Variable

Derek Robinson

To be is to be the value of a bound variable.
—WILLARD VAN ORMAN QUINE[1]

You can be anything this time around.
—TIMOTHY LEARY[2]

There is a distinction to be made between the variables employed by programmers and those employed by scientists, engineers, and mathematicians. Not that one can't straightforwardly write a program that uses computer-type variables to implement statistical algorithms. Nor is it hard to find a general logical definition good for both types. But it would not reveal the pragmatic, historical, and subcultural reasons why the word "variable" means different things to the programmer and the statistician (even if the latter's data analysis is likely performed with software written by the former). The root of the difference is that a programmer's variables are implemented on a computer, which means they must concretely exist in a computer's memory, in accordance with whose concreteness they must be named, ordered, addressed, listed, linked, counted, serialized, unserialized, encoded, decoded, raveled, and unraveled; how this happens bears little resemblance to algebraic symbols scratched on a chalkboard.

The programmer's variable is a kind of box; its name is the label written on the lid. To open the box, accomplished by the magical act of reciting its name in a prepared context, is to be granted access to what has been put "inside" it: the variable's value—one datum. Or say, what it denotes, what it "means," under a hugely impoverished notion of meaning that analytical philosophers spent much of the past century trying to shoehorn thought and language into. Cavils aside, it's in good part due to their efforts that there appeared in that century's middle third, the new science of computation.

A variable is a box stripped of sides, top, and bottom, abstracted away from geometry and physics, of no especial size or shape or color nor situated—so far as the programmer who conjures it needs to know or worry about—in any particular place. It's like there's always a spare pocket available any time there's something to be kept track of, and all it costs is to think up a name for it. (And then to remember what the name was; sadly not always so easy.) The passed

buck of reference, the regressus of signs, begins and ends in the blank affectless fact of the unfilled vessel, an empty signifier that awaits only assignment to contain a content. (In the upside-down tree-universe of Lisp, all termini point to "NIL.")

High-level computer languages relieve programmers of worrying about where values are kept in the computer's address space or how to liberate the locations they've occupied when they're no longer needed (this is done with a bit of legerdemain called garbage collection). In reality the variable is situated in a reserved area of physical memory called the Symbol Table. What is recorded in the Symbol Table is just the variable's name, paired with a pointer (a number understood as an address), which points to the location of some other cell that's allocated on demand from a heap of memory locations not currently claimed. Since all this takes place in a computer, naturally there are further layers of indirection and obliqueness between how a program accesses the variable's value and its extra-symbolic physical existence as an elaborate roundelay of trapped charges in doped silicon or mottles of switchable ferromagnetic domains on a spinning metal oxide-coated plastic disk.

The variable's role is as an index that points to something, somewhere. C. S. Peirce, grandfather of semiotics, once defined a sign as "a lesser that contains a greater."[3] Like a magical Arabian Nights tent, it appears bigger on the inside than its outside. One hears an echo of Turing's poser: "How can 2.5 kilograms of grey-pink porridge contain a whole universe?" (A hint: The finger points out of the dictionary.) A variable is a marker, a token, or placeholder staking out a position within a formal conceptual scheme. As Alan Kay[4] remarked, "The fundamental meaning of a mark is that it's there." An empty slot awaiting instantiation by being "bound" to a specific value, to be provided by someone's fingers at keyboard and mouse, or by some sensed, measured, electronically amplified, transduced, encoded alteration in the fabric of things happening elsewhere.

Some variables don't vary. A "constant" is a mnemonic stand-in conscripted simply because names are easier for people to remember and recognize than numbers. At bottom this is what any variable is: a name standing for a number that is interpreted as an address that indexes a memory location where a program is directed to read or write a sequence of bits. Electronic sensors attached to a computer are de facto variables registering external events in a set-aside range of addresses that act as portholes to view sampled digital representations of the changing voltages provided by the sensor.

In the Forth programming language, variables don't even need names. They can be values placed on top of a data stack as arguments to functions that apply operations to them and leave the results on top of the stack as arguments for subsequent functions. The necessity to name is here obviated by the specificity of place. (Forth has named variables too, but to actually use them is regarded as unsporting.) The Unix operating system has its own unnamed variables, called "pipes," for chaining together sequences of code, turning outputs into inputs, to engineer ad hoc assembly-lines of textual filters and transformers. It is this brilliant concept to which Unix owes much of its enduring success.[5]

An especially important use of variables is as arguments passed to a function subroutine. Instances of argument names found in the function body will be automatically replaced by the values of the variables that were provided when the function was invoked. Instances of argument names occurring within a function's scope act like pronouns referring to the place and time in the executing program where the arguments were last assigned values. They are pseudonyms, aliases, trails of breadcrumbs that point back up the "scope chain" of nested execution contexts. (A function "A" called from another function "B" will acquire any variable bindings found in the scope of B; likewise if B was itself called from a function "C," the latter's bindings become a tertiary part of the context of A.) In object-oriented languages there is a special argument or keyword named "this" or "self," which is used within class definitions to enable object instances at runtime to reference themselves and their internal states.[6]

The single most critical constraint on a variable's use is that it, and its every instance, must be uniquely determined in the context or "namespace" of its application, if it is to serve naming's ambition of unambiguous indication. This isn't as uncomplicated as it might seem. Namespaces are easily entangled, and before too long even 64 bits of internet addressing (allowing for 2^{64} or some 18 sextillion different designations) won't suffice to insure uniqueness. (Bruce Sterling is good on the implications of this stuff, and Mark Tansey has made a nice picture.[7]) However all that turns out, beyond the onomastic imperative of having to be uniquely determined within a context, a variable can denote, refer to or stand in place of anything that people are capable of apprehending, conceiving, and representing as a "thing."

Pronouncing upon the thingness of things has historically been considered the special preserve of philosophers, but programmers, being the practical engineering types that they are, simply had to get on with the job. The things represented in software in one way or another all ultimately reduce to patterns

of series of on-and-off switches, zeros and ones. No bit-pattern can represent anything without a program to interpret it. The meanings plied through natural language may, they say, be subject to the drift and swerve of an indefinitely deferred semiosis, but software's hermeneutic regress must finally bottom out. It's interpreters all the way down—then it's just bits.[8]

Under the hood, variables are arranged so that a specific pattern of 0s and 1s can be interpreted as a character string (and then as a word, or as several) in one context, a series of numbers, part of a picture, or maybe some music in another context. All of these pieces of information can be connected with some person, some object, or some more abstract category, and stored in a database somewhere. Ultimately they're all bits, and what software does is make sure that what one expects to find when one asks for something, and what one does find are one and the same. (Deliberately or accidentally incurred or induced violations are collected and swapped by connoiseurs of "glitch art" and "data bending.")[9]

Some things are fairly easily resolved. Numbers, still software's main stock in trade, are in the computer usually as integers (counting numbers, without decimal points) from a range between a fixed minimum and maximum (e.g., the 256 counting numbers from –128 to +127) or they are "floating point" numbers—a type of scientific notation (with exponents and mantissas) for representing non-integer values (with decimal points), which can be much larger or much smaller than integers. Alphanumeric characters have several different UTF-standardized 8-, 16-, or 32-bit-long character codes for specifying any graphic symbol used in any human language.

In the grand architectural design of Sir Tim Berners-Lee's Semantic Web, the bottomless puddle of the thingness of things is neatly sidestepped by dictating that things referenced must have URIs ("Uniform Resource Indicators," like web addresses). As long as URIs can be resolved into properly formatted truthful representations of information that people care to assert and are willing to stand by then automated proof procedures can be applied to them. Presumably, at the terminal node of the implied indefinitely extended and ramifying series of assertions asseverating the trustworthiness of other asseverations, we shall arrive at a planet-sized AI and either all our troubles are over, or they've just begun.[10]

The recent rise of markup languages[11] like HTML, CSS, XML, XSL, or SVG is recognition that in many applications, once the data have been properly set up, the ordinary kind of programming that relies on IF-THEN conditions to

alter execution flow isn't much needed. The data organization can look after the heavy lifting. Markup languages conform to the abstract data type known as "trees," branching geneologies whose member "nodes" (which can also be trees) are accessed via parent and sibling relations. Trees resemble the table of contents in a book. They are usually implemented using "list" data structures, although how these lists are implemented under the hood isn't important, as long as the lists behave like lists so that trees (and other things) created out of lists will behave like trees (or the other things).[12]

Data structures are compound, multicellular super-variables. Their purpose is to make it easy to arrange logical aggregations of data in ways that make it easy to carry out complex operations on their members. Apart from lists, whose cells can be grown and pruned and grafted in near-organic profusion, core data structures provided in most programming languages include character strings, linear arrays indexed by the counting numbers (used to make 2-D or higher dimensional data tables), and associative arrays: look-up tables whose cells are indexed with arbitrary symbols as the keys (internally turned into addresses by a hashing function,[13] or stuffed into lexicographic trees perhaps). The devil's in the details. Get the data structures right—picture and populate them, imagine traversals and topologies, strike a truce between redundancy and compression, cut a deal with the coder's old familiar foes of Time and Space, "solve et coagula," and mind the gap—and everything else will follow.

If computers can be made to agree on how data shall be represented and interpreted, encoded and decoded, then data can be shared between them the way audio, video, and text files are shared, and many different programs written in different languages running on different computer platforms can cooperatively behave as one very large distributed computer running one very large distributed program. The web is such a thing, and has gradually (if one can call the delerious growth of the past ten years gradual) been awaking to the fact. Mundane attention to marshaling and unmarshaling complex data structures in accordance with commonly agreed dialects and schemas (provision of which is the purpose of the Extensible Markup Language, XML, whose authors had the foresight to see that a data format for specifying data formats would be a good idea) is already rewriting the conduct of commercial life. A spirit of openness and peer collaboration is blowing even through hidebound proprietary holdouts like academic publishing; we await Silent Tristero's Empire and the Brittanica's demise.[14]

Notes

1. W. V. O. Quine (1939), "Designation and Existence." This phrase ("To be is to be the value of a bound variable") became a motto of Quine's, and through him, of mid-century Anglo-American analytical philosophy generally. (Reprinted in H. Feigl, and W. Sellars, *Readings in Philosophical Analysis.*

2. Dr. Timothy Leary, *You Can be Anyone This Time Around*.

3. For a summary of C. S. Peirce's philosophy of the sign, see Umberto Eco's *Semiotics and the Philosophy of Language*.

4. Alan Kay coined the term "object-oriented," headed the Learning Systems Group at Xerox PARC in the 1970s (which developed the now ubiquitous bit-mapped graphical desktop metaphor), invented the "Dynabook," and was the model for (obscure computer geek trivia alert) the Jeff Bridges video game programmer hero in the 1982 Disney film "Tron" (Kay's wife wrote the screenplay).

5. For Forth, see Leo Brodie's *Thinking Forth*, widely regarded as one of the best books about programming for anyone who programs in any language; a free PDF of the 2004 revision is available at the author's website. The Unix philosophy is summarized by Doug McIlroy (inventor of pipes) as follows: 1. Write programs that do one thing well; 2. Write programs that work together; 3. Use text streams as a universal interface.

6. For more information on scope, binding, and reference, see Harold Abelson, Gerald Jay Sussman, *The Structure and Interpretation of Computer Programs*. (A free online version can be found at the book's MIT Press website.)

7. Brian Cantwell Smith's *On the Origin of Objects* plumbs software's ontology very deeply and very densely (however it's only recommended for people not put off by infinite towers of procedural self-reflection).

8. Bruce Sterling would be the well-known science fiction writer, astute cognizer of past and present trends, peripatetic blogger, afficionado and sometime teacher of contemporary design. Recently he authored a book, *Shaping Things,* about "spimes," his neologism for a new category of post-industrially fabricated semi-software objects. Mark Tansey paints large monochromatic post-modern puzzle pictures in the high style of mid-twentieth-century illustration art. The painting referred to shows the crouching figure of (we assume) an archaeologist, bent over a small object, likely a

rock, in a desert landscape that contains many widely scattered small rocks. It has the enigmatic title, "Alain Robbe-Grillet Cleansing Everything in Sight."

9. See "Glitch," this volume.

10. Dieter Fensel, et al., *Spinning the Semantic Web*. An authoritative and up-to-date source is the World Wide Web Consortium: http://www.w3.org/.

11. Markup languages like XML acronymically descend from a typesetting language for IBM computer manuals called SGML, dating from a time (circa 1966) when IBM stood second only to the Jehovah's Witnesses as the world's biggest publisher of print materials. See Yuri Rubinsky, *SGML on the Web*.

12. John McCarthy, *LISP 1.5 Programmer's Manual*. For non-tree data structures implemented using lists, see Ivan Sutherland's *Sketchpad: A Man-Machine Graphical Communication System*—this was the first object-oriented program, the first computer aided design program, and the first "constraints-based" programming system. Utterly revolutionary at the time, it still rewards a look. In 2003 an electronic edition was released on the web.

13. Hash functions are numerical functions for mapping arbitrary character data regarded as numbers to pseudo-random addresses within a predefined range. Their great virtue is constant-time access, unlike tree-based structures. The data stored in hash-tables are (obviously) unordered, however.

14. Jon Willinsky, *The Access Principle: The Case for Open Access to Research and Scholarship*. Silent Tristero is implicated in the secret sixteenth-century postal service around whose continued existence or lack thereof the plot of Thomas Pynchon's novel *The Crying of Lot 49* revolves; elements of Pynchon's baroque conspiracy are borrowed from the Rosicrucian Brotherhood, an actual sixteenth-century conspiracy whose Invisible College perhaps only existed as carefully planted and cultivated rumors. (A mailing list of the name is frequented by white-hatted hacker types; with luck and unbending diligence in the pursuit of the art an invitation one day may arrive in your mailbox.)