Minimal Substrates for Intelligence: NAND, Universality, and the Lowest-Level Languages for AI

Douglas C. Youvan

doug@youvan.com

www.youvan.ai

November 30, 2025

The modern practice of artificial intelligence usually lives in high-level languages, massive frameworks, and sprawling hardware stacks. Yet, in principle, nothing more is required than a universal computing substrate and enough time and memory. This paper explores a simple but deep question: what is the smallest possible "language" on which an AI could be built? Using the NAND gate as a canonical minimal primitive, we examine how universality arises from extremely sparse operations, and why this is already sufficient for expressing any AI algorithm. We then compare NAND-based universality with other minimal formalisms, such as one-instruction set computers, lambda calculus, combinator systems, and universal cellular automata. Along the way, we distinguish raw expressive power from practical concerns like efficiency, programmability, and architecture. Finally, we connect these ideas to information-theoretic notions of minimal description, asking what it means to specify an AI in the shortest possible language and what this reveals about substrate independence and the nature of intelligence itself. The goal is to clarify where the true lower bounds lie: in physics, in computation, or in our ability to describe and implement minds succinctly.

Keywords: minimal computer language, NAND gate, Turing completeness, universal computation, AI substrate, one-instruction set computer, lambda calculus, SKI combinator calculus, cellular automata, functional completeness, Kolmogorov complexity, Minimum Description Length, substrate independence, it from bit, computational universality, machine architecture, learning algorithms, emergent intelligence.

A collaboration with GPT-5.1-Thinking-Extended. 45 pages.

Outline

1. Introduction: AI on the Sparsest Possible Substrate

The contemporary image of artificial intelligence is bound up with towering software stacks, hyperscale data centers, and models containing hundreds of billions of parameters. From that vantage point, it is easy to forget that all of this apparent complexity ultimately rests on a surprisingly austere foundation. At the lowest level, digital hardware computes by combining simple boolean operations and moving bits between memory locations. The fact that such minimal operations can support the full spectrum of AI systems, from simple search to large-scale deep learning, is both mathematically remarkable and philosophically suggestive.

This paper starts from a deliberately stripped-down question: what is the smallest possible language on which an AI could be built? Here "language" is used broadly to mean any primitive set of operations and composition rules capable of expressing programs. NAND, the two-input logical operation that outputs false only when both inputs are true, is a canonical example. Although trivial in isolation, it is functionally complete: all other boolean operations can be constructed from it. The central claim is that, given enough NAND gates wired appropriately and combined with mechanisms for memory and control, there is nothing in principle that cannot be implemented, including the most sophisticated AI algorithms.

By examining how far one can descend toward such minimal substrates, the discussion illuminates several layers at once. There is the formal theory of universality, which guarantees that many wildly different systems are computationally equivalent. There is the engineering practice of building usable architectures on top of minimal cores. And there is a philosophical layer, where questions about the nature of intelligence, substrate independence, and the meaning of "artificial" are reframed in terms of how much structure is actually required. The rest of the paper unpacks these themes, moving from classical results in computation theory, through specific minimal formalisms like NAND and one-instruction set computers, and finally to implications for information theory and philosophy of mind.

1.1 Framing the question of minimal languages for AI

To ask about the smallest possible language for AI is to ask what ingredients are strictly necessary for a system to count as a general-purpose computer capable of supporting intelligent behavior. The word "smallest" can be interpreted in several ways: fewest primitive operations, shortest formal specification, least hardware complexity, or minimal descriptive length in bits. All of these point toward the same underlying concern: when we strip away convenience and historical accident, what remains as the irreducible core?

Within this paper, the focus is on minimality in the sense of primitive operations. A language will be called minimal for AI if it satisfies three conditions. First, it must be capable of universal computation, in the sense that any computable function can be expressed as a program in that

language. Second, it must admit some way, at least in principle, to store unbounded information, whether through explicit memory, tape, or equivalent structures. Third, it must be sufficiently compositional that non-trivial algorithms can be constructed without adding new primitives beyond those in the core.

Artificial intelligence is then treated as a particular class of computations, including search, optimization, learning, and symbolic manipulation. If these can all be encoded within a given minimal language, then there is no further expressive requirement tied specifically to intelligence. In that sense, the question is not whether some special "AI instruction" is needed, but rather how far down one can go in the hierarchy of formalisms and still retain the ability to build such systems in principle. The comparison between NAND and more familiar languages serves as a concrete way to make this question precise.

1.2 Practical AI stacks versus theoretical lower bounds

The gap between the minimal languages of computation theory and the realities of modern AI practice is enormous. Contemporary systems are typically implemented in high-level languages such as Python, calling into optimized numerical libraries, executing on GPUs or specialized accelerators, and supported by extensive operating systems, compilers, and runtime environments. These stacks are designed to address human concerns: productivity of programmers, ease of debugging, hardware utilization, and the logistics of distributed training.

Theoretical lower bounds, by contrast, disregard these practicalities. A Turing machine, a lambda calculus expression, or a NAND-only circuit does not care about programmer ergonomics, training time, or energy costs. These models capture only the abstract capacity to compute, assuming unlimited time and memory. From that point of view, the entire modern AI stack is a convenience layer on top of a universal core that could be instantiated in many other ways.

This contrast is central to the framing of the paper. When asking what the smallest possible language for AI might be, the intention is not to advocate rebuilding real-world systems using only NAND gates or a single instruction. Rather, the aim is to map the logical basement of the edifice. Understanding where the theoretical floor lies provides a reference point for judging how much of the higher-level complexity is essential and how much is contingent. It also clarifies that powerful AI does not depend on any particular programming language or architecture, so long as the system remains universal and can manage the necessary scale.

At the same time, practical considerations cannot be entirely ignored. A minimal language might be universal yet completely unusable for any realistic AI engineering because of catastrophic inefficiencies. Part of the analysis therefore involves distinguishing universality from practicality, and exploring how the same AI algorithm might look when encoded in a sparse substrate versus

a rich one. This helps highlight the role of abstractions, compilers, and architectural choices in turning a theoretical capability into a functioning system.

1.3 Why minimality matters: clarity, universality, and philosophy of mind

The pursuit of minimal languages for AI is not just a technical curiosity. It matters for at least three reasons: clarity of explanation, understanding universality, and probing deeper questions about mind and matter. First, minimal systems often serve as excellent explanatory tools. By reducing the apparatus to a single gate or a single instruction, they make the structure of computation visible in a way that is often obscured in high-level environments. Demonstrating how a neural network or a search algorithm can be compiled down to a minimal substrate can reveal hidden assumptions about representation and control.

Second, minimality foregrounds the phenomenon of universality. The existence of many different universal systems, from NAND circuits to lambda calculus, shows that intelligence, as an instance of computation, does not depend on the quirks of any one formalism. This supports the idea of substrate independence: the same high-level behavior can, at least in principle, arise from silicon, neurons, cellular automata, or other physical realizations, as long as they implement a universal language. Examining minimal substrates helps clarify which features are truly necessary for universality and which are implementation details.

Third, minimal languages provide a testing ground for philosophical questions about the nature of artificial minds. If intelligence can, in principle, be realized on a substrate whose basic operation is as austere as NAND, this challenges intuitions that associate mind with biological richness or particular physical structures. It invites one to ask whether what matters is the pattern of computation alone, or whether some substrates might be more intrinsically hospitable to consciousness, understanding, or meaning. While this paper does not attempt to resolve such questions, it treats minimality as a lens through which they can be sharpened.

1.4 Scope and structure of the paper

The rest of the paper develops these themes systematically, moving from foundations to implications. The next section reviews the basic theory of computation and universality, introducing Turing machines, register machines, and other canonical formalisms. It emphasizes the conditions under which a system is considered universal and distinguishes expressivity from efficiency. This sets the stage for evaluating candidate minimal languages in a principled way.

Section 3 turns to Boolean logic and the role of functionally complete sets of gates, highlighting NAND as a central example. It shows how all standard logical operations can be constructed from NAND and how sequential logic, including memory elements, emerges from compositions of such gates. Section 4 then steps up one level of abstraction, outlining how full-fledged CPUs

and instruction sets can be built using only NAND-based circuitry, thereby demonstrating the universality of such hardware and its sufficiency for running arbitrary AI programs.

Section 5 surveys other minimal programming formalisms beyond NAND, including one-instruction set computers, lambda calculus, combinatory logic, and universal cellular automata. Each of these represents a different route to universality, with different trade-offs in terms of readability, structure, and spatial organization. Section 6 examines what AI specifically requires from the substrate, illustrating how learning and search algorithms can be encoded in these minimal settings, and assessing the practical costs of doing so.

Section 7 contrasts theoretical minimality with practical engineering, focusing on abstraction, modularity, and human factors. Section 8 connects the discussion to information theory, exploring how concepts like Kolmogorov complexity and Minimum Description Length bear on the idea of a minimal description of an AI in a given language. Section 9 returns to the philosophical implications, especially the notion of substrate independence and the question of whether some substrates might be qualitatively different for intelligence. Finally, Section 10 synthesizes the findings and outlines open questions for future work, followed by a reference section collecting the relevant literature.

2. Foundations: Computation, Universality, and Turing-Complete Cores

Any discussion of minimal languages for AI has to rest on the foundations of computation theory. Before one can ask how few primitives are needed, one has to be clear about what it means for a system to be computationally general. The core idea is universality: the capacity to emulate any effective procedure that could be carried out by a human following a definite algorithm. The formal machinery for talking about universality was developed in the early twentieth century, when computation was still an abstract notion, long before digital computers or machine learning.

The convergence of several independent formalisms on a single notion of computability is the background against which claims about minimal languages must be understood. Turing machines, lambda calculus, recursive functions, and various automata all turned out to describe the same class of functions. That convergence is codified as the Church–Turing thesis, an informal but deeply influential claim about the limits of mechanical procedure. This section surveys the relevant ideas, clarifies the role of state, control, and memory, and draws a sharp line between a system's expressive scope and its efficiency in performing computations.

2.1 The Church–Turing perspective on effective procedures

The Church–Turing thesis began as an attempt to pin down the intuitive notion of an “effective procedure” or “mechanical method.” Mathematicians wanted a precise way to distinguish problems that are solvable by step-by-step calculation from those that are not. Alonzo Church, working with lambda calculus and recursive functions, and Alan Turing, working with abstract machines operating on a tape, arrived independently at formal definitions of what counts as computable. Remarkably, the functions they characterized turned out to be exactly the same.

The thesis itself is not a theorem, but a claim about the relationship between these formalisms and the pre-theoretic idea of computation. It states, in essence, that anything which can reasonably be called an effective procedure can be captured by a Turing machine or an equivalent formalism. If a process can be described as a finite set of rules that a human could follow mechanically, without ingenuity, then it can be encoded as a program for such a machine.

For the purposes of this paper, the Church–Turing perspective provides a crucial constraint. When we look for the smallest possible language capable of supporting AI, we are really asking for the smallest language capable of implementing all effective procedures, including those that realize learning, search, and reasoning. Minimal languages must therefore be at least as powerful as a universal Turing machine. A language that falls short of this mark might still be useful for restricted tasks, but it could not, in principle, host all possible AI algorithms.

2.2 Turing machines, register machines, and equivalence of models

Turing’s original model is deliberately spartan. A Turing machine consists of a finite control (basically, a finite state machine), a tape divided into discrete cells extending without bound in at least one direction, and a read–write head that moves left or right one cell at a time. At each step, the machine reads the symbol under the head, consults its current state, and uses a transition function to decide what symbol to write, how to move, and what new state to enter. Despite its simplicity, this model can simulate any algorithmic process.

Other formalisms, such as register machines, rewrite systems, and partial recursive functions, were developed to capture the same intuitive class of computable functions in different ways. Register machines, for example, use a finite set of registers that hold natural numbers, and execute instructions such as increment, decrement, and conditional jump. At first glance, these seem very different from Turing machines. However, it can be shown that each can simulate the other, up to a fixed translation overhead.

This equivalence of models is central to the concept of Turing completeness. A system is Turing complete if it can emulate a universal Turing machine, meaning that for every Turing machine there is a program in the system that yields the same input–output behavior. If two systems are

each Turing complete, they are computationally equivalent in expressivity, even if their internal structure and natural idioms differ dramatically.

The existence of this large family of equivalent models has an important implication for minimal languages. It means that there is no single canonical way to present universality. One can start from tapes and heads, from registers and arithmetic, from lambda expressions and function application, or from networks of logical gates. All that matters is that, given enough resources, the system can emulate any other. Minimal languages for AI are therefore not unique; rather, they exist as a variety of formalisms, each with its own path to universal computation.

2.3 Necessary ingredients: state, control, and unbounded memory

Even though universal models can look very different, they share a common set of structural ingredients. At a high level, three features are necessary for full computational generality: state, control, and unbounded memory.

State refers to the internal configuration of the system at a given moment. In Turing machines, this is represented by the current control state and the contents of the tape cells. In register machines, it is the current instruction pointer and the values in the registers. Without some form of state that can change over time, a system is limited to computing fixed functions of its inputs, with no capacity for iterative refinement, looping, or dynamic behavior.

Control refers to the rules that determine how the system moves from one state to another. These rules must allow branching based on the current state or data, not just fixed sequences of operations. Conditional transitions, jumps, and branches are manifestations of this requirement. They are what allow algorithms to behave differently in different circumstances, to test conditions, and to follow different paths through a computation based on intermediate results.

Unbounded memory means that the system can, in principle, use arbitrarily large amounts of storage, even if it only uses a finite amount in any particular run. In Turing's model, this is represented by an infinite tape. In register machines, it appears as registers capable of holding arbitrarily large numbers, or as an unbounded number of registers. The point is not that infinite physical memory is available, but that there is no fixed upper bound built into the model that would prevent certain computations from ever completing.

These ingredients are helpful when evaluating candidate minimal languages. A system that lacks branching can only perform straight-line computations; it is not universal. A system with bounded memory, such as a finite-state machine, can solve some problems but cannot emulate all Turing machines. A system without mutable state cannot naturally express iterative processes. When we consider minimal substrates like NAND, lambda calculus, or cellular automata, part of the task is to identify how these three ingredients are realized in each formalism.

For AI in particular, these features are indispensable. Learning algorithms rely on the accumulation and modification of state, whether in weights, counts, or data structures. Search algorithms traverse large state spaces and make decisions based on intermediate evaluations. Representing knowledge, updating beliefs, and planning all presuppose structures that exploit state, control, and flexible memory. Any language that lacks these in some encoded form cannot serve as a general substrate for intelligence.

2.4 Distinguishing expressivity from efficiency

One of the most important distinctions in this context is between expressivity and efficiency. Expressivity concerns what can be computed at all in a given language or model. Efficiency concerns how much time, space, or other resources are required to carry out those computations. It is possible for two systems to be equivalent in expressivity, both being Turing complete, while differing wildly in efficiency for particular tasks.

Minimal languages often sit at an extreme point in this trade-off. They offer maximal expressivity in the sense that they can, in principle, implement any algorithm, but they may do so at the cost of enormous program length, convoluted encodings, or prohibitive runtime overhead. For example, a one-instruction set computer that uses a single arithmetic-and-branch operation can simulate any conventional instruction set, but the resulting programs are typically long and opaque, and the overhead for simple operations can be substantial.

Conversely, richer languages, with many built-in operations and data types, may offer enormous practical advantages without adding any new expressive power in the formal sense. Adding multiplication as a primitive does not make a system more than Turing complete; it merely shortens programs and improves performance. Adding specialized matrix operations, automatic differentiation, or high-level control constructs does not expand the space of computable functions, but it can transform the feasibility of implementing complex AI systems in practice.

For the project of identifying minimal languages for AI, this distinction provides a lens for evaluating candidates. The goal is to find cores that are expressively universal, not necessarily efficient. However, when discussing the implications for AI, it is crucial to acknowledge that efficiency is not an incidental concern. Large-scale learning and inference are resource-intensive, and a substrate that encodes every primitive operation in a deeply indirect way may render such systems impractical, even if they are theoretically possible.

The rest of the paper keeps this distinction in view. Minimal substrates like NAND are analyzed first for their expressive sufficiency: can they, in principle, implement arbitrary AI algorithms. Only then are the efficiency implications considered, and contrasted with the realities of contemporary AI practice. This allows a clear separation between the logical question of lower

bounds and the engineering question of how best to realize intelligence on physically constrained hardware.

3. Boolean Functional Completeness and the Centrality of NAND

To understand why a single gate like NAND can serve as the basis for an entire computing system, it is useful to step back and look at the structure of Boolean logic. Boolean algebra provides the language in which digital circuits are described: variables take values 0 or 1, and operators such as AND, OR, and NOT combine them according to specific truth tables. A striking feature of this algebra is that only a few primitive operations are needed. In fact, some single operations are already enough to generate all others when combined appropriately. Such operations are called functionally complete.

NAND occupies a special place among these minimal primitives. Defined as the negation of AND, it outputs 0 only when both inputs are 1, and 1 otherwise. It is functionally complete, meaning any Boolean function can be built from networks of NAND gates alone. This property makes NAND an ideal starting point for a discussion of minimal substrates. If one can construct the basic combinational operators, as well as memory and control structures, entirely out of NAND, then one has, in principle, the building blocks for universal digital computation.

3.1 Boolean algebra and functionally complete sets of gates

Boolean algebra is built around a small set of operations on binary variables. The most familiar are AND (conjunction), OR (disjunction), and NOT (negation). Each of these is defined by its truth table, specifying the output for every possible combination of inputs. More complex functions, such as XOR (exclusive or) or multiplexers, can be expressed as combinations of these primitives.

A set of logical operators is said to be functionally complete if every Boolean function can be expressed using only operators from that set, along with binary variables and constants 0 and 1. For example, the set {AND, OR, NOT} is functionally complete. So is the smaller set {NAND} and, similarly, the single operation NOR. The existence of such minimal bases is a nontrivial fact about Boolean algebra, and it underpins modern digital design.

Functionally complete sets matter because they provide a theoretical guarantee that no expressive power is lost when restricting oneself to those operators. If one chooses NAND as the single hardware primitive, there is no Boolean function that becomes inexpressible. Everything else can be synthesized by wiring NAND gates together in appropriate patterns. From the viewpoint of minimal languages, this establishes NAND as a serious candidate: if all Boolean logic is available, higher-level structures can be built atop it.

3.2 Constructing NOT, AND, OR, and XOR from NAND

Given that NAND is functionally complete, it must be possible to obtain the familiar operations from it. The constructions are simple but instructive. Start with the definition:

NAND(a, b) = NOT(AND(a, b)).

From this, one can derive the following:

1. NOT: To negate a single input a using only NAND, feed a into both inputs:

NOT(a) = NAND(a, a).

When a = 0, NAND(0, 0) = 1; when a = 1, NAND(1, 1) = 0, as required.

2. AND: Since NAND already computes the negation of AND, one can recover AND by negating the output of NAND:

AND(a, b) = NOT(NAND(a, b)).

Using the previous construction for NOT, this becomes:

AND(a, b) = NAND(NAND(a, b), NAND(a, b)).

3. OR: To construct OR, one can use De Morgan's law:

OR(a, b) = NOT(AND(NOT(a), NOT(b))).

Translating each subpart into NAND:

temp1 = NOT(a) = NAND(a, a)
temp2 = NOT(b) = NAND(b, b)
temp3 = AND(temp1, temp2) = NAND(NAND(temp1, temp2), NAND(temp1, temp2))
OR(a, b) = NOT(temp3) = NAND(temp3, temp3).

A more optimized version directly uses:

OR(a, b) = NAND(NAND(a, a), NAND(b, b)).

Here NAND(a, a) is NOT(a), NAND(b, b) is NOT(b), and NAND of those two gives the negation of their AND, which by De Morgan is OR.

4. XOR: Exclusive or can also be expressed in terms of NAND. One convenient construction proceeds via intermediate terms:

Let
p = NAND(a, b)
q = NAND(a, p)

r = NAND(b, p)
XOR(a, b) = NAND(q, r).

This network produces 1 exactly when a and b differ, and 0 when they are equal. While less transparent than the direct truth table for XOR, it illustrates the general principle: more complex operations are realized as small networks of NAND gates.

These constructions show explicitly how a rich set of logical functionality is emergent from repeated use of a single gate. Combinational circuits such as adders, comparators, and multiplexers are built by combining these basic operators, all ultimately reducible to NAND.

3.3 From combinational logic to sequential logic using NAND

Combinational logic captures functions where the outputs depend only on the current inputs. Many useful components fall in this category: adders produce a sum based on the current bits, multiplexers choose one input to pass through, decoders convert binary patterns to one-hot outputs. Such circuits have no memory of past inputs; they are stateless.

However, computation, and especially AI, requires more than stateless transformation. It requires the ability to store information over time, to remember intermediate results, and to update internal representations in response to new inputs. This moves the discussion from combinational logic to sequential logic, where outputs depend both on current inputs and on stored state.

Sequential behavior in digital circuits is achieved by feeding outputs back into inputs through controlled paths and using timing signals, such as clocks, to regulate when updates occur. NAND gates play a central role here as well. By arranging NAND gates in specific feedback configurations, one can construct latches and flip-flops, the basic building blocks of digital memory. The introduction of feedback breaks the simple input-output mapping of combinational circuits and creates systems with stable states that persist until explicitly changed.

In this way, the same universal primitive that suffices for combinational functions also suffices for constructing the mechanisms of state and control needed for full computation. The distinction between combinational and sequential logic is thus not a matter of different primitives, but of how those primitives are interconnected and governed in time.

3.4 Memory elements: latches, flip-flops, and state machines in NAND

A simple but profound example of memory built from NAND is the SR latch. This device has two inputs, traditionally labeled S (set) and R (reset), and two outputs, Q and not-Q, which are each connected back into the other gate to form a feedback loop. Each output is the NAND of one control input and the opposite output. When the inputs are manipulated appropriately, the

latch can be set so that Q = 1, reset so that Q = 0, or left in a state where it preserves its previous output.

From the SR latch, one can derive more sophisticated storage elements such as the D latch, which uses gating logic so that a single data input D is sampled when an enable signal is active, and held when it is inactive. Combining two latches in a master–slave configuration yields the familiar edge-triggered flip-flop, which updates its output only on a specific transition of a clock signal. All of these constructs can be built entirely from NAND gates and wiring.

Flip-flops are the core of registers, which hold multi-bit values, and of counters, which step through sequences of states. With an array of flip-flops, one can implement program counters, instruction registers, and general-purpose storage. Together with combinational logic for decoding and arithmetic, these storage elements support finite state machines that respond to inputs and transition between internal states according to specified rules.

At a higher level, a digital computer can be seen as a large, carefully organized state machine. Its state consists of the contents of memory cells, registers, and control flip-flops; its transitions are governed by clocked updates and logic derived from instructions and data. The fact that all of these structures are reducible to networks of NAND gates shows that NAND is not only functionally complete for stateless Boolean logic, but also sufficient to realize the memory and control structures needed for universal computation.

For the purposes of minimal languages for AI, this is a crucial conclusion. If one is allowed arbitrary networks of NAND gates, and the ability to feed outputs back as inputs under timing control, then one has access to the full toolkit of digital design: combinational units, memory elements, and state machines. On top of this, it is straightforward in principle to build arithmetic logic units, instruction decoders, and ultimately entire processors capable of running any AI program expressible in a conventional language. Thus, the centrality of NAND in digital logic translates directly into its status as a minimal substrate from which intelligent computation can be assembled.

4. From NAND Circuits to Universal Computers

Once NAND is accepted as a functionally complete primitive and shown sufficient to implement both combinational and sequential logic, the next step is to climb from low-level building blocks to a full digital computer. This ascent is conceptually straightforward but structurally rich. It proceeds in layers: basic arithmetic and selection circuits, storage and control elements, and finally, an organized architecture that fetches, decodes, and executes instructions. At each layer, NAND networks are reconfigured into higher-level components, until the entire system behaves like a programmable universal machine.

This section outlines that construction. It first shows how adders, multiplexers, and control circuits emerge from NAND-based logic. It then sketches a minimalist central processing unit built out of registers, a program counter, and an arithmetic logic unit. From there, it interprets the resulting instruction set as a higher-level language hosted by NAND hardware. Finally, it explains why such a machine is universal in the sense required by computation theory and draws out what that means for building AI on such a sparse substrate.

4.1 Building adders, multiplexers, and control logic from NAND

Simple arithmetic lies at the heart of most digital computation. Even high-level AI operations, such as matrix multiplications in neural networks, reduce to large numbers of additions and multiplications on binary words. At the circuit level, addition is realized by adders constructed from basic logic gates. Because AND, OR, and XOR can all be implemented using only NAND gates, adders can be built from NAND as well.

A one-bit half adder takes two input bits a and b and produces a sum bit and a carry bit. The sum is given by XOR(a, b), while the carry is given by AND(a, b). As shown earlier, XOR can be synthesized from a small network of NAND gates, and AND can be implemented by negating a NAND output. Thus the half adder is a modest composition of NAND-based subcircuits. Extending this to a full adder, which also accounts for an incoming carry bit, involves computing the sum of three bits and producing an outgoing carry. This can be done by cascading two half adders and using OR to combine the carry outputs, all of which are reducible to NAND.

Arranging multiple full adders in sequence yields a ripple-carry adder capable of adding multi-bit binary numbers. Each stage processes one bit of the operands along with the carry from the previous stage, producing a bit of the result and a new carry. Although ripple-carry adders are not the most efficient design in terms of delay, they are structurally simple and demonstrate that basic arithmetic on arbitrary-sized integers can be realized by repeating a simple NAND-based pattern.

Multiplexers are another essential component. A one-bit multiplexer selects between two input bits based on a control signal. Its output can be described as:

output = (control AND input1) OR (NOT(control) AND input0).

This expression directly translates into a small circuit composed of AND, OR, and NOT gates, each of which can be realized via NAND. Multi-bit multiplexers and demultiplexers generalize this pattern, allowing entire words or multiple signals to be routed under control of select lines. In a processor, multiplexers are used to choose which register feeds the arithmetic unit, which address goes to memory, and which value is written back into storage.

Control logic itself is mostly combinational. Given the current instruction word, the processor must produce various control signals: which operation the arithmetic unit should perform, which registers should be read or written, whether to increment the program counter or branch, and so on. Each of these decisions is encoded as Boolean functions of the instruction bits and, possibly, of status flags produced by previous computations. Because Boolean functions are expressive over NAND, all control logic can be synthesized as structured networks of NAND gates.

Together, adders, multiplexers, and control circuits built from NAND provide the basic toolkit for arithmetic and data steering inside a computer. They form the backbone of the arithmetic logic unit and the instruction decoding path, demonstrating how surprisingly complex behavior emerges from repeated use of a single gate type.

4.2 Assembling a minimalist CPU: registers, program counter, and ALU

With arithmetic and selection mechanisms in hand, the next step is to assemble a minimalist central processing unit. At a conceptual level, a CPU consists of three main components: registers for holding data, a program counter for tracking the current instruction, and an arithmetic logic unit (ALU) for performing operations on data. All of these can be composed from NAND-based storage elements and combinational circuits.

Registers are collections of flip-flops that hold multi-bit values. For example, an 8-bit register can be built from eight D flip-flops, each holding one bit. Control logic determines when the register should load a new value from a data bus and when its contents should be driven onto that bus. Since D flip-flops themselves can be constructed from NAND-based latches and gating logic, entire register files can be realized as structured arrays of NAND gates.

The program counter is a special register that holds the address of the next instruction to execute. On each clock cycle, under normal circumstances, it is incremented by one, so that the processor steps through the instruction sequence. The increment operation is performed by an adder, and the choice between incrementing and loading a branch target is made by multiplexers controlled by branch logic. All of these structures, again, are built from NAND, ensuring that the flow of control, not just data storage, remains grounded in the same primitive.

The arithmetic logic unit combines the adders and other combinational units to perform operations such as addition, subtraction, bitwise logic, and comparisons. A minimalist ALU might support only a small set of arithmetic and logical operations, selectable by a few control bits. Internally, these operations are realized by wiring together adders, inverters, and logical operators, all synthesized from NAND. The ALU outputs both the result and status flags, such as zero or negative, which can be fed back into the control logic to influence branching decisions.

Tying these elements together requires a modest control unit. The control unit receives the current instruction, decodes it using combinational logic, and produces the necessary control signals for the registers, ALU, program counter, and memory interface. It ensures that, on each clock cycle, the right values are loaded, processed, and stored. In a hardwired control scheme, this logic is fixed as a network of gates; in a microcoded scheme, an internal read-only memory encodes control sequences, but even that ROM is functionally a network of logic and storage elements built on NAND.

The resulting CPU, though minimalist, is fully capable of executing a sequence of instructions stored in memory, manipulating data, and branching based on computed results. It embodies the classical fetch–decode–execute cycle. Importantly, every part of it can be systematically reduced, in implementation, to NAND gates and wires. This demonstrates not only logical sufficiency but also architectural completeness: from a single gate type, a complete programmable computing engine emerges.

4.3 Instruction sets as higher-level "languages" over NAND hardware

Once a CPU exists, its programming is governed by its instruction set architecture (ISA). This is the vocabulary of operations that machine code can express: arithmetic, logic, memory access, control flow, and sometimes more specialized instructions. From the software perspective, an ISA is a low-level language in which algorithms are written or into which higher-level languages are compiled.

There is a clear separation between the hardware foundation and the instruction set that runs on top of it. The same NAND-based substrate can realize different ISAs, just as different microarchitectures can implement the same ISA. This separation helps clarify the notion of minimal languages for AI. At the chemical level, there is a single primitive operation; at the microarchitectural level, there is a finite state machine executing control sequences; at the ISA level, there is a small set of instructions that programmers or compilers see.

Crucially, the ISA can be exceedingly simple while still supporting universal computation. A machine might offer only load, store, add, subtract, branch-if-zero, and halt. Another might use a single instruction, such as subtract-and-branch-if-less-or-equal, and encode all other operations as patterns of that instruction. These designs illustrate how the notion of a minimal language shifts as one moves up the abstraction stack. At the lowest level, the "language" is the connectivity of NAND gates. A level above, it is the machine's instruction repertoire.

For AI, the instruction set acts as the interface between algorithmic designs and the hardware substrate. High-level descriptions of learning procedures, search strategies, or symbolic manipulations are compiled into sequences of machine instructions. Each instruction, in turn, triggers a complex but deterministic choreography of signals through the NAND networks

making up the CPU and memory. One can therefore think of the ISA as a compact, structured language for orchestrating the behavior of a vast underlying NAND circuit.

This view reinforces the idea that minimality can be considered at multiple levels. One might seek a minimal ISA capable of efficiently expressing AI algorithms, distinct from the minimal gate set needed to implement that ISA. Both are interesting questions, but only the latter addresses the ultimate logical foundation. The former is more about practical design choices that balance simplicity, performance, and programmability.

4.4 Universality of a NAND-based machine and implications for AI

A CPU built entirely from NAND gates, with a sufficiently rich instruction set and access to unbounded memory, is Turing complete. This follows from the equivalence between realistic instruction-set machines and Turing machines. Any computation that a Turing machine can perform can be encoded as a program for such a CPU, and conversely, a Turing machine can simulate the CPU by manipulating its own tape. The universal character of the device does not depend on the particular gate used in its construction, but the fact that NAND alone suffices underscores its minimality.

From the standpoint of AI, this universality means that there is no intrinsic limitation imposed by the use of NAND as the sole primitive. Any learning algorithm, search procedure, or symbolic reasoning system expressible on a standard computer can be implemented on the NAND-based machine. Deep neural networks, decision trees, reinforcement learning agents, probabilistic models, and planning algorithms all reduce to programs composed of basic arithmetic and logical operations, memory accesses, and control flow. Since each of these is available at the ISA level, and the ISA is grounded in NAND, the substrate is fully adequate.

The implications are twofold. First, AI does not require any special “intelligent” instruction or primitive operation beyond those needed for universal computation. Intelligence arises from the organization of computations, the structure of data, and the interaction with the environment, not from a special gate with semantic content. This supports a strong form of substrate independence: if a system can host universal computation and provide enough resources, it can, in principle, host AI.

Second, the fact that universality can be achieved with such a sparse gate set highlights how much of modern AI’s complexity resides in the upper layers of abstraction. Frameworks, libraries, and specialized accelerators exist to make certain patterns of computation efficient and manageable, not to expand the fundamental space of what is computable. They are responses to limits of time, energy, and human comprehension. From a minimal perspective, they are all elaborations atop the same basic capacity embodied in the NAND-based machine.

At the same time, this theoretical universality should not be confused with practical feasibility. Building a real AI system using a raw NAND-based CPU with a very minimal ISA might be possible in principle but hopelessly inefficient in practice. The number of gates required, the clock speeds attainable, and the energy consumption would all impose severe constraints. Thus, while the universality of a NAND-based machine settles the question of logical sufficiency, it leaves open the equally important question of how best to engineer AI systems that are tractable, robust, and comprehensible.

In summary, tracing the path from individual NAND gates to a universal CPU shows that the most austere digital primitive can underpin an entire hierarchy of computation. Once this hierarchy is in place, AI becomes a matter of how programs are written and organized at higher levels, rather than of what the hardware can, in principle, express.

5. Minimal Programming Formalisms Beyond NAND

While NAND provides a concrete, hardware-flavored route to universality, it is far from the only minimal substrate on which an AI could, in principle, be realized. The history of computation is full of deliberately stripped-down models that trade hardware concreteness for mathematical elegance or conceptual clarity. These formalisms show that universality does not depend on the details of digital circuits. Instead, it can emerge from arithmetic operations on registers, purely symbolic rewriting systems, or local update rules on grids of cells.

This section surveys several such minimal formalisms: one-instruction set computers such as SUBLEQ, the lambda calculus as a foundational model of computation, combinatory logic and SKI systems as instructionless cores, and universal cellular automata as spatially organized substrates. Each of these can serve as a candidate for the “smallest possible language” in its own sense. Together, they broaden the perspective beyond NAND, illustrating how different styles of minimality illuminate different aspects of computation and, by extension, of possible AI implementations.

5.1 One-instruction set computers: SUBLEQ and related designs

One-instruction set computers (OISCs) push the idea of minimality into the realm of machine code. Instead of offering a variety of instructions for arithmetic, logic, and control flow, an OISC uses a single instruction that can, through clever coding conventions, emulate all others. The most studied example is often called SUBLEQ, short for “subtract and branch if less than or equal to zero.”

In a canonical SUBLEQ machine, memory is an array of integers. Each instruction consists of three addresses, commonly denoted A, B, and C. Executing the instruction performs the

following steps: subtract the contents of memory location A from memory location B, store the result back in B, and if the result is less than or equal to zero, branch to the address C; otherwise, proceed to the next instruction. There are no explicit instructions for addition, loading, storing, or unconditional branching. Yet all of these can be encoded as sequences of SUBLEQ operations.

For example, addition can be realized by using subtraction with negated values, conditional branching can be turned into unconditional branching by appropriate use of always-true conditions, and memory moves can be constructed by subtracting to zero and then reconstructing values via sequences of operations. The resulting programs are not intuitive by human standards, but they are mechanically equivalent in expressive power to programs written for conventional instruction sets.

From the standpoint of minimal languages for AI, OISCs like SUBLEQ show that even at the level of software-visible instructions, universality can be compressed into a single primitive. Any AI algorithm expressible in a conventional machine language can, at least in principle, be translated into SUBLEQ code. High-level constructs such as loops, function calls, and data structures become patterns of the lone instruction. While the inefficiencies of such encodings make them impractical for real systems, they underscore that nothing in the nature of intelligence requires a rich instruction set. One carefully chosen operation, combined with unbounded memory, is enough.

5.2 Lambda calculus as a minimal model of computation

Where OISCs focus on machine-level arithmetic and branching, the lambda calculus approaches minimality from a purely symbolic, functional direction. Introduced as a formal system for defining functions and their application, lambda calculus is built from only three syntactic forms: variables, function abstractions, and function applications. There are no primitive numbers, booleans, or data structures; everything is encoded as functions.

A lambda term either names a variable, defines an anonymous function by binding a variable in a body, or applies one term to another. Computation proceeds by beta reduction, in which an application of a function abstraction to an argument is simplified by substituting the argument for the bound variable in the function body. Despite its simplicity, the lambda calculus is capable of expressing all computable functions. Standard arithmetic operations, logical operators, and even control structures like conditionals and recursion can be defined purely in terms of function application.

For instance, booleans can be represented as functions that choose between two alternatives, numbers as iterated applications of a successor function, and conditionals as higher-order functions that dispatch based on a boolean argument. Recursive definitions, which might seem

to require some external naming mechanism, are achieved through fixed-point combinators that provide a way for functions to refer to themselves indirectly.

As a minimal language for AI, the lambda calculus presents a starkly different picture from NAND or SUBLEQ. There are no explicit notions of memory, registers, or branching. Instead, state is carried implicitly through function arguments and return values, and control flow is realized through the structure of function application and reduction. Yet, because lambda calculus is equivalent in expressive power to Turing machines, any algorithmic description of an AI can be translated into it.

The value of lambda calculus in this context lies in its abstraction. It shows that universality, and thus the capacity for AI, does not depend on the presence of numerically flavored operations or explicit control flow constructs. It can arise from nothing more than the ability to define and apply functions, with all other structures encoded as higher-order patterns. This supports a view of intelligence as something that can, in principle, be understood as the unfolding of symbolic transformations in an extremely minimal algebraic setting.

5.3 Combinatory logic and SKI systems as instructionless cores

Combinatory logic pushes minimality even further by eliminating variables from the formalism. Instead of writing functions that explicitly name their arguments, combinatory logic represents computation as the composition of a few primitive combinators, which are themselves higher-order functions. The most famous such system is built around three combinators, traditionally denoted S, K, and I.

Informally, K is a combinator that, given two arguments, returns the first and discards the second. S is a combinator that distributes its third argument across the first two, enabling a form of generalized application. I is the identity combinator, returning its argument unchanged, and can be defined from S and K, so that in a strict sense even S and K alone suffice. Every lambda expression can be systematically translated into an equivalent expression in SKI combinatory logic, meaning that SKI is also a universal model of computation.

In a combinatory logic expression, computation proceeds by repeatedly applying reduction rules associated with the primitive combinators. For example, an application of K to arguments x and y reduces to x, and an application of S to x, y, and z reduces to applying x to z and then y to z. There is no notion of variables, environments, or explicit substitution. All structure is encoded in the tree of combinator applications and is manipulated locally by the reduction rules.

From the perspective of minimal languages, SKI systems are notable because they eliminate both variables and any explicit instruction set. There are only primitive combinators and their applications. Yet, like lambda calculus, they can encode arithmetic, logic, data structures, and control flow purely within this framework. An AI expressed in lambda calculus can be translated

into SKI form, losing all named parameters and relying only on the structural interaction of S and K.

Seen this way, SKI combinatory logic presents perhaps the starkest minimal core: a language with no variables, no data types, and no instructions in the conventional sense, yet capable of universal computation. Its relevance to AI lies not in its practicality but in its demonstration that even the notion of an instruction set is optional. Intelligence-generating computations can be seen as emergent patterns of reduction in a variable-free system of combinators, again underscoring the substrate-independent character of universality.

5.4 Universal cellular automata and computation in spatial media

A different route to minimality and universality is offered by cellular automata. Instead of focusing on symbolic expressions or register-based arithmetic, cellular automata arrange computation in space. The system consists of a grid of cells, each of which can be in one of a finite number of states. Time advances in discrete steps, and at each step, the state of each cell is updated according to a local rule that depends on the current states of that cell and its neighbors.

Some cellular automata are universal. They can simulate any Turing machine by encoding the machine's tape and state into patterns of cell states and interpreting the local update rule as implementing the machine's transition function. Famous examples include two-dimensional systems with simple sets of states and neighborhood rules, as well as one-dimensional systems that exhibit complex dynamics despite having only a small number of states per cell.

In these universal cellular automata, computation is not localized in a single processing unit but distributed throughout the grid. Logical operations and data storage emerge from the interactions of patterns of cells that propagate, collide, and transform. There is no explicit program counter or instruction register. Instead, the "program" is encoded in the initial configuration of the grid and the fixed update rule applied uniformly in time and space.

As a candidate minimal language for AI, universal cellular automata demonstrate that neither symbolic nor register-based views are necessary. Intelligence, in principle, could be realized as evolving patterns of activity in a spatially extended medium governed by a simple local rule. A learning algorithm, for example, might correspond to the self-organization of structures that preserve and propagate certain patterns while suppressing others. Search and optimization could be embodied in the dynamics of pattern interactions.

The relevance of such models goes beyond abstract possibility. They suggest that substrate independence extends to systems where computation is an emergent property of spatial dynamics, rather than an explicitly programmed sequence of instructions. For minimal language considerations, the key point is that the local update rule, which can often be described in just a

few bits, suffices as the entire “language” of the system. Everything else is encoded in the initial conditions and the ongoing evolution.

Taken together, one-instruction computers, lambda calculus, combinatory logic, and universal cellular automata show that universality can take many forms, each minimal in its own way. Some favor arithmetic operations on memory, others pure function application, still others pattern dynamics in space. All, however, share the property that they can express any effective procedure, including those that underlie artificial intelligence. This diversity of minimal formalisms strengthens the conclusion that AI is not tied to any particular style of computation. Instead, it can, in principle, be built on top of almost any universal substrate, no matter how sparse its primitives.

6. What AI Actually Requires from the Substrate

Once it is clear that many different minimal formalisms reach universality, the natural question is what, if anything, AI specifically demands from the underlying substrate. If any universal system can in principle run any algorithm, does AI place additional structural requirements? Or does it merely intensify generic demands for time, space, and energy?

From a computational point of view, most AI methods reduce to recurring patterns: mechanisms that adjust parameters based on data, procedures that search through large combinatorial spaces, and structures that represent knowledge, policies, or models in a way that can be updated and queried. These patterns do not require new primitives beyond those needed for universal computation, but they do stress the substrate in particular ways. They presuppose the ability to store large amounts of structured information, to perform many repeated numerical or symbolic operations, and to adapt internal state as new evidence arrives.

This section makes those patterns explicit. It examines learning, search, and representation as generic computation; shows how neural networks, search trees, and optimization procedures can be encoded in minimal formalisms; analyzes the time, space, and energy costs of doing so on extremely sparse languages; and finally considers the point at which minimal substrates cease to be meaningful engineering choices, even if they remain theoretically sufficient.

6.1 Learning, search, and representation as computational patterns

Artificial intelligence is often presented as a collection of distinct subfields: machine learning, planning, knowledge representation, natural language processing, and so on. At the level of computation, however, a small set of patterns recurs across these areas.

Learning, in its broadest sense, is the process of adjusting internal parameters to improve performance on a task, based on data or feedback. In supervised learning, these parameters

might be weights in a neural network; in reinforcement learning, they might be value estimates or policy parameters; in Bayesian inference, they might be posterior distributions. Computationally, learning appears as repeated updates to stored state, driven by data and some update rule. The substrate must therefore support read–modify–write cycles on structured data at scale.

Search refers to traversing a space of possibilities to find states or structures that satisfy some criterion: low loss, high reward, or consistency with constraints. This includes combinatorial search in planning or theorem proving, as well as continuous search in optimization. In computational terms, search involves generating successor states, evaluating them, and deciding which to explore further. This demands extensive branching, storage of partial results, and sometimes backtracking or stochastic sampling.

Representation is the choice of how knowledge, hypotheses, and goals are encoded so they can be manipulated. A state can be represented as a vector of real numbers, a graph, a logical formula, or a sequence of symbols. The representation determines what operations are natural and efficient: dot products, graph traversals, unification, or parsing. At the substrate level, these representations all reduce to structured arrangements of bits and a set of primitive operations that transform them.

These three patterns intertwine. Learning adjusts representations by search in parameter space; search itself can be guided by learned heuristics; representation choices constrain both. What they share is a dependence on generic computational capabilities: the ability to store and update large state, to perform conditional branching, to carry out arithmetic and comparison, and to iterate procedures many times.

Crucially, none of this requires a special “learning gate” or “search instruction.” Any universal substrate with state, control, and memory can, in principle, implement these patterns. The distinctive character of AI emerges from how these patterns are organized and scaled, not from the presence of novel primitives at the lowest level.

6.2 Encoding neural networks, search trees, and optimization in minimal formalisms

To make the connection to minimal languages more concrete, consider how three canonical AI structures could be encoded: neural networks, search trees, and optimization algorithms such as gradient descent.

A neural network is, at bottom, a composition of linear transformations and nonlinear activation functions. In a typical feedforward network, each layer computes a vector–matrix product, adds a bias, and applies a nonlinearity elementwise. On digital hardware, these operations reduce to repeated uses of addition, multiplication, and elementary nonlinear functions approximated by simple arithmetic or lookup tables. In a NAND-based CPU, addition and multiplication are

implemented by circuits built from NAND gates; the nonlinearity is likewise implemented as a piecewise function realized by comparisons and simple arithmetic. The network's weights and activations are stored in memory and updated by learning rules that themselves are sequences of additions, multiplications, and comparisons.

The same description can be carried over to a one-instruction set computer like SUBLEQ. Here, memory locations hold integer-encoded weights and activations. The basic SUBLEQ instruction is used to implement addition and subtraction by appropriate sequences and to realize branching needed for activation function logic. Matrix operations become nested loops expressed entirely in terms of the single instruction, and learning rules are unrolled into long sequences of primitive operations. While the resulting programs may be verbose and inefficient, there is no conceptual obstacle to encoding the full network and its training dynamics.

In lambda calculus or SKI combinatory logic, neural networks can be represented at an even higher level of abstraction. Vectors and matrices can be encoded as functions that apply a supplied operation across their elements, while real numbers can be approximated by rational encodings or other schemes. Linear layers are higher-order functions that accept weight and input encodings and return transformed encodings. Activation functions are simply functions over these encodings. Gradient descent updates are expressed as compositions of these higher-order functions, ultimately reducible to pure function application and reduction. Although extremely indirect compared to conventional numeric code, this shows that even purely symbolic minimal formalisms can, in principle, host continuous optimization.

Search trees offer a complementary example. A tree can be represented as nested data structures in any universal language: nodes with fields for state, children, and evaluation scores. In NAND-based machines, these are structures in memory linked by addresses. In SUBLEQ, they are arrays of locations with conventions for child pointers. In lambda calculus, a tree can be represented as a function that encapsulates its branching structure and provides accessors for traversals. Search procedures, such as depth-first or best-first search, are then expressed as recursive functions or loops that expand nodes, evaluate them, and maintain frontier data structures.

Optimization algorithms, including gradient descent, evolutionary strategies, or simulated annealing, are also straightforward to encode once arithmetic, comparison, and random sampling are available. Each iteration of such algorithms updates a set of parameters based on evaluations of an objective function, which itself is a deterministic computation over data and parameters. Minimal formalisms can express both the objective function and the update rule as compositions of their primitive operations.

These examples illustrate a general point. Once a substrate is universal, the question of whether it can encode a given AI structure is not one of possibility but of convenience and cost. The transformations needed to map high-level neural, symbolic, or search-based algorithms into minimal languages may be tedious to write by hand, but they are mechanically definable. Compilers and interpreters can, in principle, automate these translations, further confirming that AI does not demand special substrate features beyond universality.

6.3 Time, space, and energy costs of AI on extremely sparse languages

Although universality guarantees that AI algorithms can be expressed in minimal languages, it says nothing about how costly that expression will be. The overheads imposed by extremely sparse primitives can be enormous. These overheads manifest as increased program length, longer execution time, greater memory usage, and higher energy consumption for any realistic physical implementation.

In a system like SUBLEQ, every arithmetic operation must be decomposed into a sequence of subtract-and-branch steps. Multiplication becomes repeated addition or more elaborate algorithms, themselves expressed as many instructions. Conditional logic that might be a single branch instruction in a richer ISA requires careful construction of conditions and jump patterns. As a result, a relatively short program in a higher-level language expands into a long and intricate sequence of primitive instructions. Execution time grows correspondingly, as does the size of the code stored in memory.

Similar blowups occur when implementing algorithms in lambda calculus or SKI combinatory logic. Operations that are constant time on a register machine, such as indexing into an array, become logarithmic or worse when data is encoded as nested function applications. Functional encodings of numbers and data structures introduce layers of indirection. Reduction sequences can be very long relative to the conceptual structure of the algorithm. Without aggressive optimization and clever encodings, simple computations can require huge numbers of reduction steps and large intermediate expressions.

Physical implementations must also contend with energy costs. Each primitive operation, whether a NAND gate firing or a cellular automaton cell updating, consumes some energy and generates heat. Long sequences of primitives required to simulate what would be a single high-level operation lead to higher cumulative energy expenditure. In practice, hardware designers choose richer gate sets or higher-level functional units not because minimal primitives are unavailable, but because they are energetically and temporally inefficient.

Space usage presents another dimension of cost. Minimal encodings sometimes require more memory to store intermediate structures or to represent data in indirect forms. For example, representing numbers in unary rather than binary drastically increases memory requirements as

values grow. Representations chosen for their simplicity in a minimal formalism may be disastrously inefficient in a practical setting, making large-scale AI systems impossible to fit into available memory.

All of these factors mean that, while minimal substrates are theoretically sufficient, they often impose multiplicative or even exponential overheads on resource usage. For small toy problems, these overheads may be tolerable; for realistic AI workloads involving large datasets and models, they quickly become prohibitive. This does not undermine the logical point about universality, but it does limit the usefulness of minimal languages as practical foundations for high-performance AI.

6.4 When minimal substrates become practically unusable

At some point, the gap between “in principle” and “in practice” becomes too wide to ignore. A substrate may be universal and capable of expressing any AI algorithm, but still be essentially unusable for engineering purposes. The boundary between minimal but usable systems and minimal but only theoretically interesting systems depends on technology, tooling, and human cognitive limits.

One dimension of unusability is human comprehensibility. Programs written directly in a one-instruction language or in raw SKI combinators are almost impossible to understand, debug, or maintain. Even if compilers generate such code automatically, reasoning about performance, correctness, or edge cases in terms of the primitive language becomes intractable. Abstractions layered above the minimal core are needed to make programming tolerable, but these abstractions immediately move away from strict minimality at the visible level.

Another dimension is the cost of verification and testing. AI systems are already complex and difficult to analyze when written in conventional languages with strong type systems and tooling. Compressing everything into a minimal core can make formal reasoning about program behavior more uniform in theory, but in practice it obscures structure. Verification conditions may be easier to express in a high-level representation that directly reflects the algorithm’s intent than in a flattened minimal encoding.

Physical constraints further limit the viability of extreme minimalism. Hardware built from only NAND gates could, in theory, implement any higher-level functional unit, but practical designs introduce more specialized gates and modules to reduce delay, area, and power consumption. Similarly, trying to simulate floating-point arithmetic or parallel vector operations purely in terms of the most minimal operations leads to latency and energy costs that defeat the purpose of building efficient AI accelerators.

There is also an architectural aspect. Many AI workloads benefit from parallelism and locality: processing many data items simultaneously, and keeping related data close in memory. Minimal

formalisms do not preclude such organization, but they do not provide any special support for it either. Richer instruction sets and architectural features, such as vector units, caches, and specialized tensor operations, exist precisely to exploit these patterns.

Beyond a certain point, then, minimal substrates serve best as conceptual anchors rather than as practical design targets. They clarify what is logically necessary and sufficient. They allow one to prove that no new fundamental primitives are required for AI. They can serve as the target of theoretical reductions and as a common denominator across different computational models. But actual AI systems are almost always built on top of less minimal, more structured languages and architectures that strike a balance between universality and efficiency.

Recognizing when a minimal substrate has crossed from instructive to unusable is itself a design judgment. It depends on whether the goal is to reason about the limits of computation, to build pedagogical examples, or to engineer high-performance systems. For exploring the foundations of AI, minimal languages are invaluable. For deploying AI in real-world settings, they are starting points for abstraction, not destinations.

7. Practical Versus Theoretical Minimality

The existence of extremely sparse universal substrates, from NAND to one-instruction set computers and combinatory systems, settles the theoretical question: rich collections of primitives are not required for computational universality, and therefore not required for AI in principle. However, engineers, researchers, and tool builders do not operate only in the realm of principle. They work within constraints of time, money, cognitive capacity, and organizational complexity. For them, the question is not merely whether a substrate is universal, but whether it is a good basis for designing, understanding, and maintaining real systems.

This section explores that gap between theoretical and practical minimality. It examines how human factors such as readability, composability, and debugging shape language and architecture choices; how layers of abstraction connect raw gates to modern AI frameworks; what trade-offs arise between tiny cores and richer instruction sets; and why minimality, in practice, is a design parameter rather than a hard necessity. The picture that emerges is one in which universal cores are indispensable as conceptual anchors, but rarely appear in exposed form in everyday AI development.

7.1 Human factors: readability, composability, and debugging

Humans are not Turing machines. While any computable process can be simulated in a sufficiently minimal language, human programmers and theorists have limited working memory,

limited attention, and a strong need for patterns that can be recognized and reused. These constraints make human factors central in evaluating minimality.

Readability is the most immediate concern. A program written purely in terms of NAND gates, SUBLEQ instructions, or SKI combinators may be universal, but for a human reader it is almost opaque. High-level intent, such as “perform a dot product” or “expand a node in a search tree,” is buried under a mass of low-level operations. Even with good comments and tooling, understanding such programs is difficult, and any modification risks subtle errors.

Composability is closely related. AI systems are typically built as compositions of modules: layers in a neural network, components in a pipeline, subroutines in a planner. Languages and frameworks that make composition natural support the building and reuse of these modules. Minimal formalisms often lack explicit constructs for modularity; composition must be simulated through conventions and encodings. Richer languages offer functions, classes, modules, and type systems that encourage structured composition, making it easier to assemble complex systems from simpler parts.

Debugging adds another dimension. When an AI system behaves unexpectedly, developers need to inspect intermediate states, set breakpoints, and reason about control flow. Tooling can, in principle, be built on top of any substrate, but the complexity of mapping human concepts to minimal primitives grows as the visible language gets sparser. Debugging in terms of logical gate switching or primitive combinator reductions is theoretically possible but practically punishing. In contrast, debugging in terms of tensors, graphs, or abstract syntax is much closer to how humans think about the system.

These human factors do not negate the value of minimal cores, but they make it clear that practical AI work requires languages and tools that align with human cognitive strengths. Minimality must be tempered by the need for clarity, structure, and feedback, otherwise the costs in development time and reliability outweigh the benefits of theoretical elegance.

7.2 Layering abstractions: from gates to high-level AI frameworks

Modern computing systems address the tension between minimal cores and human usability by layering abstractions. At the bottom sit transistors and gates, perhaps reducible to NAND; above that, microarchitectures and instruction sets; above those, operating systems, compilers, and runtime libraries; and, at the top, high-level languages, domain-specific frameworks, and interactive tools.

Each layer hides details of the layer below while exposing a more convenient interface. Gate-level configurations are encapsulated in logic cells and functional units. Machine instructions are abstracted by assembly languages and then by high-level constructs such as loops and

functions. Numerical kernels for linear algebra and optimization become the backbone of AI libraries, which then present concepts like models, datasets, and training loops to practitioners.

For AI specifically, this layered approach manifests in frameworks that treat neural network layers, loss functions, and optimizers as basic building blocks. The programmer thinks in terms of composing layers, specifying architectures, and writing training scripts, not in terms of the underlying matrix multiplications or the instructions executed on the hardware. Autograd systems automatically derive gradients, and just-in-time compilers optimize execution. Underneath, however, all of this ultimately compiles down to some instruction set, which itself is realized with gates and flip-flops.

This stack of abstractions reconciles minimal theoretical cores with rich practical environments. The minimal core establishes that, at bottom, a relatively small set of operations suffice. The abstractions built above it provide the vocabulary and structure needed for efficient human work. In this sense, minimal languages are less competitors to higher-level frameworks than foundations upon which those frameworks are erected.

7.3 Trade-offs between tiny cores and rich instruction sets

When designing a language or an architecture, there is a continuum between extremely minimal cores and richly featured instruction sets. Moving along this continuum involves trade-offs between simplicity, performance, implementability, and flexibility.

On one end, tiny cores like NAND or SUBLEQ offer conceptual purity and a uniform foundation. Implementations can, in principle, be simple at the hardware level, since only a small set of operations needs to be supported. Formal reasoning about such systems may benefit from the homogeneity of the primitive set. However, programs targeting these cores tend to be long, inefficient, and hard to write or understand. Compilers that map high-level code onto such cores must work harder and generate more sprawling low-level programs.

On the other end, rich instruction sets provide specialized operations, such as vectorized arithmetic, cryptographic primitives, or tensor contractions, that can dramatically accelerate common tasks. For AI, specialized instructions for matrix multiplication, convolution, and activation functions can reduce execution time and energy consumption by orders of magnitude. The price is additional complexity in hardware design and verification, as well as a larger semantic surface for programmers to master.

Between these extremes lie architectures that aim for a sweet spot: a modest but expressive core of instructions, augmented by a small number of performance-critical extensions. The question is not whether the core is strictly minimal in a theoretical sense, but whether it is minimal enough to be clear and manageable while rich enough to avoid pathological inefficiencies.

Similar trade-offs appear in programming language design. A language with very few constructs may be elegant but verbose; one with many constructs may be powerful but harder to learn and analyze. For AI development, languages often prioritize expressiveness and library support over extreme minimality, as the productivity and correctness benefits outweigh the aesthetic appeal of a smaller primitive set.

7.4 Minimality as a design choice rather than a necessity

Taken together, these considerations suggest a reframing of minimality. Theoretical minimality, in the sense of reducing everything to a single gate or instruction, is not a requirement for AI. It is a proof that no fundamental new primitive is needed beyond what suffices for universal computation. Practical minimality, by contrast, is a design choice: how small and simple should the visible core be, given the purposes and constraints of a particular system?

In some contexts, pushing toward minimality makes sense. Educational tools that aim to show how computation arises from simple parts may deliberately expose a very small set of primitives. Formal verification efforts might prefer a compact core calculus onto which larger languages are compiled and whose properties can be rigorously proved. Experimental architectures for research may choose minimality to isolate essential mechanisms.

In most real-world AI engineering, however, minimality competes with other goals. Performance, developer productivity, ease of integration, and robustness all push toward more expressive cores and richer abstractions. Language and architecture designers choose a point on the minimality spectrum that aligns with these goals. The resulting systems are not minimal in the strictest sense, but they rest on the assurance that if they can be compiled down to some universal substrate, they are not missing any fundamental capabilities.

Thus, minimal languages like NAND, SUBLEQ, lambda calculus, and SKI serve primarily as conceptual reference points. They demonstrate that the space of AI-capable substrates includes extremely sparse corners. They provide canonical targets for theoretical reductions and help clarify the relationship between different computational models. But in practice, they are rarely used directly. Instead, they inform and constrain the design of the richer languages and architectures in which AI is actually built.

Recognizing minimality as a design parameter rather than a hard constraint helps integrate the foundational and practical perspectives. It acknowledges the elegance of sparse cores without sacrificing the realities of large-scale AI development. It also clarifies that debates about the “right” primitive set are less about what is necessary for intelligence and more about what is suitable for human and physical constraints. In that sense, minimality is not the destination, but a compass bearing that orients the design of systems at every level of the computing stack.

8. Minimal Descriptions, Kolmogorov Complexity, and MDL Perspectives

So far, minimality has been discussed mainly in terms of primitive operations and universal formalisms. There is another way to think about "smallest possible" that shifts the focus from languages to descriptions. Instead of asking how few primitive instructions or gates are needed, one can ask: how short can the complete description of an AI system be, in bits, given some fixed reference language or machine?

This information-theoretic viewpoint brings Kolmogorov complexity and the Minimum Description Length (MDL) principle into play. Kolmogorov complexity formalizes the idea of the shortest program that generates a given object, while MDL turns similar ideas into a practical recipe for choosing models. Both depend, implicitly or explicitly, on a choice of primitive language or universal machine. Minimal languages in the computational sense thus intersect with minimal descriptions in the information-theoretic sense.

This section explores how to describe an AI as a bit string, how Kolmogorov complexity frames the notion of the shortest description of an intelligence, how MDL can act as a heuristic for architecture design, and how different choices of primitive language reshape what counts as a "minimal" description.

8.1 Describing an AI as a bit string: programs, data, and models

Any digital AI system, no matter how complex, can be represented as a finite bit string. This bit string must encode at least three ingredients: a program specifying how the system behaves, the data on which it is trained or conditioned, and the learned parameters or model state that result from training. In practice, these components are often separated conceptually, but from an information-theoretic standpoint they can be concatenated and treated as a single composite description.

For example, consider a neural network trained for image classification. The program part of the description includes the code implementing the neural network layer types, the training loop, and perhaps the data loading and preprocessing logic. The data part includes the training examples or a summary sufficient to reproduce the learned parameters. The model part is the final set of weights after training. All of this can be written down as bits, and in principle compressed if there is redundancy.

Describing an AI system as a bit string forces one to be explicit about what is being considered part of the "system." One may treat the training data as external and only encode the final model, or consider both data and training algorithm as parts of the description of the effective intelligence. Similarly, one must decide how much of the runtime environment is assumed to be given: the instruction set, standard libraries, and operating system can either be counted as part of the description or factored out as part of the chosen reference machine.

Once these decisions are made, the minimality question can be restated: given a fixed reference machine, what is the shortest bit string that, when interpreted as a program plus optional data, produces the behavior of a given AI system? The length of this string is an instance-specific measure of descriptive complexity, which Kolmogorov complexity attempts to formalize in a machine-independent way.

8.2 Kolmogorov complexity and the shortest description of an intelligence

Kolmogorov complexity assigns to each finite object the length of the shortest program that produces that object on a fixed universal machine. Applied to AI, one can imagine the Kolmogorov complexity of an agent's behavior or of its internal policy: the length of the shortest program that, when run, exhibits the same input–output behavior as the agent.

Several points make this notion subtle but illuminating. First, Kolmogorov complexity is uncomputable in general. There is no algorithm that, given an arbitrary description, reliably finds the shortest program that produces it. This limits its use to thought experiments and upper or lower bounds, rather than exact calculations.

Second, Kolmogorov complexity depends on the choice of universal machine, but only up to an additive constant. This is the content of the invariance theorem: switching from one universal machine to another can change the complexity of an object by at most a constant number of bits, corresponding to the length of a translator between the machines. For everyday objects, this constant is insignificant; for questions about absolute minimality, it becomes conceptually important.

Third, thinking in terms of Kolmogorov complexity highlights the difference between local complexity and global regularity. A sprawling neural network with millions of parameters may be generated by a relatively simple program that encodes a training procedure and a compact dataset. Conversely, a small piece of code with hand-tuned constants may be more complex in the Kolmogorov sense if there is no simple generative story behind its specific details. Minimal descriptions are about generative simplicity, not surface size.

Applied to AI, these ideas suggest a way to frame the minimal-language question at a deeper level. Instead of asking which primitive operations are needed, one can ask how concisely an intelligence can be described. A minimal substrate is one that allows a description of the agent whose length is close to its Kolmogorov complexity, given the constraints of the chosen reference. The language that makes an AI's structure and learning process easiest to express will yield shorter descriptions than a language that forces awkward encodings.

8.3 Minimum Description Length as a design heuristic for architectures

While Kolmogorov complexity is a theoretical ideal, the Minimum Description Length principle turns related ideas into a practical heuristic. MDL proposes that, when choosing between models to explain data, one should prefer the model that minimizes the sum of two description lengths: the length of the model description and the length of the data encoded under that model. The intuition is that good models compress data well, but themselves are not overly complex.

This principle can be extended from statistical modeling to AI architectures. When designing an architecture for an AI system, one might ask: how complex is the architecture description, and how complex are the resulting trained parameters or behaviors when expressed relative to that architecture? Architectures that capture regularities in a domain may allow shorter overall descriptions because they provide reusable structure that simplifies the representation of specific tasks or datasets.

For example, convolutional architectures exploit spatial locality and weight sharing, which can be seen as a way of embedding prior assumptions about images into the model class. This reduces the effective description length of image recognition systems, compared to fully connected networks that must learn similar patterns redundantly. Transformer architectures, with their attention mechanisms and token-based representations, do something similar for sequential and linguistic data.

Designing architectures with MDL in mind means looking for ways to express powerful inductive biases in a compact form, so that many different tasks can be described concisely in terms of that architecture plus task-specific parameters. In effect, the architecture acts as a language in which specific models are written; its quality is measured by how short those model descriptions can be, without sacrificing accuracy.

At a deeper level, MDL also highlights the connection between minimal languages and efficient learning. A language that allows succinct descriptions of relevant hypotheses will support faster learning, because fewer bits need to be inferred from data. Conversely, a minimal language in the primitive sense, such as NAND-only machine code, may make model descriptions unnecessarily long, hindering both learning and practical use.

8.4 How the choice of primitive language reshapes “minimal” descriptions

Both Kolmogorov complexity and MDL depend on a choice of description language or reference machine. This is where minimal computational substrates and minimal descriptions intersect. The primitives available in the language determine how easy or hard it is to describe certain structures. A high-level functional language with rich data types may allow a complex AI

algorithm to be written in a few lines; the same algorithm, when compiled down to a one-instruction machine, expands into a long and intricate low-level program.

From a strict Kolmogorov perspective, the difference between these descriptions is bounded by a constant, because one can always prepend a compiler or interpreter for the high-level language to the low-level program. However, constants matter in practice. A language whose primitives align with the natural structure of AI algorithms will yield shorter and more transparent descriptions than one that forces them through contorted encodings.

This suggests that there is a kind of alignment between a domain of problems and the primitive language used to describe solutions. In the context of AI, languages that natively support vectors, matrices, probabilistic constructs, differentiable operations, or logical inference can drastically shorten the descriptions of learning and reasoning processes. Even if these languages are ultimately compiled down to a minimal universal core, they function as more suitable description languages at the level where design and analysis occur.

Conversely, pursuing extreme minimality in the primitive language can obscure structure and inflate description length, even if the resulting system remains universal. When every operation must be built from NAND gates or a single instruction, the information that would be captured by a higher-level primitive is spread across many low-level steps. This may be elegant from a reductionist standpoint, but it is inefficient from a descriptive and cognitive standpoint.

In this light, minimal languages and minimal descriptions are not competing notions but complementary perspectives. A minimal computational core guarantees that, in principle, nothing is missing. A well-chosen higher-level language, informed by MDL-style thinking, makes the descriptions of actual AI systems as short and as aligned with the problem structure as possible. Together, they frame the design space: one sets the theoretical floor, the other guides practical choices about how to approach it without getting lost in the details of the basement.

9. Substrate Independence and the Philosophy of Minimal Intelligence

The exploration of minimal languages and universal substrates for computation naturally leads to deeper questions about the nature of intelligence itself. If NAND, SUBLEQ, lambda calculus, or a universal cellular automaton can each, in principle, host an AI, what does that say about the relationship between mind and matter? Does intelligence belong to a particular kind of physical stuff, or is it fundamentally a pattern that can be instantiated in many ways? And if it is a pattern, do some substrates nonetheless enjoy a privileged status, either for engineering reasons or for more philosophical ones?

This section examines these questions through the lens of substrate independence and minimal formalisms. It starts from the idea of intelligence as implementation-independent computation, then brings in physical constraints from thermodynamics and noise, asks whether some substrates are more natural homes for intelligence, and finally considers how minimal languages intersect with embodiment and the notion of what counts as “artificial” in artificial intelligence.

9.1 Intelligence as implementation-independent computation

One of the strongest suggestions arising from the theory of universal computation is that many different systems can realize the same computational structure. A Turing machine, a register machine, a lambda calculus interpreter, and a NAND-based CPU are all able, with suitable encodings, to implement the same algorithm. From this perspective, an intelligent process is not tied to any one of these implementations. It is defined by its abstract behavior: how it transforms inputs into outputs, how it updates internal state in response to experience, and how it pursues goals over time.

This view supports a robust version of substrate independence. If intelligence is identified with certain computational patterns, then any physical system that realizes those patterns with sufficient fidelity can, in principle, implement an intelligent agent. Neurons, transistors, optical switches, and even configurations in a cellular automaton grid all become candidates. The minimal computational languages explored earlier show that the space of possible substrates is not only broad but includes extremely austere corners.

From this vantage point, the particular choice of primitive operations—NAND versus NOR, lambda abstraction versus combinators, SUBLEQ versus a richer instruction set—becomes a matter of convenience and efficiency rather than necessity. The fundamental content of an intelligent system is its algorithmic structure and its learned or evolved parameters, not the specific way these are encoded at the lowest level. Changing the substrate or the primitive language is akin to rephrasing the same argument in a different, but logically equivalent, dialect.

This implementation-independent view does not deny that physical details matter. Real agents exist in physical environments, and their computational processes run on finite, noisy hardware subject to constraints. But it suggests that these constraints affect how intelligence is realized, not whether the pattern of intelligence can, in principle, exist. The philosophy of minimal intelligence, in this sense, is a philosophy of patterns: what matters is the structure and dynamics of the computation, not the specific atoms that carry it.

9.2 Physical constraints: thermodynamics, noise, and realizability

Substrate independence, however, is not a claim that any imagined computational pattern can be realized in any physical system without cost. Physical law imposes stringent constraints on computation. Thermodynamics sets lower bounds on energy dissipation for certain operations,

noise sets limits on reliability, and material properties constrain speed, density, and connectivity.

For example, Landauer's principle connects logical irreversibility with a minimum energy cost per bit erased. Even if a NAND-only system is universal, each gate switching consumes energy and generates heat. As one pushes toward massive AI systems, with large numbers of operations per second, these thermodynamic considerations become central. Different physical substrates—electronic, photonic, spintronic, or quantum—offer different trade-offs in terms of energy efficiency, error rates, and scalability.

Noise and imperfections provide another source of constraint. In theoretical models, NAND gates and Turing machines operate flawlessly, with sharp distinctions between states. In physical systems, signals degrade, timing skews accumulate, and components fail. Error-correcting codes and fault-tolerant designs mitigate these issues, but at the cost of additional overhead. Some substrates may be more robust than others against particular forms of noise, making them more suitable for implementing large-scale intelligent systems.

Realizability therefore introduces a distinction between what is possible in principle and what is feasible in practice. A universal cellular automaton with a very simple local rule may be able to simulate an AI, but arranging a physical system that behaves like that cellular automaton at scale may be much harder than building a conventional digital machine. Conversely, biological neural tissue exhibits computational capabilities that are still not fully understood or matched in artificial systems, suggesting that nature has found a substrate particularly well matched to certain forms of intelligence.

The upshot is that substrate independence must be held together with an awareness of physical constraints. Minimal languages and universal formalisms tell us that many substrates are computationally adequate; thermodynamics and noise tell us that not all substrates will be equally hospitable to practical, scalable intelligence. The philosophical challenge is to reconcile the abstract equivalence of computations with the concrete differences in how they can be embodied.

9.3 Are some substrates more "natural" for intelligence than others?

Given these physical considerations, it is reasonable to ask whether some substrates are more natural hosts for intelligence. "Natural" here can mean several things: easier to evolve, easier to engineer, closer to the structures found in biological cognition, or more efficient in energy and space for certain classes of tasks.

Biological brains provide a compelling data point. They are built from neurons and synapses organized into complex networks, with rich dynamics and plasticity. This substrate seems particularly well suited to learning from noisy, continuous, sensorimotor experience. Its

massively parallel architecture, local connectivity, and chemical signaling support forms of computation that differ markedly from traditional serial digital machines. If intelligence arose first in such a substrate, perhaps that is not an accident but a reflection of its affordances.

On the artificial side, hardware optimized for linear algebra and differentiable computation appears to be a good fit for current mainstream AI techniques, especially deep learning. Graphics processors and tensor accelerators implement operations like matrix multiplication with high efficiency, making them "natural" substrates for neural networks. They are not minimal in any formal sense, but they align well with the structure of widely used algorithms.

In contrast, a one-instruction set computer or a pure SKI combinator machine could host the same algorithms in principle, but they do not line up as well with the patterns of computation common in practical AI. They are less natural in the sense that they force awkward encodings and impose large overheads. Their minimality in primitive operations comes at the cost of being misaligned with the high-level structure of intelligent behavior as currently modeled.

This suggests a nuanced position. While intelligence may be substrate-independent in the strong sense that its defining patterns can be realized in many ways, some substrates can be more natural or congenial than others for particular forms of intelligence or particular engineering pathways. Minimal substrates illustrate that no special primitive is required, but richer substrates may, in practice, be closer to the grain of the phenomena we wish to capture. The choice of substrate thus shapes not only efficiency but also which forms of intelligence are easiest to express and explore.

9.4 Minimal languages, embodiment, and the meaning of artificial

Finally, the discussion of minimal languages intersects with questions of embodiment and the meaning of "artificial" in artificial intelligence. A purely formal view might treat an AI as just a program, describable as a bit string and executable on any universal machine. Yet actual intelligent agents, whether biological or artificial, exist in environments, interact with bodies, and operate under constraints. Their intelligence is not only a property of their internal computations but also of how those computations couple to the world.

Minimal formalisms often abstract away from this coupling. A NAND circuit or a lambda term is static until inputs are provided; its interaction with an environment is mediated by an interface that is usually left unspecified. Embodied AI, by contrast, emphasizes that intelligence arises from continuous feedback between an agent's internal processes and its surroundings. Sensors, actuators, and the physical dynamics of the body play a constitutive role.

In this light, minimal languages can be seen as describing only the internal computational core of an agent. The embodiment layer—motor control, perception, physical constraints—adds additional structure that may not be easily captured in the most austere formalisms. For

example, a robot's control system might be implementable in a minimal language, but the richness of its behavior depends on the physics of its body and environment, not just on its internal computation.

The word "artificial" complicates the picture further. If intelligence is a pattern that can be implemented on many substrates, then the distinction between artificial and natural intelligence may become less about what kind of computations are occurring and more about how they arise. Biological intelligence is the result of evolution and development; artificial intelligence is designed and engineered. Minimal computational languages show that, in principle, the same patterns could emerge in both contexts, but the pathways to them differ dramatically.

Minimality itself starts to look like a design aesthetic rather than a defining feature of artificial systems. Engineers may choose minimal cores for reasons of clarity, formal tractability, or elegance, but there is no reason to believe that nature or evolution would have done the same. Brains are not minimal in any obvious formal sense; they are shaped by constraints and contingencies that differ from those guiding hardware and software design.

Thus, minimal languages help clarify that intelligence does not require a particular substrate or a particular set of primitives. They underscore the implementation-independence of computational patterns. But when one turns to questions of embodiment and artificiality, other factors come into focus: how agents are embedded in the world, how their capacities develop, how they are constrained by physical law, and how their internal structures reflect those constraints. Minimal intelligence, in the philosophical sense, is not only about the smallest language that can host an AI, but also about the simplest conditions under which something can be said to think, act, and understand within a world.

10. Conclusion: Where the Lower Bound Really Lives

The journey from NAND to neural networks, from one-instruction machines to lambda calculus, has a consistent moral. At the level of pure computability, the bar for hosting artificial intelligence is remarkably low. Many different minimal formalisms, each defined by a tiny set of primitives and simple rules of composition, are powerful enough to express any algorithmic process, including those we associate with learning, planning, and reasoning. The lower bound for AI, in this sense, lives not in any special "intelligent" operation, but in universality plus sufficient resources.

Yet the same exploration also shows that this lower bound is layered. There is a logical floor, where NAND and its cousins suffice. There is an information-theoretic floor, where the shortest descriptions of intelligent behavior are measured. And there is a physical floor, where

thermodynamics, noise, and embodiment constrain what can actually be built. The minimal computational languages discussed in this paper illuminate one of these floors very sharply; they leave the others only partially in view.

10.1 Summary: NAND sufficiency and the landscape of minimal formalisms

At the most concrete level, NAND offers a hardware-oriented proof of sufficiency. Boolean algebra guarantees that NAND is functionally complete for combinational logic, and feedback constructions show that it can implement sequential circuits and memory elements. From there, adders, multiplexers, registers, program counters, and control units can be constructed. A complete CPU, with a finite instruction set and access to unbounded memory, can be realized entirely in terms of NAND gates and wiring.

This establishes that a purely NAND-based machine is Turing complete. Any program that runs on a conventional computer, including any AI algorithm expressed in a high-level language, can in principle be compiled down to this substrate. NAND thus serves as a canonical example of a single primitive operation that suffices for universal computation and, by extension, for AI.

Beyond NAND, the paper surveyed a broader landscape of minimal formalisms. One-instruction set computers such as SUBLEQ demonstrate that even at the level of machine code, a single carefully chosen instruction can be enough. Lambda calculus and SKI combinatory logic show that universality can arise from purely symbolic systems built from function abstraction, application, or a handful of combinators, with no explicit notion of memory or branching. Universal cellular automata illustrate that computation can be spatial and local, governed by a fixed update rule applied uniformly across a grid.

Taken together, these examples reveal that there is not one smallest language for AI, but many ways to be minimal. Some minimize the number of gates, some the number of instructions, some the syntactic constructs. All of them meet the same criterion: they can emulate any effective procedure given enough time and memory.

10.2 What minimal languages clarify about AI, and what they obscure

Minimal languages clarify at least three important points about artificial intelligence. First, they show decisively that AI does not require special-purpose primitives. There is no need for an “intelligence gate” or a built-in “learning instruction.” The ingredients of AI are patterns of computation that can be decomposed into standard operations: arithmetic, comparison, control flow, and storage. Once these are present in a universal system, there is nothing missing in principle.

Second, they sharpen the concept of substrate independence. If neural networks, search algorithms, and symbolic reasoning systems can all be implemented on a NAND-only CPU, a

SUBLEQ machine, or a lambda calculus interpreter, then intelligence is not a property of any specific implementation technology. It is a property of certain computational structures that can be realized in many ways. Minimal formalisms make this independence vivid by showing how far down the hierarchy of abstraction one can descend without losing expressive power.

Third, they provide a natural bridge to information-theoretic perspectives. Describing an AI system as a bit string and asking about its Kolmogorov complexity or its Minimum Description Length becomes more precise when a reference universal machine is fixed. Minimal languages supply such reference machines in an explicit and analytically tractable form.

At the same time, minimal languages obscure other crucial aspects of AI. They say nothing, by themselves, about efficiency, scalability, or energy usage. They abstract away from physical constraints like thermodynamic limits and noise. They do not directly illuminate how architectures exploit structure in data, how learning dynamics behave in high-dimensional spaces, or how agents are embedded in real environments. They also hide the human side of AI engineering: readability, composability, debugging, and the social process of building and maintaining complex systems.

In short, minimal languages clarify what is logically necessary and sufficient for AI, but they obscure much of what makes AI difficult, interesting, and consequential in practice.

10.3 Open questions at the intersection of computation, physics, and mind

The exploration of minimal substrates raises, rather than resolves, several open questions at the junction of computation theory, physics, and the philosophy of mind.

One set of questions concerns the physical cost of intelligence. Given a universal substrate such as a NAND-based machine or a cellular automaton, what are the fundamental energy and time requirements for implementing particular classes of intelligent behavior? Can one meaningfully define an energy-efficient or thermodynamically minimal intelligence, and if so, does that favor some substrates over others?

Another set concerns the descriptive complexity of minds. While Kolmogorov complexity suggests that any agent's behavior has a shortest generative description, it does not tell us how large that description is for realistic intelligences, or how it scales with environment richness. Does the minimal description length of an agent's policy correlate with intuitions about its sophistication? Can architecture design be guided by approximations to this complexity, beyond qualitative MDL reasoning?

There are also questions about naturalness and alignment between substrate and function. Certain substrates seem particularly suited to certain styles of intelligence, as seen with biological neural tissue or specialized AI accelerators. Is this merely an engineering contingency,

or does it reflect deeper relationships between the structure of environments and the structure of effective agents?

Finally, at the philosophical level, the existence of many minimal AI-capable substrates pushes on questions about consciousness and subjectivity. If the same computational pattern can be instantiated in neurons, silicon, or a cellular automaton, does that imply anything about the distribution of conscious experience, or is consciousness sensitive to substrate in ways that bare computation does not capture? Minimal formalisms remain neutral on this issue, but they sharpen the framing of the debate by isolating the purely computational core.

10.4 Future work: toward rigorous measures of AI substrate minimality

Looking forward, there are several directions in which the theory of minimal languages and AI could be developed more rigorously.

One direction is to define quantitative measures of substrate minimality that go beyond simple counts of primitives. Such measures might combine the size of the primitive set with the average description length of a representative family of AI algorithms, the overhead needed to simulate a standard reference architecture, or the asymptotic resource blowups for key computational patterns like gradient descent or tree search. This would turn the informal notion of “sparse but usable” into something more precise.

A second direction is to integrate physical constraints into these measures. Rather than treating all primitive operations as equal, one could weight them by energy cost, error susceptibility, or latency in a given physical realization. Substrate minimality would then be evaluated not only in terms of logical sufficiency but also in terms of thermodynamic and engineering efficiency for AI workloads.

A third direction is to connect minimal-language thinking with automated architecture search and meta-learning. If architectures can be seen as higher-level languages for describing models, then searching over architectures is akin to searching for languages that minimize description length and maximize performance on given tasks. Bringing MDL-style criteria into architecture search could help formalize why certain design patterns, such as convolution or attention, are so effective.

Finally, there is room for deeper conceptual work at the interface with philosophy. Minimal substrates provide clean case studies for debates about substrate independence, implementation identity, and the status of artificial minds. Clarifying these issues may require new frameworks that can talk simultaneously about computation, physical realization, and phenomenology without collapsing one into the other.

In all of these directions, the guiding insight remains the same. The lower bound for AI lives in universality, not in any exotic primitive. Minimal languages and substrates show that the space of possible implementations is vast and that intelligence, as a computational phenomenon, is remarkably portable. The challenge for future work is to understand how this abstract portability interacts with the gritty realities of physics, engineering, and experience, and to develop tools that let us navigate the space of substrates in principled, quantitatively grounded ways.

11. References

[1] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, Series 2, 42, 230–265.

[2] Church, A. (1936). An unsolvable problem of elementary number theory. American Journal of Mathematics, 58(2), 345–363.

[3] Kleene, S. C. (1952). Introduction to Metamathematics. North-Holland.

[4] Shannon, C. E. (1938). A symbolic analysis of relay and switching circuits. Transactions of the American Institute of Electrical Engineers, 57(12), 713–723.

[5] Minsky, M. L. (1967). Computation: Finite and Infinite Machines. Prentice-Hall.

[6] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). Introduction to Automata Theory, Languages, and Computation (2nd ed.). Addison-Wesley.

[7] Sipser, M. (2013). Introduction to the Theory of Computation (3rd ed.). Cengage Learning.

[8] Barendregt, H. P. (1984). The Lambda Calculus: Its Syntax and Semantics (revised ed.). North-Holland.

[9] Church, A. (1941). The Calculi of Lambda-Conversion. Princeton University Press.

[10] Curry, H. B., & Feys, R. (1958). Combinatory Logic, Volume I. North-Holland.

[11] Wolfram, S. (2002). A New Kind of Science. Wolfram Media.

[12] Cook, M. (2004). Universality in elementary cellular automata. Complex Systems, 15(1), 1–40.

[13] von Neumann, J. (1966). Theory of Self-Reproducing Automata (A. W. Burks, Ed.). University of Illinois Press.

[14] Li, M., & Vitányi, P. (2008). An Introduction to Kolmogorov Complexity and Its Applications (3rd ed.). Springer.

[15] Cover, T. M., & Thomas, J. A. (2006). Elements of Information Theory (2nd ed.). Wiley-Interscience.

[16] Rissanen, J. (1978). Modeling by shortest data description. Automatica, 14(5), 465–471.

[17] Rissanen, J. (1989). Stochastic Complexity in Statistical Inquiry. World Scientific.

[18] Grünwald, P. D. (2007). The Minimum Description Length Principle. MIT Press.

[19] Landauer, R. (1961). Irreversibility and heat generation in the computing process. IBM Journal of Research and Development, 5(3), 183–191.

[20] Bennett, C. H. (1973). Logical reversibility of computation. IBM Journal of Research and Development, 17(6), 525–532.

[21] Bennett, C. H. (1982). The thermodynamics of computation—a review. International Journal of Theoretical Physics, 21(12), 905–940.

[22] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

[23] Russell, S. J., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.

[24] Hutter, M. (2005). Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability. Springer.

[25] Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.

[26] Moore, E. F. (1956). Gedanken-experiments on sequential machines. In C. E. Shannon & J. McCarthy (Eds.), Automata Studies (pp. 129–153). Princeton University Press.

[27] Shannon, C. E., & McCarthy, J. (Eds.). (1956). Automata Studies. Princeton University Press.

[28] Arora, S., & Barak, B. (2009). Computational Complexity: A Modern Approach. Cambridge University Press.

[29] Aaronson, S. (2013). Quantum Computing Since Democritus. Cambridge University Press.

The author has published over 4,600 AI-assisted papers at: https://www.researchgate.net/profile/Douglas-Youvan/research . Google Scholar has indexed these preprints, and if you want a particular reference, the AI mode of a Google search (Gemini) has efficiently catalogued these preprints.