15. See, for example, MacKenzie & Wajcman, *The Social Shaping of Technology.*

16. Matti Tedre et al., "Ethnocomputing: ICT in Social and Cultural Context."

17. MacKenzie & Wajcman, *The Social Shaping of Technology.*

18. Everett M. Rogers, *Diffusion of Innovations.*

19. Ibid.

20. See Ron Eglash, "Geometric Algorithms in Mangbetu Design," and Henning Bruhn, "Periodical States and Marching Groups in a Closed Owari."

21. Steve Heims, *The Cybernetics Group.*

Function

Derek Robinson

> A word is a box containing words.
> —GERTRUDE STEIN[1]

A function in programming is a self-contained section of code (one still comes across the term "subroutine," which is the same thing) that is laid out in a standard way to enable deployment and re-use at any number of different points within a program. It's a way of minimizing the duplication of intellectual effort, of making things routine, and as Alfred North Whitehead remarked, "Civilization advances by extending the number of important operations which we can perform without thinking about them."[2]

Functions are usually small and limited to performing a single task. They are active, they do things to things. Some typical examples of functions would be arithmetic operators like "plus," "times," and "square root," which can be combined with other arithmetic operations to compose expressions. If they might be useful in the future, these expressions can be named and turned into functions. Programmers will often keep personal files of utility functions for importing into projects; collections of greater breadth and size are made into libraries and maintained in repositories for use by other coders. It wasn't so long ago that libraries of machine code subroutines, with a light dusting of syntactic sugar, formed the basis of the first high level computer languages.[3]

Programming is a civic-minded activity. Politeness counts. Intense thought is expended in the hope that others, including most importantly one's future self, will not have to keep repeating the same tired phrases again and again. We try to be smart about parameterizing and abstracting, about dignifying as Variables those parts of things that vary, and as Functions the parts that do not, and which are to this degree redundant, vulnerable to automation, ripe for refactoring or removal. The activity of programming, like Jean Tinguely's famous self-destroying automaton ("Homage to New York," 1960), occupies the peculiar position, part teleological and part topological, of existing, ultimately, to obviate its own existence. (Q: "If computers are so smart, why don't they program themselves?" A: "Somebody would first have to write the program, and no-one has yet been that smart.")

When defining a function, there is some sort of preamble establishing its identity (usually a name, although sometimes not) and declaring any arguments or parameters that it will require. Something like "`function defunknose (x,y)`"—`defunknose` here being the name and x and y the arguments—followed by the function's "body," the block of code that actually carries out the computation the function was designed to perform. When later this function is called (by invoking "`defunknose (5,6)`" for example) each instance of an argument found in the function's body gets replaced by its corresponding value. In general, calling a function with different argument values produces different results. In the more sophisticated languages like Lisp or JavaScript, functions can be passed as arguments to functions (it might well be an anonymous "lambda" function that is passed). Finally it is customary (but not obligatory) for functions to return results to their callers. The code that invokes a function should have no reason to care how the result was produced.

A function's definition is a symbolic expression built up recursively from previously defined functions. The regressus of expressions composed of functions whose definitions are expressions composed of functions ultimately bottoms out in a small and irreducible set of atomic elements, which we may call the "axioms" or "ground truths" of the symbol system. In a computer these are the CPU's op-codes, its hardwired instruction set. In the system of arithmetic they would be the primitives "identity" and "successor," from which the four basic arithmetic operations can be derived and back into which they can be reduced. Such radical atomism was a favorite pastime of analytical philosophers of the mid-twentieth century, prefiguring the development of electronic giant brains designed to tirelessly carry out just this sort of task (which our little

human brains have difficulty keeping straight). (This is why writing software is so hard.)

Functional Programming

Functional Programming is an approach to programming and programming language design that uses only functions. It abjures any assignment of values to variables on the grounds that this can lead to unexpected side effects and thus compromise correct execution of programs. A function ought not, according to this philosophy, affect anything outside its scope; consequences shall owe only to results returned, and the only proper way to interact with a function is by means of the values passed to it as arguments when the function is invoked.

The first functional programming language was GEORGE, created in 1957 by Charles Hamblin for use on DEUCE, an early Australian computer. (Everything was upper case in those days.) The design was termed a zero-address architecture, because no memory was allocated for named, persistent variables; thus no symbol table was needed either. Any argument values needed by a function were accessed though a special dynamically growing and shrinking range of addresses called the "stack." (Imagine a stack of plates: the last plate added is the first removed.) A function could count on its arguments having been the last things pushed onto the stack before it was called; a stack pointer kept track of the current "top" cell as data were added to and removed from it. All calculations used the stack to store intermediate results, and the final result would be left on top of the stack as an argument for the next function in line. GEORGE programs used Reverse Polish Notation, a strange-looking syntax where operands precede their operators. Today's programming languages will often translate their code into RPN internally, and use a data stack for expression evaluation. Again, functions are recursively constructed symbolic expressions, and stacks are essential to their unraveling.[4]

Purely functional programs, despite or because of their elegant construction, are rarely found outside computer science textbooks. Most programming jobs involve states of affairs and making changes thereto conditional thereupon, but functions of the purer stripe don't acknowledge the concept of "memory"—for them there is only a continual process of transformation. It's very Zen, very committed, very macrobiotic. A function-ish style of programming, on the other hand, is encouraged in languages like Lisp, Forth, or JavaScript; it is empirically, programmer-lines-of-code-measurably a very productive way

to realize interesting and useful things in software. It's about writing many little functions that you then get to reuse inside the definitions of not yet defined little functions, and so on, and so on, bootstrapping one's way up a personal tower of metalinguistic abstraction until at the very top there is perched one final function: the program itself. (Think of a bathtub full of mousetraps, and yourself poised there, ping-pong ball in hand. Think cascades, fusillades, think detonations of denotations. Now let go, let fly.)

Functions as Mappings

But real mathematical functions aren't executable subroutines. A function is an ideal abstract consensual cerebration, and the code a programmer commits is only one out of indefinitely many possible materializations, each a pale sublunary reflection of the ideal. A function proper is propaedeutic, telling how the thing should behave, giving the theory but not concerning itself with how it is to be implemented. The "real" sine function, for example, defined over the real numbers, would require infinite-precision arithmetic—demanding an infinite supply of memory to inscribe its unscrolling digits, and asking all eternity for its satisfaction.

Our familiarity with functions like the sine curve shouldn't get in the way of a more general, modern conception of functions as mappings. Functions as understood by programmers are pretty close to the modern idea. That computers can't represent continuous values isn't really a big deal; human mathematicians, after all, share the same limitation. (Even if by dint of drill and long contemplation they learn to conceive in themselves a supple, subtle, logical intuition of the infinitely great and the vanishingly small, to the point where they may indeed come to see their occult fictions as Reality. As actually the realer Reality. As indeed, gone far and deep enough into their cups, the very thoughts of God.)[5]

A function can be regarded as a look-up table (often enough it may be implemented as one too) which is to say a mapping from a certain symbol, the look-up key, to a value associated with this key. Modern scripting languages typically include as a native data type the "associative array" (also known as hash table, dictionary, or map) for managing look-up tables of arbitrary complexity. In JavaScript associative arrays are at the same time "objects," the main building blocks (as "lists" are in Lisp) out of which all other entities are constructed. Associative arrays, as the name suggests, can, with a bit of coding cleverness, give software an associative capability, permitting programmers

to emulate (after a fashion) the more flexible, soft-edged categories of natural cognition,[6] against the all-or-nothing, true-or-false Boolean logic, which many people still seem to think is all that computers are capable of.

To briefly pursue the organic analogy: individual neurons, while glacially slow by comparison with CPU switching speeds, in their imprecise massively parallel way still vastly "outcompute" (buying the theory that computations are what brains do) the swiftest supercomputers. It's a version of the classic algorithmic trade-off between processing time and memory space, first essayed by the nineteenth-century computing pioneer Charles Babbage.[7] It may often be advantageous to precompute a function and save having to recalculate it later by compiling the results into a table of key-value pairs (with its argument vector as the look-up key and the result returned as the key's value), perhaps with a rule for interpolating (or "connecting the dots") between tabulated data values at look-up time. In cases where all one has is a collection of discrete samples—where the function that generated the data isn't known a priori, for example measurements of things and events taking place in the world—a look-up table and a rule for smoothing the data belonging to nearby or similar points is hard to beat. (Many of the techniques used in statistics and neural network modeling can be seen as wrinkles on this "nearest neighbors" idea.) Such numerical methods date back to the Ptolemys, when trigonometric tables were first compiled for use by astronomers, navigators, and builders.

Functions and Logic

A function is an abstract replica of causality. It's what it is to be a simple, deterministic machine: the same input must always map to the same output. This intuition is at the heart of logic. If repeating the same operation with the same input gives a different output, you know without a doubt that something changed: it isn't the function you thought it was, it isn't a simple machine. Or perhaps one's measuring instrument was faulty; maybe you blinked. Still you will know for certain that something went sideways since (it is of our humanness to believe) nothing happens without a reason. This inferential form was anciently termed "modus tollens." It says that "A implies B; but not B; hence not A." In other words, there is some theory "A" with testable consequence "B," but when the experiment is performed the predicted outcome wasn't observed, so we must conclude (assuming that the twin constancies of nature and reason haven't failed us) that the theory was wrong.[8]

There's a one-wayness to functions, an asymmetry. They can be one-to-one, where a single input value (which could be an argument list or vector made up of several values) is associated with a single output value. Or they can go from many-to-one: two or more inputs arrive at the same output. But they can never go from one-to-many. The same input must always—if this thing is rational, if it's a machine—produce the same result. One can't in general simply replace a function's inputs by its outputs, run the function backwards and expect to get the inputs back as the result; that isn't deterministic, it's not a function, it will not work.

The exception to the above would be a class of reversible logic functions that at some point might emerge from pure theory to find practical uses in cryptography and/or quantum computing.

Theoretically, a universal computer could be made entirely out of reversible logic gates; in principle therefore any irreversible function can be replaced by a reversible function having certain nice theoretical properties like extremely low or even nonexistent power dissipation. It will certainly be interesting to see what comes of it. There are a few well-known examples of simple reversible functions: multiplication by –1, which toggles the sign of a number; the Boolean NOT (turning 0 into 1, and 1 into 0); and EXCLUSIVE OR. This last-named is a personal favorite: given two equal-length bit-patterns as inputs, XOR will yield a bit-pattern which XORed with either of the two original bit-patterns reproduces the other one. But these simple reversible functions are not sufficient for universal reversible computation.[9]

A corollary of the asymmetry of functions is that observing a function's output, even when we know its internal mechanism, doesn't allow inferring with certainty the input that caused the output. What is past is past, nor is it logically valid to adduce absent causes from present signs: a moment's reflection will reveal that any state of affairs could be a consequence of many different possible causes. How odd then, that this native forensic mode of reasoning, in real life so relied upon, should be logically invalid. Aristotle called it the "enthymeme" or "logically fallacy" of "affirming the consequent." (An unfortunate translation: logical fallacies though fallible need not always lead to false conclusions.) To affirm the consequent reverses the deductive syllogism ("modus ponens") which states, "If A implies B, and A, then B." It is to say rather, "If A implies B, and B, then A." The American philosopher C. S. Peirce thought affirming the consequent (which he termed "abduction") was after deduction and induction, the missing but vital third form of reasoning without which any account

of logic or science would remain incomplete. It's what the palaeontologist does in reconstructing a whole brontosaurus from a brontosaur's toe-bone; or the detective, in reconstructing a crime. It is the fallible anti-logic, the "analogic," of sense perception, pattern recognition, diagnosis: how we read the signs and in-between the lines.[10] Computer science rediscovered abductive inference in the 1980s; it had been neglected since AI broke with cybernetics and information theory some twenty years before.

Abductive or analogical pattern-matching is easily realized by means of an inverted index, a variant form of look-up table where rather than having keys mapped to single values they are mapped to sets or lists of values. (An inverted index therefore isn't a function but rather a "relation.") Nothing too complicated, it's how a book index or search engine works. The words given in a search query will have already been associated by the search engine with lists of spidered web pages where these terms have occurred. The best matching pages are identified by superimposing the result lists belonging to the given terms, so that the more times a page is cited in the aggregated multiset, then the higher, all else being equal (indexes also employ statistical methods that assign numeric "weights" to terms and items to better reflect their probable relevance) it will be placed in the outcome.[11]

The index, as it were, "reverses time."[12] It is an imperfect, indeterminate, in logical terms strictly illegitimate one-to-many mapping that goes from a single "effect" (the input key) to the set of its multiple possible "causes" (the key's inverted file). It is a kind of abstract deconvolution, a way to tackle what physical scientists call the "general inverse problem." The evidential traces or signs of an event are convolutions (literally "enfoldings") of the event with whatever objects or medium its *n*th-order effects encounter and become mixed up with. The material imparts its own intrinsic bias or twist to the event record; it acts like a filter or lens. To recover an "image" of the time-and-space-distant original entails superimposing many scattered, diffuse, faint, redundant, and to unknown degree noise-corrupted[13] signals from its spreading "event-cone." A term from physical optics perfectly captures the idea: "circles of confusion." Drawn together[14] and superimposed, the circles of confusion resolve an image (doubtless rather blurry, but still useful) of the original event. (The "circles" are really cross-sectional slices of a spherical (roughly) wavefront of effects propagating outwardly in space and forward in time to intersect the plane of image formation in what I hope isn't too strained an analogy.)

Embedding Functions

Computers and software, for all that they have become ordinary parts of life and when working correctly are as much taken for granted as sewers or electricity, are scaffolded upon certain inviolable rules, "deep structures" that underlie (we are led to infer) both physical reality and the mental apparatus by whose aid we are able to recognize and grasp reality, to name and shape it. The elements of software, its functions and variables, are at bottom simple things; as equally, in the faith of scientists and philosophers, must be the elements and principles which make up the world.

But software also teaches that the simplicity is hard-won; it is hard to slow thoughts down to allow their dissection into irreducible, atomic components of structure and action that permit reconstructing them into something that behaves the way you imagined it would when you had the idea. Underwriting the ability of people to create software is a bedrock gnosis of "how things work," a kinesthetic intuition of causes and effects that is as much physical as logical. The function is a mental diagram of an ideal machine. With the development of computers, so deeply enmeshed in the semiconductor physics of the substrate, it became evident (if it wasn't before) that thought's rigging of logic—the "it-is-so"-ness of recognizing when a thing makes sense and when it doesn't, quite—is at least conditioned by the basic construction of the world; and that to know one clearly is also to know the other.[15]

Notes

1. Gertrude Stein, as quoted in William Gass, *The World Within the Word.*

2. Alfred North Whitehead, *Introduction to Mathematics.*

3. Pride of place goes to the "A-O compiler," created in 1952 for the Remington Rand UNIVAC computer by a team under Grace Hopper's admiralship. Also notworthy is Kenneth Iverson's APL, which was an early functional programming language originating in a Fortran matrix subroutine library. (K. E. Iverson, *A Programming Language.*)

4. For stacks, see Phil Koopman's definitive, *Stack Computers: The New Wave,* now available online from the author's home page.

5. No one save the mathematician or theologian could get so precise about something that by its definition is so indefinite. It's not for nothing (it was rather for aleph-null)

that the finest speculative theologist of the modern era was Georg Cantor, inventor of set theory and the transfinite numbers. (Even while his career in the higher abstraction was punctuated by periods of enforced repose in asylums for the deeply spiritually afflicted.)

6. The psychologist Roger Shepard, in an essay on the preconditions of knowing, describes how the world transduced by the sensory enfilade serves as an index into "consequential regions of psychological space" where are found and activated a suite of innate and acquired propensities and preparednesses—memories and knowledge that will likely prove equal to the circumstances at hand. Vertebrate brains seem to be organized in such a way that a high-dimensional feature vector (a "sparse population code" representing the animal's ensemble sensory nerve activity) acts as an "address" into, in effect, a very large, wet, sloppy look-up table. (R. N. Shepard, "Towards a Universal Law of Generalization for Psychological Studies.") in *Science* Vol. 237 Issue 4820.

7. Charles Babbage, *The Ninth Bridgewater Treatise.*

8. Sir Karl Popper was knighted for, among other things, having pointed out that the power of science is mostly negative, and that scientific progress proceeds by disproving erroneous theories (i.e., by modus tollens) not by proving "correct" theories true. The latter possibility obtains only in tightly circumscribed synthetic worlds like Euclidean geometry or deductive logic—or, in principle, software. Resolution theorem proving—the basis of the logic programming language Prolog—does its stuff by disproving in a reductio ad absurdum the logical complement of the proposition whose truth one wishes to prove. Obviously it can only work under the closed-world assumption that each concept has one and only one antithesis whose negation exactly reproduces and so vouchsafes true the original proposition. Full certainty can only be had in a deductive system whose logical, causal constitution has been established from the bottom up. (Descartes: "If you want to know how a body works, or a world, then build one.")

9. Pop science treatments of quantum computing (and the role of reversible functions therein) include Julian Brown, *Minds, Machines, and the Multiverse.* People who are tired of being told that quantum mechanics (and with it virtually the entire past hundred years of physics) must lie forever beyond the grasp of ordinary understanding will appreciate Carver Mead's well-credentialed demurral, *Collective Electrodynamics.*

10. On abduction: Carlo Ginzburg, "Clues: Morelli, Freud, Sherlock Holmes."

11. For best matching see Derek Robinson, "Index and Analogy: A Footnote to the Theory of Signs." The first inverted index was the Biblical concordance undertaken in 1230 by an ecclesiastical data processing department of five-hundred monks,

directed by Hugo de Sancto Caro (Hugues de Saint-Cher), a Dominican friar later made a cardinal.

12. In an intriguing if eccentric book, *Symmetry, Causality, Mind,* computer vision researcher Michael Leyton proposes that "time reversal" is the sole task undertaken by intelligence. He sees vision as a two-fold problem: first, the eye must reverse the formation of the optical image incident on the retina (a classic "inverse problem") to identify objects and the spatial relations between them; secondly, the mind must "reverse the formation of the environment"—it must adduce reasons why these objects should be where they are, and should have the forms they have, and it must ascertain as best it can the intentions or implications of these things with respect to its own needs and goals. Only then can it be in a position to decide what, if anything, to do about the situation.

13. "Noise-corrupted" is synonymous with "massively convolved with impractically many broadly diffused and attenuated traces of events that we happen not to be interested in right now."

14. "Drawn together"—see Bruno Latour's essay "Drawing Things Together" for an account of broadly analogous issues in the social production of knowledge.

15. Readers with a taste for such deliberations might enjoy following the elegant turns Paul Valery's curiosity takes in his essay, "Man and the Seashell."

■

Glitch

Olga Goriunova and Alexei Shulgin

This term is usually identified as jargon, used in electronic industries and services, among programmers, circuit-bending practitioners, gamers, media artists, and designers. In electrical systems, a glitch is a short-lived error in a system or machine. A glitch appears as a defect (a voltage-change or signal of the wrong duration—a change of input) in an electrical circuit. Thus, a glitch is a short-term deviation from a correct value and as such the term can also describe hardware malfunctions. The outcome of a glitch is not predictable.

When applied to software, the meaning of glitch is slightly altered. A glitch is an unpredictable change in the system's behavior, when *something obviously goes wrong*.