Fluency and Feasibility in Human Computing

Flyxion
Independent Researcher

November 2026

Abstract

A universal model of computation can in principle carry out any computable procedure, although a physical machine realizes it only within finite time and memory. This fact says little about whether a particular person can state the procedure precisely, run it within a useful period, or check what it returns. This essay examines that distance through input devices, command habits, programming languages, mobile assistants, and large calculations. It first separates input that transmits a symbol fixed by a known configuration from input that infers a probable one, and argues that this difference affects which motor skills can be relied upon. It then describes how recall, text expansion, shell scripts, and temporary key mappings move recurring effort into reusable structures, and how those structures can be lost when the systems supporting them change. Language-model services have brought many desktop tasks to the phone, where a sequence performed once may need little typing, while repeated work still calls for a stored procedure with access to the relevant files.

The essay next argues that general-purpose programming paradigms can share the same computational power while differing greatly in what they make easy to express and verify, and that this breadth shifts difficulty from construction toward the selection of worthwhile work. Estimates for a galactic gravity calculation and a recording from one hundred cameras show how briefly described tasks can exceed any practical schedule. Such tasks become approachable through decisions about what to leave out. Exact abbreviation and approximate reduction both lower cost but carry different obligations: the first must recover the intended instruction, whereas the second must preserve the answer to a stated question, exactly or within a stated tolerance. Natural-language interfaces reduce the cost of first expression while leaving meaning to be settled and results to be judged.

Keywords: human–computer interaction; text entry; end-user programming; expressiveness; computational complexity; approximation

1 Transmitted and Inferred Input

A physical keyboard and a gesture keyboard both produce text, but they arrive at it differently. When a key combination such as Control, Shift, and S is held, the machine receives a small set of discrete events. Pressing, releasing, holding, repeating, combining, and sequencing keys can each be told apart, and each can be assigned a meaning by an operating system, an application, or the operator. Let a be an action and κ the active configuration, meaning the layout, bindings, and settings in force. Input is called *transmitted* here when the result is a fixed function $f_\kappa(a)$ that the operator can know in advance. Some estimation still occurs beneath this level, since contact noise must be filtered before a press is registered, but under ordinary conditions the same action in the same configuration yields the same event.

Speech recognition and gesture typing are *inferred* input. A continuous signal g, whether sound or the path of a finger, is resolved into a word by a model with parameters θ that also consults the surrounding context c. In simplified form the chosen word is

$$\hat{w} = \arg\max_{w \in V} \; P_\theta(w \mid g, c), \tag{1}$$

where V is the vocabulary. The method succeeds because ordinary prose is highly redundant: context and word frequency compensate for an imprecise movement. It serves less well for an arbitrary filename, a mathematical expression, a shell command, an invented word, or an identifier in which capital letters matter, because the intended string may be improbable or absent from V.

Word-gesture keyboards, Swype among them, were nevertheless an important invention [10]. They turned a field of miniature buttons into something closer to a writing surface. A frequent word could become a practiced shape, produced without aiming separately at each letter, much like a handwritten glyph. The shape was familiar while its interpretation remained probabilistic, so an adapted dictionary or a competing word could change what an unchanged movement produced. When Swype was discontinued in 2018, its users lost more than an application. They lost an instrument on which a particular kind of skill had been built.

The touchscreen imposes further limits. A mouse separates positioning from activation: the pointer can rest over an object, reveal information about it, and move on without committing to anything. A finger on a conventional touchscreen couples the two closely and also covers the region it is acting upon. Multitouch permits simultaneous contacts, yet mainstream phone systems expose them mainly as a modest set of predefined gestures and seldom as a vocabulary of chords and modifiers that the operator can program. Part of this is physical, and part is a matter of design. Temporary modes, persistent modifier states, gesture sequences, and user-defined expansions are all technically conceivable. Keyboards on some platforms already provide modifier keys and cursor movement by sliding across the space bar, although each application still decides what it accepts.

Gestural input need not be probabilistic. Stenographic machines use simultaneous key combinations interpreted through an editable dictionary, and fluency comes from learned compression. Early stylus systems likewise explored simplified single strokes that could be recognized reliably and eventually produced without looking [5]. A phone keyboard might therefore offer two registers, one that infers words for prose and one in which pinned gestures always yield exactly what their owner assigned, with a held thumb or a mode key moving between them.

2 Habit and Recall

Skill with a computer accumulates over a long succession of environments. Command prompts on 386 and 486 machines, dual-boot installations, emulators such as DOSBox, virtual machines such as VirtualBox, the Windows Subsystem for Linux, and containers each answer the wish to have another working system within reach. Each can seem formidable at first. After enough use, creating a container or configuring remote access over SSH becomes routine, and much of the routine survives an interval of months. Particular commands change from one environment to another, while expectations about paths, options, configuration files, permissions, and error messages carry across.

Attempting to recall a command before looking it up contributes to this durability. Research on learning has found that retrieving material from memory can improve later retention more than studying it again [7]. Those experiments concerned verbal material and not system administration, so the application to commands is an analogy, though a plausible one. Looking things up remains sensible when an error would be costly. The point is that effortful recall is part of how a procedure turns from instructions that are followed into actions that are available.

3 Abbreviation, Scripts, and Composition

Earlier computing environments encouraged an economy of symbolic operations. Teletypes, slow connections, narrow terminals, restricted filename lengths, and printed output made each character consequential. Unix commands such as `ls`, `cp`, `mv`, and `rm` still show this economy, as do the abbreviated commands of the Ex editor and its descendants. Shell completion applies a related principle: the operator types only enough of a filename to distinguish it from its neighbors, and the machine recovers the rest from the names currently available. Brevity in these settings was more than a matter of style, since it answered the recurring cost of interaction.

Any long or tedious operation that recurs is a candidate for a name. A text expansion turns a few keystrokes into a phrase, a shell alias stands for a command, and a script gathers a sequence of commands into something that can be run as one. Let A be a set of short names, S a set of complete instructions, and

$$D_\kappa : A \to S \tag{2}$$

the expansion defined by configuration κ, possibly only on part of A. Let $s^* \in S$ be the instruction that the operator intends. An invocation of the abbreviation a is exact when $D_\kappa(a)$ is defined and

$$D_\kappa(a) = s^*. \tag{3}$$

Nothing in the instruction has then been discarded; the information needed to reconstruct it has been moved into the configuration. A deterministic expansion can still be wrong, because the operator may invoke the wrong abbreviation, the configuration may have changed, or the expansion may occur in an unintended context.

A script does more than save typing. It records the order of steps, accepts arguments so that one procedure applies to different files, and can test whether a step succeeded before beginning the next. The following fragment makes smaller copies of a directory of photographs, using the ImageMagick program, and leaves the originals in place.

```
#!/usr/bin/env bash
set -euo pipefail
source_dir=$1
mkdir -p "$source_dir/small"
shopt -s nullglob
for photo in "$source_dir"/*.jpg; do
    magick "$photo" -resize 1600x1600\> \
        "$source_dir/small/${photo##*/}"
done
```

Once saved and made executable, the file is a command in its own right and can be called by other scripts. Effort formerly spent on each occasion now resides in a description that can be read, corrected, and extended. The obligations move with it, since a mistake preserved in a script is repeated as faithfully as a correct step.

The interface can also be rebuilt for the duration of a single activity. During a long session of photograph editing, a tool such as AutoHotkey can assign frequent operations to the numeric keypad and release them afterward.

```
#Requires AutoHotkey v2.0
photoMode := false
F12:: {
    global photoMode
    photoMode := !photoMode
    ToolTip(photoMode ? "Photo mode" : "")
}
#HotIf photoMode && WinActive("ahk_exe editor.exe")
Numpad1::Send("^z")
Numpad2::Send("^s")
Numpad3::Send("{Right}")
Numpad4::Send("{Left}")
#HotIf
```

The program name and commands are placeholders for those of an actual editor, and the keypad names shown assume that Num Lock is on. The restriction to one window and the visible indicator matter, because a forgotten mode can send an instruction to the wrong place. The judgment about each photograph remains with the person editing it. What has been shortened is the mechanical route by which that judgment is carried out.

Shortcuts need not be planned in advance, because repeated inconvenience shows where they belong. A new one repays its cost when

$$n\,(t_{\mathrm{long}} - t_{\mathrm{short}}) > t_{\mathrm{setup}} + t_{\mathrm{upkeep}}, \tag{4}$$

where n is the expected number of future uses, t_{long} and t_{short} are the times for one use without and with it, t_{setup} is the time to create it, and t_{upkeep} includes repairing and relearning it. The inequality counts only time and omits benefits such as fewer errors. The accumulated saving from habits, configurations, scripts, and shared conventions may be called *interface capital*. It belongs partly to the relation between one operator and one system, and it can be stranded by

an incompatible update, a removed facility, a migration, or the withdrawal of a keyboard such as Swype.

4 Formal Languages as Media of Thought

For a practiced programmer, Bash or Python is often more than a code into which a finished thought is translated. A loop, a dictionary, or a pipeline can occur to the programmer as readily as a sentence pattern occurs to a speaker, and writing it can be how the intention acquires a definite form. Idiom develops as well, so that a program may be valid Python and still read as though it were Java in Python's spelling. The listener is unlike a human one. A person repairs a garbled sentence from context, whereas an interpreter rejects a malformed statement or carries out a well-formed one even when it was not what its author meant.

Tool builders have repeatedly tried to reduce the friction between intention and execution without requiring that machines understand ordinary speech. A terse failure message, or a crash that left nothing to search for, gave its reader little to act upon. A traceback that names files, lines, and the chain of calls improves matters. Structured shells let a file or process be handled as an object with named properties, sparing the programmer some extraction of columns from text. Xonsh combines Python with shell-style commands so that both are available at one prompt [9]. Other languages, COBOL, AppleScript, and Inform 7 among them, adopted English-like forms, but a sentence that looks like English must still be one the parser accepts.

Language-model systems alter this arrangement by inferring a formal procedure from an informal request. They reproduce the distinction between inferred and transmitted input at a larger scale. A request to delete old logs leaves open which files count as logs, how old they must be, and whether subdirectories are included. A sound division of labor lets the model propose a command and the shell execute exactly what is shown, with an opportunity to examine it between the two. Knowledge of the formal language remains valuable at that point, because a misplaced wildcard can look harmless in an English summary.

Language models joined to web applications have also made many tasks once associated with a desktop Linux environment available on a phone. A writer can request a draft, pass criticism from one system to another by copying and pasting, and obtain a typeset document with little use of a keyboard. A few taps and swipes can set off a large amount of work, since a short and loose request may be expanded into pages of text or code, although this expansion is inferred and may not come out the same way twice. The pipeline survives, with the clipboard serving somewhat as a pipe operated by hand and the person choosing what passes from one stage to the next. What has fallen sharply is the cost of producing and moving text. The need to decide what should be kept, and to check it, remains.

Such arrangements serve well for a sequence performed once. When each step requires further tapping, applying the sequence to a hundred files can mean repeating it a hundred times, whereas a script pays its setup cost once and then runs in batches. The boundary is not fixed by screen size: terminal applications such as Termux can run Bash scripts on Android, and an assistant with access to a code execution environment can write and run a batch operation on files supplied from a phone. Repeated work calls for access to the files and a durable place to keep the procedure, so that it can be inspected, adjusted, applied to many inputs, and run again. A desktop shell still provides these most directly, particularly when the files already

reside on that computer.

5 Equivalence and Difference Among Paradigms

Functional, procedural, object-oriented, and logic programming offer different ways of framing a problem, as algorithms, sequences of changes, interacting objects, or relations to be satisfied. General-purpose languages in these families can compute the same class of functions when idealized memory and time are granted. The statement does not extend to every formal language. A strictly total language requires each program to terminate, and thereby excludes some computable functions, including any interpreter for the whole of itself under the usual assumptions, in exchange for guarantees that unrestricted languages do not supply [8]. Proof assistants rely on restrictions of this kind in their logical cores.

Equal computational power also leaves large practical differences. Felleisen's account of expressiveness distinguishes features that can be translated by local rewriting from features whose imitation requires reorganizing a whole program [3]. Mutable state can be represented in a purely functional style, but it must then be passed through the computations that depend on it, either explicitly or through a structure such as a monad. Each paradigm makes certain things cheap to state and certain mistakes harder to commit. Learning several allows a habit acquired in one, such as writing functions without hidden effects, to be exercised in another that does not require it.

6 The Excess of Possibility

A small set of tools quickly exceeds anything its user could explore. Tools such as sed or Perl permit very general manipulation of text, Vim serves for immediate local changes and can automate them, and shell pipes join separate programs into new procedures. Beyond functional completeness, much additional syntax offers further ways of saying what can already be said. Convenience of this kind has real value, but it does not decide which programs deserve to exist.

Fluency therefore moves the principal difficulty from the question of how a thing could be done to the question of which things are creative, beautiful, interesting, salient, or practical. A familiar and limited set of tools settles many choices beforehand and leaves attention for the object being made. A cheap procedure can still be pointless, so selection cannot be reduced to saving effort. It resembles taste: a developing sense of what to build, what to simplify, and what to leave unmade.

7 Human Timescales

Three questions are easily confused. Computability asks whether any procedure can produce the answer. Complexity theory asks how required resources grow with the size of the input, and commonly treats polynomial growth as the rough boundary of tractability. Practical feasibility asks whether this procedure, on these resources, finishes soon enough to be of use. An exponential method may be instant on a small input, and a polynomial one may be hopeless on a large input. Long computations are sometimes worthwhile: the factorization of the 250-digit

challenge number RSA-250, announced in 2020, took roughly 2700 core-years, spread across many machines so that the elapsed time was far shorter [2]. A strictly serial chain of dependent operations, by contrast, cannot be made faster merely by adding processors.

Let P be a procedure, u its operator, and x a task. Let T_e be the time to write and debug P, T_r the time for one run, T_v the time to check one result, n the number of runs over which the writing is shared, and $H(u, x)$ the time the operator can spend per result. A rough condition for feasibility is

$$\frac{T_e(P, u)}{n} + T_r(P, x) + T_v(P, u, x) \leq H(u, x). \tag{5}$$

This is an average. The first result still bears the full cost of writing, the stages are assumed to follow one another, and memory, money, and energy are left aside. It nevertheless prevents execution time from being mistaken for the whole cost. A program that runs in milliseconds after hours of debugging may serve a single occasion worse than a slow one written in a minute. Exploration is especially sensitive to T_r: if one step takes seventeen hours, each conceptual error may cost a day before it becomes visible.

Consider a galaxy represented by $N = 10^{11}$ bodies. Direct calculation of gravity between every pair involves $N(N-1)/2 \approx 5 \times 10^{21}$ interactions at each time step. At an assumed rate of 10^{10} interactions per second, one step takes about sixteen thousand years. Barnes and Hut organized bodies in a tree so that a distant cluster can be treated approximately as one mass, with cost growing in proportion to $N \log N$ for suitable distributions [1]. If this required about a thousand interactions for each body, the total of 10^{14} would take a few hours at the same rate, ignoring tree construction, communication, and the considerable memory needed. Interactive study would still call for far fewer bodies.

A modest laboratory description leads to similar quantities. Suppose that one hundred cameras, each recording twelve million grayscale pixels at one byte apiece, observe an object ten times a second for an hour. There are $M = 1.2 \times 10^9$ pixel positions and $K = 36{,}000$ captures, or about 43 terabytes. Correlating every position with every other across the hour concerns about 7.2×10^{17} pairs and, at K multiply-add operations per pair, about 2.6×10^{22} operations. At an assumed rate of 10^{13} multiply-add operations per second (each is conventionally two floating-point operations), the arithmetic takes roughly eighty years, and the pairwise results at four bytes each occupy nearly three exabytes.

8 Omission as Method

Redundancy offers a way through. Neighboring pixels often vary together, and several cameras see the same surface. A low-rank approximation describes the $M \times K$ data matrix through a small number k of patterns, and randomized algorithms can compute such approximations in a few passes through the data [6]. With $k = 100$, one principal matrix multiplication requires about $MKk \approx 4.3 \times 10^{15}$ multiply-add operations, minutes at the assumed rate. Reading may dominate: at five gigabytes per second, a single pass over the recording takes more than two hours. Geometry offers another route, in which one moving surface is reconstructed and analyzed in place of the raw images, at the price of the reconstruction and its assumptions.

Each route differs in kind from an exact abbreviation. Let x be a complete state drawn from a domain Ω, let $q(x)$ be the answer to a question about it, and let $\rho(x)$ be a reduced

description. The reduction is exact for q on Ω if there is a function h such that

$$q(x) = h(\rho(x)) \quad \text{for all } x \in \Omega, \tag{6}$$

even though x itself may be unrecoverable from $\rho(x)$. Such a function exists precisely when

$$\rho(x) = \rho(y) \implies q(x) = q(y) \quad \text{for all } x, y \in \Omega. \tag{7}$$

The description may identify states that differ in many respects, provided that none of those differences changes the answer. This is the condition under which a many-to-one description loses nothing relevant to the question [4]. It guarantees that h exists, not that h can be computed, or computed within the time available, so exactness in this sense does not by itself secure feasibility. When exactness is unavailable, one may ask instead that

$$d(q(x),\, h(\rho(x))) \le \varepsilon \quad \text{for all } x \in \Omega, \tag{8}$$

where d measures the difference between answers and ε is the tolerated error. Randomized and statistical methods often guarantee less, such as meeting the bound with a stated probability, and some heuristics guarantee nothing and must be tested against cases whose answers are known.

An abbreviation recovers the instruction itself. An approximation recovers, at best, enough of an answer. A faint, brief, or highly localized event may disappear from a low-rank or sparsely sampled description while the dominant patterns look excellent. A defensible reduction therefore states which question it serves, which errors are acceptable, over what range of cases the claim holds, and how a violation would be noticed. Discovering these things is frequently harder than writing the code.

9 Conclusion

Computational universality establishes that a procedure can be expressed in principle. It does not establish that a given person can express it conveniently, execute it in time, or verify its result. Practical computing depends on stable habits, interfaces that can be reshaped, languages that make intentions precise, judgment about what is worth doing, and reductions whose consequences are understood. Fluency develops as recurring work is moved into reusable structures while its operator retains control over what they mean. Natural-language systems extend the movement by making first expression easier. They do not remove the need to settle what was meant, to decide whether it was worth doing, and to judge whether the returned result is reliable enough for its purpose.

References

[1] J. Barnes and P. Hut. A Hierarchical $O(N \log N)$ Force-Calculation Algorithm. *Nature*, 324:446–449, 1986.

[2] F. Boudot, P. Gaudry, A. Guillevic, N. Heninger, E. Thomé, and P. Zimmermann. Factorization of RSA-250. Announcement to the CADO-NFS Discussion List, February 28, 2020.

[3] M. Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, 17(1–3):35–75, 1991.

[4] Flyxion. Lossless Projection. Manuscript, 2026.

[5] D. Goldberg and C. Richardson. Touch-Typing With a Stylus. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, pages 80–87. ACM, 1993.

[6] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding Structure With Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Review*, 53(2):217–288, 2011.

[7] H. L. Roediger and J. D. Karpicke. Test-Enhanced Learning: Taking Memory Tests Improves Long-Term Retention. *Psychological Science*, 17(3):249–255, 2006.

[8] D. A. Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

[9] Xonsh Developers. The Xonsh Shell: Official Documentation. `https://xon.sh`.

[10] S. Zhai and P. O. Kristensson. The Word-Gesture Keyboard: Reimagining Keyboard Interaction. *Communications of the ACM*, 55(9):91–101, 2012.