Address Before Operator: From the Typewriter Carriage to Vim's Grammar

Flyxion
Independent Researcher

2026

Abstract

A text editor's grammar is downstream of what its hardware makes visible. This essay traces one line of that dependency from Rasmus Malling-Hansen's writing ball, whose earliest prototype was built into a table with a foot pedal driving the paper carriage, through the teletype and into ASCII, and a second, parallel line from ed's blind command language into vim's two grammars. The writing ball's single return button, which turned the carriage back and stepped the paper down in one stroke, was later split into two separate signals by the teleprinter and encoded as the adjacent ASCII control characters that still delimit the "lines" every line-oriented editor operates on; Nietzsche, who owned one of the machines and wrote a short verse about it, made a version of this essay's argument himself, in a letter to Heinrich Köselitz, about how the instrument was shaping his prose. Ed inherited that same encoded line and built a language around a constraint of its own: editing without a persistent display. Its single grammatical rule, address before command, sets the target first because nothing else on a blind terminal can. Vim inherited ed's grammar into its Ex mode essentially unchanged, address before command, while its normal-mode operators invert the order, command before address, once a persistent screen makes the address available for free. The claim advanced here is an interpretation of design affordance rather than a strict operational necessity: vim did not choose one grammar over the other, it kept both, sorted by whether the interface already supplies the location or the command still has to.

1 Before the Grammar: What the Machine Physically Did

Before any of this was syntax, it was furniture. The first machine to earn the name writing ball, invented by the Danish clergyman and educator Rasmus Malling-Hansen and first shown to the public in 1870, was not the sleek brass hemisphere museums now display; that came later. Its earliest working prototype, built between 1867 and 1869, was set into a table, and a foot pedal, not a key, drove the paper cylinder beneath the keys [5]. The production models that followed kept the pedal's job but moved it onto the machine itself: a hemisphere of over fifty spring-loaded typebars converged on a single printing point above a curved or cylindrical paper frame, and a button at the front left, pressed fully down, turned that frame concentrically back to the start of the line and stepped it down one line, a single mechanical action doing the work later split, in ASCII, into two separate signals, as the next section traces [6]. Malling-Hansen's machine reached the market four years before Remington's Sholes and Glidden typewriter, and a journalist covering the 1878 Paris World's Fair, comparing the two directly, judged the Danish machine more precise, more solid, and cheaper [9].

The writing ball solved the letter-case problem differently than the machines that followed it. Where later typewriters shared one typebar between an upper and a lower case and used a shift key to move the whole mechanism between them, physically tilting the carriage or dropping the type basket so a different point on the bar struck the ribbon, the writing ball's large hemisphere had room enough to give many characters their own dedicated key, trading a simpler keyboard for a more complex one rather than trading mechanical complexity for keyboard simplicity. Both are solutions to the same constraint, a fixed number of physical strike points against an alphabet's worth of symbols and cases; they just spend the cost in different places, one in the shift mechanism, the other in the keyboard's size. The typewriters that did use a shared typebar left a name behind them: pressing shift did not select a character electronically but physically shifted part of the machine, and holding it down for an entire capitalized word being tiring on the little finger, manufacturers added a shift lock, a mechanical catch holding the shifted position until released. When the carriage it once held in place disappeared into an electronic keyboard, the lock had nothing left to hold, and the mechanism survived as a toggle implemented in firmware, renamed Caps Lock.

The writing ball's most famous owner made the point this essay is built around explicitly, a century before ed existed. Friedrich Nietzsche, his eyesight failing, ordered a portable writing ball, serial number 125, directly from Malling-Hansen in 1882 and typed roughly sixty manuscripts on it before the machine was damaged in transit to Genoa and made worse by a mechanic unfamiliar with its repair [7]. When his friend Heinrich Köselitz remarked that the new instrument might be changing the shape of his prose, Nietzsche, typing his reply on the same machine, agreed: our writing tools are also working on our thoughts [3]. He also typed a short verse about the machine itself, dated Genoa, worth giving in full since Nietzsche's writing is long out of copyright:

> *Schreibkugel ist ein Ding gleich mir: von Eisen*
> *Und doch leicht zu verdrehn zumal auf Reisen.*
> *Geduld und Takt muss reichlich man besitzen*
> *Und feine Fingerchen, uns zu benützen.*

The writing ball is a thing like me, made of iron, yet easily thrown out of true, especially while traveling; patience and tact must be had in abundance, and fine fingers, to make use of us. Nietzsche is describing a mechanical fact, a precision instrument that a rough Italian trip could knock out of alignment, and a cognitive one in the same breath, in a letter that is itself evidence for the claim: the argument of this essay, that an interface's physical constraints leave a formal residue in whatever grammar gets built on top of it, is not a new argument. Nietzsche and Kittler made a version of it about prose style; what follows makes the same claim about command syntax.

2 The Physical Action Becomes a Code Point

The return key underwent its own version of this transformation, and it is worth tracing precisely because the result is still load-bearing in every line-oriented editor discussed below. The writing ball's return button already did two things in one mechanical stroke, turning the carriage back and stepping down a line; the standard three-row typewriters that dominated the next century typically split this into a single lever or, on the earliest Remington machines,

a treadle, but kept it as one physical action from the typist's point of view. Teleprinters split it again, this time into two separate transmitted signals, because a print head returning from the far right of a wide line takes real time to travel, and a machine reading its next character too soon could print into the middle of the return; sending carriage-return and line-feed as two distinct signals, in that order, let the second buy time for the first to finish [8]. When the American Standard Code for Information Interchange was fixed in 1963, it encoded exactly this pair of physical actions as two adjacent control characters, carriage return at decimal 13 and line feed at decimal 10, inherited directly from Teletype practice [1].

Unix, designed against terminals built on that same teletype lineage, kept only line feed as its line terminator, treating the carriage-return half of the pair as redundant once the terminal driver, rather than the application, was made responsible for any actual carriage motion; MS-DOS and later Windows kept both characters, preserving the full teletype pair for backward compatibility with the hardware convention it came from [2]. What matters for the rest of this essay is what that choice fixed permanently: the unit ed calls a line, and the unit every address in Sections 4 and 5 below selects, is not a neutral abstraction invented for text editing. It is the encoded residue of a physical carriage return, exactly as Caps Lock is the residue of a physically shifted type basket, and exactly as the writing ball's single return button is the residue later split, then re-encoded, into two adjacent ASCII characters. Ed's address grammar operates on units whose boundaries were fixed by hardware that no longer exists, in the same way its grammar itself, developed below, was fixed by a display that no longer constrains vim.

3 Two Grammars, One Vocabulary

Ed's entire command language reduces to one rule. A command may be preceded by an address or a comma-separated pair of addresses, and everything else about the command follows from what that address selects and what the one-letter verb does to it. Deleting lines three through five is `3,5d`. Inserting text before line one is `1i`. Printing every line matching a pattern is `g/re/p`. Substituting text on the current line is `s/re/repl/`. In every case the syntax is address, then command: the editor is told where before it is told what.

Vim contains two grammars descended from this one rule, not one grammar that replaced it. Its Ex mode, the colon-prefixed command line inherited through ex and vi, reproduces ed's address-first syntax with almost no alteration: `:3,5d`, `:g/re/p`, and `:s/re/repl/` are, modulo the leading colon, the same commands with the same address vocabulary and the same letters. Vim's normal-mode operator grammar, by contrast, inverts the order. The central rule of normal-mode editing is `[count]operator{motion or text object}`: `d}` deletes to the end of the paragraph, `dw` deletes to the start of the next word, `ciw` changes the word under the cursor. The operator, a lone letter exactly as in ed, comes first; the motion or text object, which supplies the address the operator will act on, comes second. Two grammars, the same underlying vocabulary of address and command, composed in opposite serial orders.

4 Ed's Address Vocabulary, Verbatim in Ex

Ed supports a small, closed set of address forms, and every one of them survives into vim's Ex syntax essentially unchanged. A bare line number is an address. The special symbol `$` addresses the last line of the buffer, and `.` addresses the current line. Relative addresses, `+N`

and `-N`, move forward or backward from the current line by a fixed count. The comma, or equivalently `%`, addresses the whole buffer, each side interpreted relative to the current line as it stood before the command ran. The semicolon is not simply a synonym for "current line to end"; it is a range separator that resets the current line to its left-hand address before the right-hand address is evaluated, which is what lets a semicolon-joined address anchor a following relative offset or search to a freshly chosen starting point rather than to wherever the buffer happened to be. Used bare, with nothing on either side, this reduces to the equivalent of `.;$`, which is where the "current line to the end" reading comes from, but that is the bare case, not the general rule. A slash-delimited pattern, `/re/`, addresses the next line matching a regular expression, and a question-mark-delimited pattern searches backward. Ed also supports marks: the `k` command labels the current or addressed line with a lowercase letter, and `'x` then addresses whatever line still carries that label. Vim's Ex range syntax reproduces essentially all of this: `$`, `.`, `+N`, `-N`, `%`, `/re/`, `?re?`, and named marks (`'a,'b`) all mean what they mean in ed, extended by only one genuinely new address form, the visual-selection range (`:'<,'>`), which names a position a full-screen editor can record that a line-oriented one, having no selection to begin with, has no occasion to.

Ed's addressing discipline exists for a specific reason: it is meant to move the burden of remembering which lines are affected out of the operator's head and into the command itself, before that detail has a chance to be forgotten in the course of deciding what to do with it. The address is written down first because it is the part most likely to be lost if it is not.

5 The Global Command as Vim's `:g`, Unchanged

Ed's global command, `g/re/cmd`, applies an arbitrary command to every line matching a pattern, and its complement, `v/re/cmd`, applies a command to every line that does *not* match. Both commands take any ed command as their argument, not merely `p` or `d`; a substitution, a move, or a sequence of commands separated by newlines can all follow the pattern delimiter. Vim's `:g` and `:v` are this command with no meaningful change: `:g/re/d` deletes every matching line, and the common idiom of running a macro across every matching line, `:g/re/normal @q`, is ed's global command applied to an arbitrary compound action in exactly the sense the original design already permitted. Substitution follows the same pattern. Ed's `s/re/repl/` supports parenthesized subexpressions recalled by backreference, `\1` through `\9`, and a trailing `g` flag to replace every match on a line rather than only the first; vim's `:s` takes the identical syntax, backreferences and trailing flag included. Where Ex is concerned, there is very little vim actually changed.

6 Why the Order Flipped

The reason normal-mode operators invert ed's order is not a matter of taste. It tracks a constraint ed's grammar was built around and vim's was built without, though the argument is best read as an account of interaction design rather than a claim about what ed's implementation strictly requires line by line. Ed does maintain a current-line pointer internally, and a compound command such as `3,5d` can run to completion without printing any of the three lines it touches; nothing in ed's machinery forces a print before every action. What the grammar tracks instead is what the *user* has access to, not what the program internally holds.

Ed edits blind. Historically run against a printing terminal with no persistent display, ed keeps no standing view of the buffer on the user's side of the interaction; the only way a user sees a line is to address it, at which point ed prints that line and nothing else, or, for a bare address entered alone, prints it by default. Setting an address is therefore, for the person typing, not merely a targeting step prior to editing, it is very often the only available way to re-establish where they are, since nothing else on the terminal retains that information for them between commands. This is the sense in which address comes first: not that ed cannot compute without it, but that the user cannot safely choose what to do next without paying, in commands, for information a persistent screen would have given for free. The grammar is address-first because, for ed's user, orientation and targeting are typically the same request, even when they are not the same operation inside ed itself.

Vim, and vi before it, removed exactly this constraint for whatever is currently on screen. A full-screen editor maintains a persistent view of the buffer and the cursor's position within it at every moment, at no marginal cost to the user; unlike ed, vim does not need a command to tell the user where they are, because the screen already does. Once orientation is free for the user, not merely available in principle to the program, the part of ed's grammar that existed to purchase that orientation, the leading address, becomes optional for anything already visible, and the grammar is free to lead with the action instead. This is what `[count]operator{motion}` is: a language that no longer spends its first token re-establishing location for the user, because the user's location was never lost, and can instead spend that token declaring intent. The motion that follows still supplies an address in exactly ed's sense, a range of text the operator will act on, but it now does so as confirmation of scope rather than as prior orientation, and in visual mode that confirmation is even visible before the operator commits, a capability ed's blind terminal could not have supported regardless of how its grammar had been arranged.

7 What Vim Kept as Ed's Grammar Rather Than Inverting It

This also explains why Ex mode was not folded into the operator grammar along with everything else. Ex commands routinely target text the user is not currently looking at: `:50,80d` deletes fifty lines without the cursor ever visiting them, exactly as the equivalent ed command would. For a target outside the visible buffer, the screen supplies no orientation at all, and the situation is structurally identical to ed's: nothing exists yet to fix location except the address itself, so the address has to come first. Vim's two grammars are not a historical inconsistency left over from incomplete modernization. They are sorted by a single variable, whether the editor's persistent view already supplies the address or not, and each grammar is address-first or operator-first for the same reason ed's single grammar was address-first throughout: the order tracks who is doing the work of locating the target, the screen or the command.

8 Modality Before Vim

Vim's division between normal mode and insert mode is also not a novel addition inherited from nowhere. Ed already distinguishes a command mode, in which addresses are set and one-letter verbs are issued, from an input mode, entered by `a` (append), `i` (insert before), or `c` (change), and exited by a lone period on its own line. This is the same modal split vim formalizes as normal and insert mode, under different names and a different exit key. What ed's

design has no antecedent for is visual mode, and for the reason already given: visual selection requires a persistent, extensible view of the buffer to highlight against, a capability that does not exist for an editor with no standing display. Visual mode is the one mode vim could add only after it had already removed the constraint that shaped the rest of ed's grammar.

9 Undo: Where Vim Broke From Ed Rather Than Reordering It

Not every difference between ed and vim is a reordering of the same primitives, and undo is the clearest exception. Ed's undo, `u`, is a single toggle rather than a history: running `u` reverses the previous command, and running `u` again reverses the undo itself, so that undo behaves as an involution over one remembered state rather than as a stack. Vim's undo is a persistent, branching tree, every change recorded as a node with multiple possible futures, navigable not only by repeated undo and redo but by explicit movement through alternate branches. This is not the address-first grammar rearranged into an operator-first one; it is a bounded mechanism replaced by an effectively unbounded one, and it belongs in this comparison as the reminder that not every gap between ed and vim reduces to the visibility argument developed above. Some of what vim added, it simply added.

10 Conclusion

The pattern repeats at every layer this essay has traced. A typewriter's shift key named a literal, physical shift, and when the physical constraint that gave the key its name disappeared, the name and a toggled version of the behavior survived it as Caps Lock. A typewriter's carriage return named a literal return of a literal carriage, and when that hardware disappeared into a teleprinter and then into ASCII, the two physical actions it once required survived as two adjacent control characters, still fixing the boundary of what every line-oriented editor in this essay calls a line. Ed's address-first grammar is offered here as a case of the same pattern one level up: not a hardware residue but an interaction residue, a grammar shaped by what a blind terminal could not show a user, structured so that the token most likely to be needed and least likely to be re-derivable, the address, came first. This is a design interpretation of why the grammar reads the way it does, not a claim that ed's internal machinery uniquely determined that ordering; other tellings of the same history are available, and the visibility account offered here is meant to be judged as an explanation of interaction design and grammatical affordance, not as a strict operational proof.

Read that way, vim's grammar was not a departure from ed's so much as the same account playing out for a second time under a different constraint. Ex mode keeps ed's order because it still routinely edits what is not on screen, where the old constraint still holds. Normal-mode operators invert that order because for anything already on screen, the address no longer needs to be purchased first, since the persistent display already supplies it. Vim did not choose a grammar; it inherited a fossil, the address-first token order, exactly as it inherited another fossil, the line itself, from a teletype that returned its carriage a century before anyone typed `dw`.

References

[1] American Standards Association. *American Standard Code for Information Interchange*, ASA X3.4-1963. New York: American Standards Association, 1963.

[2] "CR LF vs LF vs CR: Understanding Line Break Differences." codegenes.net. Accessed August 2026.

[3] Kittler, Friedrich. *Gramophone, Film, Typewriter*. Translated by Geoffrey Winthrop-Young and Michael Wutz. Stanford: Stanford University Press, 1999.

[4] Lucas, Michael W. *Ed Mastery*. Detroit: Tilted Windmill Press, 2018.

[5] "The First Prototype." The International Rasmus Malling-Hansen Society. Accessed August 2026.

[6] "The Writing Ball." The International Rasmus Malling-Hansen Society. Accessed August 2026.

[7] Blinderman, Ilia. "Discover Friedrich Nietzsche's Typewriter, the Curious 'Malling-Hansen Writing Ball' (Circa 1881)." Open Culture, November 2022.

[8] Kennard, Adrian (RevK). "CR+LF Has a Long History." *RevK's Ramblings* (blog), February 2022.

[9] "Hansen's Writing Ball, also known as Malling-Hansen Writing Ball." Science Museum Group Collection. Accessed August 2026.