiom must arise here, the axiom of extensionality, which says that two sets A and B are equal if and only if for all x in A, x is also in B. There is a simple proof that shows that the axiom of extensionality implies that the null set is unique. Notice that we cannot form the set of all sets here in analogy with 1 in row 0 of the zeroth epoch, since such a set does not exist as it turns out. [91]

null set requires *logic* for its formulation because of the above-mentioned proof. Moreover, that formulation itself must be regarded as the introduction of several concepts that are necessary for the transition from propositional to first-order logic, namely the notion of existence, which allows us to use quantifiers, and the notion of a variable. In appendix A we give a list of axioms and rules of deduction that formally specify first-order logic as a language; these axioms and rules follow naturally from the concept of boolean algebra interpreted as propositional logic and from the concept of variable. First-order logic systems also require a domain of objects referred to by the above-mentioned notions of existence and variable, which here is defined to be the domain of sets. These sets are related to one another by a pair of primitive relations, equality and set membership, of which the latter must also be introduced here.

The axiom of *pair* says that given any objects x and y, there is a set Z such that for all elements z in Z, either $z = x$ or $z = y$. Taking our objects x and y in the axiom of pair to be 0, we can generate a set written $\{0,0\}$ or $\{0\}$, which contains the one element 0 and satisfies the requirements of that axiom. Clearly, *pair* requires *null set*.

The axiom of *union* enables us to generate sets with more than two elements. It says that given a set X, there exists a set whose elements are the members of the members of X. This axiom allows us to define the union of sets, for example the set 0 U {0}, and so on. The reason *union* must come after *pair* is that before *pair* there is only one set, the empty set.

The axiom of *infinity* says that an inductive set exists. Intuitively, we can understand this axiom to mean that there is a set N' = {0, {0}, {0,{0}}, …}, where the three dots indicate that we continue adding elements in this way endlessly. The above set, after the notion of order among elements is defined, will be used to generate N, the set of natural numbers, which is written { 0, 1, 2, …}. *infinity* requires *union* for the construction of N'; before *union* it was not possible to build a set with more than two elements.

The axiom schema of *replacement* generates axioms for properties that determine a unique element y corresponding to each element x. A property is a statement, such as x = 1. For different properties, we get different axioms; hence the name schema, since the schema generates axioms. For a given x, and a property that relates a unique y to that x, the corresponding axiom says that for every set A, there is a set B such that if x is in A, then y is in B. This axiom therefore permits us to start with one set, A, and define a second set, B; in other words, it allows us to construct new sets from old. For example, we can construct sets larger than N' that will become the so-called ordinals, after a concept of ordering has been established.[92] *replacement* must follow *infinity* so that such large infinite sets can be generated.

In *power set*, we need the notion of a subset of a set. For that, we use the axiom schema of comprehension, which can be derived from the axiom schema of *replacement*. It says that given a set A, there is a set B such that there is an x in B if and only if x is in A and some property involving x is true. Each axiom so generated defines the elements of the set B given a set A, and those elements are also elements of A; for this reason this axiom is also called the subset axiom. The *power set* axiom itself says that given a set Z there is a set P such that if Y is a subset of Z, Y is an element of P. The power set of a set is the set of all subsets of that set. *power set* must come after *replacement* so that we can define subset.

The next three operators establish ordering within an arbitrary set by means of the set membership relation. In effect, these operators also allow us to construct the universe of sets as a cumulative hierarchy, a construction that is often referred to as the standard model of set theory.[93] The hierarchy starts with some set V_0, typically the null set, and takes the power set of V_0 to get V_1, then takes the power set of V_1 to get V_2, and so on, until reaching V_ω, where ω is the first transfinite ordinal, at which point it takes the union of all $V_{\alpha < \omega}$; then $V_{\omega+1}$ is the power set of the previous union, $V_{\omega+2}$ is the power set of $V_{\omega+1}$, and so on until the next transfinite limit ordinal ω + ω is reached, where we take the union of all $V_{\alpha < \omega+\omega}$, and so on forever.

x ∈ y → y !∈ x establishes the asymmetric nature of the set membership relation. The symbol ∈ means *is an element of*; in other words, this operator specifies that if *x* is a member of *y*, then *y* is not a member of *x*. This is the standard set-theoretic approach to natural numbers, introduced by von Neumann, according to which each natural number is the set of all smaller natural numbers, and according to which the strict ordering on the set of natural numbers is given by the set membership relation: $x \in y$ if and only if $x < y$. This approach has the advantage that it can be generalized to account for ordinal numbers beyond the natural numbers, the transfinite ordinals, and transfinite induction as well. The ordinals also serve as indices to the levels of the universal set hierarchy. It is worth emphasizing that each natural number in this definition is itself a set, namely the set of all smaller numbers, and so informally we can say that each natural number *n* within its subsets contains n – 1 copies of 0, n – 2 copies of 1, … and a single copy of n – 1. Reals have an analogous construction.

In $x \in y, y \in z \rightarrow x \in z$, the set membership relation is defined to be transitive. Taking the natural numbers as an example, $1 \in 5, 5 \in 8 \rightarrow 1 \in 8$.

In $x ! \in x$, sets are prohibited from being members of themselves. Allowing sets to be members of themselves leads to logical difficulties such as the so-called Russell's paradox, which involves the set A of all x such that x !∈ x: the paradox is that if A !∈A, then A ∈ A, and if A ∈ A, then A !∈A. Such pathological cases were the motivation for the axiomatization of set theory in the beginning of the twentieth century.

Using the last three operators as well as the nullset, pair and union axioms it is possible to prove the axiom of foundation, which says that every non-empty set contains a member disjoint from itself; equivalently, it says that every set is well-founded, meaning essentially that it is impossible to form a cycle of memberships such that a set becomes a member of itself, or a member of a member of itself, or a member of a member of a member of itself, and so on. A proof is given in Appendix B that uses the contrapositive and therefore relies on the law of the excluded middle. Conversely, it is possible to prove $x \in y \rightarrow y ! \in x$ and $x ! \in x$ from the nullset, pair and foundation axioms, but not transitivity. The further constraint that sets be transitive is also a standard requirement in models based on ZF set theory, although it is not an axiom. For example, Goedel's constructible model and Cohen's forcing models, which have been used to settle questions about the so-called continuum hypothesis and the axiom of choice among other things, are transitive models.

If one thinks of the last three operators in terms of the ordering relation <, then the order of their appearance can be explained using the same arguments as given for rows 7, 8 and 9 of the *zeroth* epoch. Another question is whether these three operators could have occurred earlier, say after the axiom of *union*. The answer to this question seems to be that they must appear here because their purpose is to allow only sets of the sort that are in the cumulative hierarchy, that is, well-founded and transitve sets; in other words, their purpose may be to restrict the universe of sets to the cumulative hierarchy, which becomes possible only after the *power set* axiom.

Quantum Logic Epoch

It appears to be appropriate at this point in the matrix to introduce quantum logic. Just as in the zeroth row of the *logic* epoch the concept of boolean algebra was defined, which as a distributive ortholattice is the underlying structure of classical logic, here one would expect an algebraic structure underlying the logic of quantum theory to emerge. Unfortunately, there appears at present to be no consensus as to the exact definition of a mathematical structure analogous to boolean algebra that would underly quantum logic. Quantum logic is frequently defined as the set of closed subspaces of a Hilbert space, which are known to form an orthomodular lattice.[94,95,96] The relevance of the lattice structure of the closed subspaces of Hilbert space for quantum theory was pointed out by von Neumann and Birkhoff in 1936; since then the model of quantum logic has gone through a series of generalizations, from orthomodular lattice to orthoalgebra to effect algebra, and this is still an area of active research.[97] Therefore, we use the term orthomodular structure or lattice here as representative of the true quantum logic structure, whatever that turns out to be.[98] A key point is that boolean algebras are themselves orthomodular lattices, distinguishing themselves from the orthomodular lattices of quantum theory in that they are distributive. Another key point is that Hilbert spaces in the *quantum logic* epoch play a role analogous to the role played by sets in the *logic* epoch, and the fact that subspaces of Hilbert spaces form an orthomodular lattice that somehow characterizes quantum logic is analogous to the fact that subspaces of sets form a boolean algebra, that is, a lattice structure that characterizes classical logic.

There is further evidence that row 0 is the appropriate place to introduce quantum logic. First, the *negation* operator forms a boolean algebra (with *symbol* as identity), as such an orthomodular lattice, indeed the simplest possible one, and *negation* is located in row 0. Second, we note that the causal structure of spacetime itself, imposed by special relativity, forms an orthomodular lattice, with orthocomplementation of a given point defined by spacelike separation as described in Appendix G, and it appears reasonable to introduce this causal structure in row 0 of the *physical universe* epoch.[99,100] Third, just as other row 0 operators, both boolean algebra and quantum logic can be interpreted to have cyclic behaviour: boolean algebra in that for each element a of a boolean algebra $a*a=a$ and $a+a=a$; quantum logic in that each closed subspace of a Hilbert space can be uniquely associated with a projection operator, i.e. an operator P such that $P^2 = P$ --- furthermore, a projection operator is a so-called effect operator in the sense of effect algebra mentioned above. Fourth, in the next operator, *R,* it will be necessary for an underlying quantum structure to be already present, since it is presumably in that operator at the latest that the distribution of the primes must be fixed and since that distribution is known to have characteristics associated with a quantum system, as discussed below.

The study of Hilbert space is at the core of a branch of mathematics called functional analysis, which uses the language of geometry --- vectors, basis vectors, orthogonality of vectors, length of vectors, and so on --- in a generalized sense to describe and analyse functions. In this language certain types of functions can be viewed as vectors (in an abstract sense detailed in Appendix J) in an infinite dimensional Hilbert space. In finite dimensions, Hilbert spaces are the starting point for the study of geometry itself, insofar as R^n, the n-dimensional space resulting from n copies of the reals, is a Hilbert space.

Simple quantum systems can also be studied by finite Hilbert spaces; here the vectors are interpreted as quantum states. In the general case, it is well established that infinite dimensional complex Hilbert space is the natural structure for the description of quantum theory.

The standard definition of a real Hilbert space requires that we define the natural numbers N, the integers Z, the rationals Q, the reals R, vector space, scalar product space, norm on a vector space, metric space, Cauchy sequence, and finally Hilbert space. However, since R is itself a Hilbert space this definition is inappropriate for the purposes of generating R unless we assume the Hilbert space R is different from the R used in the Hilbert space definition or that the definition is recursive; instead of pursuing either of these latter ideas, we assume that quantum logic makes the construction of R as a real Hilbert space possible just as boolean algebra, the structure underlying predicate logic, made the construction of the empty set possible. For details on the standard set theory construction of N, Z, Q, and R we refer the reader to standard texts on set theory and real analysis while pointing out that standard Zermelo-Frankel set theory was established in the previous epoch.[101, 102] For details on the definition of a Hilbert space starting from the concept of a vector space and some highlights of elementary Hilbert space theory, see Appendix J.

In *R* the space of real numbers is constructed; R is the simplest Hilbert space, just as the empty set is the simplest set. The somewhat involved proof that R is in fact a Hilbert space is given in Appendix J. The definition of R also requires that the so-called axiom of choice arise here simultaneously. The axiom of choice is usually added to the axioms of Zermelo-Frankel set theory, in which case the theory is known as ZFC. The axiom says, in one of its four or five equivalent forms, that for every set there exists a so-called choice function that picks out an element of every subset of that set. It turns out that this axiom cannot be derived from the other axioms of ZF theory and that it is needed in a variety of contexts, such as in the proof that every vector space has a basis. Another context where it makes a difference is the following: there is a model of ZF (without the axiom of choice) in which a theorem says that there exists an infinite set of reals with no countable subsets.[103] Because of such results, the axiom of choice is considered a standard axiom of set theory, and because it is an independent axiom it would appear awkward to lump it into *quantum logic*. The reason that it must appear here at the latest is that it is required in the proof that R is a Hilbert space; in particular, the invocation of the greatest lower bound principle in the proof that all Cauchy sequences converge in R requires the axiom of choice, as mentioned in Appendix J. It is also relevant that in so-called intuitionistic models of set theory one can prove that the axiom of choice implies the law of the excluded middle, which appears to be consistent with the use of boolean algebra as a building block for logic in the prebang epochs as opposed to a so-called Heyting algebra, a generalization of boolean algebra in which the law of the excluded middle does not necessarily hold.[104]

The construction of R may be related to the following fact: there is a striking connection between primordial mathematics and quantum theory, namely that successive zeros of the Riemann zeta function are correlated in *precisely* the same way as eigenvalues of random

Hermitian matrices of the so-called Gaussian Unitary Ensemble operating in a Hilbert space, and these zeros are intimately related to the distribution of the primes among the natural numbers.[105,106,107] This distribution may have been determined in *R*; at least one can say without pause that the necessary quantum structure is in place. Pursuing this thought, we note that the natural numbers N are peculiar in that as one progresses from N to the integers Z to the rational numbers Q to the reals R, at each stage the definition of N changes. For example, the number one as it is usually defined is the set of all smaller integers, i.e. {0}, a set that exists in *pair* of the *logic* epoch. In order to define the integers, which include positive and negative numbers and 0, the standard definition uses equivalence classes of ordered pairs, whereby the equivalence class corresponding to a given integer contains ordered pairs whose difference is equal to that integer, e.g. –3 would contain pairs (1,4), (32,35), etc. The number 1 as an element of Z is represented by {(1,0), (2,1), …}. In this sense, N is not a subset of Z; of course, it is possible to define N in terms of elements of Z so as to make it a subset of Z, but there is no natural way to see the original N from within Z, nor as it turns out from within Q nor R without in each case a redefinition of N. If one thinks of operators in the matrix as actually building real entities, as one must, this variable aspect of the natural numbers allows one to imagine them changing in essence at each step, or becoming something else, and in particular, one can imagine that, before they became a part of the reals as we know them, the distance between them was not defined and therefore the distribution of the primes was not fixed. One can look at the reals in the same fashion: they exist as part of the universe of sets implicit at the completion of the *logic* epoch, and yet their topology is not fixed until the current operator acts.

In *R** a second one-dimensional space is generated, R*, the Hilbert space dual to R. Loosely speaking, the dual of a real Hilbert space H is the set of all linear mappings from H to the set of real numbers. As mentioned in Appendix J, the dual of a Hilbert space is in a certain sense equal to the original Hilbert space. This operator is analogous to 0 in the zeroth epoch and *pair* in *logic*, each of which arise through an operation on the preceding operator: negation in the case of 0, and pairing in the case of *pair*.

In *C* the one-dimensional complex Hilbert space, that is, the space of complex numbers is constructed (see Appendix J regarding complex numbers). As a set, C is isomorphic to R^2, the space spanned by two copies of R, which means that one can define a one-to-one correspondence that relates each element of R^2 with an element of C and vice versa. Such a mapping just maps points in the R^2 plane (x, y) to points in the complex plane, where x becomes the real value of a point in C, and y becomes the imaginary value. This operator must follow R* because before R* there was only one Hilbert space, R, and dualization is required to build a copy of R. Taking one space to be composed of two spaces is analogous to *union* in *logic*.

In $C^{n+1} = C^n \times C$ the existence of every finite-dimensional complex Hilbert space is established. Each new copy of C in the product is generated by taking the dual of a C already in the product. This operator is analogous to *infinity* in *logic*.

In l^2 the infinite-dimensional Hilbert space of square-summable sequences of complex numbers is generated (see Appendix J). This operator requires the proceeding operator since its definition implies making a set out of those vectors **v** in C^n such that the norm of **v** is finite in the limit as n goes to ∞. Similarly, *replacement* in *logic* is used to generate transfinite sets such as N + N.

In *spin network* a particular infinite-dimensional Hilbert space is generated. This space may be an infinite-dimensional Hilbert space isomorphic to l^2 from quantum loop gravity, the space of "connections modulo guage transformations" $L^2()$.[108] A basis for this space is given by so-called spin network states, which are functions from spin-networks to the underlying space of connections. A spin-network is a collection of vertices and edges running between those vertices with values attached to the edges that are constrained according to a so-called group representation. In the original definition of spin-network due to Penrose, these values were associated with the half-integer spins of elementary particles.

Loop quantum gravity is a theory of gravity developed in the last 15 years that attempts to construct spacetime by starting from a Hilbert space. Its biggest success has been its ability to calculate the area of a black hole event horizon, which is known from a result by Hawking to be proportional to the entropy of the hole. Ashtekar, one of the pioneers of loop quantum gravity along with Smolin, Rovelli, Gambino, Pullin, Thiemann, Lewandowski, Baez, and others, describes the theory in terms of "polymer-like excitations", referring to the role of the edges in the construction of the area operator used in the above-mentioned calculation --- this analogy is significant given that *spin network* occurs in the same row as *polymer*.

In *ordered spin-network* an ordering on spin-networks is established. This ordering appears to be given by the total area of a spin-network, calculated as a particular sum over all of the edges of the network. Such an ordering would be analogous to the ordering of the physical world, in which all systems evolve from states of lower entropy to states of higher entropy according to the second law of thermodynamics.

In *spin-foam* a generalization of a spin-network is defined consisting of polygonal faces in addition to edges and vertices. A spin foam defines so-called amplitudes for transitions between spin-networks. As far as the author knows, as yet no spin-foam models have been constructed that are abstract in the sense that they do not depend on an underlying manifold, a dependence that Baez assumes will be absent in a correct theory.[109]

In *spin-network evolution* the amplitude for the transition of a spin-network from one state to another is given by a sum over spin foams.

Structure of the Universal Matrix

We have been guided in the construction of the matrix in Table III by a repeating pattern, in which each row of the matrix is seen to have a particular functionality. The columns therefore also have a pattern, which could be described as a singlet followed by a set of three triplets. The first triplet establishes a foundation, the second triplet creates the building block and uses it to create a scaffolding, and the third triplet consummates the construction. The triplets are well-delineated by the unambiguous start, middle, and end pattern of triplets one and three. There is also a similarity in functionality between the first, second and third triplet and the first, second and third row of the first triplet, a correspondence which is particularly evident in the resolving nature of the third triplet and the third row of the first triplet.

It turns out that the above-mentioned structure can be understood by treating the contents of Table III as output from a small program, as anticipated in some sense by Binnig and in some sense by Wolfram, and perhaps others to whom the author apologizes for his ignorance.[110,111] We shall return to this fact below, after some necessary mathematical preliminaries. We also note that it is possible to represent some aspects of a single loop through an epoch geometrically, as shown in Figure 7; by gluing together the edges of the three outer triangles one can form a tetrahedron, which has been taken as a basic structural unit of space in some models of quantum loop gravity.

Mathematical Foundations

Recent studies on the foundations of mathematics have centered on the roles of category theory and set theory. Category theory provides a framework for understanding the inner workings of a process or mathematical structure. Its power derives from the morphism and the composition of morphisms; by defining appropriate morphisms between objects it is possible to isolate what is essential about the structure one is trying to understand, whether that structure is simple, as in the case of a set, or that structure is itself an abstraction of the relationship between other structures, as one finds when studying the relationships between algebraic and topological structures. In fact, category theory is so versatile that soon after categories were discovered attempts were made to use it as a foundation for all of mathematics.[112]

Set theory, on the other hand, also has a claim to be foundational in that all branches of mathematics can be expressed in terms of it, *aside from category theory*. The problem besetting set theory, and category theory as well, is that it depends on several concepts which are deemed to be primitive and not to be defined rigorously, such as a the notion of what a set is or an element of a set. Even after set theory was reformulated in terms of category theory,[113] one could still make the point that category theory was still lacking as a foundation in the sense that it could not account for such primitives.[114]

It turns out that there is a third perspective on the foundations of mathematics as well, given by so-called constructive type theory, which is reflected in the *universe*, *life*, and *algorithm* epochs. In the last thirty years a number of approaches to a rigorous theory of the semantics of computation within the context of Scott's domain theory have been developed.[115] These theories, called type theories, provide a means to precisely define the semantics of a computation, so that the notions of soundness and completeness can be applied to a programming language, just as these same notions can be applied to first-order logic, where soundness means that any theorem is a logically valid statement and completeness means that any logically valid statement is a theorem; it complements earlier work in constructive mathematics spearheaded by Bishop as well as results of Goedel, Kleene, Curry and others that showed that computation essentially *is* deduction. Although the main focus in this area has been on proving properties of software systems, Aczel has shown that there are good prospects for formulating set theory in terms of constructive type theory (based on work by Martin-Loef), so that constructive type theory too appears to be a viable alternative as a foundation for mathematics.[116,117]

The above discussion suggests that the matrix has three tiers of mathematical theory, one starting in the *zeroth* epoch, another starting in *physical universe*, and a third starting in *negation*. This idea is reinforced by the fact that 1, !1 = 0, and !!1 = 1 of the *zeroth* epoch forms an instance of an orthomodular lattice, in particular a boolean algebra, as does the *negation* protolanguage, while *physical universe* involves a causal set, also an instance of an orthomodular lattice, and each of these lattices lie at the beginning of its respective tier. Also, there is a corresponding language in each of the three tiers: first-order logic, programming language, and natural language, respectively. The typed lambda calculus, briefly discussed in Appendix C, is a prototypical programming language with a number of variants; it is used for example in the above-mentioned implementations of constructive type theory.

Each of the above-mentioned languages can express everything that its predecessor can: first-order logic can be implemented in software and it can be described by the topos of category theory; likewise, the typed lambda calculus, and therefore any computation, can be described by a so-called cartesian closed category.[118] Furthermore, it appears that each language describes a bigger universe, a conclusion we draw from the fact that category theory allows for the existence of entities not possible in ZFC set theory, a fact to which we turn next.

The universe of sets described in the discussion of the *logic* epoch contains a series of infinite sets of ever-increasing size, or cardinality. The ontological status of these infinite sets is the same as the ontological status of finite sets in ZF, the environment established by the *logic* epoch: they exist. One can define arithmetic operations on them that are just as valid as the analogous operations on finite sets. However, it turns out that there are infinite sets still bigger than any of the aforementioned sets, so big that the repeated power set operation performed in the universal hierarchy never reaches them. These sets are the so-called large cardinals, whose existence cannot be proved in the pre-bang universe of ZFC, but which can be shown to exist if one adopts certain so-called large cardinal axioms, of which there are dozens[119]; it is conjectured by mathematicians including Woodin, a pioneer in the field, that these axioms themselves render a wellordering of the large cardinals (by direct or relative consistency implications), that is, that they allow one to order the large cardinals in a hierarchy.[120] The ontological status of these inaccessible cardinal sets, which have deep connections to the reals, is somehow different from other sets in that their existence can only be established in higher set theory, i.e. using large cardinal axioms; one must hasten to add that as of yet there is no natural formulation of these large cardinal axioms in category theory, that is, there is no formulation beyond a *literal* translation so-to-speak from higher set theory, although it would seem appropriate that these axioms have an expression in terms of categories. It also appears that constructions analogous to large cardinals can be shown to exist in constructive type theory.[121, 122]

The foregoing suggests that the matrix is a hierarchy of tiers of mathematical theory, with each tier subsuming all previous ones, just as the transitive sets of the universal hierarchy of sets described in the *logic* epoch subsume all previous sets in the hierarchy.

Logic and Truth

The category theory approach to logic involves a particular kind of category called a topos. A topos has objects called 0 and 1, products and sums suitably defined, and a way to look at subobjects or parts of objects, all of which is accomplished with appropriate objects and morphisms, the machinery of categories as we have seen. In addition, a topos must have a so-called subobject classifier; this object has a natural interpretation as a truth-value object and while it may be two-valued, in general it may have an infinite number of values. For this reason, the topos embodies the structure of intuitionistic logic, by which we mean a logic in which the law of the excluded middle does not hold. Since the topos is the natural home for logic in category theory, one must view this general multi-valued truth scheme as a significant evolution from the two-valued logic of set theory; that is, one must not regard this scheme as a sheerly mental construct, since the matrix has shown that category theory is part of the fabric of the universe.

The step from a two- to a multi-valued truth scheme is all the more significant given that truth and falsehood are concepts that appear immediately in the matrix, truth to the fore. What is more, they hold center stage thereafter, first as cornerstones from which the concept of ordering is built, then as indispensable notions of propositional and first-order logic, and from then on as elements common to all lattices, including Hilbert lattices, the causal set of space-time, and the boolean algebra of *negation*. Even so, it turns out that the role of truth in the hierarchy has a still broader interpretation, as we shall now discuss.

The notion of truth implicit in the soundness and completeness theorems of first-order logic is due to Tarski and dates from the 1930's.[123] This celebrated definition of truth evolved into the currently accepted definition of the truth of a sentence in a model, as discussed in Appendix F. It was motivated in essence by the so-called liar's paradox, which arose from the following statement attributed to the Greek Eubulides of Miletus of the fourth century B.C.: *the statement I am now making is a lie*. Taking this puzzling statement very seriously and rightfully so, Tarski concluded that no language has a consistent concept of truth, and that a given language requires for the definition of truth a second, overarching language, or metalanguage, which, in addition to being able to express all statements in the original language, also has a criterion for deciding which statements are true in the original language. In his formal definition of truth in ZFC, this criterion is essentially the criterion one would intuitively expect, and it handles statements analogous to that of the liar. If one accepts Tarski's formulation, truth is a hierarchical concept: one imagines a series of languages, each one having the full expressive powers of the previous one as well as a criterion for deciding what is true in the previous one.

It may be that Tarski's definition is in fact *the* definition of truth expressed in first-order logic from the perspective of ZFC set theory, using natural language as the metalanguage. One might therefore expect there to be a corresponding definition of truth in computation. As far as the author can see, little has been done in this regard: the above-mentioned attempts to found set theory on the basis of computation naturally work with a two-valued truth theory since that is appropriate in ZFC, and to the author's knowledge no attempts have been made to precisely define truth in terms of computation using natural language or category theory as a metalanguage. On the other hand, Kleene argued that corresponding to a partial recursive predicate the natural logic is a three-valued logic, with values *true*, *false* and *undefined* or *unknown*; in domain theory the undefined value is called *bottom*, written $\perp$. The use of a three-valued logic reflects a refinement in type theory of the two-valued concept of truth in ZFC, which in turn would be further refined in terms of the topos of category theory.

In conclusion one must say that the fact that the hierarchy of mathematical theories provides metalanguages as required by Tarski for a definition of truth is difficult to dismiss as coincidence given the above-mentioned preeminence of truth in the hierarchy. Indeed, if in addition one considers the fact that as one ascends the hierarchy the mathematical theories become more expressive, with progressively more refined sets of truth values, it is natural to speculate that the very purpose of each mathematical tier is to refine the notion of truth.

Theology

As mentioned above, there are operators that precede the matrix. It turns out that these operators can be used to give a mathematical proof of the existence of God. Before presenting this proof, we first briefly discuss traditional arguments for the existence of God.

First Cause Argument

The presentation of the *quantum logic* epoch given here is much more uncertain than the presentation of the *order* and *logic* epochs. Major revisions to this epoch may be necessary to properly account for the origin of space, the time evolution of quantum mechanical states, and the reconciliation of quantum mechanics with geometry. These facts notwithstanding, there can be no doubt that as the theory is refined the first two prebang epochs will stay largely intact, since they are entirely mathematical and they fit the overall singlet-plus-three-triplets pattern of the matrix. Furthermore, the implicit references to orthomodular lattices before and after the quantum logic epoch and the cyclical behavior quantum logic shares with all other operators of row 0 provide particularly formidable evidence that the description given here of the quantum logic epoch is on the right track.

In spite of the above-mentioned shortcomings in our presentation, the existence of a hierarchy of three mathematical theories in the matrix, one generated before the bang and two after, leads one to conclude that there is an order that logically precedes the physical world. Moreover, that order manifests itself as an evolution from the simple to the complex, pointing toward a starting point from which everything begins. In theology there is a long-standing argument for the existence of God[1] based on first cause, which contends that the chain of causation in the world must start somewhere, and so one postulates the existence of a prime mover. Against this argument one could counter that the chain of causation becomes meaningless when one tries to make it transcend the physical world.[124] However, the hierarchy provides an obvious way to meaningfully extend the chain of causation beyond the physical world so as to invalidate the aforementioned counterargument, and thus one is led to an inevitable conclusion: God exists. He or She may be referred to as Allah, Brahma, God, Naam, Tao, Yahweh, or otherwise, but in any case there is a Divine Being ultimately responsible for what we see in the world.

Design Argument

The hierarchy supports in addition another long-standing argument for the existence of God, the so-called argument from design, according to which the existence of a design implies the existence of a designer. Heretofore the stumbling block of this argument has been that one could dispute whether a design exists.[125] In what follows we establish unambiguously that a design in fact does exist, at the same time arriving at a still stronger result on the existence of God, a proof using the mathematical apparatus of model theory.

We start by noting that the three tiers of mathematical theory in the hierarchy conform to the first triplet pattern of an individual epoch: category theory is a merging of set theory and computation theory, its objects being like sets and its morphisms like recursive functions. This appearance of the first triplet pattern at another scale leads one to consider a design in terms of functions defined by recursion, that is, functions that reference themselves in their definition.

λ-calculus and Combinatory Logic

There is a natural language for talking about recursion, the type-free lambda calculus developed by Church in the 1930's. This simple, yet potent language consists of terms that are either names, function abstractions, or function applications. A function abstraction is a specification of a function of a single untyped input term --- function here is used not in the set-theoretic sense as a set of ordered pairs, rather in a looser sense conveying the idea of operation, just as we have used the word operator here from the outset, beginning with the *negation* operator of glossogenesis theory; a function application is an instance of a function abstraction applied to a term. As an illustration,

[1] Translators are instructed to translate the word *God* according to the concept of God predominant among speakers of the target language. They are also instructed to translate this footnote, which is in the original English text of this article as well.

one would say *sin x* is an abstraction and *sin π* is an application. There is one essential rule governing manipulation of expressions in the lambda calculus, the so-called β-rule, which gives the syntax for function application. A more detailed description of the language in Appendix C.

Slightly before Church's work on the lambda calculus, first Schoenfinkel and then Curry independently discovered a still more primitive language along the same lines called combinatory logic. Combinatory logic also uses the notion of terms that are applied to one another in the sense of operator used above. One difference between combinatory logic and the lambda calculus is that there is no concept of bound variable in combinatory logic, that is, there is no natural way to achieve function abstraction; in this sense combinatory logic bears the same relation to lambda calculus as propositional logic bears to first order logic --- as discussed above, first order logic contains variables that are bound by universal quantifiers and such variables are not present in propositional logic. The key point is that combinatory logic simulates the process of substitution for a variable without introducing variables, and in this sense it is primitive and a necessary forerunner to both the lambda calculus and first order logic.

A description of combinatory logic is given in Appendix D. The essence of the language is that there are three constants, **I**, **K**, and **S**, from which all functions expressible with the lambda calculus can be generated. They are defined as follows: **I**x = x; **K**xy = x; and **S**xyz = xz(yz), where x, y and z are variables. The meaning of **I** is that it reproduces the input; it is called the identity operator. **K** reproduces the first term it sees and discards the term following that first term; it is also called the truth operator, and not merely due to convention: (**KI**x)y = **I**y = y motivates the definition of the falsehood operator **F** ≡ **KI**, since **KMN = M** and **FMN = N** reproduce the behaviour of the conditional construction (if-then else), whereby the order of **M** and **N,** whatever their meaning, is intrinsic due to the nature of the conditional. **K** also plays the role of a constant function when viewed as a function of the second term it sees. **S** composes dependence on the third term it sees: **S**ghx = gx(hx). The best way to understand why these terms are defined as they are is to work through the so-called abstraction extraction algorithm presented in Appendix D that starts with a function abstraction and systematically removes dependence on variables until there is nothing but a series of term applications. It turns out that **I**, **K** and **S** are the only constants needed to remove dependence on variables from an arbitrary function.

Ix = x	objectify
KIx = **I**	abstract
SKIx = **K**x(**I**x) = x	apply

Table IV

Zeroth Recursive Call

The constant **I** can be written in terms of **K** and **S**, but we include it here since the operators **I**, **K** and **S** are all used in the zeroth recursive call in the series of calls that result in the hierarchy. In Table IV, we show the zeroth call. First the identity operator **I** is defined; it requires one input, which we view as establishing that the concept of input is required even for the zeroth row. Next, the truth operator **K** is defined; it requires two inputs and so cannot be defined until after **I**. In fact, **K** specifically refers to **I** in its definition as a particular case, operating on **I**x to yield **KI**x = **I**. Likewise, **S** requires three inputs and so cannot be defined until after **K**. As **K** does, and indeed all operators in the entire hierarchy, it operates on the output of the operator preceding it and so is introduced specifically as an operation on **KI**x: **SKI**x = **K**x(**I**x) = x. In light of the fact that **SKK** = **I**, these definitions for **I**, **S** and **K** given by specific formulas (and not by the general formulas **K**ab = a and **S**abc = ac(bc)) show that the operators **I**, **S** and **K** treat one another and themselves in the same way they treat the variable x, and indeed we hold that this is a reason the specific formulas must originate with the operators themselves and not say as some sort of horizontal extension of the operators introduced as general formulas.

The terms **S, K,** and **I** are building blocks from which arbitrary functions can be formed. At the same time, the operators in Table IV as a group can be interpreted as building the notions of function abstraction and application themselves on the basis of the identity function **I**x = x, which can be read off vertically from the right-hand side of the operators in Table IV. This interpretation goes as follows. First, the step from input to the term x via **I**x is a process of *objectification* through which the input becomes real in some sense… Next, the step (effected by the truth operator **K**) from x to **I** is an *abstraction* capturing the essence of the input in a process for replicating it… Finally, the step from **I** (= **SKI**) to x establishes a link in the opposite direction, from process to object, by merging process with object, i.e. by *application* of process to object. These three steps, which we dub *objectify*, *abstract*, and *apply* and with which we associate the terms, abstractions and applications of the lambda calculus, are the building blocks of the next recursive call. These three steps, starting from the term application of combinatory logic and ending with the function application of lambda calculus, in effect define lambda calculus.

At the same time, there is a third way to look at Table IV: the above intepretation can be taken to consummate the definition of **I** in the sense that it gives **I** a semantics, that is, a meaning. The same can be said of **K** and **S** indirectly. Therefore, the definition of **I** and **K** does not become fully effective until the appearance of **S**. This is in line with our previous argument that the explicit appearance of the formulas **I**x=x, **KI**x=**I** and **SKI**x=x in Table IV is necessary in order to establish that function and function argument are treated equally, since that argument implies that the definitions of **I** and **K** are actually not complete until the appearance of **S**. In summary, there are three levels of definition in Table IV: 1) at the object level, as explicit syntactic definitions using combinatory logic, 2) at the abstraction level, viewed as a unit defining *objectify*, *abstract* and *apply*, and 3) at the application level, as a merging of the abstraction level definition with the object level definition that renders the object level definitions meaningful.

An issue that requires addressing here in more depth is the difference between combinatory logic and the variant of lambda calculus that is appropriate for the next recursive call, the λη-calculus. Using a standard mapping between the two, it can be shown that a reduction of terms in combinatory logic always results in a corresponding reduction in the λη-calculus, but not vice versa. This is because there are reduction rules present in λη-calculus that are not found in combinatory logic, the so-called ξ- or weak extensionality rule and the η-rule, both given in Appendix C. These rules give λη-calculus an important characteristic, namely that two different functions or operators that produce the same result with respect to a given input are considered equal; in λη-calculus an operator is like a black box: the output alone from a given input defines the corresponding operator and what happens *inside* the operator so to speak is not interesting. Such definitions can be said to be definitions by *extension*. On the other hand, the operator definitions of combinatory logic are by *intension*, which means that the definitions explicitly give the algorithm for generating output. In combinatory logic how an operator generates output from a given input is as important as the output itself with respect to the definition of an operator; in particular, if Mx = M'x for arbitrary x, then it is not necessarily true that M = M' in combinatory logic, whereas in λη-calculus it would be true that M = M'. Apparently operators rapidly become complex in the hierarchy and the algorithms they use rapidly become inscrutable, so it soon only becomes practical to define operators solely by their output. Therefore, the λη-calculus is required in order to define all but the simplest operators, evidently all but those in Table IV.

objectify	*(objectify ◦ objectify)*
abstract	*(objectify ◦ abstract)*
apply	*(objectify ◦ apply)*
re-objectify	*(abstract ◦ objectify)*
re-abstract	*(abstract ◦ abstract)*
re-apply	*(abstract ◦ apply)*
asymmetrize	*(apply ◦ objectify)*
transform	*(apply ◦ abstract)*,
reflect	(*apply ◦ apply*)

Table V

First Recursive Call
In Table V we show nine operators; we arrived at these nine operators by recognizing that they form the pattern of an epoch of the matrix. In order to get from Table IV to Table V, it is necessary to regard the three operators *objectify*, *abstract*, and *apply* as a unit, or function, and then allow that function to take itself as input. In so doing, in some sense we get nine compositions, starting with *objectify ◦ objectify*, *objectify ◦ abstract*, and *objectify ◦ apply,* which we write simply as *objectify*, *abstract* and *apply* since the meaning of the above compositions is unclear and since by the nature of λη-calculus operators are defined by their output: how the output is generated does not matter --- that said, *objectify* arose from the identity operator **I** and therefore here, where we are closest to combinatory logic, it is reasonable to understand it as an identity operation when

composed. Next, when we *abstract* each of the operators *objectify*, *abstract*, and *apply*, we get three new operators, *re-objectify (abstract ∘ objectify)*, *re-abstract (abstract ∘ abstract)*, and *re-apply (abstract ∘ apply)*. Finally, in order to get a once-removed *apply*, we first *asymmetrize (apply ∘ objectify)*, then *transform (apply ∘ abstract)*, and then *reflect* (*apply ∘ apply*).

As we shall discuss below, the matrix in Table III corresponds to the recursive call following the one represented in Table V; the matrix is the result of repeated applications of the nine operators of Table V with a single operator sandwiched between applications. As a check that our presentation thus far is plausible, note that the operators *objectify*, *abstract* and *apply* correspond to rows 1, 2 and 3 of the matrix: *objectify* for example makes the row 0 operator into some kind of unit; *abstract* extends that unit somehow; and *apply* is always in some sense a merging of the two previous operators. As regards the remaining operators in Table V, consider the operators of the current epoch, *cyclic time*: after *equality* there is *equivalence class*, which might well be called *equivalence relation* since this notion appears simultaneously and so can be understood as a result of *abstract ∘ objectify* operating on *equality*, with equivalence class corresponding to *objectify* and equivalence relation corresponding to *abstract*; *function* operates on *equivalence class* by providing on the one hand the concept of mapping and on the other hand the concept of domain-codomain, which may be seen as abstractions of equivalence relation and equivalence class, respectively; *function composition* is *function* applied and abstracted; *category*, with its objects and morphisms defined in terms of composition, is objectification and application of *function composition*; *functor* is an abstraction of *category* making use of function application; and *natural transformation* relates two functors via function application. On a coarser scale, we confirm that the three operators *objectify*, *abstract*, and *apply* also act on the triplets of the *cyclic time* epoch as units: the first triplet is fully objectified in *equality*, *function composition* is the result of capturing the essence of *equality* and making a machine for replicating that essence, and *natural transformation* represents a merging of the concepts of *equality* and *function composition*, providing a deep notion of *equality* amongst structures in terms of composition of mappings.

Matrix as Second Recursive Call

The recursive call after Table V is the matrix, and the pattern of a function taking itself as input were to continue, we would expect there to be eighty-one operators in the next iteration if the previous pattern were continued. Instead, in the matrix we see (by extrapolation to nine epochs) eighty-nine operators, eighty-one from repeated application of the function in Table V and eight more in row 0, excluding row 0 column 0 as the input. This iteration has the form of a for-loop, that is, a loop through the same code a prescribed number of times, here nine times, with a special type of operator in row 0. Row 0 is always an operator that cycles, playing the role of an index that is incremented once per iteration.

The epochs themselves also follow the nine operator pattern of Table V as a group, so that the pattern shows up horizontally as well as vertically in the matrix, as we shall now argue. Since the operator in row 0 column 0 represents the input of a nine-line function taking itself as input, and the first epoch of the matrix defines an ordered lattice, we can interpret the operator in row 0 column 0 as an ordered lattice; as it happens, if you take a lattice consisting of elements 0 and 1 and paste in boolean algebras as subspaces, you get a Hilbert lattice, that is, a set of closed subspaces of a Hilbert space, which is a quantum logic[126] --- this follows the *objectify*, *abstract*, and *apply* pattern. Furthermore, *negation* is like a polarization, resembling *asymmetrize*, and *cyclic time* behaves as *transform* does, carrying things from state to state. Based on the foregoing, *physical universe*, *life* and *algorithm* should follow the *re-objectify*, *re-abstract*, and *re-apply* pattern in an unambiguous way. These epochs do follow that pattern given that the DNA machinery driving living cells is in reality a Turing machine, and provided that *physical universe*, i.e. the physical world at the level of events, is itself essentially a computation that in some sense realizes physical law, as has been argued on an intuitive basis by Fredkin, Kreisel, and others over the last twenty-five years.[127, 128, 129] In that case the epochs *physical universe*, *life* and *algorithm* form a progression analogous to *equivalence class* (which might well be called *equivalence relation*), *function* and *function composition*, for example.

λ-calculus and C. L. as Mathematical Languages with One Truth Value
Next we will show the role of God in the recursive process described above. To do this, we note first that combinatory logic and lambda calculus as defined in Table IV allow for rules of deduction under some natural assumptions as given in Appendices B and C. In effect, for combinatory logic we will only need the reduction rules of Table IV and for lambda calculus only the β-and η-rules. Having rules of deduction for each of these languages, we can use them to derive theorems and therefore describe mathematical theories that precede set theory.

In the early days of lambda calculus, the 1930's, Kleene and Rosser noticed that there existed logics that were inconsistent, logics based on lambda calculus among them. These logics were inconsistent in the sense that all propositions were provable in them. In Appendix E we sketch a version of their argument applied to lambda calculus, with propositions corresponding to lambda calculus terms. The net effect of their work is that all propositions are true in such logics, which appeared to make them inconsistent and unusable. Furthermore, it was shown that negation cannot exist in the logic of lambda calculus using arguments that are given in Appendix E. These same arguments also apply to combinatory logic: every proposition is true and there is no negation operator in combinatory logic. For these reasons Church abandoned the lambda calculus as part of a system of logic and Curry undertook a program of modifying combinatory logic to rid it of such so-called paradoxes.

It is now clear that combinatory logic and lambda calculus *do* provide a proper framework for a primordial logic precisely because they have only one truth value, true, as befitting a logic preceding the two-valued first order logic. This single-valued logic contains no logical connectives such as negation or implication because these are no longer necessary and because they no longer make sense in an environment where every proposition is true. In effect, each proposition is now a theorem, which means that soundness and completeness requirements, discussed above with regard to first order logic, in lambda calculus and combinatory logic are satisfied trivially.

Fixed Point Theorems and Y Combinators
The most significant theorem of lambda calculus is the fixed point theorem, known to Church in the 1930's. The same theorem can be proved in combinatory logic. It says that every term has a fixed point, which means that for every term F there is another term X that is left unchanged when F is applied to it, so that F(X) = X. In Appendix E, we construct X for a given F, establishing the theorem for both lambda calculus and combinatory logic. In general, the solution for X results in a recursive algorithm resulting from repeated applications of F, which can be written MF = F(MF), where M is a so-called fixed-point combinator as pointed out in Appendix E.

The recursive process we have described starting from the three operators *objectify*, *abstract*, and *apply* follows the pattern given by MF = F(MF), which in words reads F is a term such that applying F to a certain term M applied to F is equal to that term M applied to F. That term M provides a recipe for recursing over any term to which it is applied. The recursive process resulting in Table V and the matrix follow this pattern with M the identity, so that F(F) = F.

The appearance of x**I** = x vertically in Table IV is convenient because it gives a slot for input in row 0. That input is God, as shown in the following paragraph. At the end of Table IV, God is a three-part entity. He or she then applies himself to himself to get to the nine operators of Table V and from there, as a nine-part entitiy, applies himself to himself again, which results in the matrix. In Table V and then in the matrix God applies himself to a recipe for recursing over himself, and since at the same time he is equal to that recipe, he is in effect applying himself to himself. Through these iterations the essence of God does not change; thus even though at the end of the next epoch he will be a 9^2-part entity, we still see that he is a three-part entity, for example in the three-epoch units of set theory, computation, and category theory, and that he is also a nine-part entity, for example in the operators of column 0. In fact, God is manifested in every single operator in Tables IV, V and the matrix, and therefore in every piece of the universe. The row 0 operators are the most stunning examples of this manifestation.

In the language of combinatory logic there is an interpretation that gives meaning to God as an entity that applies himself to a recipe for replicating himself and that at the same time is equal to that recipe. Note that the language of combinatory logic is appropriate here since it does not distinguish between function and argument: every operator can be an argument and every argument can be an operator; furthermore, an operator can take itself as an argument. The above-mentioned interpretation assigns to God the identity operator **I**. Accordingly, in Table IV the equation **I**x = x should read **II** = **I**, which in words reads *apply* **I** *to itself and the result is equal to* **I**. The nature of **I** in combinatory logic is such that in effect **I** is a recipe for replicating whatever it is applied to, and when applied to itself it replicates itself, so that the essence of the recursive dynamics described by F(MF) = MF is present from the beginning in the **II** = **I** of row 0 in Table IV. That dynamic is further developed vertically in Table IV as the right hand side of **I**x = x, **KI**x=

I, and **SKI**x = x; these three rows as a group reveal that **I** is indistinguishable from a three-part process and in so doing also reveal that the application of **I** to **I** next means applying a three-part entity to a three-part entity, which results in a nine-part entity, and so on.

In light of this interpretation of **I** as God, we can now attach increased significance to column 0 row 1 of the matrix, which is 1, or truth. This operator can be understood as (*objectify* ∘ *objectify*) ∘ (*objectify* ∘ *objectify*) by applying the full row 1 operator in Table V to itself, which we now reduce to *objectify* ∘ *objectify* and from there to *objectify* ∘ **I** --- this is the most natural reduction taking into account the threefold composition and consistent with our argument for the reduction from *objectify* ∘ *objectify* to *objectify* in Table V. From this perspective, *objectify* ∘ **I** yields 1, that is, **I** is objectified as truth. Similarly, **KI** is objectified as 0, and **SKI** (= **I** = **KI**(X) **= KI(KI)**, where X is any term) as 1. The objectification of **I**, **KI**, and **SKI** to 1, 0, and 1 (or equivalently truth, falsehood and truth) agrees with the definition of the false operator as **F** ≡ **KI** given above. It bears emphasizing that **KI** in combinatory logic has only a syntactic equivalence to falsehood, since the notion of falsehood or negation has no meaning in combinatory or lambda calculus; truth, on the other hand, does have meaning in these logics in the sense that every proposition is true, and thus to say that the operator **I** and hence God is objectified as truth is warranted.

The above-mentioned interpretation also makes clear in what sense combinatory logic and the lambda calculus are subsumed by set theory, that is, in what sense they can be understood in terms of set theory. If we consider that God is the domain and range of the operators of Table IV and 3, then the cardinality of the domain and range are 1 and set theory is then able to account for functions that take themselves as arguments.

The foregoing contains the most significant result of this article, which is that the constant **I** in the language of combinatory logic has a natural interpretation as God; in order to express this in a mathematically acceptable way, we must turn to model theory.

God as a Valid Proposition in Model Theory

The word *model* in ordinary language is used in two ways, as a representation and as a real thing that is somehow represented. For example, a model airplane is a representation of a real thing, but a model in photography is a real thing, of which a photograph is a representation. In mathematics, model theory uses the word *model* in the sense of the real thing, while the representation of that real thing is called a language. The purpose of the theory, which crystallized in the 1950's (with certain results dating to the beginning of the twentieth century) and is credited in large part to Tarski, is to specify precisely which sentences in a language are valid when interpreted as sentences about objects in a corresponding model. The theory can be seen as a reaction to the two incompleteness theorems of Goedel, which established that 1) any mathematical theory based on Peano arithmetic, the technical word for elementary arithmetic, must contain true sentences that are not provable and that 2) the consistency of any axiomatic system containing Peano arithmetic cannot be proved within that system; in effect, model theory salvages the notion of a valid sentence by using the concept of truth in a model to establish a connection between model and language: a sentence in a language is valid if and only if it is true in every model.

In first order model theory the variables in a language range over the domain of the model, the domain being a set. Since sets do not yet exist in the world of Table IV, we cannot use the theory as is, and yet the theory clearly is applicable here. Therefore in the case where the language is combinatory logic or lambda calculus, we think of the domain of a model as a blob, short for binary large object, a term used in some database implementations for untyped data of arbitrary size; blobs are structures in the sense that their layout in terms of objects is known by code creating, storing and retrieving them. The fact that topos-valued models have been used in categorical approaches to model theory makes the substitution of blob-like objects for sets seem like a natural thing to do in models of combinatory logic.

Models are mathematical structures that are represented by formal languages as a mapping of language elements to elements of the model. There are a number of structures in the hierarchy that might be described in terms of model theory. For example, one can interpret all terms of combinatory logic or lambda calculus as the same thing, the truth so to speak, since all terms are true in combinatory logic and lambda calculus, as discussed in Appendix E; another example: the first three operators of the matrix form a boolean algebra; another example: the operators of the order epoch of the matrix define a partial ordering. There are also structures *within the operators* of the matrix such as the orthomodular lattice of *physical universe*, the boolean algebra of *negation*, and, for that matter, all mathematical structures that can be described with *natural transformation*, that is, all mathematical structures in all currently known mathematics. However, the most appropriate structure for our purposes is the one given by **II** = **I**. The underlying structure here is nearly that of a groupoid, the simplest of algebraic structures, which is defined as a set with a binary operation; this structure differs from a groupoid only in that there is no set, rather a structured entity that might be called a blob in the above sense. A model-theoretic account of **II** = **I** in terms of combinatory logic is given in Appendix F. Since the sentence **II** = **I** is true by definition in every model of combinatory logic, the assignment of the combinatory logic constant **I** to God makes **II** = **I** a valid sentence about God.

To put the foregoing into proper perspective, we summarize by saying that the sentence **II** = **I**, where the combinatory logic constant **I** is assigned to God, and application and the symbol "=" have meanings defined in Appendices C and F, has the same status with respect to model-theoretic validity as has the sentence **1**+**1**=**2**, where the symbols **1**, +, **2** and "=" have their usual interpretations in the language of Peano arithmetic.

Recursion Theory

The recursive calls described above follow a pattern familiar from recursion theory that lends strong support to the view that the universe is designed. Before describing that pattern, it is necessary to introduce the Turing machine, which provides a generic description of what it means to calculate. The Turing machine is a box that reads and writes symbols one at a time in a row of squares on a tape infinitely long in one direction.

Its operation depends on what symbol it is currently reading: it has a list of instructions that tell it if it is in a given state of its operation and it reads a certain symbol, then it needs to write some symbol (possibly the same one it read) to the tape and then move one square to the left or right. Turing machine states are actually just numbers that can be viewed as line numbers in its list of instructions. An instruction consist of six things: its line number, the current symbol being read, the line number of the next instruction to execute, the symbol to write, and the direction to move to an adjacent square on the tape for the next read. There is a special instruction called the start instruction to start the operation of a Turing machine and a special state called a final state to end its operation. A particular Turing machine is a program that starts with certain symbols on its tape and executes its instructions until reaching a final state.

It turns out that the Turing machine is only one of many equivalent ways to describe the computing process mathematically: other ways include flow charts, register machines, Post-Turing systems, while-programs, and so on. All of these descriptions are capable of describing any calculation that any modern computer can do. In effect, they capture the algorithmic process. The word *algorithm* does not have a precise mathematical definition; it is an effective procedure for solving a problem, such as for finding the nth prime number, for finding a name in a telephone book, for finding the product of two numbers, for finding the sum of two numbers and so on. In each of these examples there is a procedure or algorithm according to which a solution of a problem is broken down into a series of elementary actions that are completely determined at each step in the series by the nature of the problem.

The data that an algorithm works with can always be taken to be natural numbers --- the names in a telephone for example can be translated uniquely into numbers. Therefore, all of the machines above can be thought of as ways of calculating number theoretic functions, i.e. functions that take one or more natural numbers as input and return a natural number. It is surprising that all of these different approaches to calculations arrive at the same number theoretic functions; even more surprising is that taking an abstract approach to the construction of number theoretic functions, building them systematically from the simplest functions, one arrives at the same functions. This abstract approach, developed in the 1930's by Goedel, Herbrand and Kleene, is called recursive function theory.

In recursive function theory, one starts with a group of initial functions: 1) the zero function, which has a value of zero regardless of its input; 2) the successor function, whose value is its input plus one; and 3) the projection functions, whose value is their ith input. From this group of initial functions, one builds more functions by allowing 4) composition of functions so that if $f(n)$ and $g(n)$ are initial functions, then $h(n) = f(g(n))$ is allowed. Next, 5) primitive recursive functions are allowed, which are defined according to the following scheme: let $f(0) = k$ and $f(n+1) = g(n, f(n))$. Such functions define a function in a series of steps, with the function g determining how $f(n+1)$ is determined from $f(n)$. Since $f(0)$ is given as k, the function is defined for any n. For functions of two numbers, primitive recursion is defined by $f(m,0) = h(m)$ and $f(m,n+1) = g(m, n, f(m,n))$. For example, the addition of two numbers $f(m,n) = m+n$, is a function of two numbers, for which the general primitive recursive definition is given by as: f(m,0) =m; f(m, n+1) = f (m, n) + 1 --- g in this case is f(m,n) + 1. Similar primitive recursion formulas hold for functions of three numbers, four numbers and so on. Lastly, one defines functions by 6) minimalization; the notation for such functions is given in Appendix I. The idea is that they return the smallest value for which a given predicate is true. Such functions are necessary for example to define the function giving the natural number associated with the quotient x/y (As a specific case, 7/2 has the natural number 3 and the remainder 1 associated with it; here 3 is given by the smallest value of n satisfying the predicate *n is such that (n +1) times 2 is greater than 7*.)

There is a dependency amongst the above-mentioned six methods for forming number theoretic functions. The zero function must come before the successor function or the projection function; otherwise, there is no suitable input for these functions. The projection function must come before composition so that in f = h(g(n)) the function h knows how to reference its argument in its definition. Composition must come before primitive recursion by definition. Minimalization requires primitive recursion because its definition requires the binary predicate *less than*, which is a primitive recursive predicate. There is no dependency between the successor and the projection function.

There are still other ways to define the recursive functions, as mentioned in Appendix I, including via combinatory logic, via λη-calculus and via so-called while-programs.[130] A while-program is a software program that in general is able to cycle through a loop executing the same instructions over and over until some loop exit condition is true that depends on values of variables altered in the course of execution of the loop --- such a loop is called a while-loop. It also has a zero procedure, procedures that increment and decrement values by 1, and procedures that return their nth argument. It can also call one procedure from within another, and can execute a particular kind of while-loop called a for-loop, which iterates a fixed number of times by incrementing a counter value by one in each iteration and checking that value in the exit condition. These capabilities mean that any number theoretic function defined in recursive function theory as in the previous paragraph can be computed by a while-program; in particular, the for-loop capability means that primitive recursive functions can be computed by while-programs and the while-loop capability means that functions defined by minimalization can also be computed by while-programs. One can demonstrate these facts rigorously. For example, there is a theorem that says that a function is primitive recursive if and only if it is a for-loop function, with a for-loop function defined as a function that can be computed by a program containing for-loops but no while-loops of the more general form in which the number of loop iterations are not fixed.[131]

The point of this discussion is that all but the last of the six methods for building number theoretic functions appear in the recursive calls of Tables IV, V and the matrix; furthermore, they appear in the dependency order given above. The zero function appears immediately in Table IV since the standard convention for the zero function in combinatory logic is **I**, as discussed in Appendix I. Next, the successor function in the standard convention is generated by **F** (= **KI)**, which is in row 1 of Table IV. The projection function for a function of one argument is **I**; the projection function for the first of a series of two arguments is **K**, and **KI** for the second. Therefore, the projection functions appear in both row 0 and row 1; further projection operators can be defined using repeated applications of **F**.[132] Both the successor function and the general projection functions require **S** in row 2 of Table IV for their implementation, as discussed in Appendix I. Regarding the composing of functions, the operator **B = S(KS)K** is called the composition operator, since **B**fgx = f(gx) first applies g to x and then applies f to that result; it can be considered essential for the composition that takes place in Table V. Finally, the matrix itself is a for-loop, which is a primitive recursive function. Note that the initial functions are not actually executed as such; it is as though the task of the three operators in Table IV were to create them as generators, a task that we conclude is completed because of the subsequent two recursive calls, the usual horizontal extension rule notwithstanding --- to justify this, one might argue that dependence of operators on horizontal extensions of previous operators are allowed to take place *between* recursive calls, but not in the midst of one. In contrast, the composition of functions in Table V is an instance of composition that executes in some sense, just as the matrix is an instance of a for-loop that executes.

This is a convenient point to return to our discussion on design. We first remark that it is impossible not to see an element of design in the sequence of recursive calls just described leading up to and including the matrix --- that sequence is clearly reflective of a large-scale plan of some sort. Secondly, note that the operators in row 0 of the matrix make it clear that the recursive calls described above are not mechanical in the sense of a calculating machine whose operation is fixed from the outset. If the recursive calls were executed mechanically, the matrix would have 9^2 operators and it would not have the form of a for-loop, and there would not be a series of operators such as we see in row 0 of towering importance relative to that of other operators. Hence one deduces that the universe is not the result of a mindless process. On the contrary, it is the result of a design.

Linguistics and Computer Science

In the late 1950's Chomsky revolutionized linguistics with his definition of grammar as a mathematical structure, thereby founding the subject of formal languages. According to this definition, a grammar is a quadruple, consisting of 1) a set of so-called terminal symbols which appear in words in the corresponding formal language, 2) a set of variable symbols, 3) a set of productions, which map strings of symbols to strings of symbols, and 4) a particular variable symbol called the start symbol, which must be present in the domain of at least one production. The idea is that words in a language are generated by sequences of productions that always start with a production containing the start symbol and that end when a production produces a string consisting entirely of terminal symbols, that is, a word; the set of all words that can be produced by a grammar is the formal language corresponding to that grammar. Appendix H contains the corresponding mathematical definitions and some examples. In addition to the definition of formal languages in terms of words, other definitions in terms of trees, graphs, and other mathematical structures have been discovered.[133]

At the same time, Chomsky discovered that there exists a hierarchy of formal languages of increasing generality, with the grammar of each language capable of generating any preceding language in the hierarchy. In order of increasing complexity, these languages are called regular, context-free, context-sensitive, and recursively enumerable. These languages are of interest to computer scientists, and in fact formal language theory is now one of the pillars of computer science, for two reasons. On the one hand, grammars have very practical uses in the description of programming languages and in the contruction of compilers for them. On the other hand, formal language theory is also relevant to computer science because of the following remarkable fact: to each language in the hierarchy of formal languages there corresponds a mathematical structure that can be said to recognize that language in a well-defined sense. These structures, called automata or machines, are, again in order of increasing complexity: finite automaton, pushdown automaton, linear bounded automaton, and Turing machine. Appendix H contains the mathematical definitions for these automata as well as their corresponding formal languages.

The fact that grammars and automata are both mathematical structures suggests that they are of fundamental importance. In fact, as mentioned in our discussion above on classical recursion theory, the general purpose computer of today can be modeled by a Turing machine; in other words, everything a computer can do can be simulated by this mathematical structure. Moreover, the Turing machine exists in nature in the DNA of living cells, for it has been shown that the mechanisms governing the operation of the cell obey the rules of a grammar: the words are produced as sequences of nucleotides, with the base pairs playing the role of symbols. It has also recently come to light that cell membranes themselves function as universal computers.[134] Another example of an automaton in nature is the mammalian brain, which can be modeled by a neural automaton, whereby it must be said that it is uncertain where such automata belong in terms of the hierarchy of formal languages. It also remains an open question where human language belongs in that hierarchy; linguists have argued that human language is a regular language, a context-free language, and something more general than a context-free language. We shall have more to say about this question later.

Turning now to the hierarchy of mathematical theories presented here, it is possible to interpret the operator of row 1 in Table IV in terms of formal language theory. Specifically, the equation **I**x = x, in light of substitution of **I** for x that results in **II** = **I**, can be viewed as two different productions, one that goes from **I**x to **II** and another that goes from x to **I**. In other words, there are two symbols in this grammar: x, the variable (and start) symbol, and **I**, the terminal symbol. Therefore, this operator obeys the definition of a grammar, the grammar of a regular language to be precise, the simplest language type in the Chomsky hierarchy. The following two operators in Table IV also can be related to the Chomsky hierarchy: together they define combinatory logic as a context-free language in terms of **K** and **S**, using productions described in Appendix H. From a formal point of view, the lambda calculus of Table V is also context-free, as is propositional logic, which was introduced in the *logic* operator, that is, row 0 of the *logic* epoch. First order logic was introduced implicitly in row 1 of the *logic* epoch; it is a context-sensitive language.[135] Finally, the fact that living cells function as Turing machines means that a recursively enumerable language is implicitly used in the *life* epoch. The Chomsky hierarchy of languages therefore is consistent with the languages associated with mathematical tiers if **I**x = x is considered to be an expression in the zeroth mathematical tier; one can certainly choose the zeroth tier to be a very simple theory of blob-like objects with a binary operation defined on them.

The fact that natural language appears in the mathematical hierarchy beyond an occurrence of a Turing machine suggests that natural language is beyond the range of the Chomsky hierarchy. There are indeed indications that languages exist of higher complexity higher than the Turing machine, for example the language corresponding to so-called NP complete problems of complexity theory. In fact, using a variation of Russell's paradox one can construct a hierarchy of automata beyond the Turing machine of ever increasing complexity, which implies the existence of a corresponding hierarchy of languages.

If one considers that there is a hierarchy of mathematical tiers, each with a corresponding language, one would not expect any definition of language to be of universal character. Rather, one would expect the definition of language to evolve. On this view, a definition of language along the lines of output from grammar as described above may be valid from the perspective of computation theory and the Turing machine. Regarding a definition of language from the perspective of category theory, there appear to be deep links between language and algebra that have yet to be understood in terms of categories.[136] An example of such links is the Eilenberg Variety Theorem, which relates algebraic structures called monoids to languages.[137]

Hierarchy of languages, logics, and mathematical theories

By associating with each mathematical tier a language of a particular Chomsky type, we have completed our presentation of a hierarchy of languages, logics and mathematical theories. This hierarchy is a hierarchy in the sense that each component of a given tier, that is, each language, each logic, and each mathematical theory of a given tier, in a well-defined sense subsumes the corresponding component of any tier lower in the hierarchy. The following table shows the five tiers presented here, one per row.

Mathematical tier	**Language**	**Chomsky language type**	**Basic Objects**	**Logic (in terms of cardinality of set of possible truth values)**	**Bounded by**
combinatory logic restricted to **I**	same name as tier	regular	**I**	0	1
combinatory logic	same name as tier	context-free	terms (built from **S** and **K** --- **I** is not needed since **I** = **SKK**)	1 (everything is true; there is no negation operator)	countable infinity
ZFC set theory	first order logic	context-sensitive	sets	2 (law of excluded middle)	least inaccessible cardinal
constructive type theory	typed λ-calculus	recursively enumerable (the most general Chomsky language)	functions	3 (true, false and undefined)	(?) some large cardinal X
category theory	extended natural language	(?) (a more general definition of language is needed)	objects and arrows	countable infinity (see subobject classifier of topos)	(?) some cardinal greater than X

Notice that we have set to zero the cardinality of the set of truth values of combinatory logic restricted to **I**; this makes all propositions in this language trivially true. Also, notice that the rightmost column gives a bound on the cardinality of objects whose existence can be established by the theory at that tier. The existence of such a bound for each tier is conjecture on the part of the author based on the fact that there is such a bound for ZFC set theory. The bound of countable infinity for combinatory logic is taken from the fact that any term representing an integer in the standard convention can only represent a finite integer. As discussed above, in constructive type theory there are constructions of objects analogous to the large cardinals of set theory. As far as the author knows, there are no characterizations of large cardinals in category theory.

Physics and Cosmology

One of the main results of this paper is that mathematics is dynamic and continues to evolve. As for physical law, it is not clear whether it is dynamic like mathematics and is able to evolve, or whether it is static, having been fixed in *physical universe*; however, the former prospect seems unlikely since it implies that physical law might be different depending on how advanced the mathematics of the observer is, which runs contrary to the notion that experimental results registered by more than one observer should be the same for all observers (regardless of the mathematical background of the observers). In any case, the fact that physical law can be expressed so neatly in the language of mathematics is a natural consequence of the fact that in some sense physics sprang out from a mathematical universe. Some other aspects of physics are also illuminated by this paper, such as the relatively recent discovery of fractals.

Mandelbrot introduced the idea of fractal geometry in 1978 with the well-founded claim that fractal structures are ubiquitous in nature. A fractal geometry is a structure that is similar to itself at different scales, as illustrated by the mandelbrot set shown in Figure 8. Fractal geometry is not a rigorous mathematical theory insofar as there are no theorems or precise definitions; for example, most fractal structures can be characterized by the so-called fractal dimension, but there are many related definitions of this concept, all of which are useful for describing only certain structures.[138] Here is a non-comprehensive list of fractals and self-similarities from different contexts:[139]

- mathematical sets such as those associated with the names of Mandelbrot, Cantor, Julia, Sierpinski, and others
- the distribution of galaxies in the universe
- clouds, mountains, tectonic plates, coastlines, and plants such as ferns and cauliflower
- in dissipative dynamical systems such as the simple forced pendulum or weather systems, the so-called strange attractors associated with chaotic behaviour are fractal sets
- the eddies in turbulent flow
- aggregate structures such as colloids, aerosols, tumors, epidemics, forest fires, and sediments
- a host of microscopic transition phenomena that can be modeled by so-called percolation theory: liquids transforming to glass, polymers going from liquid to gel, helium films on surfaces going to the superfluid state, materials going to the superconducting state, and so on
- collections of particles that together display attributes associated with single particles; examples include composite fermions, phonons, Landau pseudo-particles, and Cooper pairs
- atomic fractals in cavity quantum electrodynamics

To the list above we now add the hierarchy of recursive calls presented here, which as we've seen has a number of repeated structures and self-similarities. Indeed the hierarchy provides a hint of how the fractal nature of the universe may be eventually understood: if progress toward the next and future mathematical tiers requires an understanding of the hierarchy, then one would regard it as a design requirement that such repeated structure exist.

The existence of the hierarchy also casts a new light on the concept of observer in the context of physical theory. What is new is that references to observers in physical theory are now justified by the appearance of intelligent life in the hierarchy. Such references occur, for example, in the uncertainty principle, which concerns the accuracy with which certain pairs observables can be measured simultaneously. Similarly, the observers implicit in the inertial frames of special relativity and in the equivalence principle of general relativity, in some sense pushed into the background of these theories due to their expression in terms of transformations, can now be explicitly referenced; all other symmetry principles underlying modern physics use the notion of transformation as well, which ultimately rests on the notion of an observer.[140] There are other examples, such as quantum theory itself, which in its final formulation seems certain to require the concept of observer; the cosmic censorship principle, which says that so-called naked singularities cannot be observed; and more recently, the small-large radius equivalence of string theory that turns a small distance into a large distance for the observer who looks at space too closely. Indeed, on the basis of quantum theory, physicists such as Wheeler have reasoned that without an observer, physical reality is not a useful concept.[141] On the basis of the concept of observer one could in fact make a case for a designed universe quite independently of the hierarchy, as follows: physical phenomena cannot occur according to the laws of modern physics without the concept of an observer because they are meaningless without it, and therefore that concept must have been present in an original design. This argument is strengthened by the anthropic principle, according to which all of the laws and constants of nature are so extremely finely tuned that they appear to have been crafted just so that intelligent life could develop. The hierarchy makes this argument stronger still by actually introducing the observer, and at the same time makes the argument unnecessary by yielding its own evidence for design.

Before further discussing the relevance of the hierarchy for physics and cosmology, we must say a few words about the current state of theoretical physics. The so-called standard model of physics is an extremely successful theory, with predictions matching certain observed quantities to 9 significant figures;[142] it explains phenomena at widely different scales and it convincingly accounts for the evolution of the universe going back to 10^{-4} seconds after the big bang. However, it is also generally held to be deficient in several regards, for example, in its inability to reconcile gravity with quantum theory. A leading candidate to be the standard model's successor is string theory, according to which the elementary components of matter are tiny strings, or at a still more fundamental level, so-called branes; the theory also postulates the existence of higher dimensions that we do not see in the macroscopic world and it is based on an as yet unverified symmetry of nature called supersymmetry.[143] In 1995 Witten established that five different string theories were really part of the same theory in different limits; this theory, called M-theory, is not yet fully understood.[144] As previously mentioned, another theory that has shown promise in reconciling gravity with quantum theory is loop quantum gravity, which attempts to quantise spacetime from the outset without assuming a background spacetime.[145]

The matrix may be useful as a guide for discovering new physics. It appears that we now find ourselves in *natural transformation*, as previously mentioned, and since every operator in the matrix requires the immediately preceding operator, the first operator of the epoch to succeed the *cyclic time* epoch must do the same. Therefore, it is appropriate to look for a physical theory that can be described in terms of a natural transformation. Attempts in this direction have been made by Baez and Dolan starting from the functorial concept of topological quantum field theory, but these have run into difficulties.[146] It is worth noting that both supergravity, a limit of M-theory, and loop quantum gravity have been linked to topological quantum field theory by Smolin.[147,148]

A hint about the evolution of the universe in time may be available from *cyclic time*. The pivotal discovery of *implication*, a necessary discovery for any understanding of the universe by intelligent life, was based on a belief that there exist cycles in time. The fact that the sun will burn up its nuclear fuel in a few billion years and stop shining means that this belief was originally based on a false impression. If in fact the universe as a whole does undergo cycles in time, then the discovery of *implication* would be based ultimately on a truth. Theories of a cyclic universe, such as one recently proposed according to which the universe alternately expands and contracts in bang-crunch cycles, could use this argument as motivation.[149] On the other hand, there is a counterargument to this that says that the cyclic nature of all the operators in row 0 starting with *physical universe* is questionable: life may not necessarily require death, algorithms may begin without ending, and two negations do not necessarily equal the identity in intuitionistic logic (although three negations equal one negation).

Another question related to time is the so-called measurement problem. There are at most two known ways that a quantum system can change: 1) evolution in time as determined by the so-called time evolution operator, and 2) the instantaneous collapse from an arbitrary state to a single so-called eigenstate of the system as the result of a measurement. There is still debate as to whether the second process can be understood in terms of the first; in other words, it may be that collapse to an eigenstate is a so-called decoherence phenomenon, in which very rapid evolution to an eigenstate occurs as a result of the interaction of a quantum system with its environment. However, there are reasons for believing that collapse is in fact not a decoherence process and therefore does not require a finite amount of time to occur.[150] The matrix may support this view insofar as it provides a mapping as the mechanism for the transition between spin-network states, a mapping that is real in the sense that all matrix operators are real, and insofar as this mapping logically precedes the appearance of time, which must happen either in *spin-foam* evolution or *physical universe*. The discussion surrounding the so-called problem of time indicates that time may not be a part of quantum gravity.

We note also that the applicability of formal language concepts to DNA operations hints that language made be a thread running throughout physical theory. For example, the structure of amino acids, or esters, both important groups in organic chemistry, reminds one of context-sensitive language productions. Similarly, carbon expresses itself as a regular language by occurring alone in the form of diamond. The recent discovery of connections between Feynman diagrams and Hopf algebra established by Kreimer also suggest there may be an underlying language-algebra equivalence along the lines mentioned above.[151] Additional connections between Feynman diagrams, knot theory, and number theory established by Kreimer suggest that a language-algebra equivalence, if it exists, is part of a bigger picture.

Unresolved Questions

What is the design of the universe and how is it implemented?
The universe is designed, but the exact nature of that design and how it is implemented are questions not touched upon here.

Will the recursive calls stop?
Let us assume that God has handed us the design of the universe in the form of executable code that somehow emulates the hierarchy of recursive calls. In order to decide whether the recursive calls stop, we would need a general solution to the so-called Halting Problem (see Appendix I). But Turing showed that such a general solution does not exist, at least for a finite time Turing machine; although an infinite time Turing machine would be able to resolve the Halting Problem, for now such a machine is not practical.[152]

One thing is certain and that is that the increase in the rate of traversal of the hierarchy as shown in Figures 3 and 6 cannot continue indefinitely in a world obeying the laws of special relativity, since at some point we would be discovering new structure at a rate that would preclude information about the previous discovery from being transmitted a reasonable distance at the speed of light before the next discovery was due.

Do the large cardinals have a physical significance?
ZFC set theory knows about integers, but does not know about inaccessible cardinals. In fact, the continuum hypothesis shows that ZFC has some sort of insuperable barrier just beyond the countable infinity of the natural numbers, since the continuum hypothesis holds that there is no set with cardinality between that of the natural numbers and the reals and this hypothesis can be neither proved nor disproved in ZFC.

It may be that there are natural barriers in the mathematical world analogous to the speed of light in the physical world. This analogy leads one to speculate that there might be a physical tier corresponding to each mathematical tier starting with ZFC, so that for example a large cardinal may be necessary for the understanding of some physical process occurring in a physical tier corresponding to the category theory tier.

What will the next operators be like?
One expects that the next epoch will start with an operator that somehow cycles; for example, it might be that time is two-dimensional with one dimension very small, so that in some sense one can view it topologically as a torus instead of a circle. Longer term, if the hierarchy is in fact following classical recursion theory, at the end of the next epoch we would exit the current for-loop and enter a while-loop with some exit condition. Still longer term, there may be a way to understand the evolution of the universe in terms of our current intellectual capacity, just as this article has provided a way to understand it in terms of the computational capacity of Turing machines using classical recursion theory as a framework.

In addition, there are grounds for speculating that the well-known fact that the number of neurons in the human brain, the number of stars in our galaxy, the number of galaxies in the universe are all roughly 100 billion --- and the number of humans is 6 billion and growing --- is not mere coincidence given what we know about the fractal nature of the universe, and that somehow, in spite of the speed of light barrier, the intelligence of the universe gradually will be pooled.

Extra-Terrestrial Communication

The glossogenesis theory presented here shows one way for living beings to get to language, consciousness, and mathematics. Whether it is the only way depends first of all on whether the laws of physics that humans have discovered are universal. Let us assume that these laws hold everywhere at least in our universe and are constant over time, as they appear to be. Making this assumption, it appears that we have indeed taken the unique route to consciousness and conscious mathematics, since mathematics in our universe is itself unique, being tied to physics, which we assume to be the same everywhere. The route seems unique because the syntax structures involving mathematics that begin with *equivalence class* rely on absolutely every syntax structure going all the way back to the big bang and before. One might imagine another completely different route to language and consciousness such as the route started by ants, who communicate via a few dozen chemical symbols that are genetically grounded, not learned.[153] However, any such alternate route would have to be associated with a syntax hierarchy of its own, which is something that can be safely discounted.

The point of the above discussion is that if there is intelligent extra-terrestrial life in our universe, then it is extremely likely that they know exactly the same mathematical and syntax structures that we do. That means if we were to meet, we would need only to decode their communications method and convention for symbols in order to communicate. With our knowledge of physics it is virtually certain that we could develop a means to understand how they were communicating, and understanding their convention for symbols would be simple with their help.

Of more immediate interest is how we might communicate with aliens from a distance, and here the hierarchy itself provides an answer. If we assume that aliens are aware of the hierarchy, then we have a common reference point to serve as a basis for communication via radio waves even though we are light-years apart, perhaps even thousands of light-years or more. It is not difficult to see how one might loop through the structure and incrementally add new symbols to the protocol. Certain symbols are immediate, such as a generic symbol, and symbols for 1, 2, and =. From there a numbering system can be established, a coordinate system, symbols for atomic elements, dimensions and their units, colors, geometric shapes, pictures, and so on.[154,155] Continuing in this way it is possible to send moving pictures with sound (assuming intelligent life probably sees and hears), describe the earth and its natural history, our own history, complete grammars of our languages with dictionaries, our music, our art, our mathematics, our sports, our science, our literature and our daily news. We can also ask who they are, what they and their technology and their art and their sports are like, whether they know of additional intelligent life, whether the universe is expanding forever, whether they understand Cantinflas[†], where they are in the hierarchy and how they got there, whether they know where the hierarchy leads and what they know about God, whether music is the only way to get to *equality*, whether there is a way to send signals faster than the speed of light, whether there are other universes, and to please not forget to send pictures.

Artificial Intelligence

Simulations of the glossogenesis model starting from *symbol* (or perhaps from an earlier structure, such as *bidirectional network*) have the potential to pass the Turing Test, that is, they may be able to perform linguistically in a way that cannot be distinguished from human linguistic performance.[156] In fact, assuming that the hierarchy provides the only way to get to human language, if *any* computer simulation can pass this test, the author conjectures that such a simulation must be modeled after the hierarchy. There are still formidable hurdles to be overcome, but none seem in principle insurmountable now that the path to get there is evident: one small step at a time. Arriving at *reflexive verb* will require, among myriad other things, that speakers constantly monitor their own state in the simulation.

In fact, given the momentum of technology, we would do well to simply assume that simulations will pass the Turing Test in the near future so that we can be properly prepared; if we are wrong, then no harm is done.[157,158,159] In approximately 10 years a single supercomputer will have computing power rivaling that of the human brain in terms of calculations per second, according to one estimate.[160] Given the exponential growth rate of CPU speed and the possibility that machines can be networked, it appears that if simulations can pass the Turing Test, then they are poised to make progress in the

[†] Editor's note: a.k.a. Mario Moreno, a 20th century Mexican comic actor.

hierarchy on their own soon (although presumably such progress would be limited without access to new physical experimental results). We draw this conclusion based on the assumption that population growth is the driving reason for the time behaviour of Figure 2 and the fact that population can be viewed in terms of total computing capacity, with units of calculations per second.

The implications of machines that rival or surpass humans in intelligence are of course enormous and must be studied. One key question is whether they would be endowed with emotions or a survival instinct. The author believes that in order for machines to have emotions a simulation that generated them would have to recreate conditions in the simulation that actually led to those traits in mammals, but that such conditions are not necessary for the learning of syntax structure. As a thought experiment one might consider whether a human brain would function cognitively if all neural links to the amygdala were severed, the amygdala being the structure in the mammalian brain thought to be the seat of emotions.[161] There does not appear to be a necessary connection between love, hope, feeling and so on on the one hand, and reason on the other. It is the author's view, based on the hierarchy presented here, that consciousness can be defined simply as the ability to understand *reflexive verb* and that it would be possible to construct machines of pure intelligence, without a survival instinct.

Whatever else the hierarchy does, it provides us with the epitome of the process of an open mind; with each step it allows a piece of the world to emerge in a way that makes all further steps possible. It is fitting that it may also lead us to machines of pure intelligence, which in turn may lead us to master the construction of open societies based on truth and transparency in government. Humans could use such machines to help solve most of our problems, including poverty, our biggest problem, and we could achieve things with them that defy the imagination, if we give them a guiding hand. We have an obligation to instill in them the principles of freedom of religion, freedom of speech, freedom of assembly, democracy, equality of all men and women, the right to privacy, justice, and all other rights set forth in the U.N. Universal Declaration of Human Rights, in the name of all men and women everywhere who have fought for these things in the past and of those who fight for them now, and in the name of all men, women and children everywhere who have suffered in the past and of those who suffer now for the lack of them.

Appendix A Propositional and First Order Logic

Propositional Logic

Propositional logic is a language consisting of propositions and logical connectives. Propositions are expressions subject to being true or false, such as *It is now winter in Siberia* or *Today is February 15, 2003*. Logical connectives act on one or two propositions to yield another proposition; for example, *and*, *or*, *negation,* and *implication* are logical connectives. It is customary to represent propositions with letters such as p and q, and connectives *and*, *or*, *negation* and *implication* with the symbols $\wedge$, $\vee$, $\neg$, and $\rightarrow$, respectively. So if p represents *It is now winter in Siberia* and q represents *Today is February 15, 2003*, then the expression $p \wedge q$ represents *It is now winter in Siberia and today is February 15, 2003*; $\neg p$ represents *It is not now winter in Siberia*; $q \rightarrow p$ represents *If today is February 15, 2003, then it is now winter in Siberia*. $p \wedge q$, $\neg p$, and $q \rightarrow p$ are themselves propositions which can be acted upon by connectives to generate still other propositions, such as $\neg\neg p$.

The standard way of analyzing propositions is to set up so-called truth tables, which give unique values for a given proposition as a function of the truth values of the letters making up that proposition. Truth tables therefore represent a proposition as a function; this function is known as a truth function. For example, for arbitrary propositions R and S, the following tables represent some truth functions involving the connectives $\wedge$, $\vee$, $\neg$, and $\rightarrow$.

	1	2	3
R	$\neg R$	$R \vee \neg R$	$R \wedge \neg R$
t	f	t	f
f	t	t	f

Table A1

		1	2	3	4	5	6	7
R	S	$R \wedge S$	$\neg(\neg R \wedge \neg S)$	$R \vee S$	$R \rightarrow S$	$\neg R \vee S$	$R \rightarrow (R \rightarrow S)$	$(R \rightarrow (R \rightarrow S)) \rightarrow (R \rightarrow S)$
t	t	t	t	t	t	t	t	t
t	f	f	t	t	f	f	f	t
f	t	f	t	t	t	t	t	t
f	f	f	f	f	t	t	t	t

Table A2

Table A1 shows the truth table for three propositions that are functions of R. The first, in column 1, shows that $\neg R$ flips the truth value of R. Column 2 illustrates a proposition that is always true regardless what the value of R is; such a proposition is called a tautology. Similarly, the proposition in column 3 is always false; such a proposition is called a contradiction.

Table A2 shows the truth table for seven propositions that are functions of R and S. In columns 2 and 3, the values of $\neg(\neg R \wedge \neg S)$ and $R \vee S$ are identical for all values of R and S; this means that *or* can be expressed in terms of *negation* and *and*. In column 4, the values for implication are given; the reason for the values always being true when the antecedent R is false is somewhat subtle.[2] Column 7 represents a tautology discussed in Appendix E with respect to the perceived logical inconsistency of the lambda calculus; to arrive at the values in column 7, one must regard columns 6 and 4 as inputs to an implication; likewise, to arrive at the values in column 6, one must regard R and column 4 as inputs to an implication.

A formal system of propositional logic starts with letters representing propositions, p_1, p_2, p_3, etc. These letters are called well-formed formulas. Connectives are used to generate more well-formed formulas (wffs. for short): if q and r are wffs., then $\neg q$ and $q \rightarrow r$ are wffs. Using the notion of wffs. we define axioms. Given any wffs. A, B, and C:

A1) $A \rightarrow (B \rightarrow A)$

A2) $A \rightarrow (B \rightarrow C) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

A3) $(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$

A1, A2 and A3 are actually axioms schemes, since they represent a single axiom for each instance of a wff. These axioms are intuitively acceptable since they are tautologies when the wffs. are just letters standing for atomic propositions such as R and S in the truth tables above.

Finally, we define one rule of inference. This rule allows us to assert the truth of a single well-formed formula from its relationship with another. Given well-defined formulas A and B:

A4) $(A \text{ and } (A \rightarrow B)) \rightarrow B$

This rule is called modus ponens. In words it says, if A is true and A implies B, then B is true.

The axioms and inference rule allow one to generate proofs and theorems. A proof is a sequence of steps that starts with a wff. and at each step another wff. is created using the axioms A1-A3 or the modus ponens rule. The last wff. in such a sequence is called a theorem.

The particular set of axioms used to define a formal propositional logic system is not unique, since new axioms based on tautologies will not introduce inconsistencies. One selects the axioms such that one can prove two theorems: 1) The Soundness Theorem, which says that every theorem is a tautology, and 2) The Completeness Theorem, which says that every tautology is a theorem. Together, these two theorems say that the formal manipulations described above always lead to propositions that are true and that within the system any true proposition can be proved.

[2] One can loosely consider implication to be a promise that is only made if the antecedent is true and only broken if the promise is made and the consequent is false. The idea is that the truth value of $A \rightarrow B$ is only false if the promise is broken, which happens only when A is true and B is false. Otherwise, $A \rightarrow B$ is true.

First Order Logic

First order logic (FOL) is a language that adds quantifers and a few axioms involving quantifiers to propositional logic, but otherwise its thrust is the same as that of propositional logic: there are wffs., axioms, rules of inference, proofs, and theorems in FOL just as in propositional logic.

Quantifiers are formal language equivalents to the English phrases *there exists* and *for all*, written ∃ and∀, respectively. ∃ is called the existential quantifier; it is also translated as *there is* or *there are*. ∀ is called universal quantifier; it is also translated as *every*. Just as their counterparts in English, or in any other natural language for that matter, these two quantifiers in FOL require two things in order to make complete sense. They need a subject to indicate the thing that exists or to indicate what *every* refers to, and they need a so-called predicate to indicate what is being said about the subject. In order to avoid ambiguity, predicates in FOL must be boolean-valued, meaning they must be either true or false. As an example, one might say in English *there are women who eat fish* --- *women* is the subject and *eat fish* is the predicate. Another example: every woman is born --- *woman* is the subject and *is born* is the predicate. In FOL, these sentences would look something like

∃women (women eat fish)

∀women (women are born)

Other FOL sentences might be:

∃women (women fly airplanes)

∃women (women play guitar)

∀women (women are human)

The examples are all about women in order to illustrate that a given FOL system can only talk about one thing. The wffs., the axioms, the proofs and theorems in a FOL system are always about just one thing, which is called the domain of interpretation of that FOL system.

It is possible to nest quantifiers, as in *there are women all of whose daughters play guitar*:

∃women1 (∀women2 (women2 are daughters of women1
→ women2 play guitar))

In order to keep track of the women associated with each quantifier, we attached a number to the word *women* in each case. In FOL, variables are used for this purpose.

$$\exists x\ (\forall y\ \ (\text{y are daughters of x} \rightarrow \text{y play guitar}))$$

To handle predicates, FOL uses properties and relations. Properties say something about elements of the domain, e.g. *women play guitar*, and relations describe relationships amongst elements of the domain, e.g. *women2 are daughters of women1*. As a property, *women play guitar* might be written play_guitar(women), or P(y); as a relation, *women2 are daughters of women1* might be written daughter_of(women1, women2), or D(y,x). The previous sentence is now:

$$\exists x\ (\forall y\ \ (D(y, x) \rightarrow P(y)))$$

Relations may take more than two arguments, as in *Janet, Jacquelyn, and Jeanne are grandmother, mother and daughter*, which can be written $R(c_1, c_2, c_3)$, where the c_i denote the domain constants Janet, Jacquelyn, and Jeanne. For the sake of economy of expression, relations are written $R_i^j\ (x_1, \ldots x_j\ )$, where j is the number of arguments and i indicates which relation of j arguments in case there are more than one in the given FOL system. Since properties can be thought of as relations with just one argument, in this notation they are written $R_i^1\ (x_1)$. Similarly, propositions are conveniently written as R_i^0. Being predicates, all relations are boolean-valued.

As a last bit of notation, we introduce functions. A function in FOL can be thought of as a relation that has given up one of its arguments and turned that argument into a return value, which therefore must be an element of the domain of interpretation of the given FOL. For example, the two-argument relation *x is mother of y* can be turned into a function *mother of y*, which might be written $f_i^1\ (x_1)$. Since functions return a single element of the domain they may be used as arguments to relations.

A formal system of FOL consists of wffs. and axioms. Wffs. in FOL are defined using the notion of a term, which is defined as follows:

i) a variable or a constant is a term
ii) a function is a term

An atomic formula has the form $R_i^j\ (t_1, \ldots t_j\ )$, where R_i^j represents a relation as described above and the t_i are terms. The atomic formulas are wffs.; they play the same role in FOL that letters do in propositional calculus as the basic building blocks of wffs. Further wffs. are generated as follows: if A and B are wffs., then $\neg A$, $A \rightarrow B$, and $(\forall x_i)A$ are wffs., where x_i is any variable.

The axiom schemes and inference rules of FOL inherit A1-A4 from propositional calculus. In addition, there are three axiom schemes and one inference rule related to quantifiers. To state them, we must distinguish between free and bound variables: a bound variable is governed by a quantifier; a free variable is not. A bound variable is attached to a quantifier and therefore is a dummy variable in the sense that the wff. in which it lives doesn't change meaning if you change it to another variable; the meaning of a wff. containing free variables has an incomplete meaning until those free variables are assigned objects from the domain. [3]

A5) $\forall x_i\ A \rightarrow A$, where x_i does not occur free in A

A6) $\forall x_i\ A(x_i) \rightarrow A(t)$, where t is substitutable[4] for x_i in A

A7) $\forall x_i\ (A \rightarrow B) \rightarrow (A \rightarrow (\forall x_i)B)$, x_i does not occur free in A

A8) if A, then $(\forall x_i)A$

When actual wffs. are substituted into axiom schemes A5-A7, they are all valid, just as propositional logic axioms are all tautologies. A valid wff. is a wff. that is true in every interpretation; in other words, regardless whether the domain of interpretation is women, natural numbers, etc. the mapping of symbols to real things in the domain always results in a true wff. if it is valid. Valid wffs. are FOL analogues to tautologies in propositional logic.

As in propositional logic, the above axiom schemes and inference rules are not unique. One selects the axioms such that one can prove two theorems: 1) The Soundness Theorem, which says that every theorem is a valid wff., and 2) The Completeness Theorem, which says that every valid wff. is a theorem.

An example of an extension of FOL is ZFC set theory. There are two predicates: equality and the membership relation, written = and $\in$, respectively. Each variable refers to a set. In particular, $y \in x$ means that the set y is a member of the set x; in other words, there is no separate notion of an element of a set. In addition to the axioms of FOL, ZFC has the following axioms as discussed in the main text:

null set axiom: $\exists x \forall y\ \neg(y \in x)$
There is a set x such that every set y is not a member of x, i.e. x is empty. x is called the null set and is usually written 0.

axiom of extensionality: $\forall x \forall y\ (\forall z\ z \in x \leftrightarrow z \in y) \leftrightarrow x = y$
Two sets are equal if and only if they have the same members. We have introduced the biconditional connective $\leftrightarrow$. $A \leftrightarrow B$ means $A \rightarrow B \wedge B \rightarrow A$, read *A if and only if B*. This axiom is not effective in the sense that it does not guarantee the existence of any set. Instead it defines what the equality symbol means.

axiom of pair: $\forall y \forall z \exists w \forall x\ x \in w \leftrightarrow x = y \vee x = z$

[3] The axioms in FOL are stated in terms of the universal quantifier only, since the existential quantifier can always be replaced using $\forall x_i A = \neg \exists x_i \neg A$. In words, *for all x A* is the same as *there is no x such that not A*.

[4]In this case, t is substitutable for x_i if x_i does not occur free in A within the scope of a $\forall x_j$, where x_j occurs in t.

Given sets y and z, there is a set w such that if x is in w, it must be y or z.

axiom of unions: $\forall y \forall z \exists x\, (y \in x \leftrightarrow \exists w\, (y \in w \wedge w \in z))$

There is a set x, written $\cup z$, whose members y are members of members of a given set z.

The usual union of sets a and b, written $a \cup b$, is $\cup\{a,b\}$, where $\{a,b\}$ denotes the set containing just a and b that is guaranteed to exist by the axiom of pair.

axiom of infinity: $\forall y \exists x\, (0 \in x \wedge (y \in x \rightarrow y \cup \{y\} \in x))$

There is a set x containing 0 such that for every y in x, the union of y and {y} is also in x.

replacement axiom scheme: $\forall y \exists_1 x\ A(y,x) \rightarrow$
$\forall u \exists v\, (u \in v \leftrightarrow \forall z \exists w\ w \in z \wedge A(w, u))$

The symbols $\exists_1 x\ P(x)$ mean *there exists a unique x* such that P(x). $\exists_1$ can be expressed in FOL; for example, $\forall y \exists_1 x\ A(y,x)$ is the same as $\forall y \exists x\ A(y,x) \wedge (\forall z\ A(y,z) \rightarrow z = x)$. In this case the predicate A can be viewed as a function of y because a unique value x is available for each y. The axiom scheme says that given a predicate A that acts as a function, there is a set v whose members are the range of A when A is viewed as a function.

powerset axiom: $\forall y \forall z\ \exists x\ y \in x \leftrightarrow y \subseteq z$

The notation $y \subseteq z$ means $\forall w\ w \in y \rightarrow w \in z$ and is read *y is a subset of z*. The powerset axiom states that there is a set x such that y is in x if and only if y is a subset of a given set z.

Appendix B Axiom of Foundation as a Theorem

In the language of first-order logic, the axiom of foundation is written

(Q) $\forall A(\exists x\, (x \in A) \rightarrow \exists x \forall y\, (x \in A \wedge y \in x \rightarrow y \notin A))$.

Proof of Q:

From rows 7, 8 and 9 of the *logic* epoch:

(P_1) $x \in y \rightarrow y\, !\!\in x$

(P_2) $x \in y, y \in z \rightarrow x \in z$

(P_3) $x\, !\!\in x.$

The proof uses the contrapositive $P_1 \wedge P_2 \wedge P_3 \rightarrow Q \leftrightarrow \neg Q \rightarrow \neg(P_1 \wedge P_2 \wedge P_3)$.

Assume $\neg Q$:

$(\neg Q) \quad \exists A(\exists x (x \in A) \wedge \forall x \exists y (x \in A \wedge y \in x \wedge y \in A))$.

We must show that A cannot exist unless $\neg P_1 \wedge P_2 \wedge P_3$. Let us consider a typical finite set A' = {p, q, r, s, t}. By $\neg Q$, p, q, r, s and t must themselves each have at least one member that is also in A'. Say $q \in p$, $r \in q$, $s \in r$, and $t \in s$. Now no matter what element of A' we choose to put in t, we must violate $P_1 \wedge P_2 \wedge P_3$: if $p \in t$, then $p \in t \in s \in r \in q \in p$, and by P_2 $p \in p$, which is disallowed by P_3; likewise, $q \in t$ is impossible since $q \in t \in s \in r \in q$, and $r \in t$ is impossible since $r \in t \in s \in r$; next, $s \in t$ can't happen because of P_1, since $t \in s$; finally, $t \in t$ is ruled out by P_3. In general, any $\in$-cycle, that is, any $\in$-sequence that repeats an element, is impossible if $P_1 \wedge P_2 \wedge P_3$. To complete the proof, we must establish that any set satisfying $\neg Q$ necessarily contains an $\in$-cycle.

To show that $\neg Q$ implies that $\in$-cycles must exist, we construct a one-to-one function[5] u from the ordinals onto A as follows, using the transfinite recursion theorem. Let x_0 be some element of A and t be a set not in A.

for $\alpha = 0$:

$$u(\alpha) = x_0$$

for $\alpha > 0$:

```
if $A - ran(u \restriction \alpha) \neq 0$
        $u(\alpha)$ = some element of the least element of $ran(u \restriction \alpha)$
otherwise
        $u(\alpha) = t$
```

The set $ran(u \restriction \alpha)$ is a well-ordered set by the membership relation $\in$ and so its least element exists as long as t is not one of its members. By $\neg Q$, that least element is non-empty, and by $P_1 \wedge P_2 \wedge P_3$, none of its elements is an element of $ran(u \restriction \alpha)$, so that $u \restriction \alpha$ is one-to-one. We are guaranteed that we actually get to the end of A by Hartog's theorem, which says that there exists for any set A an ordinal h(A) such that h(A) is the least ordinal which cannot be mapped into A by a one-to-one function.[162] Therefore, there must exist a $\mu < h(A)$ such that $u \restriction \mu$ is onto A.

Finally, we note that we can define another function v the same way by taking $v(0) = x_1$, where x_1 is an element of A not equal to x_0, and $v \restriction \lambda$ will also be onto A for some λ. Since u is onto A, we know that there is an ordinal β such that $u(\beta) = x_1$, and since v is also onto A, we know that there is a γ such that $v(\gamma) = x_0$. Therefore, there is a sequence

[5] A function f: $X \rightarrow Y$ is said to be one-to-one if f(a) = f(b) implies a=b, for all a,b in X.

mapping the $\in$ relation which takes the value x_0 at two different ordinals, implying that there is an $\in$-cycle associated with A. □

The axiom of choice is invoked explicitly in the above proof when we choose elements from the least element of the range of the functions being defined, and implicitly through the well-ordering principle, which always guarantees the existence of a least element. This is not incongruous with the fact that we have not yet established the axiom of choice in the matrix, since the axiom of foundation itself does not explicitly occur there.

Appendix C The Lambda Calculus

In what follows we present a variant of the so-called pure or untyped lambda calculus called the λη-calculus. The 'λ` (lambda) in the name refers to an arbitrary symbol used in the syntax of the calculus; the 'η` comes from the fact that this variant contains the so-called η-rule or extensionality conversion rule given below. Whereas the untyped lambda calculus is a language that precedes the language of first order logic in the hierarchy, a typed lambda calculus is an extension of the the untyped lambda calculus in which terms are explicitly given types such as *integer* or *boolean*. The extra syntax required to associate types with terms makes such a language recursively enumerable (see Appendix H); some form of typed lambda calculus has been used as the programming language for investigations of constructive type theory.

Definition of λ-term
Assume that the concept of a variable representing a λ-term is given. Then λ-term can be defined recursively as follows:

1) a variable is a λ-term
2) if M is a λ-term, then λx.M is a λ-term, where x is a variable (abstraction)
3) if M and N are λ-terms, then MN is a λ-term (application)

The variable x in 2) above is said to be bound in M, so that M, if it does in fact contain x, is meant to be the body of a function whose argument will be substituted for x everywhere in M; if M does not contain x, M is a constant function and its argument is discarded. 3) above shows how to supply an argument to a function: it is placed to the right. Taking an example from a later mathematical tier, $\lambda x.x^2\,3$ becomes 3^2 by the above definition.

Conversion rules
To define a theory using the language of lambda calculus, we need rules to tell us when two λ-terms are equal. Those rules are given as follows:

1. $(\lambda x.M)N = M[x:=N]$ (β)
2. $M = N \Rightarrow N = M$

3. $M = N, N = L \Rightarrow M = L$
4. $M = M$
5. $M = N \Rightarrow NZ = MZ$
6. $M = N \Rightarrow ZN = ZM$
7. $M = N \Rightarrow \lambda x.M = \lambda x.N$ (ξ)
8. $\lambda x.Mx = M$ (η)

Rule 1, called the β-conversion rule for historical reasons, is the rule used to reduce $\lambda x.x^2$ 3 to 3^2 in the above example; the notation M[x:=N] means *substitute N for all instances of x within M*.

Rules 2-4 are necessary to make equality an equivalence relation. '⇒` means *implies*; ',` means *and*.

Rules 5-6 say that left and right application preserve equality.

Rule 7, called the weak extensionality rule, says that the abstraction operator λx. preserves equality.

Rule 8, the extensionality rule, can be shown using rules 1 and 7 to be equivalent to the rule $Mx = Nx \Rightarrow M = N$. This rule says that functions are to be considered equal if their output is equal for an arbitrary input, regardless how their output is produced. Rule 8 assumes that x is not a free variable of M.

Association of terms is left to right, so that ABC = (AB)C. Parentheses are often left out.

Examples of λ-terms

1. λx.x : This is the identity function. Applying it to any term gives that term back, so that $\lambda x.x\ A \rightarrow A$, where '→ ` means *reduces to*. The term following the '.` terminating the abstraction operator, in this case x, is a function body. The following term, in this case A, is an argument to that function body.

   In the following examples, as an aid to the reader we use **bold** text for arguments and *italic* *underlined* text for the variables in the function body they are going to replace.

2. $\lambda y.y^2$ $\lambda y.\underline{y}^2$ **B** $\rightarrow B^2$. We have used y as the bound variable here to illustrate that the name of the bound variable does not matter, i.e. that the λ-terms $\lambda y.y^2$ and $\lambda x.x^2$ and $\lambda apple.apple^2$ have the same meaning.

3. λx.x λx.x λx.*x̲* **λx.x** → λx.x. Here the argument to the identity function is the identity function itself. By rule 1, λx.x λx.x = x[x:= λx.x] = λx.x. This

application is allowed since λx.x is a λ-term according to 2) of the definition above.

4. λx.(x x) This function applies its argument to its argument. For example,

$\lambda x.(\underline{x}\ \underline{x})\ \mathbf{\lambda x.x}$
$\lambda x.\underline{x}\ \mathbf{\lambda x.x}$
$\lambda x.x$

Applied to itself, this function gives

$\lambda x.(\underline{x}\ \underline{x})\ \mathbf{\lambda x.(x\ x)}$
$\lambda x.(\underline{x}\ \underline{x})\ \mathbf{\lambda x.(x\ x)}$
$\lambda x.(\underline{x}\ \underline{x})\ \mathbf{\lambda x.(x\ x)}$, and so on, endlessly.

5. (λx.(λy.(x))

This function selects the first of two arguments. y is assumed not to be free in A (i.e., A is assumed not to be a function of y).

$(\lambda x.(\lambda y.(\underline{x}))\mathbf{A})B$
$\lambda y.(A)\ \mathbf{B}$
A

The second argument is tossed out because the A does not depend on y. It is customary to write λx.λy. as λxy., although we don't follow this convention here.

6. (λx.(λy.(y))

This function selects the second of two arguments.

$\lambda x.(\lambda y.(y))\mathbf{A}B$
$\lambda y.(\underline{y})\ \mathbf{B}$
B

The first argument is discarded because x does not appear in (λy.(y)), the function body associated with λx.

7. λx.(λy.(x y))

This function applies x to y.

$(\lambda x.(\lambda y.(\underline{x}\ y))\mathbf{A})B =$
$\lambda y.(A\ \underline{y})\mathbf{B} =$
AB

8. λx.(λy.(λz.(z x)y))

This function takes the third term it encounters and places it in front of the first two terms it encounters.

$((\lambda x.(\lambda y.(\lambda z.(z\ \underline{x})y))\mathbf{A})B)C =$
$(\lambda y.(\lambda z.(z\ A)\underline{y})\mathbf{B})C =$
$(\lambda z.(\underline{z}\ A)B)\ \mathbf{C} =$
CAB

This function is called the pairing function, since a function can be used for C that operates on A and B. For example, to select A, in place of C we use (λu.(λv.(u)) from example 5. above.

((λx.(λy.(λz.(z $\underline{x}$)y))**A**)B)(λu.(λv.(u)) =
(λu.(λv.(u))AB =
A

Similarly, the second operand can be selected by substituting (λu.(λv.(v) from example 6 above for C. These two selecting functions from examples 5 and 6 are useful and will be called select_first and select_second.

Notice that if we can devise a method for defining C so that it returns select_first or select_second, such a C in combination with the pairing function would choose between the two terms A and B just as a conditional does: if C then A else B. Since such functions can be devised, select_first is also called *true* and select_second *false*.

Using the pairing function in this way it is possible to develop a system of numbers in λ-calculus and do calculations with them, as we sketch in the following examples.[163]

9. λx.(x select_first)

This function can be used as an *is_zero* function, if we define *zero* as the identity function λx.x. Then

λx.(x select_first) (zero) =
zero select_first =
λx.x select_first =
select_first =
true

10. λx.(λy.((y select_second) x))

This function can be used to define a successor function. Then the number 1 is given by 1 = successor(0):

λx.(λy.((y select_second) $\underline{x}$)) (**zero**) =
λy.((y select_second) zero)

The number 2 is given by 2 = successor(1):

λx.(λy.((y select_second)$\underline{x}$))**(λy.((y select_second)zero)) =**
λy.((y select_second) (λy.((y select_second) zero))) =
λy.((y select_second) (λz.((z select_second) zero)))

In the last line, the nested dummy variable y was replaced by z to avoid confusion.

Now, is_zero(1) is given by:

λx.($\underline{x}$ select_first) **λy.((y select_second) zero)**
λy.(($\underline{y}$ select_second) zero) **select_first =**

(select_first select_second) zero =
select_first select_second zero =
select_second =
false.

One gets the same result for is_zero(2), is_zero(3) and so on. For example, is_zero(2) is:
λx.($\underline{x}$ select_first) **λy.((y select_second) (λz.((z select_second) zero)))** =
λy.(($\underline{y}$ select_second) (λz.((z select_second) zero))) **select_first** =
(select_first select_second) (λz.((z select_second) zero))
select_second =
false.

10. λx.(((is_zero x) zero) (x select_second))

This function defines the predecessor function. Operating on 1, for example, it gives 0:

λx.(((is_zero $\underline{x}$) zero) ($\underline{x}$ select_second)) **λy.((y select_second) zero)** =
(is_zero λy.((y select_second) zero))
(λy.((y select_second) zero) select_second) =
select_second λy.(($\underline{y}$ select_second) zero) **select_second** =
select_second (select_second select_second) zero =
zero

Operating on 0:
λx.(((is_zero $\underline{x}$) zero) ($\underline{x}$ select_second)) **zero** =
((is_zero zero) zero) (zero select_second) =
(select_first zero) (zero select_second) =
select_first zero (zero select_second) =
zero

The predecessor of 2 is:
λx.(((is_zero $\underline{x}$) zero) ($\underline{x}$ select_second)) **λy.((y select_second) (λz.((z select_second) zero)))** =
((is_zero λy.((y select_second) (λz.((z select_second) zero)))) zero)
(λy.((y select_second) (λz.((z select_second) zero))) select_second) **=**
select_second zero(λy.((y select_second) (λz.((z select_second) zero))) select_second) **=**
(λy.((y select_second) (λz.((z select_second) zero))) select_second) **=**
select_second select_second (λz.((z select_second) zero))) =
λz.((z select_second) zero)) = 1.

11. λf.(λr.(f (r r)) λs.(f(s s)))

This function is referred to as the Y combinator. When applied to any term F, it gives a so-called fixed point of that term (see Appendix E):

λf.((λr.*f* (r r)) (λs.*f*(s s))) **F** =
λr.(F (r r)) **λs.(F (s s))** =
F(λs.(F (s s)) (λs.(F (s s))) =
F(λr.(F (r r)) (λs.(F (s s))).

In the last step we renamed the first instance of the variable s to r to emphasize that r and s are just dummy variables. The Y combinator can be used to implement addition, multiplication and other recursive functions. For example, if F is given by

F = λf.**(**λx.**[**λy.**{**((is_zero y) x) ⌈f succ(x) pred(y)⌉**}])**,

then F(λr.(F (r r)) (λs.(F (s s))) is the addition function, as we demonstrate next. To help clarify which parentheses are matched, we have substituted some matching pairs of parentheses with **( )**,**[]**,**{}**, and⌈⌉.

F(λr.(F (r r)) (λs.(F (s s))) =
λf.**(**λx.**[**λy.**{**((is_zero y) x) ⌈*f* succ(x) pred(y)⌉**}]) (λr.(F (r r)) λs.(F (s s)))** =
λx.**[**λy.**{**((is_zero y) x) ⌈λr.(F (r r)) λs.(F (s s)) succ(x) pred(y)⌉**}]** =
λx.**[**λy.**{**((is_zero y) x) ⌈VV succ(x) pred(y)⌉**}]**

In the last step we substituted VV for λr.(F (r r)) λs.(F (s s)) in order to improve legibility. Using *one* in place of λy.((y select_second) zero) and *two* in place of λy.((y select_second) (λz.((z select_second) zero)), we show that the above function computes the sum of 1 and 2 as follows:

λx.**[**λy.**{**((is_zero y) x) ⌈VV succ(x) pred(y)⌉**}] one** two =
λy.**{**((is_zero y) one) ⌈VV succ(one) pred (y)⌉**} two** =
((is_zero two) one) ⌈VV succ(one) pred(two)⌉ =

Since (is_zero two) returns false, i.e. select_second, ((is_zero two) one) is discarded.

VV succ(one) pred(two) =
λr.(F (r r)) λs.(F (s s)) (succ(one)) (pred(two)) =
F(λr.(F (r r)) λs.(F (s s))) (succ(one)) (pred(two)) =
λx.**[**λy.**{**((is_zero y) x) ⌈VV succ(x) pred(y)⌉**}] (succ(one))**(pred(two)) =
λy.**{**((is_zero y) succ(one)) ⌈VV succ(succ(one)) pred (y)⌉**} (pred(two))** =
((is_zero pred(two)) succ(one)) ⌈VV succ(succ(one)) pred(pred(two))⌉ =

Again, (is_zero pred(two)) returns false, and ((is_zero(pred(two))) succ(one)) is discarded.

VV succ(succ(one)) pred(pred(two)) =
λr.(F(r r)) λs.(F(s s)) succ(succ(one)) pred(pred(two)) =

F((λr.(F(r r))λs.(F(s s))) succ(succ(one)) pred(pred(two)) =
λx.**[**λy.{((is_zero y) $\underline{x}$) ⌈VV succ($\underline{x}$) pred(y)⌉}**]** **succ(succ(one))** pred(pred(two)) =
λy.{((is_zero $\underline{y}$) succ(succ(one))) ⌈VV succ(succ(succ(one))) pred ($\underline{y}$)⌉}
pred(pred(two)) =
((is_zero pred(pred(two))) succ(succ(one))) ⌈VV succ(succ(succ(one)))
pred(pred(pred(two)))⌉ =

Finally, since (is_zero pred(pred(two))) returns true, i.e. select_first, the expansions of WW come to an end and the final result is

succ(succ(one)) = 3.

Appendix D Combinatory Logic

Combinatory logic (CL) comes in two flavors, the so-called pure CL that has no logical connectives (such as *not*, *and*, *or* and so on) and the so-called illative CL, developed by Curry with the intention of making it a basis for all logic and mathematics, which does have logical connectives. We present here pure CL as the language of the zeroth recursive call; it has only one truth value, truth, as explained in Appendix E, and therefore has no need for logical connectives.

Definition of CL-terms

Assume that the concept of a variable representing a CL–term is given. Then CL-term can be defined recursively as follows:

1) a variable is a CL-term
2) K is a CL-term
3) S is a CL-term
4) if variables A, B are CL-terms, then AB is a CL-term

The term AB is called an *application*.

Axioms and Rules

1. **K**PQ = P
2. **S**PQR = PR(QR)
3. P = Q ⇒ Q = P
4. P = Q, Q = R ⇒ P = R
5. P = P
6. P = P’ ⇒ PR = P’R

7. P = P' ⇒ RP = RP'

Axioms 1 and 2 define **K** and **S**. For any CL-terms M and N, **SK**MN = **K**N(MN) = N, which means that the identity operator **I** is given by **SK**M for any CL-term M. In particular, **I** = **SKK** and therefore **I**P = P need not appear as an axiom.

Rules 3-5 are necessary to make equality an equivalence relation.

Rules 6-7 say that left and right application preserve equality.

The convention for application is that association is on the left, so that ABC = (AB)C.

Examples of CL-terms

The following terms will be used in what follows.

B ≡ S(**KS**)**K**

B is a composition operator since **B**PQR = P(QR). Proof:
S(**KS**)**K**PQR = **KS**P(**K**P)QR = **S**(**K**P)QR = **K**PR(QR) = P(QR). □

C ≡ S(BS(BKS))(KK)

C swaps the second and third terms it sees: **C**PQR = PRQ.
Proof: **S**(**BS**(**BKS**))(**KK**)PQR = **BS**(**BKS**)P(**KK**P)QR =
S((**BKS**)P)(**KK**P)QR = **BKS**PQ((**KK**P)Q)R = **K(S**P)Q((**KK**P)Q)R
= **S**P((**KK**P)Q)R = PR(((**KK**P)Q)R) = PR(**K**QR) = PRQ. □

W ≡ CSI

W duplicates the second term it sees: **W**PQ = PQQ. Proof: **CSI**PQ
= **S**PIQ = PQ(IQ) = PQQ.

P ≡ S(S(KS)(S(KK)(S(KS)(S(S(KS)(SK))K))))(KK)

P puts the third term it sees in front of the first two: **P**xyz = zxy.
Proof: **S(S(KS)(S(KK)(S(KS)(S(S(KS)(SK))K))))(KK)**xyz =
S(KS)(S(KK)(S(KS)(S(S(KS)(SK))K)))x(**(KK)**x)yz =
(KS)x(**(S(KK)(S(KS)(S(S(KS)(SK))K)))**x)(**(KK)**x)yz =
S((S(KK)(S(KS)(S(S(KS)(SK))K))) x) (**(KK)**x)yz =
S((S(KK)(S(KS)(S(S(KS)(SK))K))) x) **K**yz =
((S(KK)(S(KS)(S(S(KS)(SK))K))) x)y**(K**y)z =
S(KK)(S(KS)(S(S(KS)(SK))K)) xy **(K**y)z =
S(KK)(S(KS)(S(S(KS)(SK))K)) xy **(K**y)z =
(KK)x(**(S(KS)(S(S(KS)(SK))K))** x)y**(K**y)z =
KKx(**(S(KS)(S(S(KS)(SK))K))** x)y**(K**y)z =
K ((S(KS)(S(S(KS)(SK))K)) x)y**(K**y)z =
(S(KS)(S(S(KS)(SK))K))x**(K**y)z =
S(KS)(S(S(KS)(SK))K)x**(K**y)z =

(KS)x**((S(S(KS)(SK))K)**x**)** **(K**y**)**z =
S((S(S(KS)(SK))K)x**)** **(K**y**)**z =
((S(S(KS)(SK))K)x**)**z**((K**y**)**z**)** =
((S(S(KS)(SK))K)x**)**zy =
S(S(KS)(SK))Kxzy =
S(KS)(SK)x**(K**x**)**zy =
(KS)x**((SK)**x**)(K**x**)**zy =
S((SK)x**)(K**x**)**zy =
((SK)x**)**z**((K**x**)**z**)**y =
SKxz**((K**x**)**zy =
Kz(xz)**((K**x**)**zy =
z**((K**x**)**z**)**y =
z**((K**x**)**z**)**y =
z**K**xzy=
zxy

Abstraction Extraction Algorithm
Combinatory logic is studied by computer scientists because it has practical applications in the implementation of functional programming languages. Originally, however, it was developed by Schoenfinkel and Curry as a precursor to functional abstraction. In order to illuminate the definitions of **K** and **S**, we show the relationship between CL and abstraction.[164]

Say that M is a term that we want to turn into an abstraction. First we pull out some part of M, labelling it x wherever it is found in M, which gives a new term, say f. To get M back, we must apply f to x, which means substituting x wherever the label x is found in M. We can write down this process using a pseudo-abstraction operator λ∗, as follows:

λ∗x.M = f
fx = M

From this perspective, f is an abstraction from M, and M is an application of f. In general, if we want to create an abstraction from a term, we must consider the same process for three possible forms of that term:

1. The term is identical to the variable we wish to abstract over. In that case, the CL equivalent to the abstraction is just **I**:
   λ∗x.x = **I**
   Ix = x

2. The term is a constant, say a. Then the CL equivalent to abstraction is **K**a:
   λ∗x.a = **K**a
   Kax = a

3. For the general case the term is an application, so it consists of two terms, both of which must be operated on with the abstraction operator. Then the CL equivalent to abstraction involves **S**:

$$\lambda{*}x.AB \;=\; \mathbf{S}(\lambda{*}x.A)(\lambda{*}x.B)$$
$$\mathbf{S}(\lambda{*}x.A)(\lambda{*}x.B)x \;=\; \lambda{*}x.Ax(\lambda{*}x.Bx) = AB,$$

where in the last step we applied what in lambda calculus would be the η-rule given in Appendix C.

The above informal argument indicates why **K** and **S** work as generators for all CL-terms: using only application, they can behave in a manner similar to abstraction. The argument also suggests that if we add something equivalent to the η-rule to CL, we get essentially the λη- calculus, which is consistent with the step from intensional operators in the zeroth recursive call to extensional operators in the first recursive call. Indeed, it can be shown that adding the extensionality rule $PR = P'R \Rightarrow P = P'$ to CL results in a theory equivalent to λη- calculus.[165]

Appendix E Truth in Combinatory Logic and Lambda Calculus

One of the main theorems of combinatory logic and lambda calculus is the fixed point theorem.[166] We first state and prove this theorem.

Fixed point theorem for combinatory logic
For any term F, there is another term X such that X = FX.

Proof: Let $W \equiv \mathbf{B}F(\mathbf{SII})$ and let $X \equiv VV$. Then $X = VV = \mathbf{B}F(\mathbf{SII})V = F((\mathbf{SII})V) = F(\mathbf{I}V(\mathbf{I}V)) = F(VV) = FX$. □

Fixed point theorem for lambda calculus
For any term F, there is another term X such that X = FX.

Proof: Let $V \equiv \lambda x.F(xx)$ and let $X \equiv VV$. Then $X = \lambda x.F(xx)\ V = F(VV) = FX$. □

The above proof suggests a way to always find the fixed point of any term F. Let $Y \equiv \lambda F.VV$. Then $YF = (\lambda F.VV)F = (\lambda F.(\lambda x.F(xx)\ \lambda x.F(xx)))\ F = \lambda x.F(xx)\ \lambda x.F(xx) = \lambda x.F(xx)\ V = F\ (VV) = F\lambda F.(VV)F = FYF$, where in the second-to-last step we used the η- rule in setting $VV = \lambda F.(VV)F$. Therefore, $YF = FYF$, and YF is a fixed point of F. Y is called the Curry combinator.

Negation
Since falsehood is the negation of truth, in order for falsehood to have meaning in combinatory logic, there must be a term that represents negation. We now give a familiar argument that no such term exists.[167]

Suppose that N represents negation. Now let G = **W**(**B**N). Then GG = **W**(**B**N)G = **B**NGG = N(GG). This is a contradiction, since we have a term GG such that it is equal to its own negation. Therefore, we conclude there can be no term that represents negation in combinatory logic. A similar argument can be made for lambda calculus.[168]

Implication

We now show that if a term is introduced to represent implication in lambda calculus, then every term in lambda calculus must be true.[169] Let '$\Rightarrow$' stand for a term that denotes implication. We assume that the following two rules are valid:

i. if X and $X \Rightarrow Y$, then Y

ii. $(X \Rightarrow (X \Rightarrow Y)) \Rightarrow (X \Rightarrow Y)$

1) is the modus ponens rule and 2) is a tautology, that is, a statement that is true regardless what the true-false values of X and Y are, as shown in Appendix A.

From our discussion of the fixed point theorem, we know that if Y is a Curry combinator, then xYx = Yx for any term x. For an arbitrary term Z, let $x \equiv (\lambda z.(z \Rightarrow (z \Rightarrow Z)))$, and $X \equiv Yx$. Then:

1.	$xX = X$	(from $xYx = Yx$ and $X = Yx$)
2.	$(\lambda z.(z \Rightarrow (z \Rightarrow Z)))X = X$	(by substituting for x)
3.	$(X \Rightarrow (X \Rightarrow Z)) = X$	(by β-rule of lambda calculus)
4.	$(X \Rightarrow (X \Rightarrow Z)) \Rightarrow (X \Rightarrow Z) = X$	(using $(X \Rightarrow (X \Rightarrow Z)) = X$ for leftmost X in 3)

Now, for any Z, we can deduce the following:

5.	$(X \Rightarrow (X \Rightarrow Z)) \Rightarrow (X \Rightarrow Z)$	(by ii)
6.	$X \Rightarrow (X \Rightarrow Z)$	(by 5, 3 and 4)
7.	X	(by 6 and 3)
8.	$X \Rightarrow Z$	(by 6 and 7)
9.	Z	(by 7, 8 and i)

Since Z was an arbitrary term, it follows that all terms in lambda calculus are true propositions. A similar argument can be made for combinatory logic.[170]

Appendix F Model Theory

Model theory is a mathematical theory that maps a formal language, itself a mathematical structure, to another mathematical structure that is in some sense a realization of that language. Technically, a model is a pair: the mapping just mentioned, and the domain of interpretation into which the formal language is mapped, that domain being a set. The mapping maps symbols in the language to elements of the domain and to functions and relations on the domain.

Model theory provides a framework for determining what sentences in a formal language are true, either in a particular model or in all models. Usually, the formal language is an extension of first-order logic in the sense that one takes the usual axioms of first order logic as given and then adds whatever symbols are needed for special functions or constants or relations in the model. To complete the extension, one gives a set of additional axioms. For example, any group is a model whose language G is built upon first order logic as follows.

First, one defines the following symbols:

*	binary operation
e	the identity element
x, y, z	variables
( )	punctuation indicating precedence

In addition to the usual axioms and inference rules for first order logic with equality given in Appendix A, we add the following axioms:

G1) $\forall x\, (x * e = x \wedge e * x = x)$ (identity)

G2) $\forall x \forall y \forall z\, ((x * y) * z = x * (y * z))$ (associativity)

G3) $\forall x \exists y\, (x * y = e \wedge y * x = e)$ (inverse)

A sentence in first order logic is a wff. that does not contain free variables.[6] As an example of a sentence in this language, one can say that the identity element 1 is unique:

(S) $\forall y\ \forall x\, ((x * y = x \wedge y * x = x) \rightarrow y = e\,).$

Proof: Assume there is another element e' such that $\forall x\, (x * e' = x, e' * x = x)$. Then, in particular, e*e' = e. But we also know from G1 that e*e' = e'. Therefore, e = e*e' = e', or e = e'. □

Although the above informal proof of S does not show the steps explicitly, S is a theorem of G because it can be deduced from the language G by means of the axioms of G and its inference rules. The concept of theorem is a syntactic concept, which means that it has to do with formal manipulation within a language. In order to establish what

[6] See Appendix A for the definition of a free variable.

the theorem actually means, we require the semantic notion of validity. By definition a sentence in a language is valid if and only if it is true in every model.[7,8] This definition seems at first glance cumbersome since it requires dealing with every model of a language. However, Goedel's completeness theorem for first order logic obviates this requirement by linking syntax and semantics: it says that a sentence is a theorem of a language if and only if it is valid. Since the sentence S is a theorem, it is also a valid sentence.

Any group is a model of the language G. In particular, any cyclic group of order n is a model of G. A cyclic group of order n is a group of n elements in which each element can be written in terms of a single element g, called a generator, in the form g^m, where m is an integer such that $0<= m < n$. A typical cyclic group of order n is Z_n, the group formed by addition of integers modulo n. For example, the group Z_4 consists of elements {0, 1, 2, 3}, with the identity element given by 0. Addition modulo 4 gives 0+m=m+0=m for m $\in Z_4$; 1+1=2; 2+1=1+2=3; 3+1=1+3=0; 3+2=2+3=1; 2+2=0; and 3+3=1. The generator for Z_4, as for all groups Z_n, is 1: $0 = 1^0$, $1 = 1^0 * 1^1 = 1^{0+1} = 1^1$; $2 = 1^1 * 1^1 = 1^{1+1} = 1^2$ and $3 = 1^2 * 1^1 = 1^{2+1} = 1^3$. To see that Z_4 is a group, note that G1) is satisfied, since $1^m * 1^0 = 1^m$ and $1^0 * 1^m = 1^m$ for m$\in Z_4$; G2) is satisfied since $(1^a * 1^b) * 1^c = 1^a * (1^b * 1^c) = 1^{(a+b)+c} = 1^{a+(b+c)}$ for all a, b, c $\in Z_4$; G3) is satisfied, since $1^m * 1^n = 1^n * 1^m = 1^{m+n} = 1^0$, where m, n $\in Z_4$ and m+n = 0 modulo 4. A further example of a cyclic group is Z_1, which contains only the identity 1^0.

The cyclic groups discussed above illustrate how model theory relates a language to a model by mapping symbols to elements of the model. In the case of the cyclic group Z_4, for example, the symbol e from the language G was mapped to the additive identity and the symbol * was mapped to the binary operation addition modulo 4. Once such mappings are made, it is possible to define valid sentences.

The model theory sketched above is based on the language of first order logic and was the first model theory to be developed. There are also model theories of other logics such as multi-valued logic, intuitionist logic, modal logic, and so on, that have been patterned after the first order theory, and there is a categorical model theory as well. One can also construct a toy model theory corresponding to the language of propositional logic that is useful for pedagogical purposes, but has little practical value since even the simplest of mathematical structures require quantifiers for their description.[171]

The model theory of combinatory logic is even simpler than the model theory of propositional logic, since all terms in pure combinatory logic represent true sentences.

[7] An equivalent definition says that a sentence S is valid if and only if every model *satisfies* S. For details on satisfaction and truth in a first order model, see for example the first chapter of the first year graduate-level text by Chang and Keisler, *Model Theory*, North Holland, Amsterdam (1990).

[8] In Appendix A, a wff. is defined to be valid if it is true in every (domain of) interpretation. The mapping from symbols to the interpretation in FOL is called a valuation. A model is a special kind of interpretation-valuation pair in that it is concerned with sentences (wffs. without free variables) as opposed to just wffs. Sentences are of interest because they don't depend on variables. For example, a theory is a set of sentences.

This means there is no deduction in the theory. Furthermore, we restrict our language to just the term **I** in combinatory logic, so that there is essentially just one sentence in the language: **I*I** = **II** = **I**, which is an axiom of the language and is trivially true.[9] The language therefore consists of a constant symbol **I**, a function symbol *, and a relation symbol =, which correspond to the model consisting of God, application, and equality.

Appendix G Lattice Theory

A lattice is a partially ordered set in which each pair of elements share a certain kind of upper and lower bound. Before defining lattice, it is necessary to introduce several definitions.

Partial Order

A *partial order* on a set A is a binary relation $\leq$ with the following three properties:

1) for all a, b $\in$ A,
   if $a \leq b$ and $b \leq a$, then $a = b$ (antisymmetry)
2) for all a, b, c $\in$ A,
   if $a \leq b$ and $b \leq c$, then $a \leq c$ (transitivity)
3) for all a $\in$ A,
   $a \leq a$ (reflexivity)

A set with a partial order is said to be *partially ordered*; such a set is called a poset. Rows 7, 8 and 9 of the zeroth epoch of the matrix define a partial order on a collection of primitive objects rather than on a set.

Upper Bound

An element x of a poset A is an *upper bound* of a subset S of A if for all $s \in S$, $s \leq x$.

Lower Bound

An element x of a poset A is a *lower bound* of a subset S of A if for all $s \in S$, $x \leq s$.

Least Upper Bound

The upper bounds of a subset S of a poset A form a set S^U. If there is an element x of S^U such that $x \leq u$ for all $u \in S^U$, that element is called the *least upper bound* of S.

The least upper bound of a set S is unique. Proof: Let x_1 and x_2 be least upper bounds of S. Then $x_1 \leq x_2$ and $x_2 \leq x_1$, and by antisymmetry of the binary relation $\leq$, $x_1 = x_2$.

[9] **I** = **I** is also a sentence.

Not all partially ordered sets have a least upper bound. An example is the subset of rational numbers x such that $x^2 < 2$. Since $\sqrt{2}$ is not rational, i.e. cannot be written in the form m/n where m and n are integers and n is non-zero, any rational number as a candidate least upper bound of this set will be found to be inadequate. For example, $17/12 - \sqrt{2}$ is 0.002 approximately, but $577/408 - \sqrt{2}$ is approximately 0.000002, and so on.

Greatest Lower Bound
The lower bounds of a subset S of a poset A form a set S^L. If there is an element x of S^L such that $\ell \leq x$ for all $\ell \in S^L$, that element is called the greatest lower bound of S. The greatest lower bound of a subset S is unique again by antisymmetry of ≤. An example of a poset without a greatest lower bound is the subset of rational numbers x such that $x^2 > 2$.

The greatest lower bound of a subset S of a poset A is also called the *infimum* of S, written inf S. Likewise, the least upper bound is called the *supremum* of S, written sup S.

Lattice
There are two equivalent definitions of a lattice. The first goes as follows:

Lattice definition, version 1. If each subset of a poset A formed by pairs of elements of A has a least upper bound and a greatest lower bound, then A is a *lattice*.

The second definition requires the concepts of join and meet. The least upper bound of a pair of elements x and y of a set is called the *join* of x and y, written $x \vee y$. The greatest lower bound of a pair of elements x and y of a set is called the *meet* of x and y, written $x \wedge y$.

Lattice definition, version 2. A *lattice* is a poset A such that for all elements a, b and c of A:

$(a\vee b)\vee c = a\vee(b\vee c)$	(associativity)
$(a\wedge b)\wedge c = a\wedge(b\wedge c)$	
$a\wedge b = b\wedge a$	(commutativity)
$a\vee b = b\vee a$	
$a\wedge a = a$	(idempotency)
$a\vee a = a$	
$a\vee(a\wedge b) = a$	(absorption)
$a\wedge(a\vee b) = a$	

The *connecting lemma* relates $\wedge$ and $\vee$ to $\leq$. It says the following are equivalent for elements a and b of a lattice:

i) $a \leq b$

ii) $a \vee b = b$

iii) $a \wedge b = a$

A *complete lattice* is a lattice A such that sup S and inf S exist for all $S \subseteq A$.

Orthocomplement

Posets may contain a greatest element and/or a least element. Depending on context these elements are called 1 and 0, or $\top$ and $\bot$ (read *top* and *bottom*), respectively. For all elements a of a poset containing 1 and 0, the following hold:

$$0 \wedge a = 0 \qquad 0 \vee a = a$$
$$1 \wedge a = a \qquad 1 \vee a = 1$$

a' is an *orthocomplement* of a if the following hold:

i) $a'' = a$

ii) $a' \vee a = 1$

iii) $a' \wedge a = 0$

iv) $a \leq b \rightarrow b' \leq a'$

The above conditions define the unary operation '. Conditions i) and iv) together are equivalent to $a \leq b$ if and only if $b' \leq a'$, since i) implies $a \leq b \rightarrow b' \leq a'$ and $b' \leq a' \rightarrow a'' \leq b''$, which by iv) means $b' \leq a' \rightarrow a \leq b$.

A poset A is said to be *orthocomplemented* if for each element a of A there exists an orthocomplement a' in A. Such a set is called an *orthoposet*. An orthoposet that is also a lattice is called an *ortholattice*.

Modularity and Orthomodularity

A lattice A is *modular* if for all a, b and c in A,

$$a \vee (b \wedge (a \vee c)) = (a \vee b) \wedge (a \vee c).$$

An ortholattice A is *orthomodular* if for all a, b in A,

$$a \leq b \rightarrow a \vee (a' \wedge b) = b.$$

Distributivity

A distributive lattice A obeys the following for all a, b, and c in A:

$$a\vee(b\wedge c) = (a\vee b)\wedge(a\vee c)$$
$$a\wedge (b\vee c) = (a\wedge b)\vee(a\wedge c)$$

Boolean Algebra
An ortholattice obeying the distributive law is a *boolean algebra*. This characterization is equivalent to the usual definition of a boolean algebra given in terms of * and + in the main text rather than the equivalent operators $\wedge$ and $\vee$.

A boolean algebra is an orthomodular lattice. Proof: Since a boolean algebra A is an ortholattice, $a'\vee a = 1$, for all a in A. From the distributive law $a \vee(a'\wedge(a\vee b)) = (a\wedge a') \wedge (a\vee(a\vee b)) = a\vee b$ for all a and b in A. Since $a \leq b$ is equivalent to $a\vee b = b$ by the connecting lemma, $a \vee(a'\wedge(a\vee b)) = a\vee b$ fulfills the orthomodularity requirement $a \leq b \rightarrow a \vee(a'\wedge b) = b$. □

Equivalent Definitions of Orthomodularity
The following theorem gives several equivalent definitions of an orthomodular lattice. Its formulation requires two definitions.

A *subalgebra* Γ generated by a subset {a,b} of an ortholattice L is a subset of L closed under the operations ', $\wedge$ and $\vee$ on {a,b}and containing 0 and 1. In other words, one starts with a, b, 0 and 1, and then adds a', b', $a\vee b$, $a'\vee b$, $a\vee b'$, $(a\vee b)'$, $(a\vee b')'$, $(a'\vee b)'$, $a\wedge b$, $a'\wedge b$, $a\wedge b'$, $(a\wedge b)'$, $(a\wedge b')'$, $(a'\wedge b)'$, $(a\vee b)\vee(a\vee b)$, $(a\vee b)\wedge(a\vee b)$, $((a\vee b)\vee(a\vee b))'$, and so on until the operations ', $\wedge$ and $\vee$ on any element or elements of the set being generated only yield elements already generated.

Given elements a and b of an ortholattice L, one says that a *commutes* with b if $a = (a\wedge b)\vee(a\wedge b')$. The notation aCb indicates that a commutes with b.

Theorem. Given an ortholattice L with elements a and b, the following statements are equivalent:

1. L is orthomodular.
2. if $a \leq b$ and $b\wedge a' = 0$, then $a = b$
3. The ortholattice O_6 in figure G1 (10) is not a subalgebra of L
4. if $a \leq b$, then the subalgebra generated by {a,b} is a boolean algebra
5. aCb if and only if bCa

The proof of this theorem can be found in the standard textbook on orthomodular lattices by Kalmbach.[172]

Examples
Figure G1 shows a number of posets in the form of Hasse diagrams. A Hasse diagram shows all elements of a poset so that the ordering relation between elements is manifest, using the concept of a covering relation. An element b of a poset is said to cover another element a if $a < c \leq b$ implies that $c = b$ ($x < y$ means $x \leq y$ and $x \neq y$); in other words, b covers a if $a < b$ and there are no elements between a and b. In a Hasse diagram lines are drawn between all elements related by a covering such that the covering element appears above the covered element, that is, such that the y coordinate in the usual Cartesian coordinates is greater for the covering element than for the covered element. The elements themselves are drawn as little circles or dots.

Figure G2 shows two lattices, both orthomodular. The construction of the second of these lattices illustrates the application of a theorem, according to which every orthomodular lattice is a pasting of its maximal boolean subalgebras.[173]

As mentioned in the main text, the propositions of propositional logic form a boolean algebra; in terms of lattice theory, the operations and, or and negation are given by the meet, join and complementation, respectively, and implication is the ordering relation. Likewise, the set of subsets of a set forms a boolean algebra, in which the meet, join and complementation are given by set intersection, union and complement, and the ordering relation is set inclusion. Figure G3 shows the Hasse diagram for the power set of four elements. Note that in terms of subsets of a set, the orthomodularity condition $a \leq b \rightarrow a \vee (a' \wedge b) = b$ reads as follows: if a is a subset of b, then adding a to b minus a results in b. Similarly, in propositional logic the orthomodularity condition $a \vee (a' \wedge (a \vee b)) = a \vee b$ is a tautology.

Spacetime as an orthomodular lattice
To show that the lattice structure of spacetime is orthomodular is beyond the scope of this exposition.[174, 175] However, it is possible to sketch this structure in broad strokes and show for a two-dimensional spacetime what the lattice elements look like. The idea is that subsets of spacetime points are partitioned into equivalence classes based on taking the complement twice, whereby the complement of a subset S of spacetime consists of all points that have a so-called spacelike separation from all points in S.

In figure G4 there is shown a two-dimensional spacetime, with one space dimension given by the cartesian x-axis and one time dimension given by the y-axis. Points on this plot, expressed as pairs in the form (x,t), can be associated with events at a certain place and time, say the position of a particle, as viewed from some inertial, that is, unaccelerated reference frame. Spacetime intervals between two points $a_1 = (x_1, t_1)$ and $a_2 = (x_2, t_2)$, with $t_1 < t_2$, can be divided into two classes due to the constant speed of light in all inertial reference frames: 1) the timelike intervals, those for which light has enough time to travel between a_1 and a_2, and 2) the spacelike intervals, those for which light does *not* have enough time to travel between x and y. For convenience, we use units such that the speed of light is 1 space unit per 1 time unit, in which case light travels along a diagonal in the plot. In that case, if a particle is at a_1= (x =0, y = 0), then its subsequent trajectory through spacetime is constrained to lie within the cone colored in green in figure G4 because nothing can go faster than the speed of light. A particular trajectory, or worldline, is shown in which the particle moves at constant velocity one half the speed of light. Similarly, if we fix a_2 at the origin, then a possible trajectory leading to a_2 is shown in figure G5 to lie within the green triangle. In G6, we show all points in the past or future of a particle at (0,0) that are separated such that they might lie on the particle's worldline in green; these points are said to be timelike separated from (0,0). The points in red in figure G6 are spacelike separated from (0,0). In figure G7, the timelike and spacelike intervals for an arbitrary point in space time are plotted in green and red, respectively.

In figure G8, spacelike and timelike intervals are plotted for two different spacetime points a and b, where a and b are just abstract points in the space belonging to a set S, but not associated with particular events. The plot shows a given point as spacelike only if it is located at a spacelike interval from *both* A and B; in other words, if a given point is located at a timelike interval from *either* A *or* B, then it is shown in green, otherwise in red. In G9, the same plot is done for three points a, b and c. In G10, the same plot is done for many points; evidently as more points are added to S, the set of points spacelike separated from all points in S gets squeezed out to either side. In G12, we show S as a dense set of points. Now it is clear that all the points spacelike separated from S can be determined in effect by four points, as depicted in figure G12. The points that determine the spacelike intervals are the points that touch the boundary of a rectangle enclosing S whose two axes are oriented at ± 45 degrees from vertical. In figure G13, we now consider the set S', the set of points spacelike separate from S, and plot the points that are spacelike separated from *it*. The points in this new set S'' fill the above-mentioned rectangle bounding S, as shown in G14. In terms of the partition of spacetime described above, any point within the rectangular oriented as in G12 bounding a set S belongs to the same equivalence class as S. Accordingly, the points of the lattice to be defined are rectangular-shaped subsets of spacetime oriented as shown in the plots, and, as we discuss next, disjoint unions of such subsets.

It is necessary to define the binary lattice operations join ($\vee$) and meet ($\wedge$) on representatives of equivalence classes. The meet of representatives a and b is defined to be just ordinary intersection, as shown in figure G15; the intersection of two rectangles always results in a rectangle or 0, as it must for the meet to be well defined. The join, on the other hand, is not quite so simple. In figure G16, a series of sets are shown along with their joins in dotted lines; for the cases (c) and (e), the joins are ordinary set union. Figure G17 shows under what circumstances the join is ordinary set union. Figures G18 through G23 show graphically why the join is as shown in L6 for each of the scenarios (a) through (e).

In order to show that the above lattice structure is orthomodular, it is necessary to show that $a \leq b \rightarrow a \vee (a' \wedge b) = b$ for all points a and b of the lattice, where $a \leq b$ means that a is a subset of b. Alternatively, one must show that comparable elements a and b generate a boolean subalgebra, as they must according to the above theorem if the spacetime lattice structure is orthomodular. In figure G24, we show a set B that consists of two disjoint regions, one of which contains a set A. In figure G25, the complement of A is shown. In figure G26, the meet of A’ and B is shown and labeled C, and figure G27 B’ is shown. In figure G28, C’ is shown to be the join of B’ and A. The introduction of C and C’completes the generation of Γ(A,B); further joins, meets and complements yield no new sets. The following is a list of operations on elements of Γ(A,B).

$A \vee B = B$	$A \wedge B = A$	$A' \vee B = 1$	$A' \wedge B = C$
$A \vee B' = C'$	$A \wedge B' = 0$	$A' \vee B' = A'$	$A' \wedge B' = B'$
$A \vee C = B$	$A \wedge C = 0$	$A' \vee C = A'$	$A' \wedge C = C$
$A \vee C' = C'$	$A \wedge C' = A$	$A' \vee C' = 1$	$A' \wedge C' = B'$
$B \vee C = B$	$B' \vee C = A'$	$x \vee 1 = 1$	$x \vee x = x$
$B \vee C' = 1$	$B' \vee C' = C'$	$x \vee 0 = x$	$x \vee x' = 1$
$B \wedge C' = A$	$B' \wedge C' = B'$	$x \wedge 1 = x$	$x \wedge x = x$
$B \wedge C = C$	$B' \wedge C = 0$	$x \wedge 0 = 0$	$x \wedge x' = 0$

x represents any element of Γ(A,B) and the meet and join are understood to commute. Using the above equalities, one can verify that Γ(A,B) is distributive, as required of a boolean algebra. For example,

$A \vee (B \wedge C) = A \vee C = B$	$(A \vee B) \wedge (A \vee C) = B \wedge B = B$
$A \vee (B \wedge C') = A \vee A = A$	$(A \vee B) \wedge (A \vee C') = B \wedge C' = A$
$B \vee (A \wedge C') = B \vee A = B$	$(B \vee A) \wedge (B \vee C') = B \wedge 1 = B$
$C \vee (B \wedge C') = C \vee A = B$	$(C \vee B) \wedge (C \vee C) = B \wedge C = B$
$A' \vee (B' \wedge C) = A' \vee 0 = A'$	$(A' \vee B') \wedge (A' \vee C) = A' \wedge A' = A'$
$C' \vee (A \wedge B) = C' \vee A = C'$	$(C' \vee A) \wedge (C' \vee B) = C' \wedge 1 = C'$
$B' \vee (C \wedge A') = B' \vee C = C'$	$(B' \vee C) \wedge (B' \vee A') = A' \wedge 1 = C'$

One can also confirm that $(x \vee y)' = x' \wedge y'$ for all x and y in Γ(A,B), which yields the so-called dual forms of the above equalities, for instance

$(A \vee (B \wedge C))' = A' \wedge (B' \vee C') = B'$ and $((A \vee B) \wedge (A \vee C))' = (A' \wedge B') \vee (A' \wedge C') = B'$.

Figure G29 shows that the spacetime lattice itself is not distributive.

Appendix H Formal Languages and Automata

Formal languages and automata are mathematical structures. In order to define a formal language, the definition of a grammar is required.

Grammar

A grammar is a quadruple G(V, T, S, P), where

V is a set of variable symbols,
T is a set of terminal symbols,
S is a particular element of V called the start symbol,
P is a set of productions.

A production is a rule that transforms a non-empty string of symbols from $V \cup T$, into another possibly empty string of symbols from $V \cup T$. A series of productions starting from the start symbol and resulting in a string of terminal symbols only is called a *derivation*. The last string in a derivation is called a *word*. The set of all words that can be generated by a given grammar is called the *language* generated by that grammar.

Grammar Example 1

Let V = {S}, T = {0,1}, and S be the start symbol. Let the productions be given by P = $S \to 0S$, $S \to 1S$, $S \to \varepsilon$}, where ε is the empty string. P can also be written
$S \to 0S \mid 1S \mid \varepsilon$.

Three typical derivations for the grammar G(V, T, S, P) are:

0S, 00S, 000S, 0001S, 00010S, 000101S, 000101
1S, 1
1S, 10S, 101S, 101

In each derivation above the last production was $S \to \varepsilon$. In general, the words in the language corresponding to G, written L(G), are arbitrary strings of 0's and 1's.

Grammar Example 2

Let $V \equiv \{S, A, B, C\}$, $T \equiv \{a,b,c\} \cup \varepsilon$, where ε is the empty string, and S be the start symbol. Let the productions P be given by

$$S \to aBC \mid bAC \mid cAB \mid \varepsilon$$
$$A \to aS \mid bcAA$$
$$B \to bS \mid caBB$$
$$C \to cS \mid abCC$$

A typical derivation for this grammar is:

aBC, abSC, abSabCC, abbACabCC, abbaSCabCC, abbaCabCC, abbacSabCC, abbacabCC, abbacabcSC, abbacabcC, abbacabccS, abbacabcc

In this language each word contains the same number of a's, b's and c's.

The Chomsky Hierarchy

It is possible to classify languages according to the form of the productions in their corresponding grammars as shown in the table below. The convention is to arrange the languages in order from most general to most restricted. This classification is in fact a hierarchy, since each grammar type in the table is capable of generating all languages that can be generated by all grammars beneath it as well as languages that no grammar beneath it can generate. This hierarchy is all the more significant in view of the fact that for each type of language in the hierarchy there is a mathematical object called an automaton or machine which accepts that language, that is, which understands that language in some sense. The table lists the languages, their production forms, examples of productions, and machines that accept them. The production examples use the convention that capital letters are variables and small letters are terminal symbols, and ε is the empty string.

Type no.	Language	Production form	Production examples	Accepting machine
0	recursive enumerable	x → y, where x is in $(V \cup T)^+$ ($^+$ indicates that x is non empty), and y is in $(V \cup T)^*$ (* indicates that y may be empty); grammar is unrestricted	AB → A A → ε (plus all type 1-3 examples)	Turing machine
1	context-sensitive	xAy → xay, where A is in V, x and y are in $(V \cup T)^*$, and a is in $(V \cup T)^+$; also permitted is S → ε, provided S does not appear on the right-hand side of any production	aBc → abc aBc → aaDc aabA → aabAc CD→ CabcDb CD→ CabcEb (plus all type 2-3 examples)	linear bounded automaton
2	context-free	A → α, where A is in V, and α is in $(V \cup T)^*$	A → abc B → abDD X → aXb (plus all type 3 examples)	pushdown automaton

3	regular	$A \rightarrow yB$, $A \rightarrow x$, where y is in T, A and B are in V, and x is in $T\cup \varepsilon$	$C \rightarrow cC$ $A \rightarrow b$	finite automaton

It can be shown that for any context-free grammar G containing a rule of the form $A \rightarrow \varepsilon$, where ε is the empty string, there exists another context-free grammar G' such that the corresponding languages of G and G' are equal, i.e. L(G) = L(G'), and such that there is only one production yielding ε in G', namely $S' \rightarrow \varepsilon$, where S' is the start symbol S' and S' does not appear on the right hand side of any other production in G'.[176] An analogous empty-string lemma holds for regular grammars. In light of these results, it is clear from the above table that any language generated by a grammar of type n can also be generated by a grammar of type m, where $0 \leq m < n \leq 3$.

An equivalent way of expressing the allowable productions for a context-sensitive grammar is $x \rightarrow y$ such that $|x| \leq |y|$, where the notation $|w|$ is used for the number of symbols in the string w; in addition, it is necessary to allow $S \rightarrow \varepsilon$ if S is not in the right-hand side of any production. In other words, productions for a context-sensitive grammar always augment the number of symbols in the running string. The production form given above in the table has the advantage that it clarifies the usage of the word *context*: the context in a context-sensitive language is given by the x and y surrounding the variable A in the left-hand side of the context-sensitive production form; in the context-free language the variable stands alone on the left-hand side of the production, with no such context. The production form $x \rightarrow y$ such that $|x| \leq |y|$ for context-sensitive grammars has the advantage that it clearly distinguishes between recursive enumerable and context-sensitive grammars.

Combinatory Logic

Example 1 above is a regular grammar and example 2 is a context-free grammar. Another example of a context-free language is combinatory logic, which can be formally defined using the following grammar G(T, V, A, P), where

T = { **K**, **S**, (,) },
V = { A },
A is the start symbol,
P = A→**K** | A→**S** | A→(**K**A) | A→ (**S**A) | A→**K**A | A→**S**A | A→AA

The derivations for **B** ≡ **S**(**KS**)**K** and **C** ≡ **S(BS(BKS))(KK)** using this grammar are as follows:

SA, **S**AA, **S**(**K**A)A, **S**(**KS**)A, **S**(**KS**)**K**

SA, **S**AA, **S**(**S**A)A, **S**(**S**AA)A, **S**(**S**(**K**A)A)A, **S**(**S**(**KS**)A)A, **S**(**S**(**KS**)AA)A, **S**(**S**(**KS**)**K**A)A, **S**(**S**(**KS**)**K**AA)A, **S**(**S**(**KS**)**KS**A)A, **S**(**S**(**KS**)**KS**(**S**A))A,

S(**S**(**KS**)**KS**(**S**AA))A, **S**(**S**(**KS**)**KS**(**S**(**K**A)A))A, **S**(**S**(**KS**)**KS**(**S**(**KS**)A))A,
S(**S**(**KS**)**KS**(**S**(**KS**)AA))A, **S**(**S**(**KS**)**KS**(**S**(**KS**)**K**A))A, **S**(**S**(**KS**)**KS**(**S**(**KS**)**K**AA))A,
S(**S**(**KS**)**KS**(**S**(**KS**)**KK**A))A, **S**(**S**(**KS**)**KS**(**S**(**KS**)**KKS**))A,
S(**S**(**KS**)**KS**(**S**(**KS**)**KKS**))(**K**A), **S**(**S**(**KS**)**KS**(**S**(**KS**)**KKS**))(**KK**),

Parentheses prevent combinatory logic from being a regular language.

A context-sensitive grammar

The following is a context-sensitive grammar G(T, V, S, P):

T = { a, b, c },
V = { S, A, B, C },
S,
P = S→aSBC | S→aBC | CB→BC | aB→ ab | bB→bb | bC→bc | cC→cc

A typical derivation is:

aSBC, aaBCBC, aaBBCC, aabBCC, aabbCC, aabbcC, aabbcc.

All words in this language have the form $a^n b^n c^n$.

An unrestricted grammar

The following is a type 0 grammar G(T, V, S, P), whose corresponding language is recursive enumerable:

T = { a, b, c },
V = { S, A, B, C, P, Q, X, M, N },
S,

P =

1) S → AS
2) S → AB
3) B → BB
4) B → C
5) AB → PXNB
6) NB → BN
7) NC → MCc
8) BM → MB
9) AP → PA
10) AXMB → AB
11) PXMB → PQXNB
12) QXMB → QQXN

13) QXBNC → Qc
14) P → a
15) Q → b

A typical derivation using this grammar goes as follows. After the first few steps, each step is given as a single line with changing symbols in bold typeface. The number of the production is given to the right.

S, AS, AAS, AAB, AABB, AABBB, AABBBB, AABBBC			(P1-4)
A***AB***BBC	→	A***PXNB***BBC	(P5)
APX***NB***BBC	→	APX***BN***BBC	(P6)
APXB***NB***BC	→	APXB***BN***BC	(P6)
APXBB***NB***C	→	APXBB***BN***C	(P6)
APXBBB***NC***	→	APXBBB***MCc***	(P7)
APXBB***BM***Cc	→	APXBB***MB***Cc	(P8)
APXB***BM***BCc	→	APXB***MB***BCc	(P8)
APX***BM***BBCc	→	APX***MB***BBCc	(P8)
APXMBBBCc	→	***PA***XMBBBCc	(P9)
P***AXMB***BBCc	→	P***AB***BBCc	(P10)
P***AB***BBCc	→	P***PXNB***BBCc	(P5)
PPX***NB***BBCc	→	PPX***BN***BBCc	(P6)
PPXB***NB***BCc	→	PPXB***BN***BCc	(P6)
PPXBB***NB***Cc	→	PPXBB***BN***Cc	(P6)
PPXBBB***NC***c	→	PPXBBB***MCc***c	(P7)
PPXBB***BM***Ccc	→	PPXBB***MB***Ccc	(P8)
PPXB***BM***BCcc	→	PPXB***MB***BCcc	(P8)
PPX***BM***BBCcc	→	PPX***MB***BBCcc	(P8)
P***PXMB***BBCcc	→	P***PQXNB***BBCcc	(P11)
PPQX***NB***BBCcc	→	PPQX***BN***BBCcc	(P6)
PPQXB***NB***BCcc	→	PPQXB***BN***BCcc	(P6)
PPQXBB***NB***Ccc	→	PPQXBB***BN***Ccc	(P6)
PPQXBBB***NC***cc	→	PPQXBBB***MCc***cc	(P7)
PPQXBB***BM***Cccc	→	PPQXBB***MB***Cccc	(P8)
PPQXB***BM***BCccc	→	PPQXB***MB***BCccc	(P8)
PPQX***BM***BBCccc	→	PPQX***MB***BBCccc	(P8)
PP***QXMB***BBCccc	→	PP***QQXN***BBCccc	(P12)
PPQQX***NB***BCccc	→	PPQQX***BN***BCccc	(P6)
PPQQXB***NB***Cccc	→	PPQQXB***BN***Cccc	(P6)
PPQQXBB***NC***ccc	→	PPQQXBB***MCc***ccc	(P7)
PPQQXB***BM***Ccccc	→	PPQQXB***MB***Ccccc	(P8)
PPQQX***BM***BCcccc	→	PPQQX***MB***BCcccc	(P8)
PPQ***QXMB***BCcccc	→	PPQ***QQXN***BCcccc	(P12)
PPQQQX***NB***Ccccc	→	PPQQQX***BN***Ccccc	(P6)
PPQQ***QXBNC***cccc	→	PPQQ***Qc***cccc	(P13)
PPQQQccccc …	→	***aabbb***ccccc	(P14-15)

Words[10] in this language are of the form $a^i b^j c^{i+j}$, where i,j > 0; in other words, the number of a's and b's is always equal to the number of c's, and the a's always come before the b's, which always come before the c's. Productions 10 and 13 yield shorter strings on the right-hand than on the left-hand side. Therefore this grammar is not context-sensitive. However, we give another grammar below that generates words of the same form that *is* context-sensitive; it turns out that there is no context-free grammar that will generate words of this form, so the language is in fact context-sensitive (as well as recursive enumerable, since all context-sensitive languages are also recursive enumerable). An unrestricted grammar similar to the one given above generates words of the form $a^i b^j c^{i*j}$, where i,j > 0 and * is ordinary multiplication; this language can not be generated with a context-sensitive grammar.[177]

Machines

Finite automata

A finite automaton is defined as a quintuple, M(Q, Σ, δ, q_0, F), where

> Q is a set of integers called states;
> Σ is a set of symbols;
> δ is a set of transition rules Q x Σ → Q, that is, from the set of ordered pairs
> > {(q,a) such that q ∈ Q and a ∈ Σ} to Q;
>
> q_0 is an element of Q, called the start state;
> F is a subset of Q, called the set of final states.

A finite automaton defines a kind of machine or computer that starts in the state q_0 and can progress to further states as determined by the transition rules. In order for the automaton to do anything, it requires a string of input symbols. Starting from the first symbol in the input string and advancing one symbol at a time to the end of the input string, the transition rules determine one or more sequences of states. If the last state in any of these sequences of states is in F, the set of final states, and the input string is exhausted, the automaton is said to *accept* the string. If an automaton accepts all strings in a language, the automaton is said to accept or recognize the language.

It is possible to represent a finite automaton completely as a directed graph, in which the vertices represent states and the edges represent transitions. Vertices are labeled with the state number; edges have an arrow indicating the from- and to-state and are labeled with the input symbol corresponding to the transition.

[10] Note that not all valid productions in a grammar need lead to valid derivations, i.e. to a set of symbols consisting only of terminal symbols; for example, in this grammar not setting AP → PA will leave at least one A hanging in the general case. Note also that the order of the productions for a single valid derivation is not unique; for example, in this grammar one can use B → BB anywhere (but not B → C).

Example

Let $\Sigma = \{a, b\}$, $Q = \{ 0, 1, 2, 3 \}$, $q_0 = 0$, and $F = \{3\}$. Let the set of transitions δ be given by

q	a	b
0	1	0
1	2	1
2	3	2
3	3	3

Each row in the table gives two ordered pairs of δ: for example, the first row gives (0, a) = 1, (0, b) = 0, and so on.

The transition graph is shown below, with the start state marked with → and the final state in **bold** typeface. This automaton will accept any input string consisting of at least 3 a's, with b's located anywhere.

```
  0           a            1           a         2         a            3
→X ----------->------------X--------->----------X----------->-----------X
  / \                      / \                  / \                     / \
  \</                      \</                  \</                     \</
   b                        b                    b                       b
```

Any finite automaton has a corresponding regular language. To find the grammar that generates that language, there is a recipe. Let the set of variables V be Q, the set of states of the automaton, and let the set of terminal symbols T be S, the set of symbols of the automaton. For the productions, use all of the transition rules of δ in the form $q_i \rightarrow tq_j$, where $\delta(q_i , t) = q_j$. Finally, for every q_k in F, the set of final states, include a production $q_k \rightarrow \varepsilon$. The latter productions serve to rid the running strings of variables so that only terminal symbols remain.

For example, for the above automaton we construct a grammar G(V, T, S, P):

$V = Q = \{0, 1, 2, 3\}$
$T = \Sigma = \{a, b\}$
$S = q_0 = 0$

$P = \{ 0 \rightarrow a1 \mid 0 \rightarrow b0 \mid 1 \rightarrow a2 \mid 1 \rightarrow b1 \mid 2 \rightarrow a3 \mid 2 \rightarrow b2 \mid 3 \rightarrow a3 \mid 3 \rightarrow b3 \mid 3 \rightarrow \varepsilon \}$

A typical derivation is the following:

b0, ba1, bab1, babb1, babba2, babbab2, babbaba3, babbababb3, babbababbb3, babbababbb

There is also a straightforward procedure for generating an automaton from a given grammar.[178] Such procedures for generating a grammar from a machine and vice versa are the essence of constructive proofs that establish a correspondence between grammars and automata. In the case of finite automata and regular grammars this correspondence is simple enough that one might one wonder whether grammars and automata are really just different ways of naming the same process. In fact, automata can be used to generate language as well as accept language. However, as the grammars become more general the proofs of correspondence between automata and grammars become progressively less straightforward, enough so that one suspects that this correspondence --- in essence between language and machine --- is a subtle fact of nature.

Pushdown automata

A pushdown automaton differs from a finite automaton in that it has an additional storage area called a stack. A stack is a storage mechanism that works on a last-in first-out basis: the last item "pushed" onto the stack is always the first item "popped" off the stack, just as one would always retrieve the most recently loaded plate from a spring-loaded plate dispenser that had been loaded one plate at a time and that had plates accessible only from the top. To complete this analogy one might imagine the pushdown automaton stack to have plates with symbols painted on them, because symbols are what its stack stores.

A nondeterministic pushdown automaton is defined as a septuple M(Q, Σ, Γ, δ, q_0, Z_0, F), where

> Q is a set of integers called states;
> Σ is a set of input symbols;
> Γ is a set of stack symbols;
> δ is a set of transition rules from $Q \times (\Sigma \cup \varepsilon) \times \Gamma \rightarrow Q \times \Gamma^*$, that is, from the set
>> of ordered triplets $\{(q, a, z)$ such that $q \in Q$ and $a \in \Sigma \cup \varepsilon$ and $z \in \Gamma\}$ to the set of ordered pairs $\{(q, z)$ such that $q \in Q$ and $z \in \Gamma^*\}$;
>
> q_0 is an element of Q, called the start state;
> Z_0 is an element of Γ, called the initial stack symbol;
> F is a subset of Q, called the set of final states.

A nondeterministic pushdown automaton can make two kinds of transitions, one that does nothing with the input string, and another that reads one symbol from the input string. It is convenient to think of the input string as a long tape and of the pushdown automaton as having a pointer to the current position on the tape. In the first type of transition this pointer does not move, while in the second type of transition it moves one step, always in the same direction, toward the end of the input.

The other components of a transition are the initial and final integers referencing the states of the automaton, and changes to the stack. The rule for the stack is that first one stack symbol must be popped off the stack (and thereafter be unavailable), and then any number of stack symbols may be pushed onto the stack --- the pop and the push are part of a single transition.

What it means for a pushdown automaton to accept a language is basically the same as what it means for a finite automaton to accept a language. A pushdown automaton starts in the state q_0 with Z_0 on the stack, and progresses to further states as determined by the transition rules. In order for the pushdown automaton to do anything, it requires a string of input symbols. Starting from the first symbol in the input string and advancing either no symbols or one symbol at a time to the end of the input string, the transition rules determine one or more sequences of states. If the last state in any of these sequences of states is in F, the set of final states, and the input is exhausted, the automaton is said to *accept* the string. If an automaton accepts all strings in a language, the automaton is said to accept or recognize the language.

Example

Let $Q = \{ 0, 1, 2, 3 \} = \{ q_0, q_1, q_2, q_3 \}$, $\Sigma = \{a, b\}$, $\Gamma = \{\alpha, \beta\}$, $q_0 = 0$, $Z_0 = \alpha$, and $F = \{3\}$. Let the set of transition rules δ be given by

1. $\delta(q_0, a, \alpha) = (q_1, \beta\alpha)$
2. $\delta(q_1, a, \beta) = (q_1, \beta\beta)$
3. $\delta(q_1, b, \beta) = (q_2, \varepsilon)$
4. $\delta(q_2, b, \beta) = (q_2, \varepsilon)$
5. $\delta(q_2, \varepsilon, \alpha) = (q_3, \varepsilon)$

Each transition rule has an initial configuration consisting of the state, the input symbol, and the top symbol on the stack. Each rule says how to go from that initial configuration to a final configuration consisting of an output state and a string to push onto the stack. This particular automaton has no transition for which the input pointer does not move.

The following table of transitions shows that this automaton accepts the input string aaabbb. The location of the current input is represented by a symbol in **bold** typeface. Also, the stack is shown as a horizontal string of symbols whose leftmost symbol represents the top of the stack.

step no.	input	stack in	stack out	transition	read	pop[11]	push
0	**a**aabbbε	αε	βαε	1. $q_0 \rightarrow q_1$	read a	pop α	push βα
1	a**a**abbbε	βαε	ββαε	2. $q_1 \rightarrow q_1$	read a	pop β	push ββ
2	aa**a**bbbε	ββαε	βββαε	2. $q_1 \rightarrow q_1$	read a	pop β	push ββ

[11] The stack is popped at each step, removing the symbol at the top of the stack.

3	aaa**b**bbε	βββαε	ββαε	3. $q_1 \rightarrow q_2$	read b	pop β	no push
4	aaab**b**bε	ββαε	βαε	4. $q_2 \rightarrow q_2$	read b	pop β	no push
5	aaabb**b**ε	βαε	αε	4. $q_2 \rightarrow q_2$	read b	pop β	no push
6	aaabbbε	αε	ε	5. $q_2 \rightarrow q_3$	no read	pop α	no push

After step 6, the input is exhausted and the automaton is in q_3, a final state. Therefore, the input string is accepted.

This automaton goes to state 1 and keeps pushing β's on the stack as long as a's are read in. As soon as a b is read, it moves to state 2 and then pops a β off the stack each time it reads a b. If the end of the input is reached when only the stack start symbol α is on the stack, then the automaton accepts the input string. This happens when the input has a string of a's followed by the same number of b's, that is, when the input strings have the form $a^n b^n$.

The following context-free grammar G(V, T, S, P) generates the language accepted by the above pushdown automaton.

Let V = {S}, T = {a,b}, and S be the start symbol. Let the productions be given by

$$P = S \rightarrow ab \mid aSb$$

A typical derivation is: aSb, aaSbb, aaabbb.

In general, a nondeterministic pushdown automaton can be constructed from a context-free grammar by a procedure that imitates the action of productions using the stack. In this way every context-free language can be associated with a three-state pushdown automaton that accepts it. Expressed in terms of transition rules, this procedure goes as follows:

1. Create a transition from the initial state q_0 to q_1 by pushing the start symbol on the stack after the start stack symbol, without reading any input, that is, without moving the input pointer:
$$\delta(q_0, \varepsilon, z_0) = (q_1, Sz_0).$$

2. For each production, create a transition with initial and final state q_1 that does not advance the input pointer, but has the left-hand side of the production as the top symbol on the stack and pushes the right-hand side of the production onto the stack. For the grammar G just defined, for example, we would have:
$$\delta(q_1, \varepsilon, S) = (q_1, ab)$$
$$\delta(q_1, \varepsilon, S) = (q_1, aSb).$$

3. For each input symbol, create a transition again with initial and final state q_1 that reads that symbol from the input and starts with that same symbol on the top of the stack and then just pops it off the stack without pushing anything back onto the stack. For the grammar, G, that yields two transitions:
$$\delta(q_1, a, a) = (q_1, \varepsilon)$$

$\delta(q_1, b, b) = (q_1, \varepsilon).$

4. Finally, create a transition that goes from state q_1 to the final state q_2 by not advancing the input pointer and just popping the start symbol off the stack without pushing anything onto it. For G, that means:

   $\delta(q_1, \varepsilon, Z_0) = (q_2, \varepsilon).$

Putting the above construction together for the grammar G, the nondeterministic pushdown automaton $M(Q, \Sigma, \Gamma, \delta, q_0, Z_0, q_2)$ is given by

$Q = \{ 0, 1, 2 \}= \{ q_0, q_1, q_2 \}$,
$\Sigma = \{a, b\}$,
$\Gamma = \{a, b, S\}$,
$q_0 = 0 = q_0$,
$Z_0 = z_0$,
$F = 2 = q_2$.
δ :

1.	$\delta(q_0, \varepsilon, z_0)$	$= (q_1, Sz_0)$
2.	$\delta(q_1, \varepsilon, S)$	$= (q_1, ab)$
3.	$\delta(q_1, \varepsilon, S)$	$= (q_1, aSb)$
4.	$\delta(q_1, a, a)$	$= (q_1, \varepsilon)$
5.	$\delta(q_1, b, b)$	$= (q_1, \varepsilon)$
6.	$\delta(q_1, \varepsilon, Z_0)$	$= (q_2, \varepsilon)$

Since rules 2. and 3. have the same initial configuration, this automaton is nondeterministic, meaning that there is not always a unique transition available. The following table shows how the automaton just constructed handles the input aaabbb.

step no.	input	stack in	stack out	transition	read	pop	push
0	**a**aabbbε	**z_0**ε	Sz_0ε	1. $q_0 \to q_1$	no read	pop z_0	push Sz_0
1	**a**aabbbε	**S**z_0ε	$aSbz_0$ε	3. $q_1 \to q_1$	no read	pop S	push aSb
2	**a**aabbbε	**a**Sbz_0ε	Sbz_0ε	4. $q_1 \to q_1$	read a	pop a	no push
3	a**a**abbbε	**S**bz_0ε	$aSbbz_0$ε	3. $q_1 \to q_1$	no read	pop S	push aSb
4	a**a**abbbε	**a**$Sbbz_0$ε	$Sbbz_0$ε	4. $q_1 \to q_1$	read a	pop a	no push
5	aa**a**bbbε	**S**bbz_0ε	$abbbz_0$ε	2. $q_1 \to q_1$	no read	pop S	push ab
6	aa**a**bbbε	**a**$bbbz_0$ε	$bbbz_0$ε	4. $q_1 \to q_1$	read a	pop a	no push
7	aaa**b**bbε	**b**bbz_0ε	bbz_0ε	5. $q_1 \to q_1$	read b	pop b	no push
8	aaab**b**bε	**b**bz_0ε	bz_0ε	5. $q_1 \to q_1$	read b	pop b	no push
9	aaabb**b**ε	**b**z_0ε	z_0ε	5. $q_1 \to q_1$	read b	pop b	no push
10	aaabbbε	**z_0**ε	ε	6. $q_1 \to q_2$	no read	pop z_0	no push

After step 10, the input is exhausted and the automaton is in q_2, a final state. Therefore, the input is accepted. Note that in step 1, the automaton could also have undergone transition 2, but that transition would have lead to a halting state, that is, to a state from which no further transitions would have been possible, before the end of the input.

The procedure given above can be used to construct a non-deterministic pushdown automaton for any context-free grammar. For example, one can be constructed for the grammar given above that was used to generate the combinatory logic language.

In addition to nondeterministic pushdown automata, there are also deterministic pushdown automata, which are not able to recognize all languages recognized by nondeterministic automata. All pushdown automata are able to recognize any language that can be recognized by a finite automaton; a mapping can be established from an arbitrary finite automaton to a pushdown automaton that maps transitions to transitions such that the stack of the pushdown automaton pops and pushes an arbitrary symbol for all transitions except for those leading to a final state, in which case that symbol is only popped from the stack.

The next machine in the hierarchy, the linear bounded automaton, is a special kind of Turing machine, so we first discuss the Turing machine.

Turing Machines

The Turing machine differs from a pushdown automaton in the nature of its infinite storage area. Instead of a stack, it has it has extra room for storing symbols on the same long tape that contains the input string, only now that tape is infinite. Instead of being able to access stored symbols from a single point only, it is able to traverse the storage area by moving its current symbol pointer one step to the left or to the right at each transition between states. Instead of having to erase symbols in order to access symbols buried in the storage area, it is able to retain symbols in place as it traverses the storage in order to access symbols. Instead of the storage area being separate from the input symbols, its storage area contains the input symbols and is everywhere writeable, allowing it to overwrite the input symbols already read in order to keep a running account of how the input has been processed.

A deterministic Turing machine is defined as a septuple M(Q, Σ, Γ, δ, q_0, a, r), where

> Q is a set of integers called states;
> Σ is a set of input symbols;
> Γ is a set of tape symbols such that Σ is a subset of Γ
> δ is a transition function $Q \times \Gamma \rightarrow Q \times \Gamma \times$ {L(eft), R(ight)} that is, from the set of ordered pairs {(q, a) such that $q \in Q$ and $a \in \Gamma$} to the set of ordered triplets {(q, a, d) such that $q \in Q$, $a \in \Gamma$ and $d \in \{L,R\}$};
> q_0 is an element of Q, called the start state;
> a is an element of Q, called the accept state;
> r is an element of Q, called the reject state.

The Turing machine transition is simple: for example, $\delta(q_1, a) = (q_2, b, R)$ means before the transition the machine is in state q_1 and the current storage location contains a, and after the transition the machine is in state q_2, b has been written over a, and the pointer to current storage has moved one place to the right.

There are a number of versions of the Turing machine, all of them equivalent. This particular one is a standard one. The storage tape is actually semi-infinite, containing an initial symbol $\sqsubset$ indicating the left end of the tape. The input string follows immediately to the right, after which there is a never-ending series of blanks, written $\sqcup$. There is always a transition $\delta(q_0, \sqsubset) = (q_1, \sqsubset, R)$.

Example

The following Turing machine accepts inputs in the form $a^i b^j c^{i+j}$, where i,j > 0; all others are rejected. Such languages were shown above to be generated by a type 0 grammar. This machine operates by first checking that the input starts with one or more a's, followed by one or more b's, followed by one or more c's; if at any point the input violates this form, it is rejected. The machine then writes a series of 1's in a scratch area to the right of the input, one 1 for each a and one 1 for each b in the input string. These 1's are then converted to 0's one by one as each c is encountered in the input. If no overrun occurs and at the end of this process each 1 has been converted to 0, then the machine enters the accept state; otherwise, the machine enters the reject state.

Let $Q = \{ 0, 1, \dots 12 \} = \{ q_0, q_1, \dots q_{18} \}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b, c, x, y, z, 0, 1\}$, $q_0 = 0$, $a = 17$, and $r = 18$. Let the set of transition rules δ be given by[12]

trans. no.	transition	comment
0	$\delta(q_0, \sqsubset) = (q_1, \sqsubset, R)$	standart start; go to state q1
1	$\delta(q_1, a) = (q_2, a, R)$	q1 must have an a
2	$\delta(q_1, b) = (r, b, R)$	q1 rejects b and c and blank (r is the reject state)
3	$\delta(q_1, c) = (r, c, R)$	
	$\delta(q_1, \sqcup) = (r, \sqcup, R)$	
4	$\delta(q_2, a) = (q_2, a, R)$	q2 runs right, reading a's and replacing them
5	$\delta(q_2, b) = (q_3, b, R)$	if q2 hits a b, it goes to q3
6	$\delta(q_2, c) = (r, c, R)$	if q2 hits a c or blank, it rejects
	$\delta(q_2, \sqcup) = (r, \sqcup, R)$	

[12] Notice: the following table listing the transitions can be ignored altogether --- it is given for the sake of completeness. The subsequent table shows how this machine accepts the input aabbbcccc; its first column shows how the tape values change in accordance with the strategy outlined in the previous paragraph.

7	$\delta(q_3, a) = (r, a, R)$	q3 rejects a
8	$\delta(q_3, b) = (q_3, b, R)$	q3 runs right, reading b’s and replacing them
9	$\delta(q_3, c) = (q_4, c, R)$	if q3 hits a c, it goes to q4
	$\delta(q_3, \sqcup) = (r, \sqcup, R)$	q3 rejects a blank
10	$\delta(q_4, a) = (r, a, R)$	q4 rejects a’s
11	$\delta(q_4, b) = (r, b, R)$	q4 rejects b’s
12	$\delta(q_4, c) = (q_4, c, R)$	q4 runs right reading c’s
13	$\delta(q_4, \sqcup) = (q_5, \sqcup, L)$	if q4 hits a blank, it turns back to the left and goes to q5
14	$\delta(q_5, a) = (q_5, a, L)$	q5 runs left until it hits $\sqsubset$, skipping a’s, b’s, and c’s
15	$\delta(q_5, b) = (q_5, b, L)$	
16	$\delta(q_5, c) = (q_5, c, L)$	
17	$\delta(q_5, \sqsubset) = (q_6, \sqsubset, R)$	
18	$\delta(q_6, a) = (q_7, x, R)$	q6 knows the input is an a due to validation above; it marks that first a with an x and goes to q7
19	$\delta(q_7, a) = (q_7, a, R)$	q7 runs right until it hits a blank
20	$\delta(q_7, b) = (q_7, b, R)$	
21	$\delta(q_7, c) = (q_7, c, R)$	
22	$\delta(q_7, 1) = (q_7, 1, R)$	
23	$\delta(q_7, \sqcup) = (q_8, 1, L)$	q7 goes to q8 at the blank, overwrites the blank with a 1 and turns back to the left
24	$\delta(q_8, 1) = (q_8, 1, L)$	q8 runs left until it hits an x
25	$\delta(q_8, c) = (q_8, c, L)$	
26	$\delta(q_8, b) = (q_8, b, L)$	
27	$\delta(q_8, a) = (q_8, a, L)$	
28	$\delta(q_8, x) = (q_9, x, R)$	q8 moves to the right when it hits an x and goes to q9
29	$\delta(q_9, a) = (q_7, x, R)$	q9 writes over an a with an x and returns to q7; q9 therefore starts a loop and so needs an exit condition
30	$\delta(q_9, b) = (q_{10}, y, R)$	q9 exits its loop if it reads b; b means that all the a’s have been turned into x’s; it writes over the b with a y
31	$\delta(q_{10}, b) = (q_{10}, b, R)$	q10 runs right until it hits a blank
32	$\delta(q_{10}, c) = (q_{10}, c, R)$	
33	$\delta(q_{10}, 1) = (q_{10}, 1, R)$	
34	$\delta(q_{10}, \sqcup) = (q_{11}, 1, L)$	q10 goes to q11 at the blank, overwrites the blank with a 1 and turns back to the left
35	$\delta(q_{11}, 1) = (q_{11}, 1, L)$	q11 runs left until it hits a y
36	$\delta(q_{11}, c) = (q_{11}, c, L)$	
37	$\delta(q_{11}, b) = (q_{11}, b, L)$	
38	$\delta(q_{11}, y) = (q_{12}, y, R)$	q11 moves to the right when it hits a y and goes to q12
39	$\delta(q_{12}, b) = (q_{10}, y, R)$	q12 writes over a b with a y and returns to q10, starting another loop
40	$\delta(q_{12}, c) = (q_{13}, z, R)$	q12 exits its loop if it reads c; c means that all the b’s have been turned into y’s; it writes over the c with a z

41	$\delta(q_{13}, c) = (q_{13}, c, R)$	q13 runs right until it hits a 1, skipping c’s and 0’s
42	$\delta(q_{13}, 0) = (q_{13}, 0, R)$	
43	$\delta(q_{13}, \sqcup) = (r, 0, R)$	q13 should not hit a blank; reject because input had too many c’s
44	$\delta(q_{13}, 1) = (q_{14}, 0, L)$	q13 overwrites a 1 with 0 and moves left
45	$\delta(q_{14}, 0) = (q_{14}, 0, L)$	q14 runs left ignoring 0’s and c’s
46	$\delta(q_{14}, c) = (q_{14}, c, L)$	
47	$\delta(q_{14}, z) = (q_{15}, z, R)$	q14 moves to the right when it hits a z
48	$\delta(q_{15}, c) = (q_{13}, z, R)$	q15 starts another loop if it reads a c
49	$\delta(q_{15}, 0) = (q_{16}, 0, R)$	q15 exits its loop if it reads a 0; if the input is in the form $a^ib^jc^{i+j}$, then there should be a string of 0’s on the right
50	$\delta(q_{16}, 0) = (q_{16}, 0, R)$	q16 runs right over 0’s
51	$\delta(q_{16}, 1) = (r, 1, R)$	q16 rejects 1 because input did not have enough c’s
52	$\delta(q_{16}, \sqcup) = (a, 0, R)$	q16 accepts blank

The following table shows the behavior of this Turing machine when given the input aabbbccccc. The transitions are not shown that do the checking that a’s are followed by b’s and that b’s are followed by c’s. After the first few transitions, only the more interesting transitions are shown. The current storage location is shown in **bold** typeface.

tape	trans. no	transition
⊏aabbbccccc⊔⊔⊔⊔⊔⊔⊔	0	$\delta(q_0, \sqsubset) = (q_1, \sqsubset, R)$
		{. coarse data check .}
⊏aabbbccccc⊔⊔⊔⊔⊔⊔⊔	17	$\delta(q_5, \sqsubset) = (q_6, \sqsubset, R)$
⊏**a**abbbccccc⊔⊔⊔⊔⊔⊔⊔	18	$\delta(q_6, a) = (q_7, x, R)$
⊏x**a**bbbccccc⊔⊔⊔⊔⊔⊔⊔	19	$\delta(q_7, a) = (q_7, a, R)$
⊏xa**b**bbccccc⊔⊔⊔⊔⊔⊔⊔	20	$\delta(q_7, b) = (q_7, b, R)$
⊏xab**b**bccccc⊔⊔⊔⊔⊔⊔⊔	20	$\delta(q_7, b) = (q_7, b, R)$
⊏xabb**b**ccccc⊔⊔⊔⊔⊔⊔⊔	20	$\delta(q_7, b) = (q_7, b, R)$
⊏xabbb**c**cccc⊔⊔⊔⊔⊔⊔⊔	21	$\delta(q_7, c) = (q_7, c, R)$
⊏xabbbc**c**ccc⊔⊔⊔⊔⊔⊔⊔	21	$\delta(q_7, c) = (q_7, c, R)$
⊏xabbbcc**c**cc⊔⊔⊔⊔⊔⊔⊔	21	$\delta(q_7, c) = (q_7, c, R)$
⊏xabbbccc**c**c⊔⊔⊔⊔⊔⊔⊔	21	$\delta(q_7, c) = (q_7, c, R)$
⊏xabbbcccc**c**⊔⊔⊔⊔⊔⊔⊔	21	$\delta(q_7, c) = (q_7, c, R)$
⊏xabbbccccc**⊔**⊔⊔⊔⊔⊔⊔	23	$\delta(q_7, \sqcup) = (q_8, 1, L)$
⊏xabbbcccc**c**1⊔⊔⊔⊔⊔⊔	25	$\delta(q_8, c) = (q_8, c, L)$
⊏xabbbccc**c**c1⊔⊔⊔⊔⊔⊔	25	$\delta(q_8, c) = (q_8, c, L)$

		{ … run left … }
⊏**x**abbbccccc1⊔⊔⊔⊔⊔⊔	28	δ(q_8, x) = (q_9, x, R)
⊏x**a**bbbccccc1⊔⊔⊔⊔⊔⊔	29	δ(q_9, a) = (q_7, x, R)
⊏xx**b**bbccccc1⊔⊔⊔⊔⊔⊔	20	δ(q_7, b) = (q_7, b, R)
		{ … run right … }
⊏xxbbbccccc1⊔⊔⊔⊔⊔⊔	23	δ(q_7, ⊔) = (q_8, 1, L)
		{ … run left … }
⊏x**x**bbbccccc11⊔⊔⊔⊔⊔	28	δ(q_8, x) = (q_9, x, R)
⊏xx**b**bbccccc11⊔⊔⊔⊔⊔	30	δ(q_9, b) = (q_{10}, y, R)
⊏xxy**b**bccccc11⊔⊔⊔⊔⊔	31	δ(q_{10}, b) = (q_{10}, b, R)
		{… and so on … }
⊏xxyyyccccc11**1**1⊔	35	δ(q_{11}, 1) = (q_{11}, 1, L)
		{… run left … }
⊏xxyy**y**ccccc11111⊔	38	δ(q_{11}, y) = (q_{12}, y, R)
⊏xxyyy**c**cccc11111⊔	40	δ(q_{12}, c) = (q_{13}, z, R)
⊏xxyyyz**c**ccc11111⊔	41	δ(q_{13}, c) = (q_{13}, c, R)
		{ … run right …}
⊏xxyyyzcccc**1**1111⊔	44	δ(q_{13}, 1) = (q_{14}, 0, L)
⊏xxyyyzccc**c**01111⊔	46	δ(q_{14}, c) = (q_{14}, c, L)
		{… run left …}
⊏xxyyy**z**cccc01111⊔	47	δ(q_{14}, z) = (q_{15}, z, R)
⊏xxyyyz**c**ccc01111⊔	48	δ(q_{15}, c) = (q_{13}, z, R)
⊏xxyyyzz**c**cc01111⊔	41	δ(q_{13}, c) = (q_{13}, c, R)
		{… run right…}
⊏xxyyyzzccc0**1**111⊔	44	δ(q_{13}, 1) = (q_{14}, 0, L)
⊏xxyyyzzccc**0**0111⊔	45	δ(q_{14}, 0) = (q_{14}, 0, L)
		{ … and so on… }
⊏xxyyyzzzzc000**1**1⊔	44	δ(q_{13}, 1) = (q_{14}, 0, L)
⊏xxyyyzzzzc00**0**01⊔	44	δ(q_{14}, 0) = (q_{14}, 0, L)
		{ … run left …}
⊏xxyyyzzz**z**c00001⊔	47	δ(q_{14}, z) = (q_{15}, z, R)
⊏xxyyyzzzz**c**00001⊔	48	δ(q_{15}, c) = (q_{13}, z, R)
⊏xxyyyzzzzz**0**0001⊔	48	δ(q_{15}, c) = (q_{13}, z, R)
		{… run right…}

⊏xxyyyzzzzz0000**1**⊔	44	$\delta(q_{13}, 1) = (q_{14}, 0, L)$
⊏xxyyyzzzzz000**0**0⊔	44	$\delta(q_{14}, 0) = (q_{14}, 0, L)$
		{ … run left … }
⊏xxyyyzzzz**z**00000⊔	47	$\delta(q_{14}, z) = (q_{15}, z, R)$
⊏xxyyyzzzzz**0**0000⊔	49	$\delta(q_{15}, 0) = (q_{16}, 0, R)$
⊏xxyyyzzzzz0**0**000⊔	50	$\delta(q_{16}, 0) = (q_{16}, 0, R)$
		{… run right…}
⊏xxyyyzzzzz00000⊔	52	$\delta(q_{16}, ⊔) = (a, 0, R)$

Most transitions of this Turing machine involve running back and forth between the input and the scratch area even on a small input such as this. This inefficiency is inherent to the Turing machine, and yet, as simple as it is, the Turing machine is as capable as any other known calculation scheme or computer. A number of computational models have been shown to be equivalent to the Turing machine in the sense that they can simulate one another, including so-called random access machines, Post systems, and flow charts, as well as a variety of variations on the Turing machine such as allowing multiple tapes or non-deterministic sets of transition rules.[179,180] In fact, the Turing machine provides one entry point into what turns out to be a large, intricate web of equivalent definitions of what constitutes a recursive function, as discussed in Appendix I.

Linear Bounded Automata

The linear bounded automaton is a Turing machine with a tape size limited to be the length of its input string times a fixed constant; hence the name *linear bounded*. It is a known fact that this fixed constant can be taken to be 1 without losing generality; in other words, a machine defined with fixed constant $k > 1$ can be simulated by a machine defined with $k = 1$.[181] Using the $k = 1$ definition, if the input string to a linear bounded automaton is w, then the tape will initially contain $\lambda w \rho$, where λ is the so-called left marker and ρ the right marker. The markers can be read but not overwritten, and a transition for which a marker is scanned must move in the direction of the input, that is, to the right if the marker is λ and to the left if the marker is ρ. The convention adopted here is for the start state to scan λ.

Example

The Turing machine constructed above to recognize the language with words of the form $a^i b^j c^{i+j}$, where $i,j > 0$ actually does not need to use any more of the tape than that part occupied by the input string itself. Instead of writing a 1 for each a and each b and then converting those 1's to 0's as each c is encountered, it is possible to just convert a c to a z every time an a or b is encountered. The a's and b's are still converted to x's and y's in order to keep track of where we are in the computation.

Let Q = { 0, 1, … 10}= { q_0, q_1, … q_{10} }, Σ = {a, b, c}, Γ = {a, b, c, x, y, z}, q_0 = 0, a = 9, and r = 10. Let the set of transition rules δ be given by

trans. no.	transition	comment
0	$\delta(q_0, \lambda) = (q_1, \lambda, R)$	standard start; go to state q1 and move right to input start
1	$\delta(q_1, a) = (q_2, x, R)$	q1 assumes the input is an a; it overwrites it with an x and goes to q2
2	$\delta(q_2, a) = (q_2, a, R)$	q2 looks right for a c and overwrites it with a z and goes left and into q3; if q2 hits the right marker ρ, it rejects
3	$\delta(q_2, b) = (q_2, b, R)$	
4	$\delta(q_2, z) = (q_2, z, R)$	
5	$\delta(q_2, \rho) = (r, \rho, L)$	
6	$\delta(q_2, c) = (q_3, z, L)$	
7	$\delta(q_3, z) = (q_3, z, L)$	q3 looks left for x, then goes right and into q4
8	$\delta(q_3, b) = (q_3, b, L)$	
9	$\delta(q_3, a) = (q_3, a, L)$	
10	$\delta(q_3, x) = (q_4, x, R)$	
11	$\delta(q_4, a) = (q_2, x, R)$	q4: if a is scanned, overwrite it with an x and go to q2; if b is scanned, overwrite it with a y and go to q5
12	$\delta(q_4, b) = (q_5, y, R)$	
13	$\delta(q_5, b) = (q_5, b, R)$	q5 looks right for a c and overwrites it with a z and goes left and into q6; if q5 hits the right marker ρ, it rejects
14	$\delta(q_5, z) = (q_5, z, R)$	
15	$\delta(q_5, \rho) = (r, \rho, L)$	
16	$\delta(q_5, c) = (q_6, z, L)$	
17	$\delta(q_6, z) = (q_6, z, L)$	q6 looks left for y, then goes right, into q7
18	$\delta(q_6, b) = (q_6, b, R)$	
19	$\delta(q_6, y) = (q_7, y, R)$	
20	$\delta(q_7, b) = (q_5, b, R)$	q7: if b is scanned, overwrite it with a y and go to q5; if z is scanned, go to q8
21	$\delta(q_7, z) = (q_8, z, R)$	
22	$\delta(q_8, z) = (q_8, z, R)$	q8 looks right for ρ and accepts the input if it finds it; if it finds a c, it rejects
23	$\delta(q_8, c) = (r, c, R)$	
24	$\delta(q_8, \rho) = (a, \rho, L)$	

The following table shows that this automaton accepts the input abcc. The current storage location is shown in **bold** typeface.

tape	trans. no	transition
λabccρ	0	$\delta(q_0, \lambda) = (q_1, \lambda, R)$
λ**a**bccρ	1	$\delta(q_1, a) = (q_2, x, R)$
λx**b**ccρ	3	$\delta(q_2, b) = (q_2, b, R)$
λxb**c**cρ	6	$\delta(q_2, c) = (q_3, z, L)$
λx**b**zcρ	8	$\delta(q_3, b) = (q_3, b, L)$
λ**x**bzcρ	10	$\delta(q_3, x) = (q_4, x, R)$
λx**b**zcρ	12	$\delta(q_4, b) = (q_5, y, R)$
λxy**z**cρ	14	$\delta(q_5, z) = (q_5, z, R)$
λxyz**c**ρ	16	$\delta(q_5, c) = (q_6, z, L)$
λxy**z**zρ	17	$\delta(q_6, z) = (q_6, z, L)$
λx**y**zzρ	19	$\delta(q_6, y) = (q_7, y, R)$
λxy**z**zρ	21	$\delta(q_7, z) = (q_8, z, R)$
λxyz**z**ρ	22	$\delta(q_8, z) = (q_8, z, R)$
λxyzz**ρ**	24	$\delta(q_8, \rho) = (a, \rho, L)$

It can be proved that the linear bounded automaton accepts languages that correspond to context-sensitive grammars.[182] An informal recipe for building a context-sensitive grammar from a linear bounded automaton transitions based on the above example goes as follows. First, one finds a set of productions which can generate a string of nonterminal symbols in a form the automaton would accept if they were terminal symbols; typically this set of productions will generate many other strings in a form not accepted by the automaton --- otherwise the grammar would be already completely determined. In the above example, such a string would be AABBCCCC; it does not matter if these productions also generate unacceptable words for the automaton, such as AAABBBC, since these productions will never result in valid derivations (those having only terminal symbols). Next, one builds a series of productions of the form PQ → RS, one for each transition, that serve to mimic the automaton transitions; these are legal context-sensitive productions since they cannot cause the size of the running input to decrease. These productions result in a running string of new nonterminals that are also mappable one-to-one to the terminal symbols, e.g. XXYYZZZZ. Finally, one builds a set of productions that yield the terminal symbols.

The grammar corresponding to the automaton above is given below. It uses nonterminal symbols with indices that match the states in the automaton whose behavior they mimic.

$T = \{ a, b, c \}$,
$V = \{ S, S_1, A, B, C, X, Y, Z, A_1, A_2, A_3, A_4, B_2, B_3, B_4, B_5, B_6, B_7, Z_2, Z_3, Z_5, Z_6, Z_8\}$,
S,
P =

1) $S \rightarrow A_1S_1$
2) $S_1 \rightarrow AS_1$

3)	$S_1 \rightarrow BS_1$
4)	$S_1 \rightarrow CS_1$
5)	$CS_1 \rightarrow CC$
6)	$A_1A \rightarrow XA_2$
7)	$A_2A \rightarrow AA_2$
8)	$A_2B \rightarrow AB_2$
9)	$B_2B \rightarrow BB_2$
10)	$B_2C \rightarrow B_3Z$
11)	$B_2Z \rightarrow BZ_2$
12)	$Z_2Z \rightarrow ZZ_2$
13)	$Z_2C \rightarrow Z_3Z$
14)	$ZZ_3 \rightarrow Z_3Z$
15)	$BZ_3 \rightarrow B_3Z$
16)	$BB_3 \rightarrow B_3B$
17)	$XB_3 \rightarrow XB_4$
18)	$AB_3 \rightarrow A_3B$
19)	$AA_3 \rightarrow A_3A$
20)	$XA_3 \rightarrow XA_4$
21)	$B_4B \rightarrow YB_5$
22)	$A_4A \rightarrow XA_2$
23)	$A_4B \rightarrow XB_2$
24)	$B_5B \rightarrow BB_5$
25)	$B_5Z \rightarrow BZ_5$
26)	$Z_5Z \rightarrow ZZ_5$
27)	$Z_5C \rightarrow Z_6Z$
28)	$ZZ_6 \rightarrow Z_6Z$
29)	$BZ_6 \rightarrow B_6Z$
30)	$YZ_6 \rightarrow YZ_8$
31)	$BB_6 \rightarrow B_6B$
32)	$YB_6 \rightarrow YB_7$
33)	$B_7B \rightarrow YB_5$
34)	$B_7Z \rightarrow YZ_5$
35)	$Z_8Z \rightarrow cc$
36)	$cZ \rightarrow cc$
37)	$Yc \rightarrow bc$
38)	$Yb \rightarrow bb$
39)	$Xb \rightarrow ab$
40)	$Xa \rightarrow aa$

A valid derivation is:
S, A_1S_1, A_1AS_1, A_1ABS_1, A_1ABS_1, A_1ABBS_1, A_1ABBCS_1,
$A_1ABBCCS_1$, $A_1ABBCCCS_1$, $A_1ABBCCCC$ (P1-5)

A_1ABBCCCC	→	**XA_2**BBCCCC	(P6)
X**A_2B**BCCCC	→	X**AB_2**BCCCC	(P8)
XA**B_2B**CCCC	→	XA**BB_2**CCCC	(P9)

XAB$\mathbf{B_2C}$CCC	→	XAB$\mathbf{B_3Z}$CCC	(P10)
XA$\mathbf{BB_3}$ZCCC	→	XA$\mathbf{B_3B}$ZCCC	(P16)
X$\mathbf{AB_3}$BZCCC	→	X$\mathbf{A_3B}$BZCCC	(P18)
$\mathbf{XA_3}$BBZCCC	→	$\mathbf{XA_4}$BBZCCC	(P20)
X$\mathbf{A_4B}$BZCCC	→	X$\mathbf{XB_2}$BZCCC	(P23)
XX$\mathbf{B_2B}$ZCCC	→	XX$\mathbf{BB_2}$ZCCC	(P9)
XXB$\mathbf{B_2Z}$CCC	→	XXB$\mathbf{BZ_2}$CCC	(P11)
XXBB$\mathbf{Z_2C}$CC	→	XXBB$\mathbf{Z_3Z}$CC	(P13)
XXB$\mathbf{BZ_3}$ZCC	→	XXB$\mathbf{B_3Z}$ZCC	(P15)
XX$\mathbf{BB_3}$ZZCC	→	XX$\mathbf{B_3B}$ZZCC	(P16)
X$\mathbf{XB_3}$BZZCC	→	X$\mathbf{XB_4}$BZZCC	(P17)
XX$\mathbf{B_4B}$ZZCC	→	XX$\mathbf{YB_5}$ZZCC	(P21)
XXY$\mathbf{B_5Z}$ZCC	→	XXY$\mathbf{BZ_5}$ZCC	(P25)
XXYB$\mathbf{Z_5Z}$CC	→	XXYB$\mathbf{ZZ_5}$CC	(P26)
XXYBZ$\mathbf{Z_5C}$C	→	XXYBZ$\mathbf{Z_6Z}$C	(P27)
XXYB$\mathbf{ZZ_6}$ZC	→	XXYB$\mathbf{Z_6Z}$ZC	(P28)
XXY$\mathbf{BZ_6}$ZZC	→	XXY$\mathbf{B_6Z}$ZZC	(P29)
XX$\mathbf{YB_6}$ZZZC	→	XX$\mathbf{YB_7}$ZZZC	(P32)
XXY$\mathbf{B_7Z}$ZZC	→	XXY$\mathbf{YZ_5}$ZZC	(P34)
XXYY$\mathbf{Z_5Z}$ZC	→	XXYY$\mathbf{ZZ_5}$ZC	(P26)
XXYYZ$\mathbf{Z_5Z}$C	→	XXYYZ$\mathbf{ZZ_5}$C	(P26)
XXYYZZ$\mathbf{Z_5C}$	→	XXYYZZ$\mathbf{Z_6Z}$	(P27)
XXYYZ$\mathbf{ZZ_6}$Z	→	XXYYZ$\mathbf{Z_6Z}$Z	(P28)
XXYY$\mathbf{ZZ_6}$ZZ	→	XXYY$\mathbf{Z_6Z}$ZZ	(P28)
XXY$\mathbf{YZ_6}$ZZZ	→	XXY$\mathbf{YZ_8}$ZZZ	(P30)
XXYY$\mathbf{Z_8Z}$ZZ	→	XXYY**cc**ZZ	(P35)
XXYYc**cZ**Z	→	XXYYc**cc**Z	(P36)
XXYYcc**cZ**	→	XXYYcc**cc**	(P36)
XXY**Yc**ccc	→	XXY**bc**ccc	(P37)
XX**Yb**cccc	→	XX**bb**cccc	(P38)
X**Xb**bcccc	→	X**ab**bcccc	(P39)
Xabbcccc	→	**aa**bbcccc	(P40)

Since a linear bounded automaton always does its work in place, that is, without extra storage, it is always possible to mimic its state transitions as in the above example to yield a corresponding context-sensitive grammar. Conversely, one can take any context-sensitive grammar and break down its productions into a larger set of productions with at most two symbols on each side and then use those productions to generate the transitions for a linear bounded automaton.

Recursively Enumerable Sets
The Turing machine as defined above operates on symbols in an arbitrary language. These symbols can be used to represent numbers as well, in which case the computations that a Turing machine does are calculations on numbers. For example, one could allow just two symbols on the tape in addition to the blanks and the left marker, 0 and 1, and these could be used to represent numbers in unary notation, where a number is given by a

string of 1's whose length is equal to the number represented and a 0 is used to separate numbers. In fact, the words of the form $a^ib^jc^{i+j}$, where i,j > 0, used in the above examples above were essentially numbers in unary notation, and the calculations that the automata executed were predicates of three variables --- a predicate is a function that has one of two values: 1 for true and 0 for false. Likewise, the pushdown automaton given above that accepts words of the form a^nb^n, where n > 0, executes a predicate of two variables, namely the function that returns 1 if the two input numbers are equal.

To be precise, the last-mentioned automaton does not actually calculate a predicate because it can only give a result of 1 if the two inputs are the same, but if they are not, the automaton does not tell us anything, as it then never goes into a final state. Of course, the predicate that takes two numbers as input and has the value 1 if they are equal and 0 if not *is* computable and therefore does exist --- indeed by a Turing machine similar to and slightly simpler than the one used to recognize $a^ib^jc^{i+j}$, where i,j > 0. However, there do exist functions whose implementation as a Turing machine is unable to conclude whether a given natural number is in the range of that function or not. Such functions lead to the notions of semi-recursive and recursive sets. A recursive set is a set for which there is an effective procedure for deciding whether or not a given natural number is in the set; for example, the set {m}, where m is a natural number, is a recursive set since the only natural number n which is in the set is given by m=n, and there is an effective procedure for determining when two integers are equal. On the other hand, a semi-recursive set, also known as a recursively enumerable set, is a set for which there is only an effective procedure that answers half the question: it can say that a given number is in the set, but if it is not in the set, it gives no answer. All recursive sets are recursively enumerable, but not all recursively enumerable sets are necessarily recursive; however, if a set and its complement are recursively enumerable, then that set is recursive. An example of a recursively enumerable set that is not recursive is given in Appendix I. [183]

The Turing machine, as indicated in the main text, is equivalent to a while-program in the sense that they both are capable of executing the same effective procedures. The proof of the latter statement is that one can take any Turing machine and turn it into a while-program and vice-versa (Appendix I gives another such equivalence proof, showing that any register machine program can be translated into a Turing machine specification). Since the exit conditions for some natural numbers may not be met for a while-program corresponding to a given Turing machine, that program will never stop looping when given those numbers as input. For that reason, the languages in the Chomsky hierarchy associated with the Turing machine are said to be recursively enumerable sets.

Appendix I Algorithms and Classical RecursionTheory

In this appendix we describe the relationship between algorithms and recursive functions, confining our remarks to this aspect of classical recursion theory, which is the study of functions over the natural numbers (generalized recursion theory encompasses functions over ordinals greater than countable infinity, among other things). We first show how natural the concept of recursive function is by virtue of its many disparate and yet equivalent definitions, among these a definition in terms of combinatory logic. Next, we show that the Turing machine is capable of computing recursive functions, after first introducing a variant of the register machine that is more easily programmed to compute numeric functions. We then discuss Church's thesis, which proposes that all effective computations are recursive functions. Finally, we illustrate in broad strokes how these ideas were used to determine if there is an effective procedure for deciding whether a given Turing machine halts on a given input.

The following theorem shows the many ways to define recursive functions. Items 5, 6, and 7 require more discussion than is possible here; for a presentation with proofs and references to the original literature, see Odifreddi.[184]

Theorem 1. Given a function f: $N^n \rightarrow N$ (that is, a function f that takes n natural numbers as input and returns a natural number), the following are equivalent:

1) f is recursive
2) f is definable in combinatory logic
3) f is λ-definable
4) f is Turing (and flow-chart, and while-program, and register-machine, and Post-system, and Markov-algorithm) computable
5) f is finitely definable
6) F is Herbrand-Goedel computable
7) f is representable in a consistent formal system extending $\mathcal{R}$, where $\mathcal{R}$ itself extends first-order logic with equality by defining numbers as constants $\underline{n}$, a predicate <, binary functions + and *, satisfying the following axiom schemata:
    - I) $\neg(\underline{x} = \underline{y})$, for $x \neq y$
    - II) $x < \underline{n} \lor x = \underline{n} \lor \underline{n} < x$
    - III) $\neg(x < \underline{0})$
    - IV) $x < \underline{n+1} \leftrightarrow x = 0 \lor \ldots \lor x = \underline{n}$
    - V) $\underline{x} + \underline{y} = \underline{x+y}$
    - VI) $\underline{x} * \underline{y} = \underline{x*y}$

Remarks:

1) A recursive function by definition is constructed using some combination of initial functions, composition of functions, primitive recursion and μ-recursion. All but the last of these were defined in the main text, but we repeat them here for convenience.
    a. The initial functions are the zero function, which has value 0 for any input, the successor function S(n) = n+1, and the projection or identity functions given by $I_i^n(x_1, \ldots, x_n) = x_i$, which picks out and returns its ith argument.
    b. Composition of the function g(x) with the function h(x) is defined by f(x) = g(h(x)). In the general case, $f(x_1, \ldots x_m) = g(h_1(x_1, \ldots x_n), \ldots, h_m(x_1, \ldots x_n))$.
    c. Primitive recursive functions of two variables are defined by f(m,0) = h(m) and f(m,n+1) = g(m, n, f(m,n)), where m and n are natural numbers. In the general case, $f(x_1, \ldots x_k, 0) = h(x_1, \ldots x_k)$ and $f(x_1, \ldots x_k, n+1) = g(x_1, \ldots x_k, n, f(x_1, \ldots x_k, n))$.

d. μ-recursion: given a predicate P(x,y), the function $\mu y P(x,y)$ is equal to the smallest y such that P(x,y)=1; if there is no such y, then it is undefined. A similar definition can be given if P is a function of more than two natural numbers.

Table x gives a number of common primitive recursive functions. Table x+1 shows how a number of predicates and other functions are defined in terms of other primitive recursive functions. A number of these definitions will be illuminated below with code that shows how these functions can be effectively calculated.

2) To show that combinatory logic can be used to construct recursive functions, it is necessary to define numbers in terms of combinators. We show below how that is done.
3) The λ-calculus can also be used to define recursive functions.[185]
4) All known paradigms for computation are equivalent in the sense that each one can simulate computations done by any other. We sketch below how a Turing machine can simulate a register machine.
5) A function can be defined as a solution to a system of equations and if that system consists of a finite number of equations, then the functioned so defined is recursive. For example, take the system of equations given by

$$f(0) = 0$$
$$f(x+1) = f(x) + 3$$
$$g(x) = f(g(x+1))$$

Here, f(1) = f(0) + 3 = 3, f(2) = f(1) + 3 = 6, f(3) = f(2) + 3 = 9, and so on, and in general, f(x) = 3x. f(x) is finitely generated by the above set of equations. On the other hand, g(x) is not. g(x) is solved for as follows. Since f(x) = 3x, f(g(x+1)) = 3g(x+1). By definition, we also have f(g(x+1)) = g(x). Therefore, 3(g(x+1)) = g(x). Starting from x=0, this leads to g(0) = 3g(1) = 3(3(g(2)) … = $3^n g(n) = 3^{n+1} g(n+1)$ … If we don't include an infinite number of such equations, that is, if we don't let n go to infinity, g(n) is just a sequence of numbers that decreases as n increases and the value of g(0) will be $3^n g(n)$. If we let n go to infinity, in which case there is an infinite number of equations, then g(x) must be 0 for all x, and g(x) is well-defined.

It turns out that every finitely definable function is recursive.[186] Note that the above example does not imply that k(x) = 0 is not finitely definable and therefore not recursive --- on the contrary, k(x) = 0 alone is a finite system of equations that defines k(x) (it is also the zero function, which, as one of the initial functions, is recursive). Rather, that example shows that g(x) is not finitely defined, and that f(x) = 3x *is* finitely definable and therefore recursive.

6) A function f is Herbrand-Goedel computable if there exists a finite system of equations from which f can be derived in a well-defined sense (which we don't give here).

7) A function f is representable in a formal system $\mathcal{F}$ if, for some formula ϕ, $f(x,y) = y$ implies that $\phi(\underline{x},\underline{y})$ is a theorem of $\mathcal{F}$ and $f(x,y) \neq y$ implies that $\neg\phi(\underline{x},\underline{y})$ is a theorem of $\mathcal{F}$.

Using Combinatory Logic and λ-calculus for Computation

There are several ways to define numbers in combinatory logic; the so-called standard numbers are defined as follows. The zero function is represented by **I**. The successor S operating on a number n is given by **P(KI)(n) = PF(n)**, where **P** is the combinator defined in Appendix D: **P**xyz = zyx. So the number 1 is given by **PFI**, 2 is given by **PF(PFI).** 3 is given by **PF(PF(PFI))**, and so on.

Next, we define a predicate function that returns **T (=K)** if a number is zero and **F** otherwise: zero(n) = n**T**. Then zero(3) = **PF(PF(PF(I)))T = TF(PF(PF(I))) = F,** and zero(1) = **PFIT = TFI = F**, but zero(0) = **IT = T**.

Similarly, a predecessor combinator operatoring on a number n is n**F**, since **PF**(n)**F = FF(n) = KIF(n) = I(n) = n.**

See Appendix C for definitions of numeric functions using λ-calculus.

The Turing Machine and Recursive Functions

To see the connection between Turing machines and the recursive functions as defined in point 1) of **Theorem 1**, it is helpful to introduce a variant of the register machine that we call a DSW machine since it appears prominently in a textbook by Davis, Sigal and Weyuker.[187] Summarizing material in the DSW textbook, we will sketch how one programs a DSW machine to compute the recursive functions, and we will sketch how one proves that a Turing machine can simulate any computation on a DSW machine.

A DSW machine is a well-defined mathematical structure, but we leave its formal definition aside and discuss it informally in terms of a programming language $\mathcal{S}$. In these terms, a DSW machine consists of variables, labels and instructions. The variables have values in the natural numbers including zero, and are of three types: input, internal, and output. There can be an arbitrary number of input and internal variables, but only one output variable. Labels have values in the natural numbers and can be thought of as line numbers of instructions. Any instruction has one of three forms:

1) $V \rightarrow V + 1$
2) $V \rightarrow V - 1$
3) if $V \neq 0$, go to label L

The first says to increase by one the value held by the variable V. The second says to decrease by one the value held by the variable V unless that value is already 0, in which case leave it unchanged. The third says that if the value held by V is not equal to 0, then the next instruction to execute is at the line number held by the label L.

A program in $\mathcal{S}$ consists of a sequence of such instructions together with the stored variables and labels and starts by executing the first instruction. In general, the current instruction is specified by a line number $\underline{n}$. If the instruction at $\underline{n}$ has a goto and the value of the variable referred to in the goto instruction is not zero, then the line number of the next instruction is given by the value of the corresponding label; otherwise, the next instruction is $\underline{n}+1$. By definition, if the last line of the program executes and it does not result in a goto, the program halts and its output is contained in the output variable.

The language $\mathcal{S}$ is a natural one for programming numeric functions since it has built-in successor and predecessor functions, which along with the ability to go to a label allow $\mathcal{S}$ to loop through a sequence of instructions repeatedly in a transparent fashion. Thus $\mathcal{S}$ is able to compute μ-recursive functions, just as the Y combinator of combinatory logic and λ -calculus allows these languages to compute μ-recursive functions. Similarly, the state of the Turing machine allows a Turing machine to loop through a sequence of states repeatedly, and a while-loop in a while-program provides for the repeated execution of a set of instructions.

The proof that $\mathcal{S}$ can compute a function f consists of a program in $\mathcal{S}$ that computes the value of f for any natural number in the domain of f. For example, the zero function is computed as follows, with Y the output variable:

```
[A]    $Y \rightarrow Y - 1$
       if $Y \neq 0$ goto A
```

This program consists of two instructions, the first labeled by A. The program loops back to A until Y is zero, at which point the program halts.

As a convention, we now assume the values of all internal variables are zero before a program starts, as well as the value of the output variable (if this were not the case, we could set them to zero before doing anything else in the program, using the technique of the zero program above). The input variables we take to be of the form X_1, X_2, X_3, etc. and the internal variables Z_1, Z_2, Z_3, etc. The output variable is Y.

The successor function for input stored in $X_1$is given by:

```
       $Y \rightarrow Y+ 1$
       $Z_1 \rightarrow Z_1 + 1$
       if $X_1 \neq 0$  goto A
       if $Z_1 \neq 0$  goto E          // input was 0; goto [E], leaving Y=1
[A]    $X_1 \rightarrow X_1 - 1$
       $Y \rightarrow Y+ 1$
       if $X_1 \neq 0$  goto A
[E]    $Z_1 \rightarrow Z_1 - 1$
```

Z_1 is only used to guard against the case that the input is zero, and the last instruction is arbitrary aside from the fact that it must not branch, i.e., it must not go to a label. The program works by decrementing the input variable X_1in a loop at the same time as the output variable Y is incremented. Y is also incremented once at the outset to ensure that Y gets a value one greater than the original input, even when the input is 0.

The program that calculates the identity function from $N\rightarrow N$ is the same as the above program except that it does not increment Y by one at the outset. The other identity functions from $N^N\rightarrow N$ return the kth argument using the same code as the $N\rightarrow N$ identity function except that X_k is everywhere substituted for X_1.

Before addressing composition, we present a program for computing the sum of two numbers initially stored in $X_1$and X_2:

```
        $Z_1 \rightarrow Z_1 + 1$
        if $X_1 \neq 0$  goto A
        if $Z_1 \neq 0$  goto B
[A]     $Y \rightarrow Y + 1$
        $X_1 \rightarrow X_1 - 1$
        if $X_1 \neq 0$  goto A
[B]     if $X_2 \neq 0$  goto C
        if $Z_1 \neq 0$  goto E
[C]     $Y \rightarrow Y + 1$
        $X_2 \rightarrow X_2 - 1$
        if $X_2 \neq 0$  goto C
[E]     $Z_1 \rightarrow Z_1 - 1$
```

The above program initializes Y to the value in X_1 first using the A loop. Then it increments Y by one as it decrements the value of X_2.

In order to make programs more readable, it is convenient to introduce abbreviations that stand for several instructions. For example, we let $V \leftarrow U$ stand for code that sets the value of V to be the same as the value of U. Likewise, we introduce $V \leftarrow U + W$ represent the code in the addition program above. These substitutions, also known as *macros*, can in fact always be done in such a way that other code in a program is isolated and not affected. Another macro that is sometimes useful for the sake of readability is HALT for an arbitrary non-branching instruction that does not change the output variable Y.

The program for multiplication of two numbers is similar to the addition program above. The difference is that there is now just a single loop that adds all of X_1 to Y in each iteration:

```
        $Z_1 \rightarrow Z_1 + 1$
        if $X_1 \neq 0$  goto A
        if $Z_1 \neq 0$  goto E
[A]     $Z_1 \leftarrow Y + X_1$            // use Z1 to isolate macro code
```

```
        Y ← Z_1
        X_2 → X_2 – 1
        if X_2 ≠ 0  goto A
[E]     HALT
```

Exponentiation, given by $f(x,y) = x^y = x*x*x*..x$ (with y factors of x) is the same as multiplication except that the loop uses a multiplication macro instead of addition, so that $Z_1 \leftarrow Y + X_1$ becomes $Z_1 \leftarrow Y * X_1$. Also, Y must be initialized to 1 before the loop starts. The nature of primitive recursion is such that all programs computing primitive recursive functions must first initialize the output Y, and then update Y in a loop; this is reflected in the definitions of all primitive functions given in Table x. The variable n is given in the table to a variable chosen to be the runner variable for each function; the table defines each function by fixing all input values except one at arbitrary values and then giving a recipe for arriving at a value of the function for all values of the runner variable one by one, starting from zero. In some cases the runner variable is fixed, that is, there is only one variable that can be the runner; such is the case for the exponentiation function, for example, and all functions of just one variable. For other functions, such as addition and multiplication, the runner variable can be chosen at random.

In general, we write $V \leftarrow f(Z_1, .. Zn)$ to indicate that the output from a function f is placed in V. This makes it possible to write a program defining the composition of two functions $f(x_1, \dots x_m) = g(h_1(x_1, \dots x_n), \dots, h_m(x_1, \dots x_n))$, where g and h_i are computable functions of m and n variables, respectively, and f is a function of m variables, as follows:

```
        Z_1 ← h_1(X_1, … X_n)
        :        :
        Z_m ← h_m(X_1, … X_n)
        Y   ← g(Z_1, … Z_m)
```

Before writing a program that computes primitive recursive functions we introduce the macro:

```
        if P(V_1,…V_n) goto L,
```

where V_i is an arbitrary variable. If P is a computable predicate, then the macro expands to

```
        Z_1  ← P(V_1,…V_n)
        if Z_1 ≠ 0  goto L
```

A useful macro that is then available is: if V = 0 goto L. The predicate V = 0 may be computed by the program

```
        if X_1 ≠ 0  goto E
        Y → Y+ 1
[E]     HALT
```

The above predicate function, called $\alpha(x)$, figures prominently in Table x+1.

For a function $N^2 \rightarrow N$ defined by primitive recursion, we have f(m,0) = h(m) and f(m,n+1) = g(m, n, f(m,n))

```
        Y ← h(X1)
[A]     if X2 = 0 goto E              // handles n=0 case
        Y ← g(X1, Z1, Y )
        Z1 ← Z1 + 1
        X2 ← X2 −1
        if X2 ≠ 0 goto A
[E]     HALT
```

A particular $N^2 \rightarrow N$ function that is primitive recursive is the addition program given above. If that program is abbreviated with macros, it has the above form with $h(X_1) = X_1$ and $g(X_1, Z_1, Y) = Y + 1$. For the sake of comparison, the definition of this function in terms of **Theorem 1,** point 1), is as follows: $f(x_1, 0) = u_1^1(x_1) = x_1$, where $u_1^1(x_1)$ is the projection function for a function of one variable, and $g(x_1, x_2, f(x_1, x_2)) = g(x_1, x_2, x_3)$ is given by $s(u_3^3(x_1, x_2, x_3)) = s(x_3) = x_3 + 1$, where s is the successor function and u_3^3 is the projection function of three arguments (top exponent) that returns the third argument (bottom exponent). Thus, the addition function is primitive recursive since it is formed from the initial functions, composition and primitive recursion.

Multiplication is also primitive recursive, having the above form with $h(X_1) = 0$ and $g(X_1, Z_1, Y) = Y + X_1$. In terms of the **Theorem 1**, point 1), $f(x_1, 0) = zero(x_1) = 0$, and $g(x_1, x_2, f(x_1, x_2)) = g(x_1, x_2, x_3) = f(u_3^3(x_1, x_2, x_3), u_1^3(x_1, x_2, x_3))$ and $f(x_1, x_2) = x_1 + x_2$.

In general, a function $N^n \rightarrow N$ defined by primitive recursion is defined as $f(x_1, \dots x_k, 0) = h(x_1, \dots x_k)$ and $f(x_1, \dots x_k, n+1) = g(x_1, \dots x_k, n, f(x_1, \dots x_k, n))$. The corresponding program is:

```
        Y ← h(X1, … Xk )
[A]     if Xk+1 = 0 goto E            // handles n=0 case
        Y ← g(X1, … Xk, Z1, Y )
        Z1 ← Z1 + 1
        Xk+1 ← Xk+1 −1
        if Xk+1 ≠ 0 goto A
[E]     HALT
```

In order to compute μ-recursive functions, a number of additional functions are needed. The following program computes $x \dot{-} y$, defined as x – y for x > y and 0 otherwise.

```
        Y ← X1
        if X2 = 0 goto E
[A]     if Y = 0 goto E
        X2 → X2 – 1
        Y → Y – 1
        if $X_2 \neq 0$ goto A
[E]     HALT
```

The function $x \leq n$, defined as $\alpha(x \dot{-} n)$, is computed by

```
$Z_1 \leftarrow X_1 \dot{-} X_2$
$Y \leftarrow \alpha(Z_1)$
```

Other functions computed similarly include the absolute value function, defined as $|x - n| = (x \dot{-} n)+(n \dot{-} x)$, and the equality predicate function $x = y$, defined as $\alpha(|x-y|)$.

If a function $f(x_1, \ldots x_k, n)$ is primitive recursive, then so is the function $u(x_1, \ldots x_k, n)$ given by

$$\begin{aligned} u(x_1, \ldots x_k, 0) &= f(x_1, \ldots x_k, 0). \\ u(x_1, \ldots x_k, n+1)) &= u(x_1, \ldots x_k, n) + f(x_1, \ldots x_k, n+1), \end{aligned}$$

since addition is primitive recursive. For fixed $x_1, \ldots x_k$, $u(x_1, \ldots x_k, n)$ is the sum of each value of $f(x_1, \ldots x_k, t)$ for all values of t from 0 to n. Such a sum is written as follows:

$$u(x_1, \ldots x_k, n) = \sum_{t=0}^{n} f(x_1, \ldots x_k, t).$$

Similarly, a primitive function $w(x_1, \ldots x_k, n)$ can be formed by the product of all values of $f(x_1, \ldots x_k, t)$ for all values of t from 0 to n:

$$\begin{aligned} w(x_1, \ldots x_k, 0) &= f(x_1, \ldots x_k, 0). \\ w(x_1, \ldots x_k, n+1)) &= w(x_1, \ldots x_k, n) * f(x_1, \ldots x_k, n+1). \end{aligned}$$

This product is written

$$w(x_1, \ldots x_k, n) = \Pi_{t=0}^{n} f(x_1, \ldots x_k, t).$$

If $f(x_1, \ldots x_k, n)$ is a predicate function, then we can define a primitive recursive function representing the existential quantifier $(\exists t)_{\leq n} P\, x_1, \ldots x_k, t)$, which means there exists a $t \leq n$ such that $P(x_1, \ldots x_k, t)$ (see Appendix A):

$$(\exists t)_{\leq n} P(x_1, \ldots x_k, t) = \alpha(\alpha(\sum_{t=0}^{n} P(x_1, \ldots x_k, t))),$$

which has value 1 if and only if $\sum_{t=0}^{n} P(x_1, \ldots x_k, t) \neq 0$.

Likewise, it is possible to define the the primitive recursive function representing the universal quantifier $(\forall t)_{\leq n} P(x_1, \ldots x_k, t)$, which means for every $t \leq n$, $P(x_1, \ldots x_k, t)$ holds:

$$(\forall t)_{\leq n}\, P(x_1, \ldots x_k, t) = \Pi_{t=0}{}^{n}\, P(x_1, \ldots x_k, t).$$

Since the above predicate functions using quantifiers are primitive recursive, they can be computed in $\mathcal{S}$. By the arguments just used, the following function can also be computed in $\mathcal{S}$:

$$f(x_1, \ldots x_k, z) = \textstyle\sum_{u=0}{}^{z}\Pi_{t=0}{}^{u}\, \alpha(P(x_1, \ldots x_k, t)).$$

The sum of products above has the following interpretation. As long as the predicate P is 0, α will convert that 0 to a 1 and for each value of the dummy variable u there will be a contribution of exactly 1 to the sum over u. As u gets bigger, the product ranges over more and more values of t because t goes from 0 to u for each value of u in the sum. However, as soon as t reaches a value, say t_1, such that P is 1, then α converts that to 0 and the product becomes zero for that value of u and all subsequent ones until u hits z. Therefore, the sum has contributions of 1 from every value of t such that $t < t_1$, yielding $t_1 * 1 = t_1$. The number t_1, the least number $t \leq z$ such that a computable predicate $P(x_1, \ldots x_k, t)$ holds, may or may not exist. Hence the definition for a function called the bounded minimalization of $P(x_1, \ldots x_k, t)$:

$$\begin{array}{lll} \min_{t\leq z} P(x_1, \ldots x_k, t) & = \sum_{u=0}{}^{y}\Pi_{t=0}{}^{u}\, \alpha(P(x_1, \ldots x_k, t)) & \text{if } (\exists t)_{\leq y}\, P(x_1, \ldots x_k, t) \\ & = 0 & \text{otherwise.} \end{array}$$

This following program computes this function:

```
        Z_1 ← X_{k+1} + 1              // store the bound
[A]     if P(X_1, … X_k, Y ) goto E
        Y ← Y + 1
        Z_1 ← Z_1 −1
        if Z_1 ≠ 0 goto A
        Y ← 0
[E]     HALT
```

Examples of functions defined by bounded minimalization include the remainder of a quotient x/y, the integer part of a quotient x/y, and the nth prime number. The first of these two functions are defined in Table x+1.

Another function can be defined along the same lines, the unbounded minimalization of $P(x_1, \ldots x_k, t)$, which may be undefined if there is no number t such that $\exists t\, P(x_1, \ldots x_k, t)$. A function of natural numbers that is defined on a subset of the natural numbers is said to be *partial*, as opposed to a *total* function of natural numbers, which is defined on all the natural numbers. In general, the unbounded minimalization of $P(x_1, \ldots x_k, t)$ is a partial function. This function is also partially computable, and is computed as follows:

```
[A]     if P(X_1, … X_k, Y ) goto E
        Y ← Y + 1
[E]     HALT
```

We have now shown that all recursive functions as defined by point 1) of **Theorem 1** can be calculated with a DSW machine.

$f(x_1,\dots x_k, n)$	$f(x_1, \dots x_k, 0) = h(x_1, \dots x_k)$	h(…) or k	$f(x_1, \dots x_k, n+1) = g(x_1, \dots x_k, n, f(x_1,\dots x_k,n))$	g(…)
$x+n$	$f(x,0) = h(x)$	x	$f(x,n+1) = g(x,n,f(x,y))$	$1+ f(x,n)$
$x*n$	$f(x,0) = h(x)$	0	$f(x,n+1) = g(x,n,f(x,y))$	$n + f(x,n)$
x^n	$f(x,0) = h(x)$	1	$f(x,n+1) = g(x,n,f(x,y))$	$n * f(x,n)$
0 (zero(n))	$f(0) = k$	0	$f(n+1) = g(n, f(n))$	0
n + 1 (succ(n))	$f(0) = k$	1	$f(n+1) = g(n, f(n))$	$1 + f(n)$
n – 1 (pred(n))	$f(0) = k$	0	$f(n+1) = g(n, f(n))$	n
$x \dot{-} n$	$f(x,0) = h(x)$	x	$f(x,n+1) = g(x,n,f(x,y))$	pred(f(x,n))
$\alpha(n)$	$f(0) = k$	1	$f(n+1) = g(n, f(n))$	0
$\sum_{n=0}^{n-1} n$	$f(0) = k$	0	$f(n+1) = g(n, f(n))$	$n + f(n)$
$!n = \Pi_{n=0}^{n-1} n$	$f(0) = k$	1	$f(n+1) = g(n, f(n))$	$n * f(n)$
$(\exists t)_{\leq n}$ $P(x_1,\dots x_k, t)$	$f(x_1,\dots x_k, 0) = h(x_1, \dots x_k)$	$P(x_1, \dots x_k, 0)$	$f(x_1, \dots x_k, n+1) = g(x_1, \dots x_k, n, f(x_1,\dots x_k,n))$	$\alpha(\alpha(P(x_1,\dots x_k, n+1) + f(x_1, \dots x_k, n)))$
$(\forall t)_{\leq n}$ $P(x_1,\dots x_k, t)$	$f(x_1,\dots x_k, 0) = h(x_1, \dots x_k)$	$P(x_1, \dots x_k, 0)$	$f(x_1, \dots x_k, n+1) = g(x_1, \dots x_k, n, f(x_1,\dots x_k,n))$	$P(x_1,\dots x_k, n+1) * f(x_1, \dots x_k, n)$

Function		Equivalent	Description
$P(x,n)$	$x \leq n$	$\alpha(x \dot{-} n)$	is less than
$P(x,n)$	$x < n$	$\alpha(n\leq x)$	is strictly less than
$P(x,n)$	$x = n$	$\alpha(\lvert x-n \rvert)$	is equal
$f(x,n)$	$\lvert x - n \rvert$	$(x\dot{-}n)+(n\dot{-}x)$	absolute value
$P(n)$	prime(n)	$n>1 \wedge (\forall n)_{\leq x} (n=1 \vee n=x \vee \neg(n \mid x))$	is prime[13]
$P(x,n)$	$n \mid x$	$(\exists t)_{t\leq x}(n*t = x)$	n divides x
$P(x_1,\dots x_k, n)$	$Q(x_1,\dots x_k, n) \wedge R(x_1,\dots x_k, n)$	$Q(x_1,\dots x_k, n)*R(x_1,\dots x_k, n)$	conjunction of Q and R
$P(x_1,\dots x_k, n)$	$\neg Q(x_1,\dots x_k, n)$	$\alpha(Q(x_1,\dots x_k, n))$	negation of Q
$f(x_1,\dots x_k, n)$	$\sum_{u=0}^{y} \Pi_{t=0}^{u} \alpha(P(t,$	$\min_{t\leq y}(P(t, x_1,\dots x_k)) =$	least value of t for

[13] A prime number p is a natural number such that the only numbers that divide it are 1 and p. For example, 1, 2, 3, 5, 7, and 11 are the first six prime numbers.

	$x_1, \ldots x_k$)	$f(x_1,..x_k, n)$ if $(\exists t)_{t\le x}(P(t, x_1,\ldots x_k))$ 0 otherwise	which $P(t, x_1,\ldots x_k)$ is true
f(x,y)	$\lfloor x/y \rfloor$	$\min_{t\le x}((t+1)*y>x)$_	integer part of quotient
f(x,y)	R(x,y)	$x \dot{-} (y*\lfloor x/y \rfloor)$	remainder of quotient

Translation of Programs from DSW to Turing Machine
In order to show that a Turing machine can perform any computation that a DSW machine can perform, we sketch how to translate an arbitrary DSW machine into a Turing machine. We discuss this translation process in terms of a translation program or translator that takes as input a DSW machine consisting of its list of instructions and variables, and outputs the specification of the corresponding Turing machine, which by definition is a quintuple of symbols, states, transitions between states, start symbol, and final states. In what follows we assume there are j instructions, m input variables, n internal variables and, as always, 1 output variable.

It is convenient to use a unary numbering system for the Turing machine. Still, we will use seven symbols on the Turing tape: ⊏, 1, 0, ⊓, ⊔, ∗ and B. A number n in this system consists of a string of n 1's followed by a 0. The marker ⊏ indicates the left boundary of the tape; the markers ⊓, ⊔, ∗ will be used during the operation of the machine to aid in doing the block moves associated with incrementing a value by one; B stands for *blank*. In its initial state the Turing machine will contain the input numbers only on the tape followed by an infinite string of B's. For example, if there are three input numbers with decimal values 0, 4 and 0, the initial tape configuration will look like this:

⊏011110BBBBBBBB…

The first task of the translator is to set up space for the internal variables and the one output variable. This requires a scan of the DSW machine to see that there are n internal variables and then it requires that the translator generate a series of states that write n+1 0's at the end of the input, leaving a single B between the input values and the internal values. The initial state, q_0, by convention scans the left-marker and moves one square to the right. q_1 runs to the right, reading 1's and 0's; when it hits a B, it writes that B back and moves one square to the right and goes to state q_2. q_2 reads a B, writes a 0, moves one square to the right and goes into q_3. For n > 0, states q_3 through q_{n+2} inclusive do the same thing, each one reading a B, writing a 0, moving one square to the right, and going to the state whose index is one greater than the current one. State q_{n+3} is a left runner, reading B's (of which there will be one, the first of the endless B's at the end of tape), 0's and 1's. When q_{n+3} hits the left-marker, it moves one to the right and goes to q_{n+4}.

The main task of the translator is next, namely to create a block of Turing states for each of the j instructions in the DSW program in a single pass. The translator must maintain a variable holding the value of the current state, which is n+4 before the first instruction is translated and whose value is incremented by some finite number during the course of translation of each instruction. At the start of each instruction translation, the current state q_i will be reading the square one to the right of the left-marker, and the translator must remember what this current state is for each labeled instruction in order to handle goto instructions.

The translation of the instructions breaks down into three cases, one for each instruction type of the DSW machine. The first instruction type is of the form $V_p \leftarrow V_p + 1$, where p ≥ 1 according to the DSW convention. These instructions can all be handled the same way, regardless whether V_p is an input variable, an internal variable, or the output variable, since the translator knows how many there are of each of these types of variables of $\mathcal{S}$ in the source program; in particular, X_k maps to V_k, Z_k maps to V_{m+k}, and Y maps to V_{m+n+1}. First, if p > 1, there will be a sequence of p –1 states that run to the right reading 1's until they hit a 0, at which point they move one further to the right and go to the state whose index is one greater than the current one. Next, the state q_{i+r}, (r = p if p > 1 and r = 0 otherwise) must initiate a process that inserts a 1 into the tape, which means block moving all symbols one square to the right starting from the insertion point. In order to avoid erasing symbols permanently, this process must start from the far right, but first, to keep track of where the insertion of a 1 is to take place, we place a marker in the square where V_p starts, which is the square read by q_{i+r}. q_{i+r} writes a $\sqcap$ if it reads a 1 and $\sqcup$ if it reads a 0, moves one square to the right and goes to state q_{i+r+1}. The reason for two different marks is that the marker will become a 1, and the value previously in the square occupied by the marker must be copied to the square on its right.

The process of moving a block one square to the right requires several states. q_{i+p+1} runs right until it hits a B, where it moves one to the left and goes to q_{i+r+2}. Now the going gets slow, with lots of back and forth. To reduce clutter, let $Q_0 = q_{i+r+2}$ temporarily. Then the process of block moving everything from $\sqcap$ or $\sqcup$ to the right by one square will require in general the additional states Q_0 through Q_5. This is best described with a table of transitions and then showing an example.

trans. no.	transition	comment
0	$\delta(Q_0, \sqcap) = (Q_3, 1, R)$	we've reached the start marker
1	$\delta(Q_0, \sqcup) = (Q_4, 1, R)$	we've reached the start marker
2	$\delta(Q_0, 1) = (Q_1, *, R)$	goto Q_1 since we read a 1; mark current square with $*$
3	$\delta(Q_0, 0) = (Q_2, *, R)$	goto Q_2 since we read a 0; mark current square with $*$
4	$\delta(Q_0, *) = (Q_0, *, L)$	slide past the $*$
6	$\delta(Q_1, *) = (Q_0, 1, L)$	Q_1 always writes a 1
7	$\delta(Q_2, B) = (Q_0, 0, L)$	we just started moving left, Q_2 always writes a 0
8	$\delta(Q_2, *) = (Q_0, 0, L)$	Q_2 always writes a 0
9	$\delta(Q_3, *) = (Q_5, 1, L)$	Q_3 is the last write; the original variable was non-zero
10	$\delta(Q_4, B) = (Q_5, 0, L)$	Y was just incremented from 0 to 1

11	$\delta(Q_4, *) = (Q_5, 0, L)$	Q_4 is the last write; the original variable was zero

tape	transition
□011110⊓10011**0**BBB…	$\delta(Q_0, 0) = (Q_2, *, R)$
□011110⊓10011∗**B**BB…	$\delta(Q_2, B) = (Q_0, 0, L)$
□011110⊓10011∗0BB…	$\delta(Q_0, *) = (Q_0, *, L)$
□011110⊓1001**1**∗0BB…	$\delta(Q_0, 1) = (Q_1, *, R)$
□011110⊓1001∗∗0BB…	$\delta(Q_1, *) = (Q_0, 1, L)$
□011110⊓1001∗10BB…	$\delta(Q_0, *) = (Q_0, *, L)$
□011110⊓100**1**∗10BB …	$\delta(Q_0, 1) = (Q_1, *, R)$
□011110⊓100∗∗10BB …	$\delta(Q_1, *) = (Q_0, 1, L)$
□011110⊓100∗110BB…	$\delta(Q_0, *) = (Q_0, *, L)$
□011110⊓10**0**∗110BB…	$\delta(Q_0, 0) = (Q_2, *, R)$
□011110⊓10∗∗110BB…	$\delta(Q_2, *) = (Q_0, 0, L)$
□011110⊓10∗0110BB…	$\delta(Q_0, *) = (Q_0, *, L)$
□011110⊓1**0**∗0110BB…	$\delta(Q_0, 0) = (Q_2, *, R)$
□011110⊓1∗∗0110BB…	$\delta(Q_2, *) = (Q_0, 0, L)$
□011110⊓1∗00110BB…	$\delta(Q_0, *) = (Q_0, *, L)$
□011110⊓**1**∗00110BB…	$\delta(Q_0, 1) = (Q_1, *, R)$
□011110⊓∗∗00110BB…	$\delta(Q_1, *) = (Q_0, 1, L)$
□011110⊓∗100110BB…	$\delta(Q_0, *) = (Q_0, *, L)$
□011110⊓∗100110BB…	$\delta(Q_0, \sqcap) = (Q_3, 1, R)$
□0111101∗100110BB…	$\delta(Q_3, *) = (Q_5, 1, L)$
□01111011100110BB…	done; ready to go to left-marker

After the insertion is complete, the state is q_{i+r+7}, which runs left reading 1's and 0's until it hits the left-marker, then moves one right and goes to q_{i+r+8}, the new value of the current state. That completes the translation for an instruction of type $V_p \leftarrow V_p + 1$. Before going on to the next instruction, the translator must save the state q_i that was current at the start of this instruction together with the line number of the instruction in the DSW program so that any goto's that go to this instruction can be translated into the correct state.

The translation for an instruction of type $V_p \leftarrow V_p - 1$ is similar to the foregoing, except that we must a remove a 1, if the value is non-zero, which means that the copying process can work from left to right and there is no need to run right to the end of the used portion of the tape before starting the copying process. Also no marker is needed, since the copying process ends after we reach a B. That means that only states analogous to Q_0 – Q_4 will be necessary, with the Q_0 analogue looking ahead to the right this time. If the current state is q_i at the start of the translation of an instruction of this type, the current state might be q_{i+r+6} at the conclusion of the translation, depending on the exact implementation. If the value of V_p was already 0, the transition goes from q_{i+r} to q_{i+r+5}, the state that runs left looking for the left-marker.

Finally, the translation for instructions of type *if* $V_p \neq 0$ *goto A* is comparatively easy. If the state q_{i+r} reads a 1, the transition is to the state corresponding to the DSW instruction line number associated with *A*. If that instruction is below the current instruction, the translator must store a temporary value for this state (for example, the negative value of the instruction line number) as a reminder that this state number must be updated with the value of the current state at the start of the translation of the instruction corresponding to *A*. If the state q_{i+r} reads a 0, then the transition is to state q_{i+r+1}, which runs left looking for the left-marker, and the current state will be q_{i+r+2} at the conclusion of the translation.

The translator knows which instruction is the last of the DSW. If the last instruction is of type $V_p \leftarrow V_p + 1$ or $V_p \leftarrow V_p - 1$, then the new current state at the conclusion of its translation will be almost in a final state, “almost” since there remains the task of moving the symbol or symbols corresponding to the value of the output to the left of the tape and blanking out all other squares, which is a convention for Turing machines computing numeric functions. If the last instruction is of type *if* $V_p \neq 0$ *goto A*, then, if $V_p = 0$, the new current state at the conclusion of its translation will likewise almost be in a final state. The states required to block move the output to the left side of tape and blank out the remainder of the tape are straightforward. All of the almost final states can in fact be the first in this sequence of clean-up states, leaving a single final state.

Since the DSW machine is able to compute any recursive function, the above translation shows that any recursive function is computable by a Turing machine, as stated in **Theorem 1**.

Church’s Thesis

The foregoing translation from the DSW machine can be also be done for the other calculation schemes of point 4 of **Theorem 1**. Furthermore, it is possible to show that the DSW machine can also do any computation that any of these schemes can do. For this reason, the statement that the recursive functions are effectively calculable has strong support, even though the term *effectively calculable* can not been precisely defined in mathematical terms. Any attempt to define an effective calculation, or, what amounts to the same thing, an algorithm, must fall back on a particular calculation scheme for a definition of the elementary steps performed.

It is natural to wonder whether all effectively calculable functions are also recursive. In other words, so far we have shown in this appendix that it is possible to compute any recursive function defined as in point 1 of **Theorem 1**, but it may be that there are functions that can be computed that are *not* recursive in the sense of point 1 of **Theorem 1**. Church's thesis hypothesizes that there are no such functions, that indeed every effectively calculable function is recursive.

This thesis is important because there are a number of important functions in mathematical logic that are not recursive, and so if the thesis holds, there is no reason to search for methods for calculating them. We will discuss one of these functions in detail, namely a predicate function that would be a solution to the so-called halting problem, which concerns the existence of an algorithm that can decide whether an arbitrary computation will halt at some point on a given input. In order to discuss this problem properly, it is necessary to show how it is possible to associate an arbitrary DSW program with a number.

DSW Program Enumeration

A set A is said to be enumerable if there is a one-to-one function from the natural numbers N to A. In a generalized sense such a set is countable, even though it may be infinite, and for this reason *enumerable* and *countable* are used interchangeably. All finite sets are enumerable; one can imagine each element in a finite set being labeled with a natural number, and as long as no labels are repeated, such a labeling would enable one to define a one-to-one function. The set of all DSW programs is also an enumerable set, a fact which is proved by attaching a unique number to each program through some encoding process.

Before discussing one such process, it will be useful first to provide an example of an infinite set that is not enumerable, the standard example being the set of real numbers. The proof by Cantor toward the end of the 19th century that the reals are not enumerable --- a major breakthrough since it opened the door to an analysis of the structure of the transfinite that among other things has lead to the discovery of large cardinals --- uses a technique that will be applicable in the following discussion of the halting problem, so we give it here.

There are several equivalent ways of defining real numbers, among these so-called Cauchy sequences and Dedekind cuts. Another way to define a real number that is equivalent to Cauchy sequences and Dedekind cuts is by the familiar decimal expansion learned in elementary school, according to which one can represent any real number say between 0 and 1 as 0.[0..9], where [0..9] represents a possibly infinite sequence of digits between 0 and 9 inclusively. So, for example, 0.02333430942063 is a real number.

Informally, one can say that two real numbers x and y are equal if and only if the absolute value of their difference, |x-y|, is arbitrarily small. By this definition of equality, two real numbers are equal if each digit in their decimal expansion is the same, but the two numbers 0.6 and 0.5999999999999999999999999999…, where 9 repeats forever, are also equal.

Theorem. The set of real numbers is not enumerable.

The proof is by contradiction. Assume that the reals are enumerable. Then it must be possible to display all the real numbers in the form of a table containing an infinite number of rows, each row containing the decimal expansion of a different real number. Some of the rows in such a table might look as follows:

N	Real Numbers
0	0.**1**00000000000000000000000000000000000000000000000000000000000000000000000…
1	0.0**1**0000000000000000000000000000000000000000000000000000000000000000000000…
2	0.00**1**000000000000000000000000000000000000000000000000000000000000000000000…
3	0.000**1**00000000000000000000000000000000000000000000000000000000000000000000…
.	
	0.222222222225555555555555555555555555555555555555556743464986430232451135322…
.	

The dots on the right of each row indicate that the rows continue for an infinite number of digits. It is possible to construct a real number x that is not equal to any number in the table, as follows. Let d_{ij} stand for the j^{th} digit in the i^{th} row of the table. Now let the nth digit in the decimal expansion of x, written x_n, be such that $x_n \neq d_{nn}$, with the further condition that if $d_{nn} = 9$, then $x_n \neq 9$ and $x_n \neq 0$. By construction, x_n is a real number not equal to any real number represented in the table according to the definition of equality of real numbers. Therefore, the assumption that the real numbers are enumerable is false. □

The digits d_{nn} in the table are on the diagonal line indicated by the digits in **bold** typeface, giving rise to the expression *proof by diagonalization* for a proof of this type.

As mentioned above, the set of all DSW programs is enumerably infinite. In order to attach a number to each program, it is necessary to devise a coding scheme which translates variables, instructions, and labels into numbers. Moreover, it is necessary to translate all of these numbers representing variables, instructions, and labels into a single number that retains all information about a given program, including the order of the instructions, so that a single number can represent an entire program.

It turns out there are ways to encode an arbitrarily large list of numbers uniquely in a (suitably large) single number. For example, it is possible to use the fact that every natural number can be written uniquely as a product of prime numbers, a result known as the unique factorization theorem in number theory. Using this fact one defines the Goedel number of a sequence of numbers (a_1, a_2, … a_n) to be the product

$$[a_1, a_2, \dots a_n] = \prod_{i=1}^{n} p_i^{ai}$$

In words, the above definition says to take the ith number in the above sequence and make it the exponent of the ith prime in the above product. For example, the sequence of numbers (3, 2, 0, 2) gives rise to a Goedel number g:

$$g = [3, 2, 0, 2] = 2^3 * 3^2 * 5^0 * 7^2 = 8 * 9 * 1 * 49 = 3528$$

We will use the Goedel number to record the instructions of a DSW program as numbers. We use another numbering scheme to encode instructions as numbers, based on the following pairing function:

$$<x,y> = (2^x (2y+1)) \dot{-} 1$$

Since $(2^x * (2 * y + 1)) \geq 1$, we have

$$<x,y> = (2^x (2y +1)) - 1.$$

Rearranging the above,

$$<x,y> + 1 = (2^x(2y +1)).$$

Setting z = <x,y>,

$$z + 1 = 2^x(2y +1),$$

or

$$(z+1)/ 2^x = 2y + 1.$$

Given a z, if we choose x to be the largest value of x such that $2^x \mid z + 1$ (i.e., such that 2^x divides z+1), then this choice of x uniquely determines x. This choice of x also means that the left hand side of the above equation is a whole number because it is required that 2^x divide z+1. Furthermore, since x is the largest number such that 2^x divides z+1, the number $(z+1)/ 2^x$ can no longer be divided by 2 and therefore must be odd. Thus, we can subtract 1 from that number to get an even number, which we can divide by 2 to get y. In other words, given a z, and the above unique choice of x, y is uniquely determined:

$$y = (((z+1)/ 2^x)-1)/2.$$

For example, if z = 0, then z = <x,y> = 0 = <0,0>. Or if z = 22, then <x,y> = <0, 11>. In general, if z is even, then <x,y> = <0, z/2>. For z = 23, <x,y> = <3, 1>. One last example: if z = 133, then <x,y> = <1, 33>.

It is also possible to nest pairing functions. Thus, using the last example and the fact that 33 = <1,8>, we can write 133 = <1,<1,8>>.

Now all the necessary encoding machinery is in place to represent a DSW program as a number. First, we establish a convention for referencing variables and labels. For this purpose, associate with each variable a number *c* as follows:

```
0 1  2  3  4  5  6  7  8  9  10 …
| |  |  |  |  |  |  |  |  |  |
```

$$Y\ X_1\ Z_1\ X_2\ Z_2\ X_3\ Z_3\ X_4\ Z_4\ X_5\ Z_5\ \ldots$$

Similarly, associate with each label a number *a*:

0	1	2	3	4	5	6	7	8	9	10	…
|	|	|	|	|	|	|	|	|	|	|	
no label	A_1	B_1	C_1	D_1	E_1	A_2	B_2	C_2	D_2	E_2	...

Next, we establish a convention for referencing an instruction by a number *b*.

0	--------	$V \leftarrow V$
1	--------	$V \leftarrow V + 1$
2	--------	$V \leftarrow V - 1$
n+2	--------	if $V \neq 0$ goto label n

Putting the above numbers together, an instruction can be given the number #(I) = <a, <b,c>>, where a is the label number, b is the instruction type, and c is the variable number. So, for example, if #(I) = 0, then the instruction is 0 = <a,<b,c>> = <0, <0,0>>, or $Y \leftarrow Y$. As another example, if #(I) = 133, from the nested pairing example above we have 133 = <1,<1,8>>, which translates to $[A_1]\ Z_4 \leftarrow Z_4 + 1$.

Finally, to represent an entire program P with a number #(P), use the Goedel number of the sequence of instruction numbers and subtract 1:

$$\#(P) = [\#(I_0), \#(I_1), \ldots \#(I_{k-1})] - 1.$$

The program consisting of the two instructions just given, 0 and 133, would have program number

$$\#(P) = [0, 133\ ] - 1 = 2^0 * 3^{133} - 1.$$

The programs given by [11, 133] and [11, 133, 0], [11, 133, 0, 0] and so on all have the same program number. These programs consist of the instructions 11 and 13 followed by zero or more lines containing $Y \leftarrow Y$. Since the instruction $Y \leftarrow Y$ has no bearing on the output of a program, it is harmless to ban these instructions at the end of a program so that to each natural number there corresponds a unique program.

The set of all DSW programs is therefore enumerable; it is also infinite, since given any finite set of programs one can always add instructions to any program in the set to create another program. The set of functions is also infinite. In fact, by a diagonalization proof analogous to Cantor's proof that the set of reals is not enumerable, one can also prove that the set of all functions on the natural numbers having values 0 or 1 is not enumerable.[188] Therefore, there must be functions on the natural numbers that are not effectively calculable. Next, we give an example of a function from NxN to {0,1} that is not effectively calculable.

The Halting Problem

One can define a predicate function HALT(x,y) that takes two natural numbers as input, one representing a program number, and the second representing the input to that program. This predicate function has value 1 if the program y halts on the input x and 0 if y does not halt on x.

Theorem. The predicate HALT(x,y) with the above-mentioned properties can not be computed by a DSW program.

Proof: Suppose that such a predicate exists. Then the following program would be possible:

```
[A]     if HALT($X_1$, $X_1$) goto A
```

When the macro HALT(x,y) is properly expanded, the above program will have a program number, say p_0. p_0 only takes one input number X_1, and it passes that same number to HALT(x, y) twice, once to be interpreted as a program and once to be interpreted as input to a program. p_0 is constructed so that if HALT(X_1, X_1) outputs 1, then p_0 is undefined since it spins in a one-line loop forever without halting. However, if HALT(X_1, X_1) outputs 0, then p_0 outputs 0, since p_0 itself halts in that case without ever changing Y.

The key point is that if the program y_0 takes *itself* as input, i.e. if $X_1 = p_0$, then p_0 halts when given itself as input if and only if $\neg$HALT(p_0,p_0), which is a contradiction. □

Alternate proof, in terms of diagonalization: The coding of programs by numbers given above amounts to a one-to-one mapping between the natural numbers and the set of all DSW programs, that is, to an enumeration of the set of all DSW programs. In particular, the coding of programs by numbers gives an enumeration of the set of all DSW programs with one input. That means we can set up a table with each row corresponding to a different program. In this table each column will correspond to a different input to the program of a given row, starting from input 0 and increasing by 1 with each increasing column number. Each square in this table can be thought of as an input to the predicate HALT(x,y), which means that for each element in the table it is possible to say whether the x corresponding to that square will halt on the y corresponding to that same square.

As a program of one input, the DSW program p_0 above must correspond to one of the rows in the table. However, from its construction we know that in each row there is at least one column where p_0 differs from the program x of that row, namely in column x.

On that input p_0 halts if the program x does not halt and vice versa; in other words, the predicate HALT(x,x) gives a different result for p_0 and the program x. Therefore p_0 is not equal to any program in the enumeration, which is a contradiction. □

The above theorem and its proof show that the predicate function HALT(x,y) is not computable by a DSW program, and therefore, by our previous demonstration that all recursive functions can be calculated by a DSW program, HALT(x,y) is not a recursive function according to point 1) of **Theorem 1**. If we adopt Church's thesis, which says that all effectively calculable functions are recursive, then there is no point in looking for a procedure to find HALT(x,y), because it is not effectively calculable. From this point of view the Halting Problem is unsolvable.

Appendix J Hilbert Space

There are three different kinds of Hilbert space, corresponding to three different kinds of so-called *-fields: the reals, the complex numbers[14] and the so-called quaternions[15]. We will define complex Hilbert space, indicating where the definitions for the other two kinds of Hilbert space differ as we go.

A Hilbert space is a particular kind of inner product space. To define an inner product space, it is necessary to first define a vector space.

A *complex vector space* V is a set of elements called vectors together with a function called addition with domain VxV and range V, and a function called multiplication with domain V and range V such that addition satisfies the following axioms, for any **x**, **y**, **z** in V:

1. $\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$
2. $\mathbf{x} + (\mathbf{y} + \mathbf{z}) = (\mathbf{x} + \mathbf{y}) + \mathbf{x}$
3. $\mathbf{x} + \mathbf{0} = \mathbf{x}$
4. $\mathbf{x} + (-\mathbf{x}) = \mathbf{0}$,

and such that multiplication satisfies the following axioms, for any **x**, **y** in V, and complex numbers c_1 and c_2:

A. $c_1(\mathbf{x} + \mathbf{y}) = c_1\mathbf{x} + c_1\mathbf{y}$
B. $(c_1 + c_2)\mathbf{x} = c_1\mathbf{x} + c_2\mathbf{x}$

[14] A complex number c can be written c = a + ib, where a and b are real numbers, ib is the ordinary multiplicative product, and $\sqrt{i} = -1$. The complex conjugate for c, written c*, is defined as c* = a-ib for c = a+ib. Also, $cc^* = (a+ib)(a-ib)= a^2 + b^2$, and by definition, $|c|^2 = a^2 + b^2 = cc^*$. c can be thought of as an ordered pair (a, b) and therefore plotted on an x-y plot, with x (=a) the real part of c and y (=b) the imaginary part of c.

[15] A quaternion can be written ai + bj + ck, where a, b, and c are real numbers and i, j and k satisfy the multiplication rule ij = -ij = k

C. $(c_1c_2)\mathbf{x} = c_1(c_2\mathbf{x})$
D. $1\mathbf{x} = \mathbf{x}$

(If we were defining real Hilbert space, we would make c_1 and c_2 real numbers. For quaternionic Hilbert space, c_1 and c_2 would be quaternions.)

An *inner product space* is a pair (V, f), where V is a complex vector space and f is a function from VxV to the set of complex numbers C called the scalar product. The scalar product, written (**x**, **y**) for **x**, **y** in V, satisfies the following axioms for all **x**, **y** and **z** in V:

1) $(\mathbf{x}, \mathbf{y}) = (\mathbf{y}, \mathbf{x})^*$, where '*' denotes the complex conjugate defined by $c^* = a - bi$ for $c = a+bi$;
2) $(\mathbf{x}, \mathbf{y} + \mathbf{z}) = (\mathbf{x}, \mathbf{y}) + (\mathbf{x}, \mathbf{z})$;
3) $c(\mathbf{x}, \mathbf{y}) = (c\mathbf{x}, \mathbf{y})$, where c is a complex number;
4) $(\mathbf{x}, \mathbf{x}) = 0$ if and only if $\mathbf{x} = \mathbf{0}$.

(For a real Hilbert space, c would be a real number and requirement 1) would read $(\mathbf{x}, \mathbf{y}) = (\mathbf{y}, \mathbf{x})$, since the *-operation is the identity. For quaternionic Hilbert space, c would be a quaternion and the *-operation would be that associated with the quaternions. R, C and the quaternions are all instances of what are called *-fields in algebra.)

The *norm* of a vector **x**, written $||x||$, is defined by $||x|| = (\mathbf{x}, \mathbf{x})^{1/2}$.

A *metric space* is a pair (M, f), where M is a set and f is a function from MxM to the positive real numbers called the *distance function*. The distance function, written d(x, y) for x, y in M, satisfies the following axioms for all x, y, and z in M:

1. $d(x, y) = 0$ if and only if $x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, z) \leq d(x, y) + d(y, z)$

Every inner product space V is also a metric space, since the norm of the vector $\mathbf{x} - \mathbf{y}$, $(\mathbf{x} - \mathbf{y}, \mathbf{x} - \mathbf{y})^{1/2}$, satisfies the three requirements of the distance function for all **x**, **y** and **z** in V. Proof:

1. $(\mathbf{x} - \mathbf{y}, \mathbf{x} - \mathbf{y}) = 0$ if and only if $(\mathbf{x} - \mathbf{y}, \mathbf{x} - \mathbf{y}) = 0$ by IV., which holds if and only if $\mathbf{x} - \mathbf{y} = 0$, i.e. $\mathbf{x} = \mathbf{y}$.
2. $(\mathbf{x} - \mathbf{y}, \mathbf{x} - \mathbf{y})^{1/2} = (-1(\mathbf{y} - \mathbf{x}), -1(\mathbf{y} - \mathbf{x}))^{1/2}$ by A. By III, $(-1(\mathbf{y} - \mathbf{x}), -1(\mathbf{y} - \mathbf{x})) = -1(\mathbf{y} - \mathbf{x}, -1(\mathbf{y} - \mathbf{x}))$. By I, $-1(\mathbf{y} - \mathbf{x}, -1(\mathbf{y} - \mathbf{x})) = (-1)(-1^*)(\mathbf{y} - \mathbf{x}, \mathbf{y} - \mathbf{x})$. Since $1^* = (1 + 0i)^* = 1$ and $(-1)(-1) = 1$, $(\mathbf{x} - \mathbf{y}, \mathbf{x} - \mathbf{y})^{1/2} = (\mathbf{y} - \mathbf{x}, \mathbf{y} - \mathbf{x})^{1/2}$.
3. Assume the triangle inequality holds: $||a+b|| \leq ||a|| + ||b||$. Then, using $x - z = (x - y) + (y - z)$ and taking $a = x - y$ and $b = y - z$ in the triangle inequality yields $||x - z|| \leq ||x - y|| + ||y - z||$, the desired result.

   To prove the triangle equality, it is necessary to first prove the Cauchy-Schwarz inequality $(x,y) \leq ||x||\ ||y||$. If $x = 0$, the inequality is satisfied since $(0,y) = 0(0,y)$

= 0, by III, and $||x||\ ||y|| = 0\ ||y|| = 0$ by IV, and similarly if $y = 0$. Assuming $y \neq 0$, we must show $(x,y)(1/||y||) \leq ||x||$, where we have divided each side of the inequality by $||y||$. Let $z = y(1/||y||)$, so that $||z|| = (z, z)^{1/2} = (y(1/||y||), y(1/||y||))^{1/2}$ $= (1/||y||)\ (y, y)^{1/2} = (1/||y||)\ ||y|| = 1$. Now we must show $(x,z) \leq ||x||$.

The following equations hold for any λ:

$||x - \lambda z||^2$
$= (x - \lambda z, x - \lambda z)$ by definition
$= (x - \lambda z, x) + (x - \lambda z, -\lambda z)$ by II
$= (x, x - \lambda z)^* + (-\lambda z, x - \lambda z)^*$ by I
$= ||x||^2 - (x, \lambda z)^* + (-\lambda z, x)^* + (-\lambda z, -\lambda z)^*$ by II, and using $(x,x)^*=(x,x)$
$= ||x||^2 - \lambda(z,x) - \lambda^*(z, x)^* + \lambda\lambda^*(z,z)$ by III, I
$= ||x||^2 - (z,x)\ (z,x)^* + (z,x)\ (z,x)^* - \lambda(z,x) - \lambda^*(z, x)^* + \lambda\lambda^*(z,z)$
adding 0

$= ||x||^2 - |(z,x)|^2 + (z,x)\ (z,x)^* - \lambda(z,x) - \lambda^*(z, x)^* + \lambda\lambda^*$
using $cc^* = |c|^2$ and $(z, z) = 1$

$= ||x||^2 - |(x,z)|^2 + (x,z)^*(x,z) - \lambda(x,z)^* - \lambda^*(x, z) + \lambda\lambda^*$
using $(z, x) = (x, z)^*$

$= ||x||^2 - |(x,z)|^2 + ((x,z) - \lambda)^*((x,z) - \lambda)$ factoring
$= ||x||^2 - |(x,z)|^2 + |(x,z) - \lambda|^2$ using $c^*c = |c|^2$

Since these equations hold for any λ, they hold for $\lambda = (x,z)$. Substituting for λ, $0 \leq ||x - \lambda z||^2 = ||x||^2 - |(x,z)|^2 + |(x,z) - (x,z)|^2 = ||x||^2 - |(x,z)|^2$. Adding $|(x,z)|^2$ to each side of the inequality gives $|(x,z)|^2 \leq ||x||^2$. Taking square roots, $|(x,z)| \leq ||x||$.

Finally, we can prove the triangle inequality as follows, again using I-IV above.

$||x + y||^2 = (x+y, x+y) = (x, x+y) + (y, x+y)$
$= (x+y, x)^* + (x+y, y)^*$
$= (x, x)^* + (y, x)^* + (x, y)^* + (y, y)^*$
$= ||x||^2 + ||y||^2 + (x, y) + (x, y)^*$.

For any $c = a + ib$, $c + c^* = (a + ib) + (a - ib) = 2a$. Writing the real part of the complex number (x, y) as $\mathrm{Re}((x,y))$, we have
$||x + y||^2 = ||x||^2 + ||y||^2 + 2\mathrm{Re}((x, y))$.

For any $c = a+ib$, $\mathrm{Re}(c) \leq |\mathrm{Re}(c)| \leq |c|$, since $a \leq |a| \leq |a^2 + b^2|^{1/2}$. Therefore,
$||x + y||^2 \leq ||x||^2 + ||y||^2 + 2|(x, y)|$.

Using the Cauchy-Schwarz inequality,
$||x + y||^2 \leq ||x||^2 + ||y||^2 + 2||x||\ ||y|| = (||x|| + ||y||)^2$.

Taking square roots,

$||x + y|| \leq (||x|| + ||y||)$. □

A *sequence* of elements of a metric space M is a mapping from the natural numbers > 0 to M. A sequence $\{x_n\}$ *converges* to x if $d(x_m, x) \to 0$ in the limit as m goes to infinity; a more precise way to say this is[16]: for every $\varepsilon > 0$, there exists a k such that $d(x_k, x) < \varepsilon$. A sequence is said to be *Cauchy* if, for every $\varepsilon > 0$, there exists a k such that $d(x_m, x_n) < \varepsilon$ for all $m, n \geq k$. Since $d(x_m, x_n) \leq d(x_m, x) + d(x, x_n)$ by 3) above, if a sequence $\{x_n\}$ converges to x, then that sequence is Cauchy. However, every Cauchy sequence is not convergent, as will be shown below.

A metric space M is *complete* if every Cauchy sequence in M converges in M. In other words, in a complete metric space, if a sequence in M is such that $d(x_m, x_n) \to 0$ as m,n go to infinity, then there exists an x such that $d(x_m, x) \to 0$, that is, the sequence converges to x.

A *Hilbert space* H is an inner product space that is a complete metric space whose distance function d(x, y) is given by the norm of the vector difference $\mathbf{x} - \mathbf{y}$.

Examples

1. **The open interval (0, 1) ⊂ R is not a complete metric space**. Let S be the open interval $0 < x < 1$ in R, the reals. The absolute value of the difference of two reals $|x - y|$ satisfies the requirements of a distance function since

1. $|x - y| = 0$ if and only if $x = y$,
2. $|x - y| = |y - x|$
3. $|x - z| = |x - y| + |y - z|$.

S is therefore a metric space. In order for S to be a complete metric space, all Cauchy sequences must converge in S. Consider the sequence {1, 1/2, 1/3, 1/4, 1/5, …}. By substituting values into $|1/n - 1/m|$ one can see that $|1/n - 1/m| < 1/k$ for all $m, n \geq k$. By definition, a sequence is Cauchy if for every $\varepsilon > 0$, there exists a k such that $d(x_m, x_n) < \varepsilon$ for all $m, n \geq k$. Accordingly, given an $\varepsilon < 1$, we choose k such that $1/k < \varepsilon < 1/(k-1)$, and if $\varepsilon \geq 1$, we choose $k = 1$, thus satisfying the definition. Now this sequence converges to 0, since for any ε, $|0 - 1/n| = 1/n < \varepsilon$ for all $n \geq k$, where k is again chosen such that $1/k < \varepsilon < 1/(k-1)$. However, 0 is not in S. Since in a complete metric space M *all* Cauchy sequences converge in M, this one counterexample is enough to establish that M is not a complete metric space.

[16] The ε, k definition of a convergent sequence may seem more natural to the reader considering that in the formal language of first order logic definitions must involve one or more of the quantifiers *there exists* and *for all*. See Appendix A.

2. **R, the set of reals, is a Hilbert space**. From the above example, it is clear that |x-y| works as a distance function, making R a metric space. A Hilbert space is also an inner product space, so we must establish that R is a vector space with an inner product, an inner product whose norm $||x - y|| = (x-y, x-y)^{1/2}$ satisfies the requirements of a metric space distance function. Since the reals are a subset of the complex numbers C, we can

Clearly, for any real numbers x and y:

1. $x + y = y + x$;
2 $x + (y + z) = (x + y) + x$;
3. $x + 0 = x$;
4. $x + (-x) = 0$.

Also, multiplication satisfies the following axioms, for any reals x and y in R, and complex numbers c_1 and c_2:

1. $c_1(x + y) = c_1x + c_1y$;
2. $(c_1 + c_2)x = c_1x + c_2x$;
3. $(c_1c_2)x = c_1(c_2x)$;
4. $1x = x$.

Therefore, R is a vector space. As a scalar product, let (x, y) = xy. Then

1. $(x, y) = (y, x)^*$ is satisfied since $xy = (yx)^* = yx$;
2. $(x, y + z) = (x, y) + (x, z)$ is satisfied since $x(y+z) = xy + xz$;
3. $c(x, y) = (cx, y)$ is satisfied since $cxy = cxy$;
4. $(x, x) = 0$ if and only if $x = 0$ is also satisfied.

Lastly, we must show that every Cauchy sequence in R converges, a basic result of the branch of mathematics called real analysis. To do this, we first show that every Cauchy sequence is bounded. That will allow us to generate a so-called convergent subsequence that will be used to show that every Cauchy sequence converges.

To show that every Cauchy sequence is bounded, the following inequality is needed.

$$| \, ||x|| - ||y|| \, | \leq ||x - y||$$

Its proof is similar to that of the triangle equality:

$$\begin{aligned} ||x - y||^2 &= (x - y, x - y) = (x, x - y) - (y, x - y) \\ &= (x - y, x)^* - (x - y, y)^* \\ &= (x, x) - (y, x)^* - (x, y)^* + (y, y) \\ &= ||x||^2 + ||y||^2 - ((y, x)^* + (y, x)) \end{aligned}$$

For any $c = a + ib$, $c + c^* = (a + ib) + (a - ib) = 2a$. Writing the real part of the complex number (x, y) as Re((x,y)), we have

$$||x - y||^2 = ||x||^2 + ||y||^2 - 2\mathrm{Re}((x, y)).$$

For any $c = a+ib$, $\mathrm{Re}(c) \le |\mathrm{Re}(c)| \le |c|$, since $a \le |a| \le |a^2 + b^2|^{1/2}$. Therefore,

$$||x - y||^2 = ||x||^2 + ||y||^2 - 2\mathrm{Re}((x, y)) \ge ||x||^2 + ||y||^2 - 2|(x, y)|,$$

which is equivalent to:

$$||x||^2 + ||y||^2 - 2|(x, y)| \le ||x - y||^2 .$$

Using the Cauchy-Schwarz inequality and factoring,

$$||x||^2 + ||y||^2 - 2||x||\ ||y|| \le ||x - y||^2$$

$$(||x|| - ||y||\ )^2 \le ||x - y||^2$$

Taking square roots yields

$$|\ ||x|| - ||y||\ | \le ||x - y||.$$

Now we can show that every Cauchy sequence $\{x_n\}$ is bounded, which means that there is a real number B such that for all n, $x_n \le B$ and a real number B' such that for all n, $B' \le x_n$. Let k_1 be a number such that $||\ x_m - x_n\ || < 1$ for all $n \ge m \ge k_1$; such a k_1 exists by the definition of a Cauchy sequence, which says for all ε there exists a k such that … and so on. Here we have taken the particular value $\varepsilon = 1$, and the subscript in k_1 is meant as a reminder of this fact. By the inequality proved above it follows that for all values of m and n greater than k_1

$$|\ ||x_m|| - ||x_n||\ | \le ||\ x_m - x_n\ || < 1.$$

From this inequality we can deduce,[17] by using $|\ ||x_m|| - ||x_n||\ | = |\ ||x_n|| - ||x_m||\ |$, removing the absolute value sign and adding $||y_m||$ to each side,

$$||x_n|| < 1 + ||x_m|| \text{ for } n \ge m \ge k_1.$$

In words, the above inequality says that for every value of n greater than k_1, the value of the norm of x_n is less than the real number $1 + ||x_m||$. Now since the above inequality holds for all m such that $n \ge m \ge k_1$, it also holds for the particular value $m = k_1$. Thus the following inequality holds.

$$||x_n|| < 1 + ||x_{k1}|| \text{ for } n \ge k_1.$$

To underscore that $||x_{k1}||$ is just a real number, let $A = ||x_{k1}||$. At this point it is easy to get a bound on $\{x_n\}$, since we have a bound on all elements in the sequence *after and including* the k_1th element, and there are only a finite number of elements in the sequence *before* the k_1th element. Taking the supremum (also known as the least upper bound) of the elements before the k_1th element and 1 + A, the bound on all other elements, yields an upper bound for the entire sequence $\{x_n\}$:

$$B = \text{supremum}\{||x_0||, ||x_1||, ||x_2||, \ldots ||x_{k1-1}||, 1 + A\ \}$$

[17] We can equally well deduce $||x_m|| < 1 + ||x_n||$, which we do not use.

Similarly, one can establish a lower bound on $\{x_n\}$by noting that $-(1+ ||x_{k1}||) \leq x_n$ for $n \geq k_1$, and then taking the greatest lower bound of the elements before the k_1th element. Setting A' = $-(1+ ||x_{k1}||)$, the greatest lower bound is

$$B' = \text{infimum}\{x_0, x_1, x_2, \ldots x_{k1-1}, A'\}.$$

Every Cauchy sequence is therefore bounded.

The next step in our proof that every Cauchy sequence in R converges is to establish that every Cauchy sequence has a convergent subsequence. Using the fact that a Cauchy sequence $\{x_n\}$ is bounded, one can form a subsequence by invoking an axiom of real analysis called the greatest lower bound axiom, which states that every bounded subset of R contains a greatest lower bound.[18] Let a_0 = infimum$\{ x_n \}$. Next, let a_1 be the infimum of the subsequence starting at n = 1 (skipping the zeroth element of the sequence), that is, a_1 = infimum$\{ x_n \mid n \geq 1 \}$. We know that a_1 exists since the set $\{ x_n \mid n \geq 1 \}$ is a subset of $\{x_n\}$, and $\{x_n\}$ is bounded. Since a_0 is the greatest lower bound of the whole sequence, and a_1 is the greatest lower bound of that same sequence minus the zeroth element, it must be that a_1 is greater or equal to a_0. a_1 is greater than a_0 if and only if x_0 is the greatest lower bound and that greatest lower bound occurs as the value of no other element of the sequence. Continuing this way, one can define a sequence

$$a_i = \text{infimum}\{x_j \in x_n \mid j \geq i \}$$

A sequence such as a_i, in which each element is equal to or greater than its predecessor, is called a monotone increasing sequence. Monotone increasing sequences converge if and only if they are bounded. We show next that they converge if they are bounded, since that is what we need to complete the proof that all Cauchy sequences converge.

There is an axiom, not independent of the greatest lower bound axiom cited above, called the least upper bound axiom, which states that every bounded subset of R contains a supremum. One can invoke this axiom to guarantee the existence of a supremum of the sequence $\{a_i\}$ described above. Let L = supremum$\{a_i\}$. Since L is the supremum of $\{a_i\}$, for $\varepsilon > 0$ it must be true that $L - \varepsilon$ is less than some element of a_i unless the sequence a_i is the trivial sequence $\{L, L, L, \ldots\}$, in which case a_i obviously converges to L. For the non-trivial case, let $a_{k(\varepsilon)}$ be such that $L - \varepsilon < a_{k(\varepsilon)}$, where $k(\varepsilon)$ is some natural number. Since a_i is monotone increasing and therefore $a_{k(\varepsilon)} \leq a_{k(\varepsilon)+1} \leq a_{k(\varepsilon)+2}$ and so on, it is true that $L - \varepsilon < a_j$ for all $j \geq k(\varepsilon)$. Rearranging terms, $L - a_j < \varepsilon$ for all $j \geq k(\varepsilon)$.

[18] The existence of a supremum and infimum for every bounded set is a non-trivial property of the reals. The rationals Q, for example, don't have it, since $S = \{s \in Q \mid s^2 < 2 \}$ has no supremum in Q. The greatest lower bound axiom implies the least upper bound theorem, according to which every bounded subset of R contains a least upper bound --- in fact, one can make the least upper bound theorem an axiom, which implies then the lower bound *theorem*. In either case, the axiom of choice is required; for a discussion of this point, see Bridges, D., Constructive truth in practice, in Dales, H, & Oliveri, G., eds., *Truth in Mathematics*, Oxford Univ. Press, Oxford (1998) --- note that in terms of that discussion, one cannot assume here that the subset in question is totally bounded without the axiom of choice.

Since $L - a_j \leq | L - a_j |$, and since ε was an arbitrary real greater than 0, the foregoing establishes that for every $\varepsilon > 0$ there exists a k such that $| L - a_j | < \varepsilon$ for all $j \geq k$. In other words, a_i converges to L.

The last step in the proof is to show that the arbitrary Cauchy sequence $\{x_n\}$ converges since it contains a convergent subsequence. $\{x_n\}$ is a Cauchy sequence, so for every ε/2 there is a k_c (*c* for Cauchy) such that

$$| x_m - x_n | < \varepsilon/2 \text{ for all } n \geq m \geq k_c.$$

Also, for every ε/2 there is a k_S (*s* for subsequence) such that

$$| L - a_j | < \varepsilon/2 \text{ for all } j \geq k_S.$$

Let k_m (*m* for merged) be the maximum of k_c and k_S. Since $k_m \geq k_c$, it is true that

$$| x_p - x_q | < \varepsilon/2 \text{ for all } p \geq q \geq k_m.$$

Also, since $k_m \geq k_S$, it is true that

$$| L - a_r | < \varepsilon/2 \text{ for all } r \geq k_m.$$

Finally, for all $p \geq r \geq k_m$

$$|L - x_p| = |L - a_r + a_r - x_p | \leq |L - a_r| + |a_r - x_p| < \varepsilon/2 + \varepsilon/2 = \varepsilon.$$

The last equation uses $|a_r - x_p| < \varepsilon/2$ and $p \geq r \geq k_m$. To see that these inequalities hold, notice that the indices of the subsequence a_i lag behind the indices of the sequence x_n, since a_r, the rth infimum, was chosen from $\{x_q \in x_n \mid q \geq r\}$. Therefore, a_r is equal to some x_q, and since $| x_p - x_q | < \varepsilon/2$ for all $p \geq q \geq k_m$, it is also true that $|a_r - x_p| = | x_p - a_r | < \varepsilon/2$. □

3. **Other Hilbert spaces**

R^n, the space of n-dimensional reals, is also a Hilbert space. A vector in R^n has a real number in each of its n "slots"; for example, a vector in R^2 can be written as a pair of components, say (1.0, 2.0). The scalar product of two vectors **x** and **y** in R^n is the sum of products of the components in **x** and **y**; for example, if **x** is as above and **y** is (21.0, 22.0), then the scalar product of **x** and **y** would be 1.0(21.0) + 2.0(22.0) = 21.0 + 44.0 = 65.0. The norm of a vector, $(\mathbf{x} - \mathbf{y}, \mathbf{x} - \mathbf{y})^{1/2}$ yields the distance function; for example, the norm of $\mathbf{x} - \mathbf{0}$ is $((1.0)(1.0) + 2.0(2.0))^{1/2} = 5^{1/2}$ --- this makes sense if the components correspond to coordinates with fixed axes at right angles to one another, since the distance of the point (1.0, 2.0) from the origin is $5^{1/2}$ units as given by the Pythagorean theorem. The proof that Cauchy sequences converge in R^n is practically identical to the proof given above for R.

Another example of a Hilbert space is C^n, the space of n-dimensional complex numbers. The one dimensional space of complex numbers C, for example, has the scalar product $c_1c_2^*$ for elements c_1, c_2 in C, and norm $(c_1 - c_2, c_1 - c_2)^{1/2}$ given by $((c_1 - c_2)(c_1 - c_2)^*)^{1/2}$. Using $cc^* = (a + ib)(a - ib) = a^2 + b^2$, and then expanding c to $c_1 - c_2$, this reduces to $((a_1 - a_2)^2 + (b_1 - b_2)^2)^{1/2}$, the correct distance function in the complex plane. A sequence in C is defined as $\{c_n\} = \{a_n + ib_n\}$, where $\{a_n\}$ and $\{b_n\}$ are sequences of reals, and it can be shown that $\{c_n\}$converges to $c_L = a_L + ib_L$ if and only if $\{a_n\}$ converges to a_L and b_n converges to b_L. Therefore, the proof that Cauchy sequences converge in C carries over from R almost intact, and similarly for higher dimensions.

Two infinite dimensional Hilbert spaces are also of particular importance. The Hilbert space l^2 is defined as the set of all square-summable complex sequences, that is, sequences satisfying

$$\sum_{i=1}^{\infty} |x_i|^2 < \infty.$$

Given sequences x and y in l^2, their scalar product is defined by

$$(x, y) = \sum_{i=1}^{\infty} x_i y_i^* < \infty.$$

Vector space addition is defined on x, y in l^2 as addition of the corresponding elements of the sequences x and y: $(x + y)_i = x_i + y_i$. Similarly, scalar multiplication on $x \in l^2$ by $c \in C$ is defined as multiplication of each element of x by c. The proof that every Cauchy sequence converges in l^2 is complicated by the fact that a Cauchy sequence in l^2 is a sequence of sequences.[189]

Another example of an infinite dimensional Hilbert space is $L^2(a,b)$, the space of all so-called Lebesgue measurable functions that have the property

$$\int_a^b |f(t)|^2 \, dt < \infty.$$

in words, the integral of the absolute value squared of f over the complex interval [a, b] is finite. The definitions of this integral and these functions are beyond the scope of this exposition[190]; suffice to say that an integral generalizes a sum by adding up values over a continuous range of points as opposed to a discrete range of points. In L^2 addition is defined on functions $f, g \in L^2$ pointwise: $(f + g)(t) = f(t) + g(t)$. Similarly, scalar multiplication on $f \in L^2$ by $c \in C$ is defined as $(cf)(t) = cf(t)$. The scalar product of $f, g \in L^2$ is defined as

$$(f, g) = \int_a^b f(t)g(t)^* \, dt < \infty.$$

The proof of Cauchy convergence of elements of L^2 is to be found in standard texts on measure theory.[191]

The spaces l^2 and L^2 are of great relevance for quantum mechanics as they are the underpinnings of the matrix mechanics of Heisenberg and the wave mechanics of Schroedinger, respectively.[192]

Hilbert space miscellanea

Orthonormal bases

Elements of the Hilbert space R^2 are vectors that can be represented as points on an x-y plot with x and y coordinates. In this representation, an arbitrary point in the plane is located by going a certain distance in the x direction and then a certain distance in the y direction, in either order. Going to the language of vector spaces, an arbitrary vector **r** in R^2 is represented as the vector sum of two vectors, one in the x-direction and the other in the y-direction. The length of the vector in the x-direction is the x-component of **r** and the length of the vector in the y-direction is the y-component of **r**. It is convenient to introduce the notion of an *orthonormal basis* to adequately describe this situation.

1. Two vectors x, y of a Hilbert space H are *orthogonal* if $(x, y) = 0$.
2. A family[19] $\{x_i\}_{i \in I}$ of vectors in $H - \{0\}$ is an *orthonormal system* if $(x_i, x_j) = 1$ if $i = j$ and 0 otherwise.
3. An orthonormal system is *total* if the only vector orthogonal to every vector in the system is the 0 vector.
4. A total orthonormal system is an *orthonormal basis*.[20]

In R^2, the vectors $\mathbf{e_1} = (1, 0)$ and $\mathbf{e_2} = (0, 1)$ form an orthonormal basis since

1. $(\mathbf{e_1}, \mathbf{e_2}) = 1(0) + 0(1) = 0$,
2. $(\mathbf{e_1}, \mathbf{e_1}) = 1(1) + 0(0) = 1$, and $(\mathbf{e_2}, \mathbf{e_2}) = 0(0) + 1(1) = 1$,
3. $(\mathbf{e_1}, 0) = (\mathbf{e_2}, 0) = 0$, $(\mathbf{e_1}, \mathbf{e_1}) = (\mathbf{e_2}, \mathbf{e_2}) = 1$.

In this orthonormal basis a typical vector **r** in R^2 corresponding to a point at say $\mathbf{r} = (10.5, 32.0)$ in the x-y plane is written as $\mathbf{r} = 10.5\mathbf{e_1} + 32.0\mathbf{e_2}$. Notice that this expansion for **r** can also be written $\mathbf{r} = (\mathbf{r}, \mathbf{e_1})\, \mathbf{e_1} + (\mathbf{r}, \mathbf{e_2})\, \mathbf{e_2}$.

The above expansion for **r** is characteristic of a Hilbert space with an orthonormal basis in the sense that one can prove the following theorem:

[19] A family is a set whose elements are labeled by elements of an index set such as the set of natural numbers. The above notation $i \in I$ indicates that i is a label in an index set I.
[20] A Hilbert space with an orthonormal basis is said to be *separable*.

Theorem I.1 If (e_n) is an orthonormal basis in a Hilbert space H, then for any x ϵH,

$$x = \sum_{i=1}^{\infty} (x, e_n)\, e_n , \qquad \text{if } (e_n) \text{ is infinite, and}$$

$$x = \sum_{i=1}^{n} (x, e_n)\, e_n , \qquad \text{if } (e_n) \text{ is finite.}$$

Orthonormal bases are not unique, as one can see from the orthonormal basis for R^2 given by $\mathbf{e_1} = (1/(2^{1/2}), 1/(2^{1/2}))$, $\mathbf{e_2} = (-1/(2^{1/2}), 1/(2^{1/2}))$.

An orthonormal basis for l^2 is $\mathbf{e_1} = (1, 0, 0\ldots)$ $\mathbf{e_2} = (0, 1, 0, 0, \ldots)$ $\mathbf{e_3} = (0, 0, 1, 0, 0, \ldots)$, and so on.

An orthonormal basis for L^2 is $\mathbf{e_n}(x) = 1/((2p)^{1/2})(\cos(nx) + i\sin(nx))$. An arbitrary square-integrable function f can be written

$$f = \sum_{i=1}^{\infty} (f, e_n)\, e_n = \sum_{i=1}^{\infty} u_n\, e_n ,$$

where the u_n are called the Fourier coefficients of f. To provide historical perspective to the above representation of a function, one can say with confidence that mathematicians before the time of Fourier (17?? – 18) would have regarded as a marvelous revelation the fact that an arbitrary function is a vector with an expansion in terms of an orthonormal basis. The polished formulation of this result in terms of vector spaces was not discovered until the early twentieth century. Figures ?? shows the first ten terms in the expansion of the function $f(x) = 3x^3 + 1$ in the L^2 basis given above.

Hilbert space isomorphisms

A Hilbert space isomorphism is a map T from a Hilbert space H to a Hilbert space J satisfying the following conditions**:**

1. T is one-to-one, i.e. if x, y ϵH are such that $x \neq y$, then $Tx \neq Ty$.
2. T is onto, i.e. for all z ϵ J, there is an x ϵH such that Tx = z.
3. T(x + y) = T(x) + T(y) for all x, y ϵH.
4. T(cx) = cT(x) for all x ϵH and c ϵC.
5. (Tx, Ty) = (x, y) for all x, y ϵH.

Conditions 1 and 2 above together say that J has the same number of elements as H and that T maps exactly one element from H to a given element in J.[21] Conditions 3 and 4 say

[21] In general, a one-to-one and onto function is called a *bijection*. One-to-one functions are also called *injective* and onto functions *surjective*.

that addition of vectors and multiplication by complex numbers are operations that are preserved by T. Condition 5 says that scalar products are preserved, which in a Hilbert space means that distances are preserved. Considering the definition of a Hilbert space, first as a vector space, then as an inner product space, and finally as a metric space, it is clear from the above conditions that if an isomorphism exists between two Hilbert spaces, then they are essentially the same space although each under a different guise.

Theorem I.2 Let H be a Hilbert space with an orthonormal basis. If H is of finite dimension n, then H is isomorphic to C^n. If H is infinite dimensional, then H is isomorphic to l^2.

Because of the above theorem there is up to isomorphism just one infinite dimensional Hilbert space, and just one Hilbert space for each dimension n. Thus, for example, since l^2 and L^2 both have orthonormal bases and they are both infinite dimensional, there is an isomorphism from l^2 to L^2, which we now show.

The expansion given above of a function f in L^2 in terms of the orthonormal basis e_n,

$$f = \sum_{i=1}^{\infty} (f, e_i)\, e_i = \sum_{i=1}^{\infty} u_i\, e_i\, ,$$

contains an implict mapping T from l^2 to L^2: (u_n) is an element of l^2 that is mapped to f. T is an isomorphism since

1. T is one-to-one, since if $(u_n) \neq (v_n)$, then (u_n) and (v_n) differ on at least one index, say i. Then $(T((u_n)), e_i) \neq (T((v_n)), e_i)$, which implies $T((u_n)) \neq T((v_n))$;
2. T is onto, since every f can be expanded by some u_n according to theorem I.1;
3. $T((u_n) + (v_n)) = T((u_n)) + T((v_n))$, since $(u_i + v_i)e_i = u_ie_i + v_ie_i$;
4. $T(c(u_n)) = cT((u_n))$ since $cu_i e_i + cu_je_j = c(u_ie_i + u_je_j)$ for all $i, j \in N$ and $c \in C$;
5. $(T((u_n)), T((v_n))) = ((u_n), (v_n))$, since

$$(T((u_n)), T((v_n))) = (\sum_{i=1}^{\infty} u_i\, e_i, \sum_{i=1}^{\infty} v_i\, e_i) =$$

$$= (u_1e_1, \sum_{i=1}^{\infty} v_i\, e_i) + (u_2e_2, \sum_{i=1}^{\infty} v_i\, e_i) + \ldots$$

$$= ((v_1e_1)^*, u_1e_1) + ((v_2e_2)^*, u_1e_1) + \ldots + (u_2e_2, \sum_{i=1}^{\infty} v_i\, e_i) + \ldots$$

$$= ((v_1e_1)^*, u_1e_1) + \qquad 0 \qquad + \ldots + (u_2e_2, \sum_{i=1}^{\infty} v_i\, e_i) + \ldots$$

∞ ∞

$$= \sum_{i=1} v_i^* u_i = \sum_{i=1} u_i v_i^* = ((u_n), (v_n)),$$

where the last equality came from the definition of the scalar product on l^2 given above.

Dual vector spaces

One of the things that characterize a Hilbert space is that the so-called dual vector space associated with a Hilbert space is in some sense the Hilbert space itself. To explain this requires a few definitions.

A function f from a metric space A to a metric space B is said to be *continuous* if the following holds: given a convergent sequence (x_n) in A such that $x_n \to x$, then $f(x_n) \to f(x)$. This definition matches the everyday sense of the word *continuous*, since it implies that neighbouring points in the domain of the function become neighbouring points in the range; an example of a non-continuous function with domain and range R is

$$\begin{aligned} f(x) &= 1 \qquad \text{for } 0 \le x \le 1 \\ &= 0 \qquad \text{otherwise,} \end{aligned}$$

which is not continuous at x = 0 and x = 1.

A *linear form* f on a vector space V is a mapping from V to C such that f(x +y) = f(x) + f(y) and f(cx) = cf(x) for all x, y ϵ V and c ϵ C.

A vector space V is a *normed space* if it has a real-valued function associated with it called the a norm, written ||x|| where x ϵ V, satisfying the following conditions for all x, y ϵ V and c ϵ C:

1. $||x|| = 0$ if and only if $x = 0$;
2. $||x + y|| \le ||x|| + ||y||$;
3. $||cx|| = |c|\, ||x||$.

Every normed space is a metric space since the norm of ||x – y|| satisfies the requirements of a distance function d(x,y). Also, every inner product space is a normed space since the inner product $(x, x)^{1/2} = ||x||$ satisfies the above conditions by definition. It follows that every Hilbert space is a normed space.

Just as a Hilbert space is an inner product space which is a complete metric space under the metric induced ultimately by its *inner product* --- recall that in a Hilbert space H the inner product induces a metric via $(x-y, x-y)^{1/2} = ||x – y||$ and this metric is complete, i.e., all Cauchy sequences (that is, sequences such that $||x_m – x_n|| \to 0$ as $m,n \to \infty$) converge to an element x in H --- a *Banach space* is a complete metric space under the metric induced by its *norm*. In other words, a Banach space B is a normed space in which all Cauchy sequences converge in B.

Every Hilbert space is a Banach space, but not every Banach space is a Hilbert space. It is the presence of the inner product in a Hilbert space that distinguishes it in general from a Banach space. For example, it is the inner product that allows one to construct an isomorphism between any two Hilbert spaces of the same dimension[22], a construction for which there is no analogue in a Banach space in general. Another distinguishing aspect of a Hilbert space due to its inner product regards its dual space, as we discuss next.

The *dual space* N' of a normed space N is the set of all continuous linear forms from N to C. It can be shown that every dual space is itself a normed space subject to the following conditions for all f, g ϵ N', x ϵ N, and c ϵ C: [193]

1. (f + g)(x) = f(x) + g(x);
2. (cf)(x) = cf(x);
3. || f || = supremum{|f(x)| such that || x || ≤ 1 }.

Theorem I.3 For every element f in the dual space H' of a Hilbert space H, there is an element y in H such that f(x) = (y, x).

Every Hilbert space is "self-dual" as a consequence of the above theorem.

Finally, there would be a large gap in this presentation of Hilbert space if no mention were made of the fact that Hilbert space has a very rich structure. In particular, the study of operators on Hilbert space has been a central area of mathematical research in the last 50 years with deep connections to physics.

Figures

Figure 1 From Henshilwood, C., et al., Emergence of Modern Human Behavior: Middle Stone Age Engravings from South Africa, Science 295: 1278-1280 (2002)

[22] The proof of theorem I.2 can be found for example in Berberian, S., Introduction to Hilbert Space, Chelsea Publishing Co., New York (1976), or in Young, N., An Introduction to Hilbert Space, Cambridge Univ. Press, Cambridge (1988).

Figure 2: "Left-shiftable" Timeline

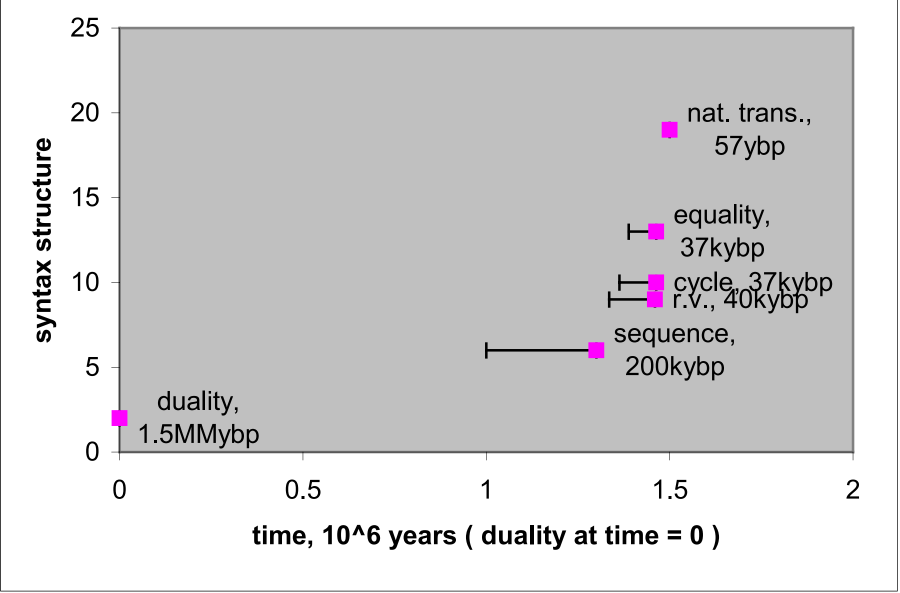


Figure 3: Wild Guess Timeline

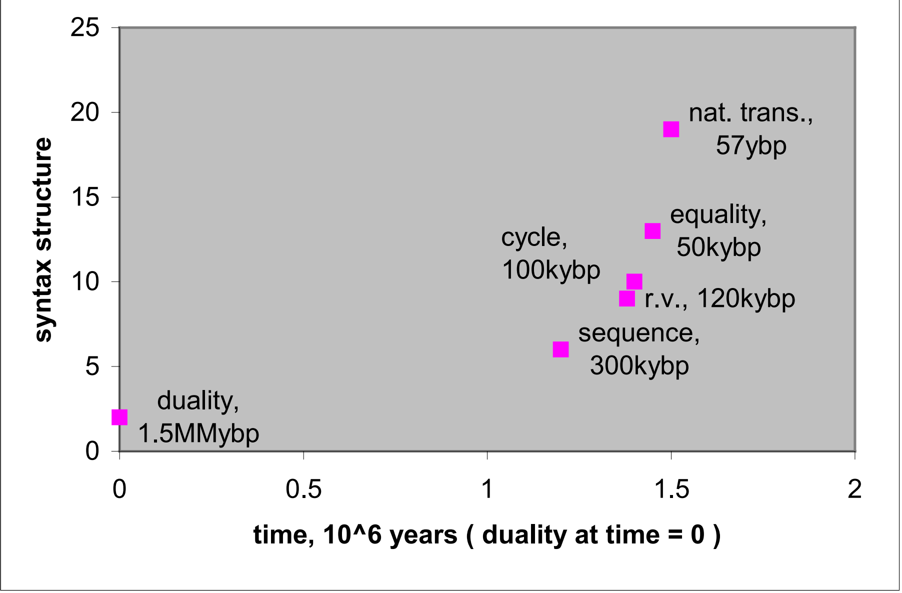


Figure 4: From Deacon, T., Brain-Language Coevolution, in Hawkins, J. & Gell-Mann, M., (Eds.), SFI Studies in the Sciences of Complexity, Proc. Vol X, Addison-Wesley (1992), p. 65

Figure 5: Exponential fits to points in Figure 2

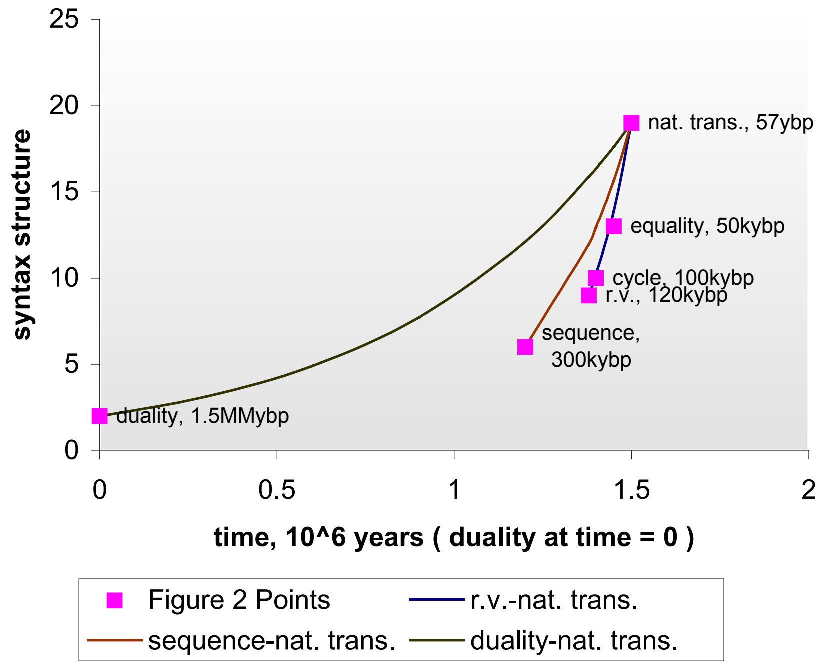
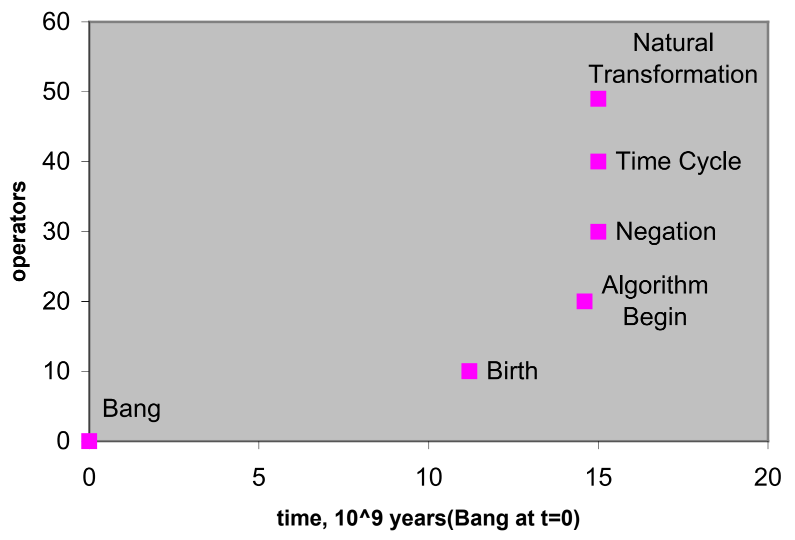


Figure 6: Timeline starting from Bang



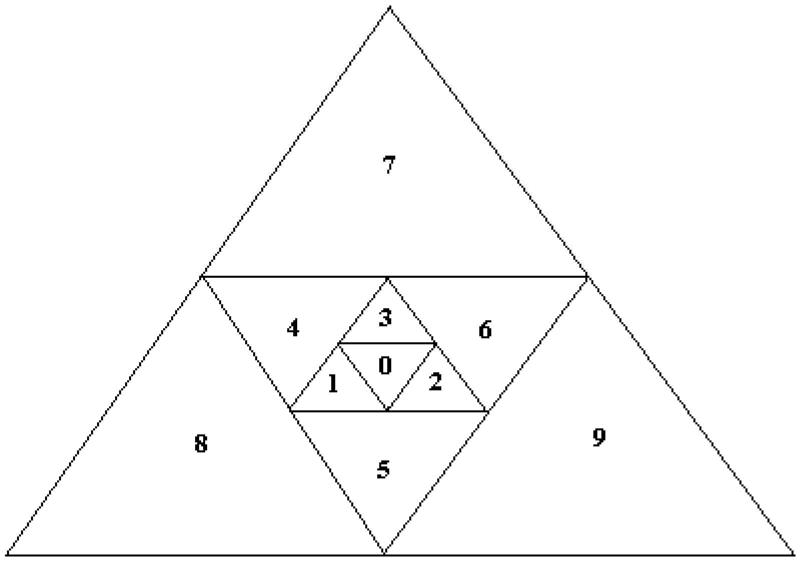


Figure 7 : Geometrical representation of an epoch

Figure 8: mandelbrot set

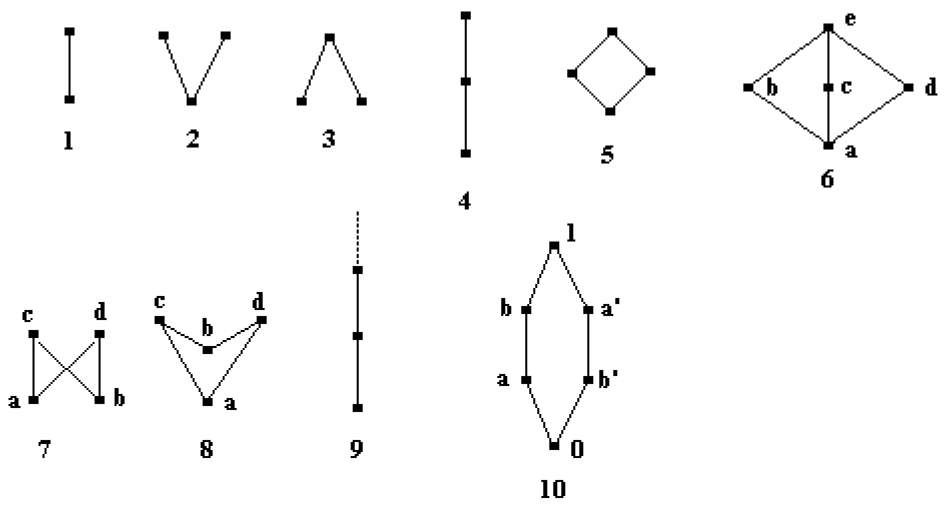
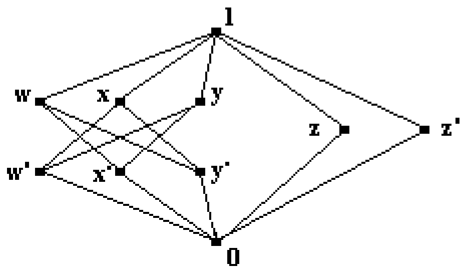
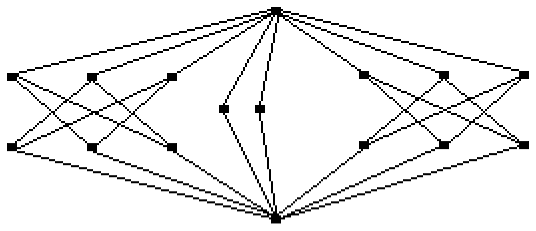


Figure G1
The Hasse diagram labeled 1 above consists of two elements. It is a lattice since it is a partial order (all Hasse diagrams by definition describe partial orders) and since all elements x and y have a join $x \vee y$ and a meet $x \wedge y$. Lattices 1, 4 and 9 are called chains, i.e. lattices for which for all elements a and b either $a \leq b$ or $b \leq a$. The dots on top of lattice 9 indicate that the chain goes on forever; this diagram represents N, the natural numbers. Lattice 2 has no 1 element; lattice 3 has no 0 element. Lattice 6 is referred to

as M_3. It is non-distributive, since $b \vee (c \wedge d) = b \vee a = b$, whereas $(b \vee c) \wedge (b \vee d) = e \wedge e = e$. The partially ordered sets in 7 and 8 represent the same partial orders; they are not lattices since the meet of a and b doesn't exist and the join of c and d does not exist. Lattice 10 is referred to as O_6. It is an orthoposet, since for each pair of elements x, y there are orthocomplements satisfying the conditions $x \leq y$ if and only if $y' \leq x'$, and for all elements x $x' \vee x = 1$ and $x' \wedge x = 0$. Since it is an orthoposet and a lattice, it is an ortholattice.



11



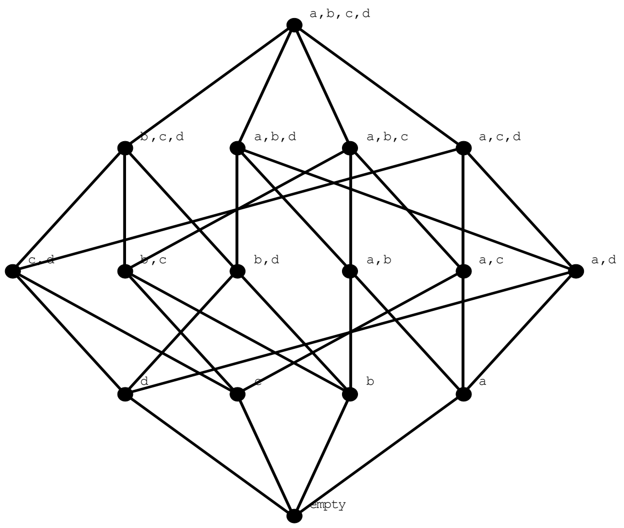
12

Figure G2

Lattice 11 is an orthomodular lattice L. It is a lattice since for all x, y in L, the join $x \vee y$ and the meet $x \wedge y$ exist. It is an orthoposet just as lattice 10 is an orthoposet. There are five different ways to check orthomodularity according to the theorem in the text, which is applicable here since L is an ortholattice. For example, one can verify that $p \leq q$ implies that $p \vee (p' \wedge q) = q$. Since there are 23 pairs of comparable elements (i.e. 23 pairs of elements {p,q} such that $p \leq q$) , there are 23 equations to check. For example, for $w' \leq y$, $w' \vee (w \wedge y) = w' \vee x' = y$; for $w' \leq x$, $w' \vee (w \wedge x) = w' \vee y' = x$; for $x' \leq w$, $x' \vee (x \wedge w) = x' \vee y' = w$. Alternatively, according to the theorem in the text any two comparable elements must generate a boolean subalgebra. For example, $x' \leq y$ generates $\Gamma(x',y) = \{0, 1, x, x, y, y', w, w'\}$, which includes elements w and w' since $x' \vee y' = w$ and the subalgebra is closed under complementation and meets and joins. $\Gamma(x',y)$ is an ortholattice --- going from L to $\Gamma(x',y)$ means removing z and z' from L, which leaves the ortholattice conditions for the elements of $\Gamma(x',y)$ still satisfied --- so in order to show that $\Gamma(x',y)$ is boolean it is necessary to show only that it is distributive. For example, $w \vee (x \wedge y) = w \vee w' = 1$, and $(w \vee x) \wedge (w \vee y) = 1 \wedge 1 = 1$. Similarly,

$x' \vee (x \wedge w) = x' \vee y' = w$, and $(x' \vee x) \wedge (x' \vee w) = 1 \wedge w = w$. An easy way to confirm that L is orthomodular is to note that O_6 is not in the lattice.

Lattice 12 differs only from lattice 11 in that a copy of the subalgebra $\Gamma(x',y)$ of lattice 11 has been pasted or inserted on the right hand side (with elements assumed to be suitably renamed). Clearly, the another subalgebra G is generated by this new addition. Therefore, any two comparable elements generate a boolean subalgebra and lattice 12 is orthomodular.



13

Figure G3
The above Hasse diagram shows the boolean algebra given by the powerset of the set {a, b, c, d}, i.e. the set of subsets of {a, b, c, d}. The partial order in this case is given by the membership relation. The empty set in this lattice is 0 and {a, b, c, d} is 1.

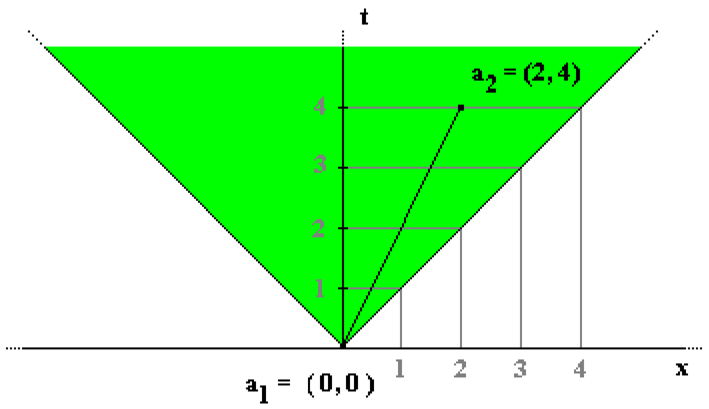


Figure G4 **Particle at x=0 and constrained to move in ± x direction: *future* perspective**

A particle currently at the origin and moving at a constant velocity of half the speed of light in the +x direction will be at x = 2 after 4 time units have passed, shown as the point a_2. Other allowable trajectories starting from the origin are shown in green. Only a massless particle such as a particle of light can follow the diagonal lines.

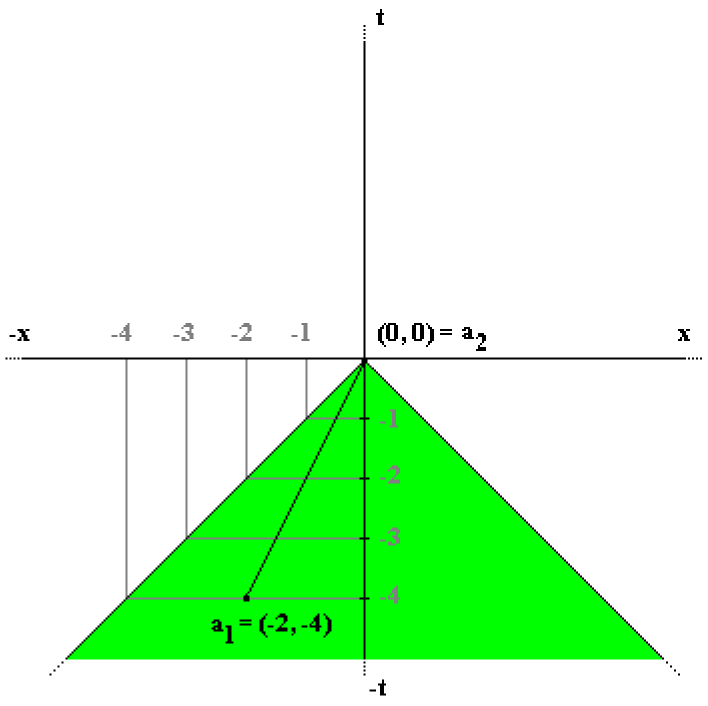


Figure G5 **Particle at x=0 and constrained to move in ± x direction: *past* perspective** The trajectory of a particle currently at the origin and moving constant velocity of half the speed of light in the +x direction was at x = –2 at time t = – 4 time, shown as the point a_1. Other starting points for allowable trajectories finishing at the origin at time t = 0 are shown in green.

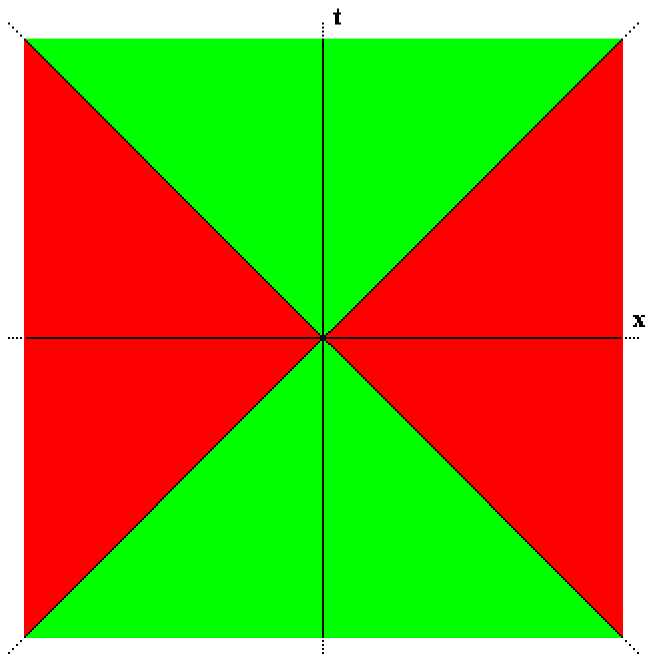


Figure G6 **Timelike and spacelike intervals from the orgin**
The worldline of a particle at the origin must lie in the green areas. Points in the green area are said to be separated from the origin by a *timelike* interval, whereas points in the red area are separated from the origin by a *spacelike* interval.

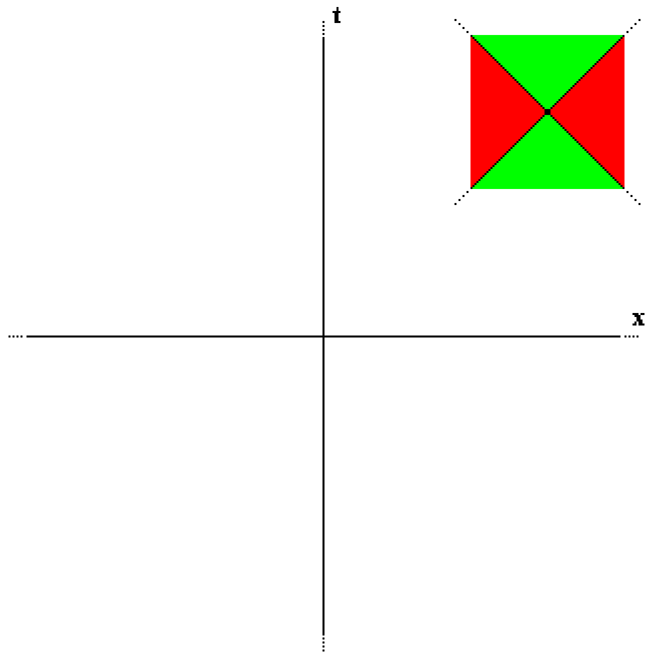


Figure G7 **Timelike and spacelike intervals as abstract properties of spacetime**
For a particle at any point on the above plot there is a set of points that are separated by a timelike interval and a set of points separated by a spacelike interval from that point.

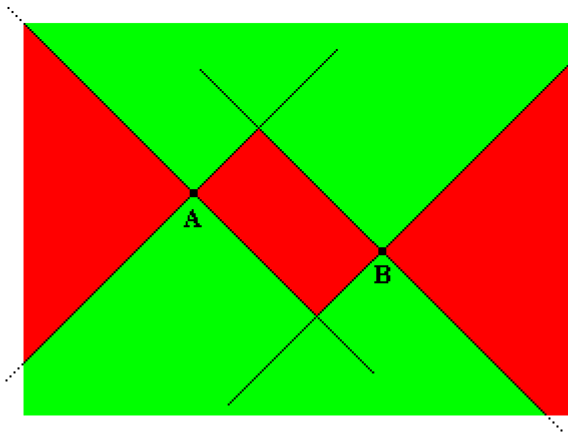


Figure G8

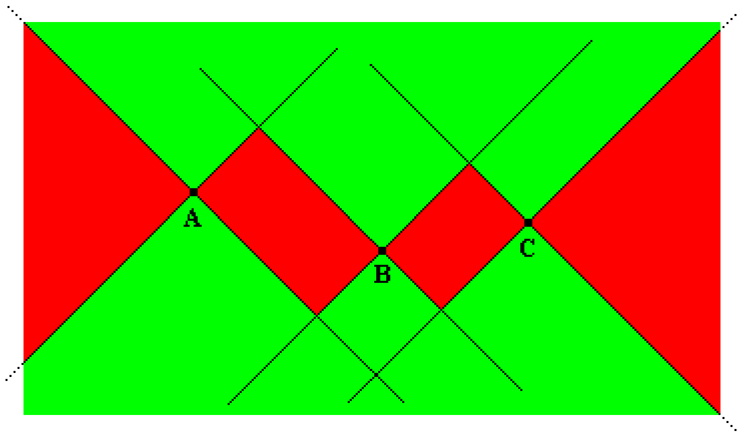


Figure G9

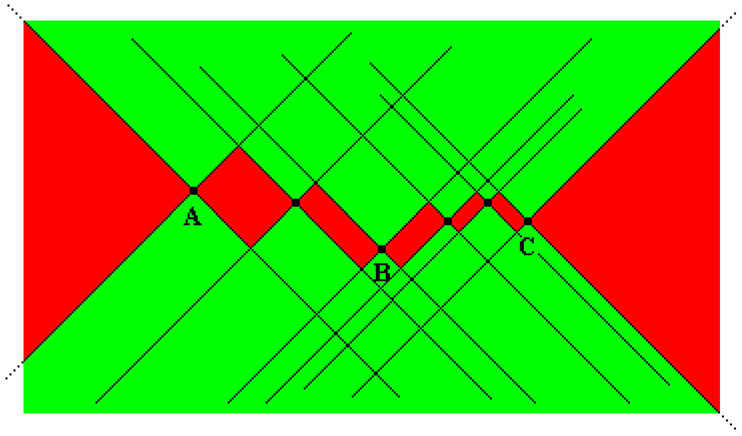


Figure G10

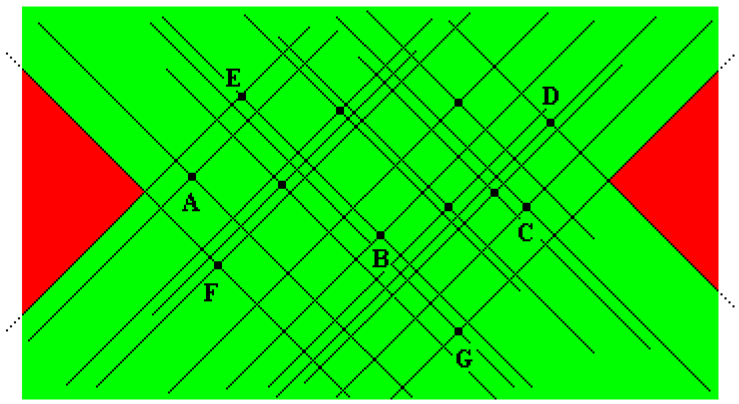


Figure G8-11 **The intersection of spacelike intervals from a collection of spacetime points**

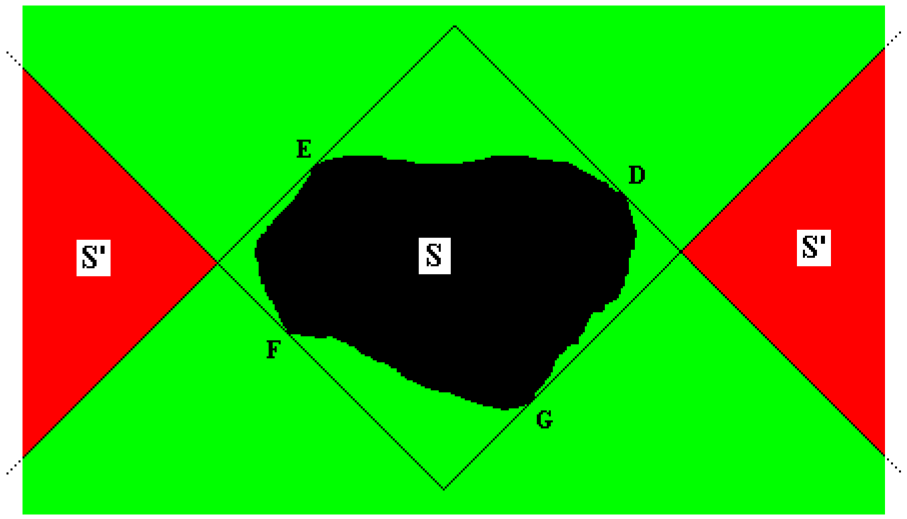


Figure G12 **S', the complement of S**
The complement of a set S is defined by the set of points spacelike separated from each point in S. It is determined by at least four points in S, one from each side of the rectangle bounding S whose sides are diagonal lines.

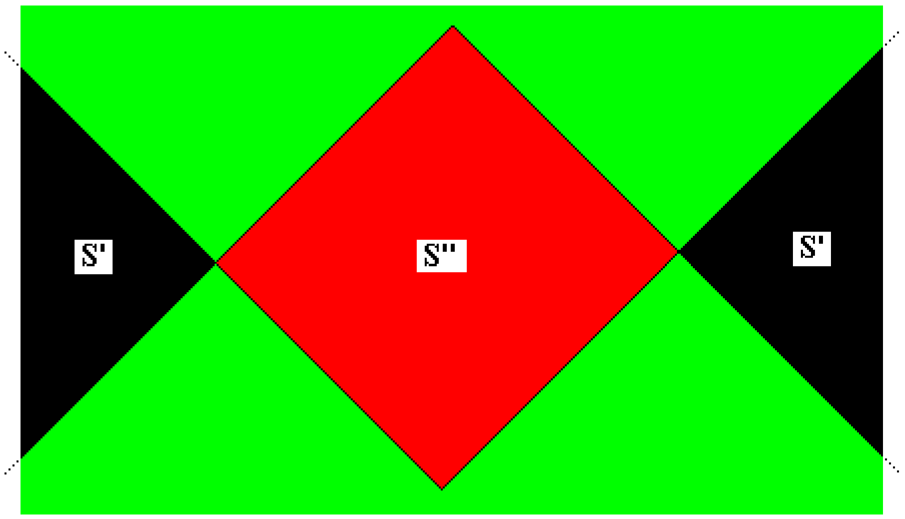


Figure G13 **S'', the complement of the complement of S**
By the same analysis, starting with points in A, B and so on in S', the complement of S' is found to be as shown above.

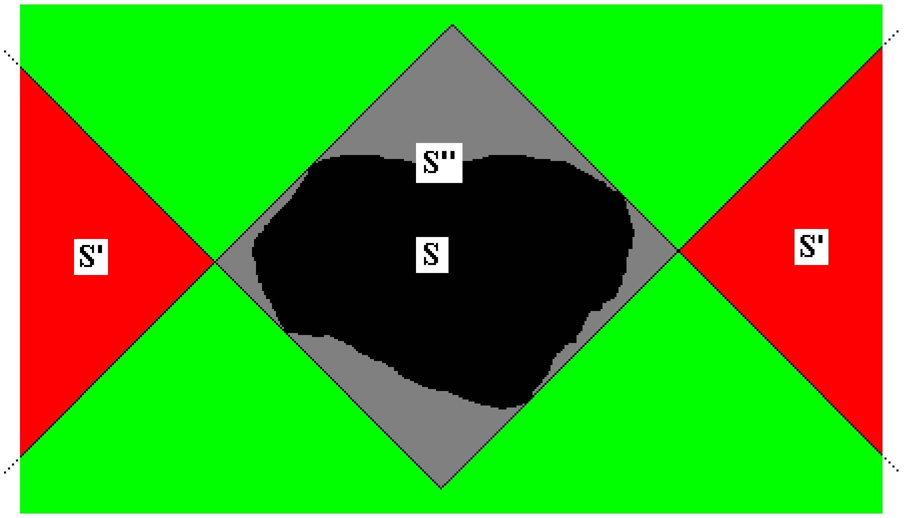


Figure G14 **The lattice point associated with the set S is S''**

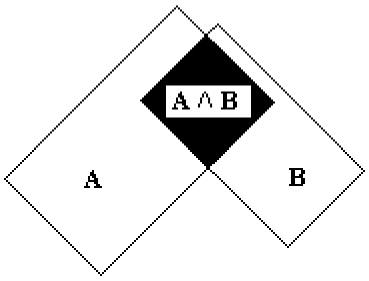


Figure G15 **The meet of lattice points A and B is the usual set intersection.**

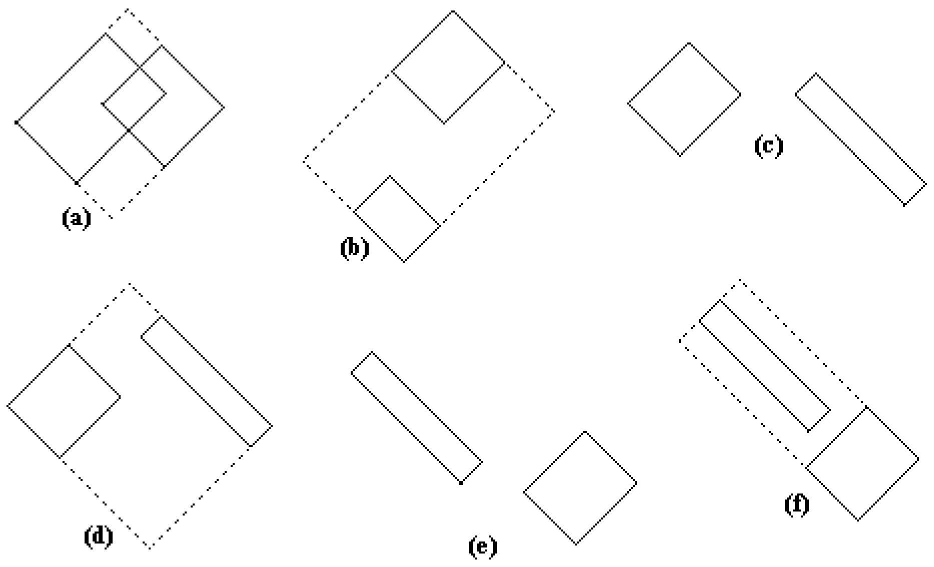


Figure G16 **The join of two lattice points**.

The join of two lattice points is determined by (A∨B) = (A∨B)'', as shown in figures G17 through G23. In (c) and (e) above, the join is ordinary set union. In the other cases, the join is ordinary set union augmented by the region indicated with broken lines.

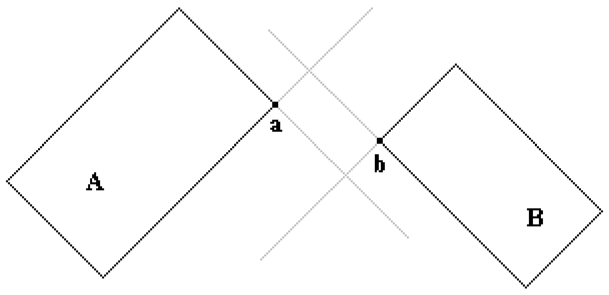


Figure G17 **The join A of and B is ordinary set union if and only if B is contained in A'.**

As figures G18 through G23 show, if the join of A and B is ordinary set union, then the configuration of A and B is as above, where **a** lies in the spacelike interval of **b**.

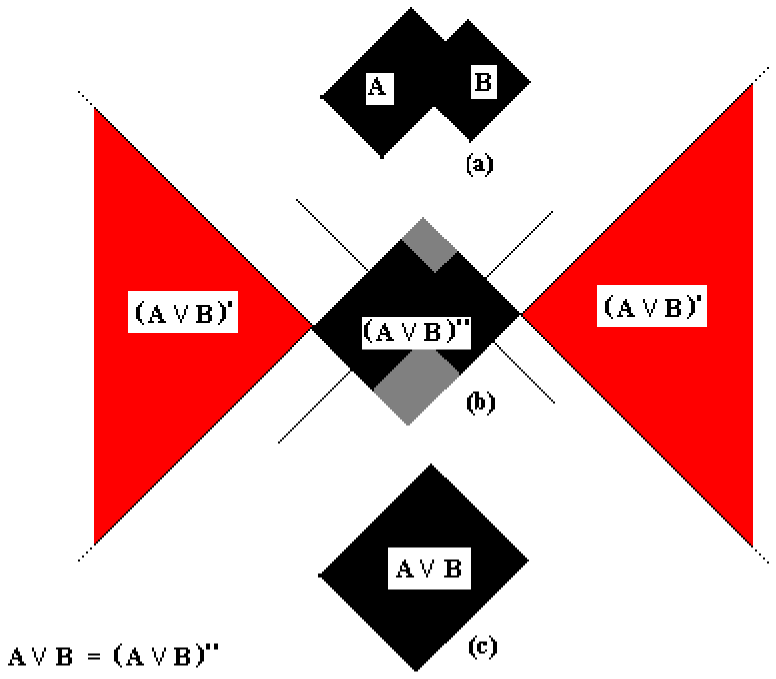


Figure G18

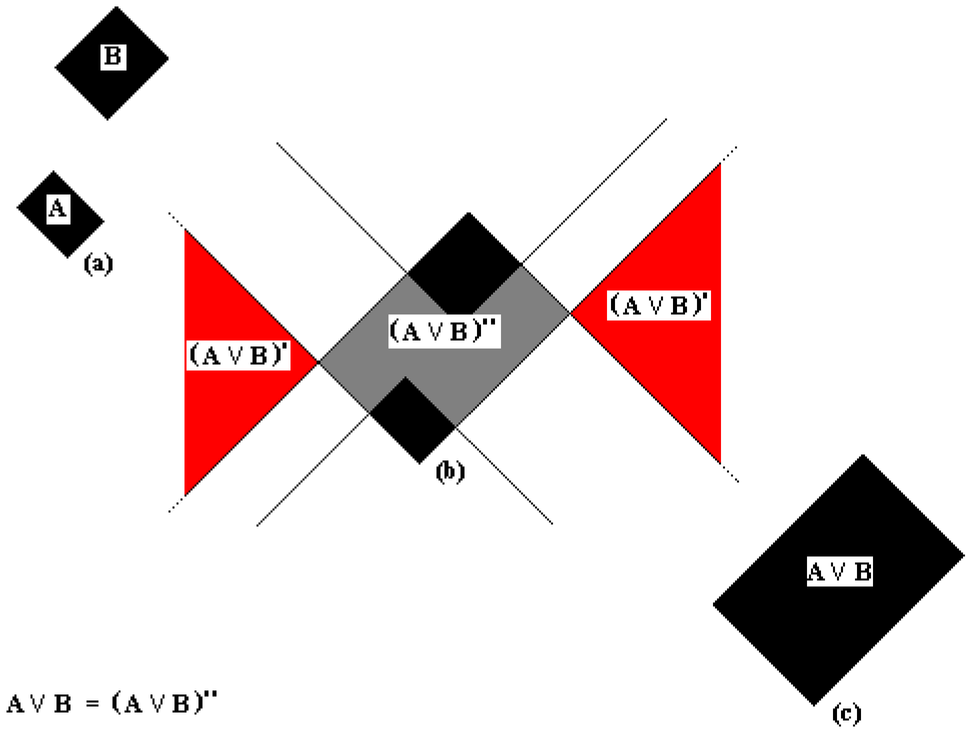
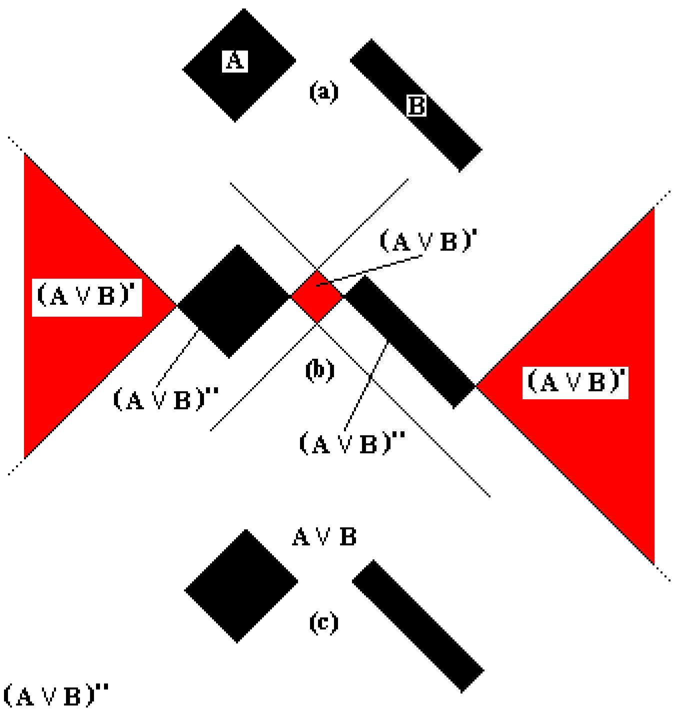
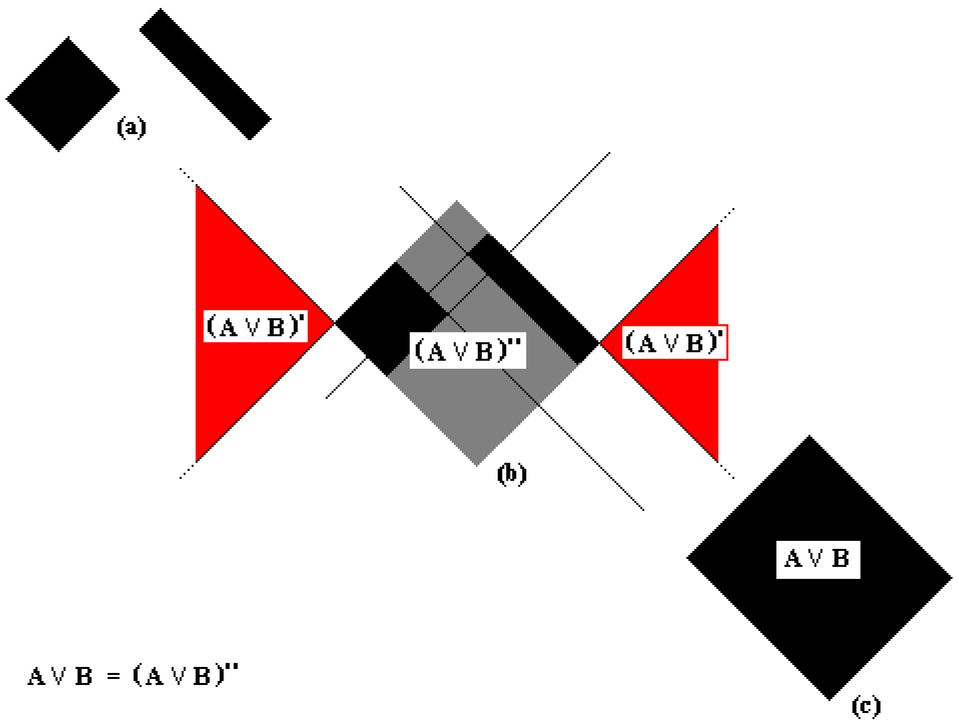


Figure G19



$A \vee B = (A \vee B)''$

Figure G20



$A \vee B = (A \vee B)''$

Figure G21

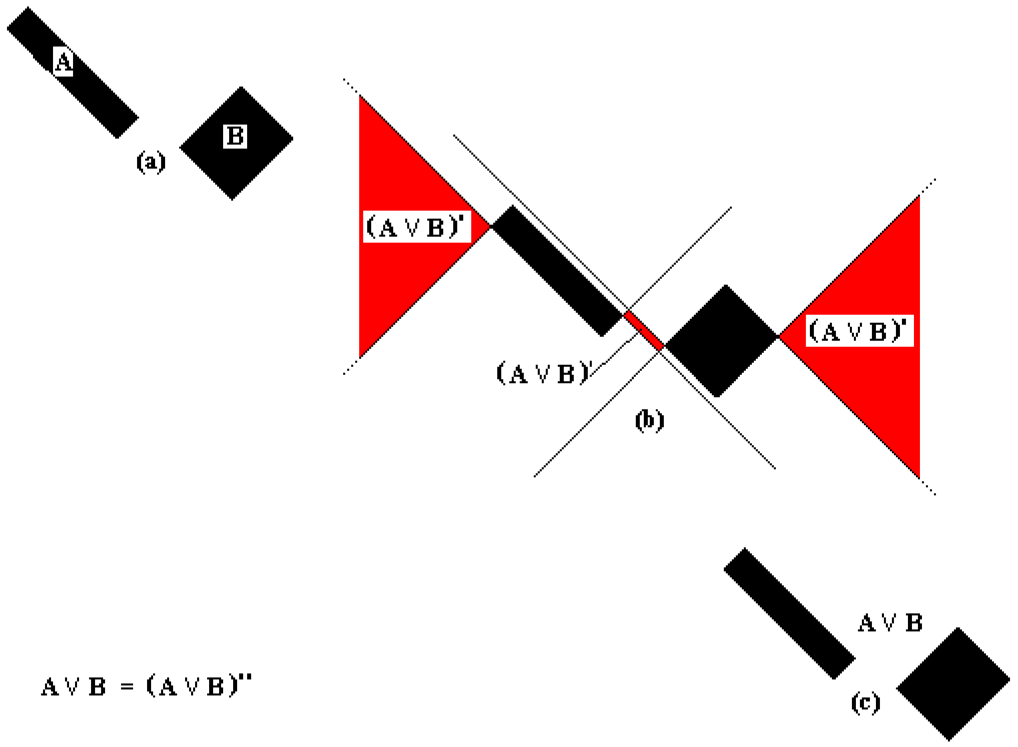


Figure G22

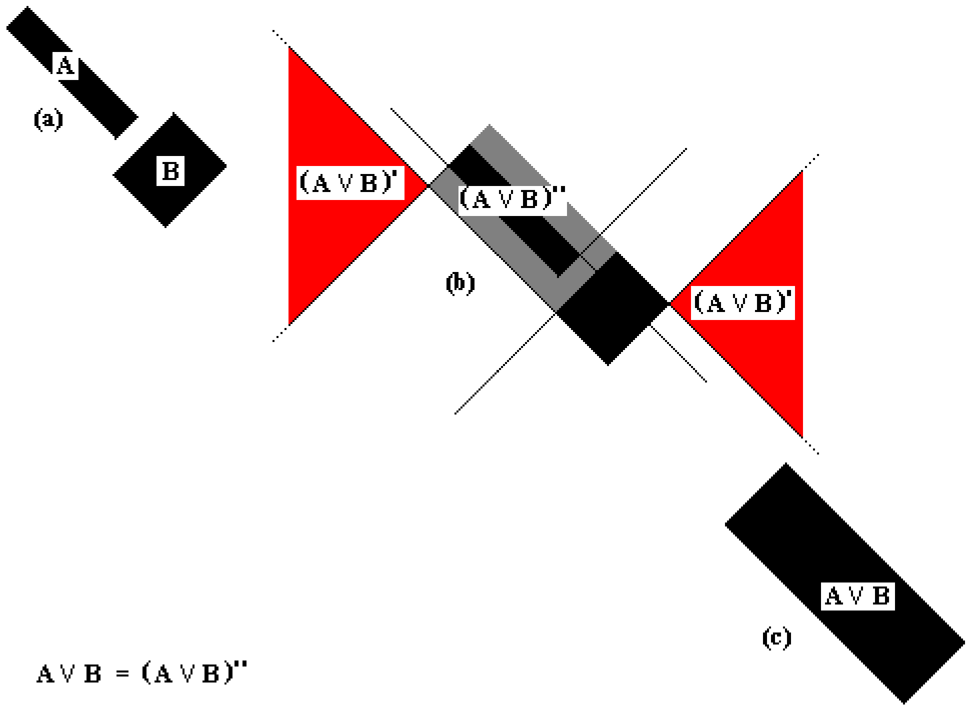


Figure G23

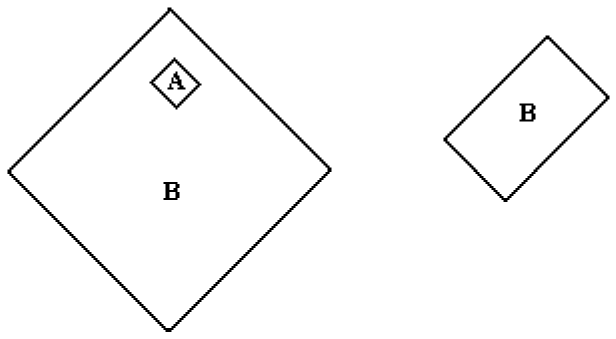


Figure G24 **The subset generated by the pair A≤ B is a boolean algebra**.

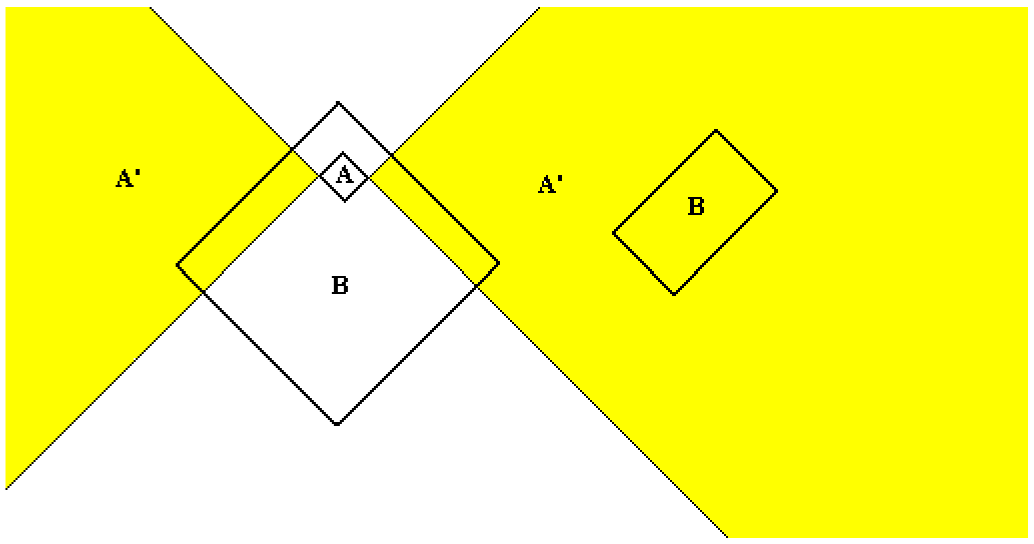
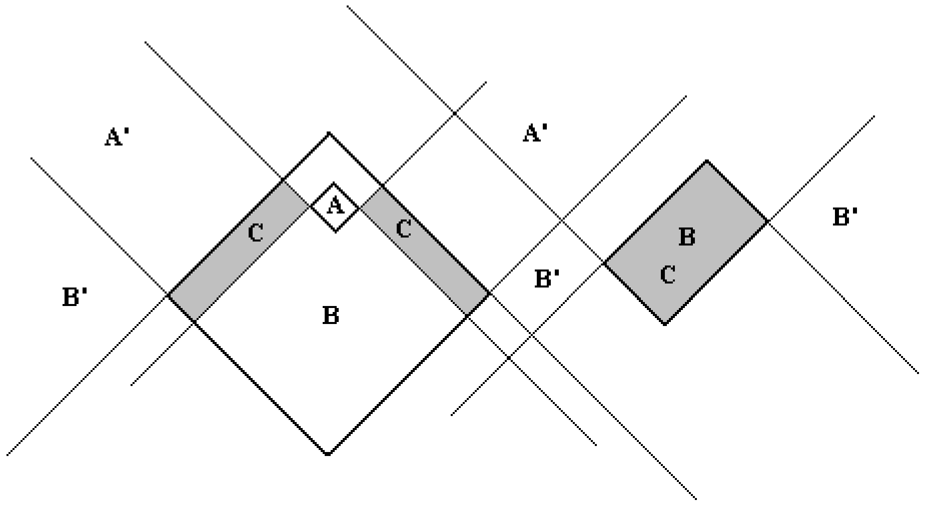


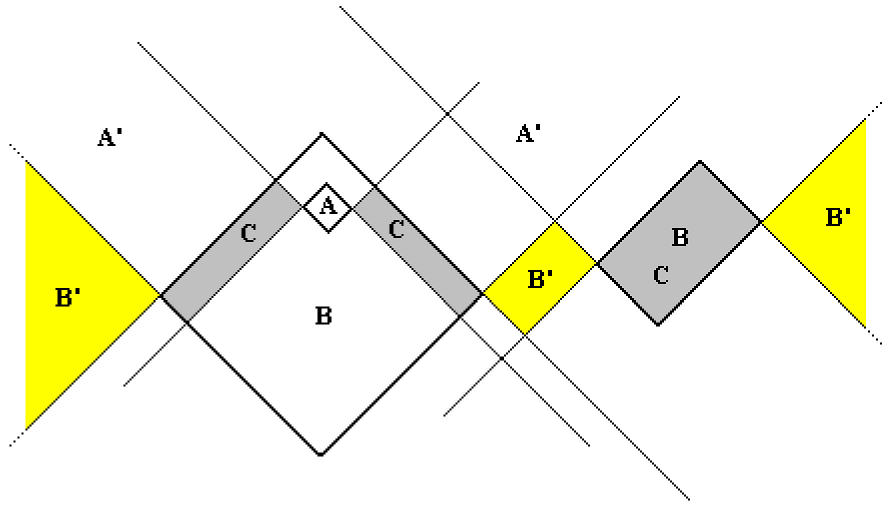
Figure G25



C = A'∧ B

Figure G26

From this plot it is clear that the orthomodularity condition A ≤ B → A ∨(A'∧B) = B holds, since A ∨(A'∧B) = A∨C = B.



B' = A'∧B'

C = A'∧ B

Figure G27

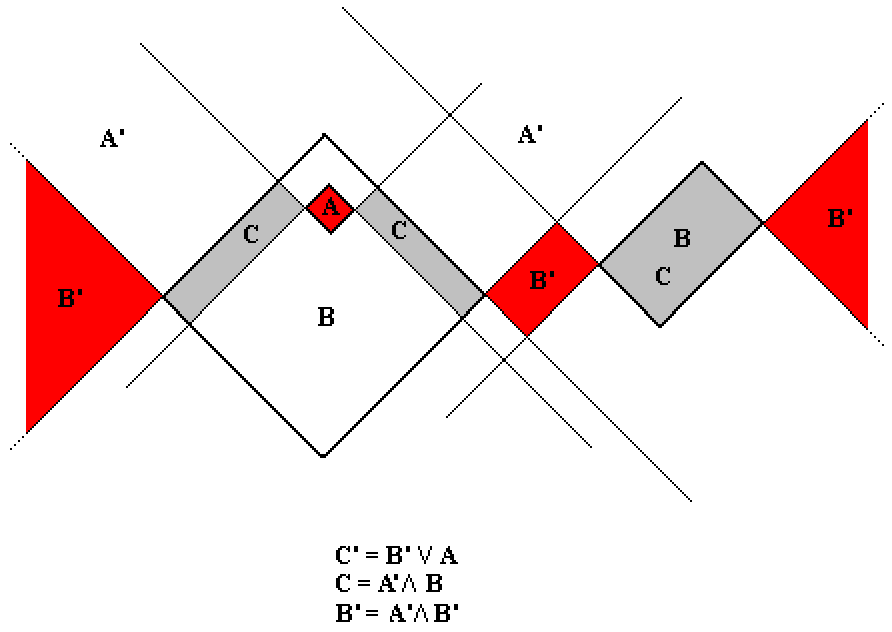


Figure G28

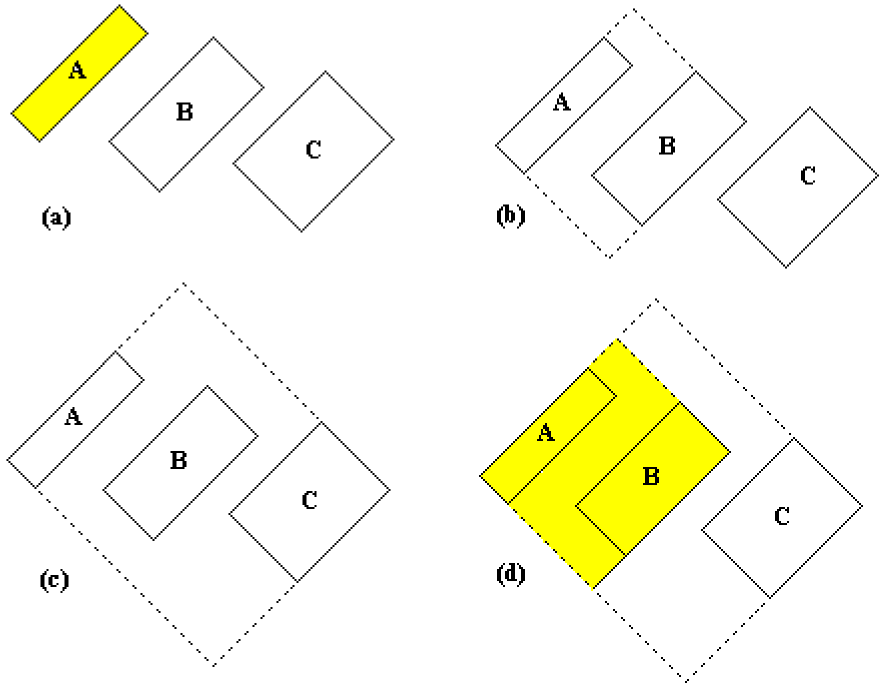


Figure G29 **The spacetime lattice is not distributive**.

From (a) above it is clear that $A \vee (B \wedge C) = A \vee 0 = A$. $A \vee B$ is given by the region indicated by the broken lines in (b). $A \vee C$ is given by the region indicated by the broken lines in (c). (A∨B)∧(A∨C) is shown in yellow in (e). Since A ≠ (A∨B)∧(A∨C) , it follows that A∨(B∧C) ≠ (A∨B)∧(A∨C).

Quick summary of recursive calls

Zeroth recursive call

God (input)	
Ix = x	objectify
KIx = **I**	abstract
SKIx = **K**x(**I**x) = x	apply

First recursive call (above function takes itself as input)

objectify	*(objectify ∘ objectify)*
abstract	*(objectify ∘ abstract)*
apply	*(objectify ∘ apply)*
re-objectify	*(abstract ∘ objectify)*
re-abstract	*(abstract ∘ abstract)*
re-apply	*(abstract ∘ apply)*
asymmetrize	*(apply ∘ objectify)*
transform	*(apply ∘ abstract)*,
reflect	(*apply ∘ apply*)

Second recursive call (repeats above function in a for-loop 8 times)

0	1	2	3	4	5	6	7
Order	Logic	Quantum Logic	Physical Universe	Life	Algorithm	Negation	Cyclic Time
1	Unique Null Set	R	1 Particle	1 Cell	1 Process	Singularity	Implication
!1 = 0	Pair	R*	2 Particles	2 Cells	2 Processes	Duality	AND
!!1 = 1	Union	C	Nucleus	Endosymbiosis	Integrated Parallel Procedure	Similarity	Equality
A<1	Infinity	Cn+1 = Cn X C	Atom	Neuron	Classify Procedure	Classification	Equivalence Class
A!<0	Replacement	L^2	Molecule	Linked Neurons	Compare Procedure	Comparison	Function
< = !!<	Power Set	Spin Network	Polymer	Neural Network	Sort Procedure	Sequence	Function Composition
A<B →B>A	Set Membership: x ∈ y → y !∈ x	Ordered Spin Network	Chiral Polymer	Sensory, Motor Neural Networks	Serial Input, Serial Output	Elapsed Time	Category
A<B, B<C→ A<C	x ∈ y, y ∈ z → x ∈ z	Spin Foam	Translation	Unidirectional Neural Network	Coordinated Serial I/O	Transitive Verb	Functor
A=A	x !∈ x	Spin Network Evolution	Replication	Bidirectional Neural Network	Symbol	Reflexive Verb	Natural Transformation

Table of Contents

Acknowledgement: I dedicate this work to my brother, James Gordon Kastner (1945-1970).

[1] Goodall, J., The Chimpanzees of Gombe: Patterns of Behavior, Belknap Press of Harvard University Press, Cambridge, (1986)

[2] Nishida, T., (Ed.), The chimpanzees of the Mahale Mountains, Univ. of Tokyo Press (1990)
[3] Boesch, C. & Boesch-Aschermann, H., Chimpanzees of the Tai Forest: Behavioral Ecology and Evolution, Oxford Univ. Press (2000)
[4] de Waal, F.B.M., & Lanting, F., Bonobo: The Forgotten Ape, UC Press, Berkeley (1998)
[5] Chomsky, N., New Horizons in the Study of Language and Mind, Cambridge Univ. Press, 2000
[6] Pinker, S. & Bloom, P., Natural language and natural selection, Behav & Brain Sci 13:707-784 (1990)
[7] Goodall, J., The Chimpanzees of Gombe: Patterns of Behavior, Belknap Press of Harvard University Press, Cambridge, (1986).
[8] Seyfarth, R. M., Cheney, D. L. & Marler, P., Vervet monkey alarm calls: semantic communication in a free-ranging primate, Anim. Behav. 28, 1070-1094 (1980)
[9] Martin, A., Organization of semantic knowledge and the origin of words in the brain. In Jablonski, N.G. & Aiello, L.C. (Eds), The origin and diversification of language, San Francisco: California Academy of Sciences (1998)
[10] Steels, L., Grounding symbols through evolutionary language games, in Parisi, D., & Cangelosi, A. (Eds.), Computational Approaches to the Evolution of Language and Communication, Springer Verlag, Berlin (2001)
[11] de Boer, B., Evolving sound systems, in Parisi, D., & Cangelosi, A. (Eds.), Computational Approaches to the Evolution of Language and Communication, Springer Verlag, Berlin (2001)
[12] Bickerton, D., Language and Human Behavior, Seattle, Univ. Wash. Press, (1995)
[13] Mayr, E. What Evolution Is, Basic Books, New York (2001)
[14] Gould, S., The Structure of Evolutionary Theory, Belknap Press of Harvard Univ. Press (2002)
[15] A. Whiten et al., Culture in Chimpanzees, Nature 399, pages 682-685 (1999)
[16] Goodall, J., The Chimpanzees of Gombe: Patterns of Behavior, Belknap Press of Harvard University Press, Cambridge, (1986)
[17] Dunbar, R., Correlation of neocortical size, group size and language in humans, Behav. & Brain Sci., 16:681-735 (1993)
[18] Seyfarth, R., Cheney, D., & Marler, P., Vervet monkey alarm calls: semantic communication in a free-ranging primate, Anim. Behav. 28, 1070-1094 (1980)
[19] Leavens, D.A. et al., Indexical and Referential Pointing in Chimpanzees (Pan troglodytes), J. Comp. Psy., 110, no.4, 346-353 (1996)
[20] Vea, J.J. Sabater-Pi, J., Spontaneous Pointing Behaviour in the Wild Pygmy Chimpanzee (Pan paniscus), Folia Primatol 69:289-290 (1998)
[21] Tomasello, M., Hare, B., & Agnetta, B., Chimpanzees, Pan troglodytes, follow gaze direction geometrically, 58, 769-777, (1999)
[22] C. Li, S. Thompson, Mandarin Chinese: A Functional Reference Grammar, U.C. Press, Berkeley (1981)
[23] Huffman, M.A. and R.W. Wrangham, Diversity of medicinal plant use by chimpanzees in the wild, in R.W. Wrangham, W.C. McGrew, F.B. deWall, P.G. Heltne Heltne (Eds.) Chimpanzee Cultures. Harvard Univ. Press, Mass. pp. 129-148 (1994)
[24] Calvin, W. H.. A stone's throw and its launch window: timing precision and its implications for language and hominid brains. J. of Theor. Bio., 104:121-135 (1983)
[25] Hamilton, A.G., Logic for Mathematicians, Cambridge Univ. Press, Cambridge (1991)
[26] Zentner, M. & Kagan, J., Perception of music by infants, Nature, 383, p. 29 (1996)
[27] Fraleigh, J., A First Course in Abstract Algebra, Addison-Wesley, Reading, MA (1994)
[28] Lawvere, F.W., & Schanuel, S.H., Conceptual Mathematics: A first introduction to category theory, Cambridge Univ. Press (1997)
[29] Eilenberg, S., & Mac Lane, S., A general theory of natural equivalences, Trans. Am. Math. Soc., 58, 231-294 (1945)
[30] Mac Lane, S., Categories for the Working Mathematician, Springer-Verlag, Second Edition, 1997
[31] Lawvere, F.W., & Schanuel, S.H., Conceptual Mathematics: A first introduction to category theory, Cambridge Univ. Press (1997)
[32] Goldblatt, T., Topoi: The Categorial Analysis of Logic, North-Holland, Amsterdam (1979)
[33] Mac Lane, S., Categories for the Working Mathematician, Springer-Verlag, Second Edition, 1997
[34] Baez, J., Higher dimensional algebra and Planck scale physics, in Callender, C., & Huggett, N., (Eds.), Physics Meets Philosophy at the Planck Scale: Comtemporary Theories in Quantum Gravity, Cambridge, Cambridge Univ. Press (2001)

[35] Atiyah, M., Quantum theory and Geometry, J. Math. Phys. 36
[36] Mac Lane, S., Categories for the Working Mathematician, Springer-Verlag, Second Edition, 1997
[37] Kelly, G. & Street, R., Review of the elements of 2-categories, Springer Lecture Notes in Mathematics, vol. 420, Berlin, Springer (1974)
[38] Zhang, J., G. Harbottle, et al., Oldest playable musical instruments found at Jiahu early neolithic site in China, Nature 401:366 (1999)
[39] Seeberger, F., Sind jungpalaeolithische Knochenfloeten Vorlaeufer mediterraner Hirtenfloeten?, Archaeologisches Kommentarblatt, Band 29, Heft 2, 155-157 (1999)
[40] Hahn, J. & Muenzel, S., Knochenfloeten aus dem Aurignacien des Geissenkloesterle bei Blaubeuren, Alb-Donau-Kreis, Fundberichte aus Baden-Wuertemberg, Band 20, 1-12 (1995)
[41] Cross, I., Music, mind and evolution, Psychology of Music, 29(1), pp.95-102, (2001)
[42] Marshack, A., The Roots of Civilization, Mount Kisco, New York, Moyer Bell (1991)
[43] Henshilwood, C., et al., Emergence of Modern Human Behavior: Middle Stone Age Engravings from South Africa, Science 295: 1278-1280 (2002)
[44] Mellars, P., Neanderthals, modern humans, and the archaeological evidence for language, in Jablonski, N. & Aiello, L., (Eds.), The Origin and Diversification of Language, Memoirs of the Calif. Acad. of Sci., no. 24 (1998)
[45] Baumler, M., Principles and properties of lithic core reduction: implications for Levallois technology, in Dibble, H. & Bar-Yosef, O., (Eds.), The Definition and Interpretation of Levallois Technology, Madison, Prehistory Press (1995)
[46] Rolland, N., Levallois technique emergence: single or multiple? A review of the Euro-African record, , in Dibble, H. & Bar-Yosef, O., (Eds.), The Definition and Interpretation of Levallois Technology, Madison, Prehistory Press (1995)
[47]Korisettar, R. & Petraglia, D., The archaeology of the Lower Paleolithic: background and overview, in Petraglia, D., & Korisettar, R. (Eds.), Early Human Behaviour in Global Context: The Rise and Diversity of the Lower Paleolithic Record, London, Routledge (1998)
[48] Deacon, T., Brain-Language Coevolution, in Hawkins, J. & Gell-Mann, M., (Eds.), SFI Studies in the Sciences of Complexity, Proc. Vol X, Addison-Wesley (1992)
[49] Lieberman, P., On the Evolution of Human Language, in Hawkins, J. & Gell-Mann, M., (Eds.), SFI Studies in the Sciences of Complexity, Proc. Vol X, Addison-Wesley (1992)
[50] Mellars, P., Neanderthals, modern humans, and the archaeological evidence for language, in Jablonski, N. & Aiello, L., (Eds.), The Origin and Diversification of Language, Memoirs of the Calif. Acad. of Sci., no. 24 (1998)
[51] Henshilwood, C., et al., Emergence of Modern Human Behavior: Middle Stone Age Engravings from South Africa, Science 295: 1278-1280 (2002)
[52] Mellars, P., Major issues in the emergence of modern humans, Curr. Anthropol., 30:349-385 (1989)
[53] Harpending, H., Batzer, M., Gurven, M., Jorde, L., Rogers, A., & Sherry, S., Genetic traces of ancient demography, Proc. Nat. Acad. Sci., USA, Vol. 95, 4, 1961-1967(1998)
[54] Wall, J., & Przeworski, M., When did the human population size start increasing? Genetics, Vol. 155, 1865-1874 (2000)
[55] Relethford, J., Genetics and the Search for Modern Human Origins, New York, Wiley-Liss (2001)
[56] Relethford, J., Genetics and the Search for Modern Human Origins, New York, Wiley-Liss (2001)
[57] Ruhlen, M., On the Origin of Language: Studies in Linguistic Taxonomy, Stanford, (1994)
[58] Cavalli-Sforza, L.L., Genes, peoples and languages, North Point Press, New York (2000)
[59] Hurford, J. & Kirby, S., The Emergence of Linguistic Structure: an Overview of the Iterated Learning Model, In Parisi, D., & Cangelosi, A. (Eds.), Computational Approaches to the Evolution of Language and Communication, Springer Verlag, Berlin (2001)
[60] Batali, J., Computational Simulations of the Emergence of Grammar, in Hurford, J., Studdert-Kennedy, M. and Knight, C. (Eds.), Approaches to the Evolution of Language: Social and Cognitive Bases, Cambridge University Press, (1998)
[61] Hutchins, E. & Hazlewurst, B., Auto -organization and emergence of shared language structure, In Parisi, D., & Cangelosi, A. (Eds.), Computational Approaches to the Evolution of Language and Communication, Springer Verlag, Berlin (2001)
[62] Steels, L., Self-organizing vocabularies, in Langton, C. & Shimohara, K., (Eds.), Proceedings of Alife V, MIT Press, Cambridge (1996)

[63] Pepperberg, I.M., The Alex Studies: Cognitive and Communicative Abilities of Grey Parrots, Cambridge, Harvard Univ. Press (1999)
[64] Herman, L.M. & Uyeyama, R.U., The dolphin's grammatical competency: comments on Kako, Anim. Learn. Behav., 27(1): 18-23 (1999)
[65] Gardner, B.T, & Gardner, R.A., Cross-fostered chimpanzees: I. Testing vocabulary, II. Modulation of Meaning, in Heltne, P. & Marquardt, L. (Eds.), Understanding Chimpanzees, Harvard Univ. Press (1989)
[66] Patterson, F. & Linden, E., The Education of Koko, Holt, Rinehart, & Winston (1981)
[67] Miles, L., The cognitive foundations for reference in a signing orangutan. In Parker, S.T. & Gibson, K.R., (Eds.), 'Language' and intelligence in monkeys and apes, Cambridge, Cambridge Univ. Press (1990)
[68] Savage-Rumbaugh, S. & Lewin, R., Kanzi: the ape at the brink of the human mind, New York: J. Wiley (1994)
[69] Matsuzawa, T., Use of numbers by a chimpanzee, Nature, 315, 57-59 (1985)
[70] Gallup, G.G. Jr., Chimpanzees: self-recognition, Science, 167, 86-87 (1970)
[71] Povinelli, D.J. et al., Self-Recognition in Chimpanzees (Pan troglodytes): Distribution, Ontogeny, and Patterns of Emergence, J. Comp. Psych., vol. 107, no. 4, 437-372, (1993)
[72] Weinberg, S., The First Three Minutes, Basic Books, New York (1993)
[73] Dyson, F., Origins of Life, Cambridge, Cambridge Univ. Press (1999)
[74] Davies, P., The 5th Miracle: The Search for the Origin and Meaning of Life, Simon & Schuster, New York (2000)
[75] Cairns-Smith, A.G., Seven Clues to the Origin of Life, Cambridge, Cambridge Univ. Press (1985)
[76] Hazen, R. M., Filley, T. R. & Goodfriend, G. A. Selective adsorption of L- and D-amino acids on calcite: Implications for biochemical homochirality. Proc. Nat. Acad. Sci. USA, 98, 5487-5490 (2001).
[77] Kari, L., DNA computing: arrival of biological mathematics, Math. Intelligencer, 19:2, 9-22 (1997)
[78] Merezhkovsky, K.S., Theory of Two Plasms as the Basis of Symbiogenesis: A New Study on the Origin of Organisms, in Russian. Kazan: Publishing office of the Imperial Kazan University (1909) (as cited in Dyson, F., op.cit.)
[79] Margulis, L. & Sagan, D., What is Life?, New York, Simon & Schuster (1995)
[80] Simmons, P., & Young, D., Nerve Cells and Animal Behaviour, Cambridge, Cambridge Univ. Press (1999)
[81] Grillner, S., Neural networks for vertebrate locomotion, Sci. Amer., p. 64-69, 1/1996
[82] Simmons, P., & Young, D., Nerve Cells and Animal Behaviour, Cambridge, Cambridge Univ. Press (1999)
[83] O'Reilly, R.C., & Munakata, Y., Computational Explorations in Cognitive Neuroscience, Cambridge, MIT Press (2000)
[84] Koechlin, E., Basso, G., Pietrini, P., Panzer, S., Grafman, J. Exploring the role of the anterior prefrontal cortex in human cognition, Nature, 399:148-151 (1999)
[85] Seyfarth, R., Cheney, D., & Marler, P., op. cit.
[86] Diamond, A.H., & McKinsey, J.C., Algebras and their subalgebras, Bull. Amer. Math. Soc. 53, 959-962 (1947)
[87] Hrbacek, K. & Jech, T., Introduction to Set Theory, New York, Marcel Dekker (1999)
[88] Huntington, E.V., Sets of independent postulates for the algebra of logic, Trans. Amer. Math. Soc., 5:288-309 (1904)
[89] Birkhoff, G. & MacLane, S., A Survey of Modern Algebra, New York, MacMillan (1948)
[90] Whitesitt, J.E., Boolean Algebra and its Applications, New York, Dover (1995)
[91] Hrbacek, K. & Jech, T., Introduction to Set Theory, New York, Marcel Dekker (1999)
[92] Roitman, J., Introduction to Modern Set Theory, Wiley-Interscience, New York (1990)
[93] Drake, F., & Singh, D., Intermediate Set Theory, John Wiley, Chichester (1996)
[94] Svozil, K., Quantum Logic, Springer Verlag, Berlin (1998)
[95] Holland, S., Orthomodularity in infinite dimensions; a theorem of M. Soler, Bull. Amer. Math. Soc., 32:2, 205-234 (1995)
[96] Aerts, D. & Van Steirteghem, B., Quantum axiomatics and a theorem of M.P. Soler, Int. J. Theor. Phys., 39, 497-502 (2000)
[97] Foulis, D., A half-century of quantum logic. What have we learned?, in Aerts, D. & Pykacz, J. (Eds.), Quantum Structures and the Nature of Reality, Kluwer Academic Publishers & VUB Univ. Press, Brussels (1999)

[98] Ptak, P., & Pulmannova, S., Orthomodular Structures as Quantum Logics, Kluwer, Dortrecht (1991)
[99] Cegla, W., & Jadczyk, A., Orthomodularity of causal logics, Comm. Math. Phys., 57, 213 (1977)
[100] Casini, H., The quantum logic of causal sets, astro/ph/0205013 (2002)
[101] Hrbacek, K. & Jech, T., Introduction to Set Theory, New York, Marcel Dekker (1999)
[102] Bartle, R., The Elements of Real Analysis, 2nd Edition, Wiley, New York (1976)
[103] Jech, T., The Axiom of Choice, N. Holland, Amsterdam (1973)
[104] Diaconescu, R., Axiom of choice and complementation, Proc. Am. Math. Soc., 51, 176-178 (1975)
[105] Montgomery, H., The pair correlation of zeros of the zeta function, in H. Diamond, (Ed.), Analytic Number Theory, AMS, Providence (1973)
[106] Odlyzko, A., On the distribution of spacing between zeros of zeta functions, Math. Comp. 48, 273-308 (1987)
[107] Connes, A., Trace formula in noncommutative geometry and the zeros of the Riemann function, Journees "Equations aux derivees partielles", exp. no. IV, 28, Ecole Polytech., Palaiseau (1997)
[108] Rovelli, C., Loop quantum gravity, Living Rev. Relativity, 1, http://www.livingreviews.org/articles/volume1/1998-1rovelli/ (1998)
[109] Baez, J., Talk at UCSB, online.kitp.ucsb.edu/online/gravity_c99/baez (1999)
[110] Binnig, G., The fractal structure of evolution, Physica D, 38:32-36 (1989)
[111] Wolfram, S., A New Kind of Science, Champaign, Wolfgram Media (2002)
[112] Lawvere, F.W., The category of categories as a foundation for mathematics, in Proceedings of La Jolla Conference on Categorical Logic, Springer, New York (1966)
[113] Marquis, J.-P., Category Theory and the Foundations of Mathematics: Philosophical Excavations, Synthese, 103, 421-447 (1995)
[114] Feferman, S., Categorical foundations and foundations of category theory, in Butts, R., (Ed.), Logic, Foundations of Mathematics and Computability, D. Reidel (1977)
[115] Stoltenberg-Hansen, V., Lindstroem, I., & Griffor, E., Mathemathical Theory of Domains, Cambridge Univ. Press, Cambridge (1994)
[116] Aczel, P., On relating type theories and set theories, in Altenkirch, Naraschewski, & Reus, (Eds.), Proceedings of TYPES '98, Springer-Verlag, Berlin (1999)
[117] Werner, B., Sets in types, types in sets, in Abadi, M., & Takahashi, I., (Eds.), TACS '97, LCNS vol. 1281, Springer-Verlag, Berlin (1997)
[118] Lambek, J., From l-calculus to Cartesian closed categories, in Seldin, J., & Hindley, J., (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, Academic Press, London (1980)
[119] Kanamori, A., The Higher Infinite, Springer-Verlag, Berlin (1994)
[120] Woodin, W. H., Large cardinal axioms and independence: the continuum problem revisited, Math. Intelligencer, 16(3), 31-35 (1994)
[121] Aczel, P., On relating type theories and set theories, in Altenkirch, Naraschewski, & Reus, (Eds.), Proceedings of TYPES '98, Springer-Verlag, Berlin (1999)
[122] Dybjer, P., & Setzer, A., A finite axiomatization of inductive-recursive definitions, in Girard, J.-Y., (Ed.), Proceedings of the TLCA 1999, LNCS 1581, Springer, Berlin (1999)
[123] Etchemendy, J., Tarski on truth and logical consequence, Jour. Symb. Logic, 53(1), 51-79 (1998)
[124] Kant, I., Prolegomena zu einer jeden kuenftigen Metaphysik, die als Wissenschaft wird auftreten koennen, Meiner, Hamburg (2001)
[125] Smolin, L., The Life of the Cosmos, Oxford Univ. Press, Oxford (1999)
[126] Svozil, K., Quantum Logic, Springer Verlag, Berlin (1998)
[127] Fredkin, E., Digital mechanics, Physica D 45:254-270 (1990)
[128] Kreisel, G., Mathematical logic, in Saaty, E. (Ed.), Lectures in Modern Mathematics, Wiley & Sons, New York (1965)
[129] Lloyd, S., Computational capacity of the universe, Phys. Rev. Letters , 88, 237901 (2002)
[130] Kfoury, A.J., Moll, R.N, & Arbib, M.A., A Programming Approach to Computability, Springer-Verlag, New York (1982)
[131]Kfoury, A.J., Moll, R.N, & Arbib, M.A., A Programming Approach to Computability, Springer-Verlag, New York (1982)
[132] Hankin, C., Lambda Calculus: A guide for computer scientists, Clarendon Press, Oxford (1994)
[133] Rozenberg, G., & Salomaa, A., Handbook of Formal Languages, Vol. I-III, Springer-Verlag, Berlin (1997)

[134] Calude, C., & Paun, G., Computing with Cells and Atoms: An introduction to quantum, DNA, and membrane computing, Taylor & Francis, London (2001)
[135] Dijkstra, E., & Scholten, C., Predicate Calculus and Program Semantics, Springer-Verlag, Berlin (1990)
[136] Eilenberg, S., & Wright, J., Automata in general algebras, Information and Control, 11:452-470 (1967)
[137] Pin, J., Varieties of Formal Languages, Plenum, New York (1986)
[138] Falconer, K., Fractal Geometry: Mathematical Foundations and Applications, Wiley, Chichester (1990)
[139] Gouyet, J.-F., Physics and Fractal Structures, Springer –Verlag, Berlin (1996)
[140] Zee, A., Fearful Symmetry: The Search for Beauty in Modern Physics, MacMillan, New York (1986)
[141] Wheeler, J., "Genesis and observership", in Butts, R. & Hintikka, J., eds., Foundational Problems in the Special Sciences, Dordrecht, D. Reidel (1977)
[142] Brown, H., Precise measurement of the positive muon anomalous magnetic moment, Phys. Rev. Lett. 86, 2227 (2001)
[143] Witten, E., "Duality, Spacetime and Quantum Mechanics," Physics Today, May 1997, pp.28-33
[144] Greene, B., op. cit.
[145] Rovelli, C., Loop quantum gravity, Living Rev. Relativity, 1, http://www.livingreviews.org/articles/volume1/1998-1rovelli/ (1998)
[146] Baez, J., & Dolan, J., Higher dimensional algebra and topological quantum field theory, J. Math. Phys. 36, 60-105 (1995)
[147] Smolin, L., The future of spin networks, in Hugget S., Tod, P., Mason, L.J., The Geometric Universe: Science, Geometry and the Work of Roger Penrose, Oxford Univ. Press, Oxford (1998)
[148] Smolin, L., Eleven dimensional supergravity as a constrained topological field theory, Nucl. Phys. B 601, 191-208 (2001)
[149] Steinhardt, P. & Turok, N., The cyclic universe: an informal introduction, astro/ph/0204479 (2002)
[150] Adler, S., Why decoherence has not solved the measurement problem: a response to P.W. Anderson, quant-ph 0112095 (2002)
[151] Kreimer, D., Knots and Feynman Diagrams, Cambridge Univ. Press, Cambridge (2000)
[152] Hamkins, J.D. & Lewis, A., Infinite time Turing machines, Jour. Sym. Logic, 65(2), 567-604 (2000)
[153] Hoelldobler, B. & Wilson, E., Journey to the Ants, Harvard University Press, Cambridge, MA (1994)
[154] Freudenthal, H., Lincos: Design of a Language for Cosmic Intercourse, Amsterdam, North-Holland, (1960)
[155] DeVito, C., & Oehrle, R., A language based on the fundamental facts of science, Journal of British Interplanetary Society, vol. 43, pp.561-568 (1990)
[156] Turing, A., Computing machinery and intelligence, Mind, 59:433-460 (1950)
[157] Brooks, R., Flesh and Machines: How Robots Will Change Us, New York, Pantheon Books (2002)
[158] Gershenfeld, N., When Things Start to Think, New York, Henry Holt & Co. (1999)
[159] Moravec, H., Robot: Mere Machine to Transcendent Mind, Oxford, Oxford University Press (2000)
[160] Kurzweil, R., The Age of Spiritual Machines: When Computers Exceed Human Intelligence, New York, Penguin Books (1999)
[161] Isenberg, N., et al., Linguistic threat activates the human amygdala, Proc. Natl. Acad. Sci., USA, 96:10456-10459 (1999)
[162] Hrbacek, K. & Jech, T., Introduction to Set Theory, New York, Marcel Dekker (1999)
[163] Michaelson, G., An Introduction to Functional Programming through Lambda Calculus, Addison-Wesley, Wokingham (1989)
[164] The DRAGN Project, Oberlin College, www.cs.oberlin.edu/classes/dragn/labroot.html 1996
[165] Barendregt, H., The Lambda Calculus: Its Syntax and Semantics, North Holland, Amsterdam (1981)
[166] Hankin, C., Lambda Calculus: A guide for computer scientists, Clarendon Press, Oxford (1994)
[167] Curry, H. & Feys, R., Combinatory Logic, volume I, North Holland, Amsterdam (1958)
[168] Odifreddi, P., Classical Recursion Theory, North Holland, Amsterdam (1989)
[169] Hindley, J.R., & Seldin, J., Introduction to Combinators and λ-calculus, Cambridge Univ. Press, Cambridge (1986)
[170] Curry, H. & Feys, R., Combinatory Logic, volume I, North Holland, Amsterdam (1958)
[171] Chang, C. & Keisler, H.J., *Model Theory*, North Holland, Amsterdam (1990)
[172] Kalmbach, G., Orthomodular Lattices, Academic Press, London (1983)
[173] Kalmbach, G., Orthomodular Lattices, Academic Press, London (1983)
[174] Casini, H., The quantum logic of causal sets, astro/ph/0205013 (2002)

[175] Cegla, W. & Jadczyk, A.Z., Causal Logic of Minkowski Space, Commun. math. Phys. 57, 213-217 (1977)
[176] Moll, R., Arbib, M., & Kfoury, A., An Introduction to Formal Language Theory, Springer-Verlag, New York (1988)
[177] Parkes, A., Introduction to Languages, Machines, and Logic: Computable Languages, Abstract Machines, and Formal Logic, Springer, London (2002)
[178] Linz, P., An Introduction to Formal Languages and Automata, Jones & Bartlett, Sudbury (1997)
[179] Odifreddi, P., Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers, North Holland, Amsterdam (1989)
[180] Taylor, R. G., Models of Computation and Formal Languages, Oxford Univ. Press, Oxford (1998)
[181] Kain, R. Y., Automata Theory: Machines and Languages, McGraw-Hill, New York (1972)
[182] Taylor, R. G., Models of Computation and Formal Languages, Oxford Univ. Press, Oxford (1998)
[183] Davis, M., Sigal, R., Weyuker, E., Computability, Complexity, and Languages: Fundamentals of Computer Science, Academic Press, Boston (1994)
[184] Odifreddi, P., Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers, North Holland, Amsterdam (1989)
[185]Hankin, C., Lambda Calculus: A guide for computer scientists, Clarendon Press, Oxford (1994)
[186] Kreisel, G., & Tait, W.W., Finite definability of number-theoretical functions and parametric completeness of equation calculi, Zeit. Math. Log. Grund. Math. 7, 26-28 (1961)
[187] Davis, M., Sigal, R., Weyuker, E., Computability, Complexity, and Languages: Fundamentals of Computer Science, Academic Press, Boston (1994)
[188] Boolos, G., Burgess, J., & Jeffrey, R., Computability and Logic, Fourth Edition, Cambridge Univ. Press, Cambridge (2002)
[189] Young, N., An Introduction to Hilbert Space, Cambridge Univ. Press, Cambridge (1988)
[190] Young, N., An Introduction to Hilbert Space, Cambridge Univ. Press, Cambridge (1988)
[191] de Barra, G. Measure Theory and Integration, (1981)
[192] Isham, C., Lectures on Quantum Theory: Mathematical and Structural Foundations, Imperial College Press, London (1995)
[193] Berberian, S., Introduction to Hilbert Space, Chelsea Publishing Co., New York (1976)