Established 1845 SCIENTIFIC AMERICAN September 1984 Volume 251 Number 3

Computer Software

Presenting a single-topic issue on the concepts and techniques needed to make the computer do one's bidding. It is software that gives form and purpose to a programmable machine, much as a sculptor shapes clay

by Alan Kay

Computers are to computing as instruments are to music. Software is the score, whose interpretation amplifies our reach and lifts our spirit. Leonardo da Vinci called music "the shaping of the invisible," and his phrase is even more apt as a description of software. As in the case of music, the invisibility of software is no more mysterious than where your lap goes when you stand up. The true mystery to be explored in this issue of *Scientific American* is how so much can be accomplished with the simplest of materials, given the right architecture.

The materials of computing are the tersest of markings, stored by the billions in computer hardware. In a musical score the tune is represented in the hardware of paper and ink; in biology the message transmitted from generation to generation by DNA is held in the arrangement of the chemical groups called nucleotides. Just as there have been many materials (from clay to papyrus to vellum to paper and ink) for storing the marks of writing, so computer hardware has relied on various physical systems for storing its marks: rotating shafts, holes in cards, magnetic flux, vacuum tubes, transistors and integrated circuits inscribed on silicon chips. Marks on clay or paper, in DNA and in computer memories are equally powerful in their ability to represent, but the only intrinsic meaning of a mark is that it is there. "Information," Gregory Bateson noted, "is any difference that makes a difference." The first difference is the mark; the second one alludes to the need for interpretation.

The same notation that specifies elevator music specifies the organ fugues of Bach. In a computer the same notation can specify actuarial tables or bring a new world to life. The fact that the notation for graffiti and for sonnets can be the same is not new. That this holds also for computers removes much of the new technology's mystery and puts thinking about it on firmer ground.

As with most media from which things are built, whether the thing is a cathedral, a bacterium, a sonnet, a fugue or a word processor, architecture dominates material. To understand clay is not to understand the pot. What a pot is all about can be appreciated better by understanding the creators and users of the pot and their need both to inform the material with meaning and to extract meaning from the form.

INTANGIBLE MESSAGE embedded in a material medium is the essence of computer software. Here the message is made visible in a voltage-contrast image: a scanning-electron micrograph of a small part of an Intel 80186 microprocessor. The features of the image are formed not by the conductors and transistors on the chip but by the signals passing through them. The trajectory of the secondary electrons emitted in response to the microscope beam is affected by electromagnetic fields at the surface of the chip: regions of higher voltage attract electrons, weakening the image-forming signal. The microscope beam is pulsed on only when the microprocessor is in a particular electronic state: when certain logic elements are "on." The colors of the lines indicate the voltages in metal communications lines leading to logic elements. Where a signal is traveling along a line there is a region of high voltage. The false-color image has been processed so that such regions, and thus "messages," are seen in light blue. Low-voltage regions are green, intermediate-voltage regions yellow. The red lines are conductors at ground potential, or zero volts. The micrograph was made by Timothy C. May of the Intel Corporation.

There is a qualitative difference between the computer as a medium of expression and clay or paper. Like the genetic apparatus of a living cell, the computer can read, write and follow its own markings to levels of self-interpretation whose intellectual limits are still not understood. Hence the task for someone who wants to understand software is not simply to see the pot instead of the clay. It is to see in pots thrown by beginners (for all are beginners in the fledgling profession of computer science) the possibility of the Chinese porcelain and Limoges to come.

Here I need spend no more time on computing's methods for storing and reading marks than molecular biology does on the general properties of atoms. A large enough storage capacity for marks and the simplest set of instructions are enough to build any further representational mechanisms that are needed, including even the simulation of an entire new computer. Augusta Ada, Countess of Lovelace, the first computer-software genius, who programmed the analytical engine that Charles Babbage had designed, understood well the powers of simulation of the general-purpose machine. In the 1930's Alan M. Turing stated the case more crisply by showing how a remarkably simple mechanism can simulate all mechanisms.

The idea that any computer can simulate any existing or future computer is important philosophically, but it is not the answer to all computational problems. Too often a simple computer pretending to be a fancy one gets stuck in the "Turing tar pit" and is of no use if results are needed in less than a million years. In other words, quantitative improvements may also be helpful. An in-

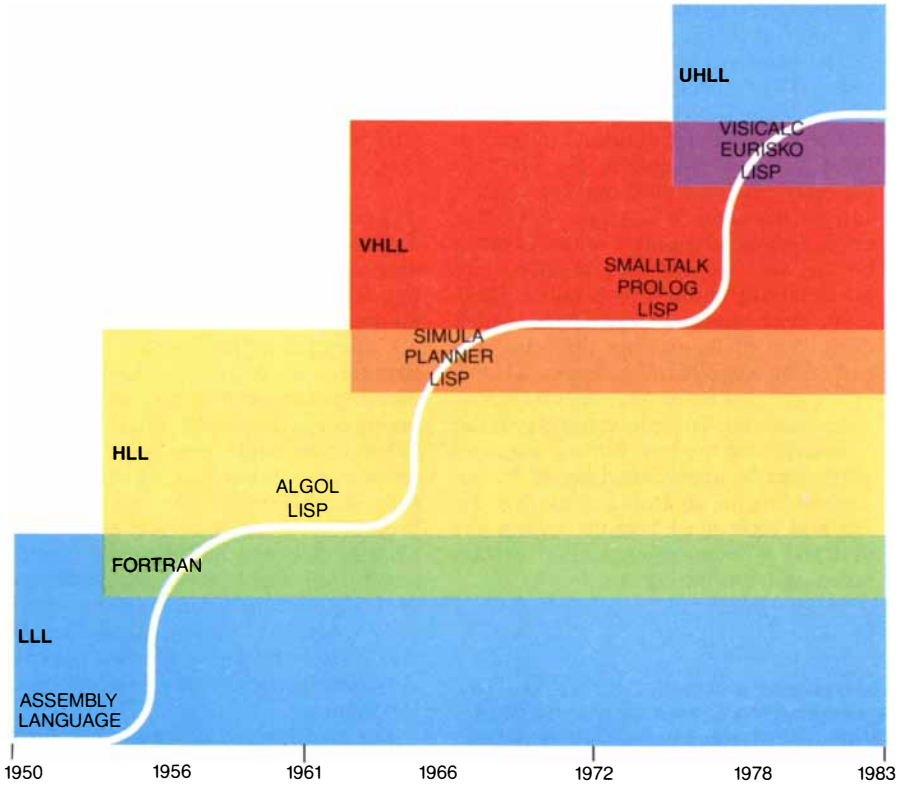
crease in speed may even represent a qualitative improvement. Consider how speeding up a film from two frames per second to 20 (a mere order of magnitude) makes a remarkable difference: it leads to the subjective perception of continuous movement. Much of the "life" of visual and auditory interaction depends on its pace.

As children we discovered that clay can be shaped into any form simply by shoving both hands into the stuff. Most of us have learned no such thing about the computer. Its material seems as detached from human experience as a radioactive ingot being manipulated remotely with buttons, tongs and a television monitor. What kind of emotional contact can one make with this new stuff if the physical access seems so remote?

One feels the clay of computing through the "user interface": the software that mediates between a person and the programs shaping the computer into a tool for a specific goal, whether the goal is designing a bridge or writing an article. The user interface was once the last part of a system to be designed. Now it is the first. It is recognized as being primary because, to novices and professionals alike, what is presented to one's senses *is* one's computer. The "user illusion," as my colleagues and I called it at the Xerox Palo Alto Research Center, is the simplified myth everyone builds to explain (and make guesses about) the system's actions and what should be done next.

Many of the principles and devices developed to enhance the illusion have now become commonplace in software design. Perhaps the most important principle is WYSIWYG ("What you see is what you get"): the image on the screen is always a faithful representation of the user's illusion. Manipulating the image in a certain way immediately does something predictable to the state of the machine (as the user imagines that state). One illusion now in vogue has "windows," "menus," "icons" and a pointing device. The display frames called windows make it possible to present a number of activities on the screen at one time. Menus of possible next steps are displayed; icons represent objects as concrete images. A pointing device (sometimes called a mouse) is pushed about to move a pointer on the screen and thereby select particular windows, menu items or icons.

All of this has given rise to a new generation of interactive software that capitalizes on the user illusion. The objective is to amplify the user's ability to simulate. A person exerts the greatest leverage when his illusion can be manipulated without appeal to abstract intermediaries such as the hidden programs needed to put into action even a simple word processor. What I call direct leverage is provided when the illusion acts as a "kit," or tool, with which to solve a problem. Indirect leverage will be attained when the illusion acts as an "agent": an active extension of one's purpose and goals. In both cases the software designer's control of what is essentially a theatrical context is the key to creating an illusion and enhancing its perceived "friendliness."



SOFTWARE GENRES succeed one another at sporadic intervals, as is shown here through the example of some programming languages. Languages are categorized rather arbitrarily by level, although the levels (*colored bands*) overlap. There are low-level languages (LLL), high-level languages (HLL), very-high-level languages (VHLL) and ultrahigh-level languages (UHLL). In the evolution of programming languages a genre is established (*horizontal white lines*), then after a few years an improvement is made (*curved white lines*). In time the improved language is seen to be not merely a "better old thing" but an "almost new thing," and it leads to the next stable genre. The language Lisp has changed repeatedly, each time becoming a new genre.

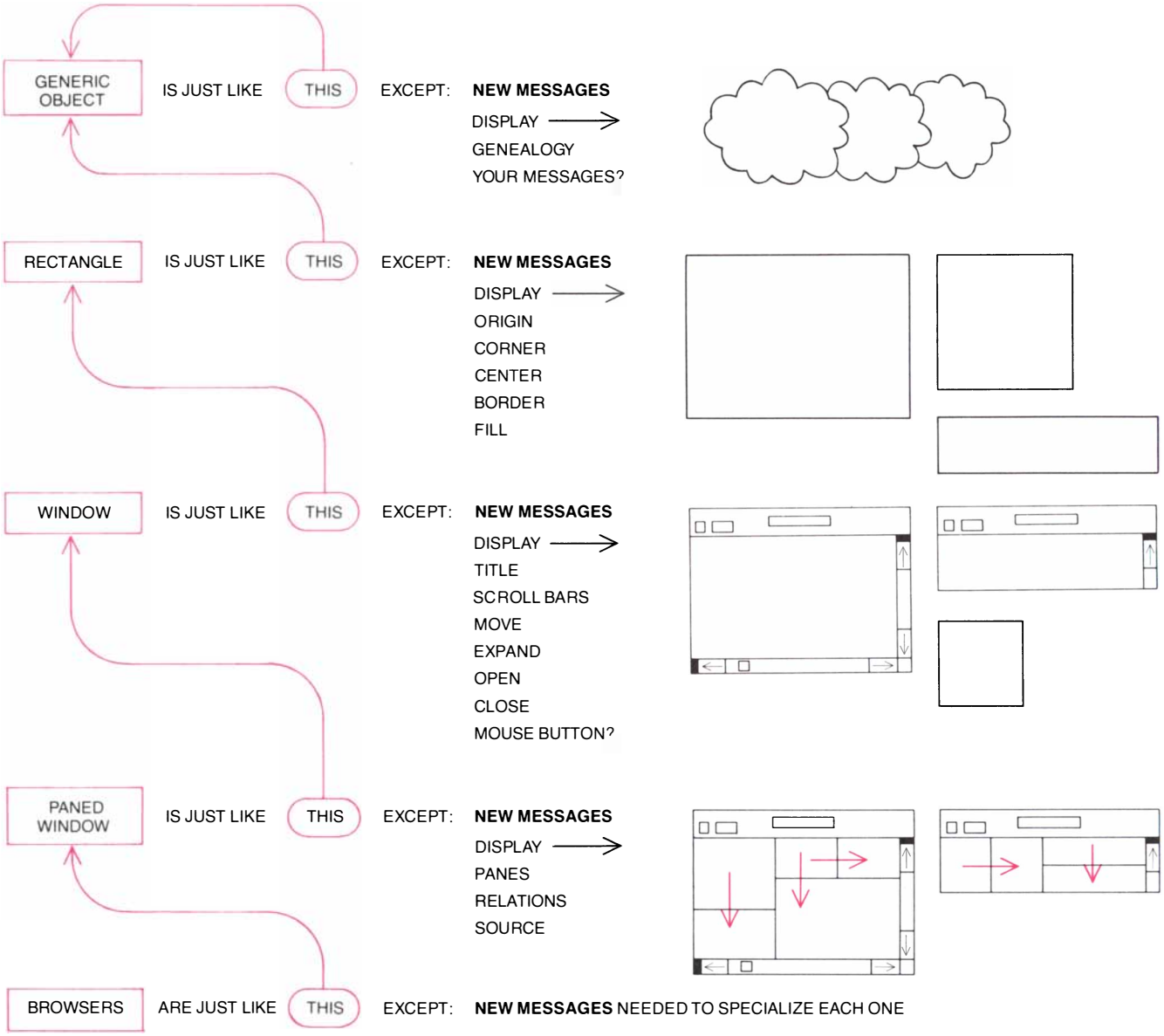
The earliest computer programs were designed by mathematicians and scientists who thought the task should be straightforward and logical. Software turned out to be harder to shape than they had supposed. Computers were stubborn. They insisted on doing what was said rather than what the programmer meant. As a result a new class of artisans took over the task. These test pilots of the binary biplane were often neither mathematical nor even very scientific, but they were deeply engaged in a romance with the material—a romance that is often the precursor of new arts and sciences alike. Natural scientists are given a universe and seek to discover its laws. Computer scientists make laws in the form of programs and the computer brings a new universe to life.

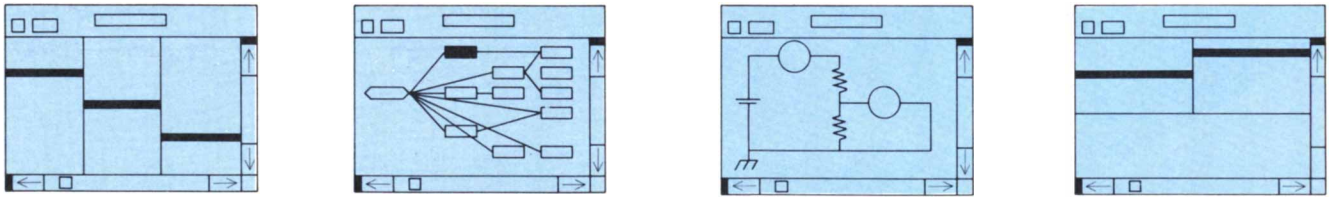
Some programmers breathed too deeply of the heady atmosphere of creating a private universe. They became what the eminent designer Robert S. Barton called "the high priests of a low cult." Most discovered, however, that it is one thing to be the god of a universe and another to be able to control it, and they looked outside their field for design ideas and inspiration.

A powerful genre can serve as wings or chains. The most treacherous metaphors are the ones that seem to work for a time, because they can keep more powerful insights from bubbling up. As a result progress is slow—but there is progress. A new genre is established. A few years later a significant improvement is made. After a few more years the improvement is perceived as being not just a "better old thing" but an "almost new thing" that leads directly to the next stable genre. Interestingly, the old things and their improvements do not disappear. Strong representatives from each past era thrive today, such as programming in the 30-year-old language known as FORTRAN and even in the ancient script known as direct machine code. Some people might look on such relics as living fossils; others would point out that even a very old

species might still be filling a particular ecological niche.

The computer field has not yet had its Galileo or Newton, Bach or Beethoven, Shakespeare or Molière. What it needs first is a William of Occam, who said "Entities should not be multiplied unnecessarily." The idea that it is worthwhile to put considerable effort into eliminating complexity and establishing the simple had a lot to do with the rise of modern science and mathematics, particularly from the standpoint of creating new aesthetics, a vital ingredient of any growing field. It is an aesthetic along the lines of Occam's razor that is needed both to judge current computer software and to inspire future designs. Just how many concepts are there really? And how can metaphor, the magical





INHERITANCE PROGRAMMING shows the power of differential description. A generic "object" (*top*) is displayed as a cloud. One can make a rectangle from the undifferentiated object by saying, in effect, "I want something just like that, except . . . ," and then specifying such properties as the location of the origin (the upper left corner), the width, the height and so on. A further elaboration of the idea is a "window," a rectangular area of the display screen that gives a view of the output of a program. In creating a window one can allow it to "inherit" applicable properties of the rectangle and add new features such as scroll bars (to move the window about over the material being viewed), a title and facilities for changing the window's size and position. A more complex window with panes is made by adding new display methods to shape the panes and establish communications among them (*colored arrows*). Paned windows can be manipulated to make "browsers": systems enabling one to retrieve resources without remembering names. Four examples of browsers are shown (*bottom*).

process of finding similarity and even identity in diverse structures, be put to work to reduce complexity?

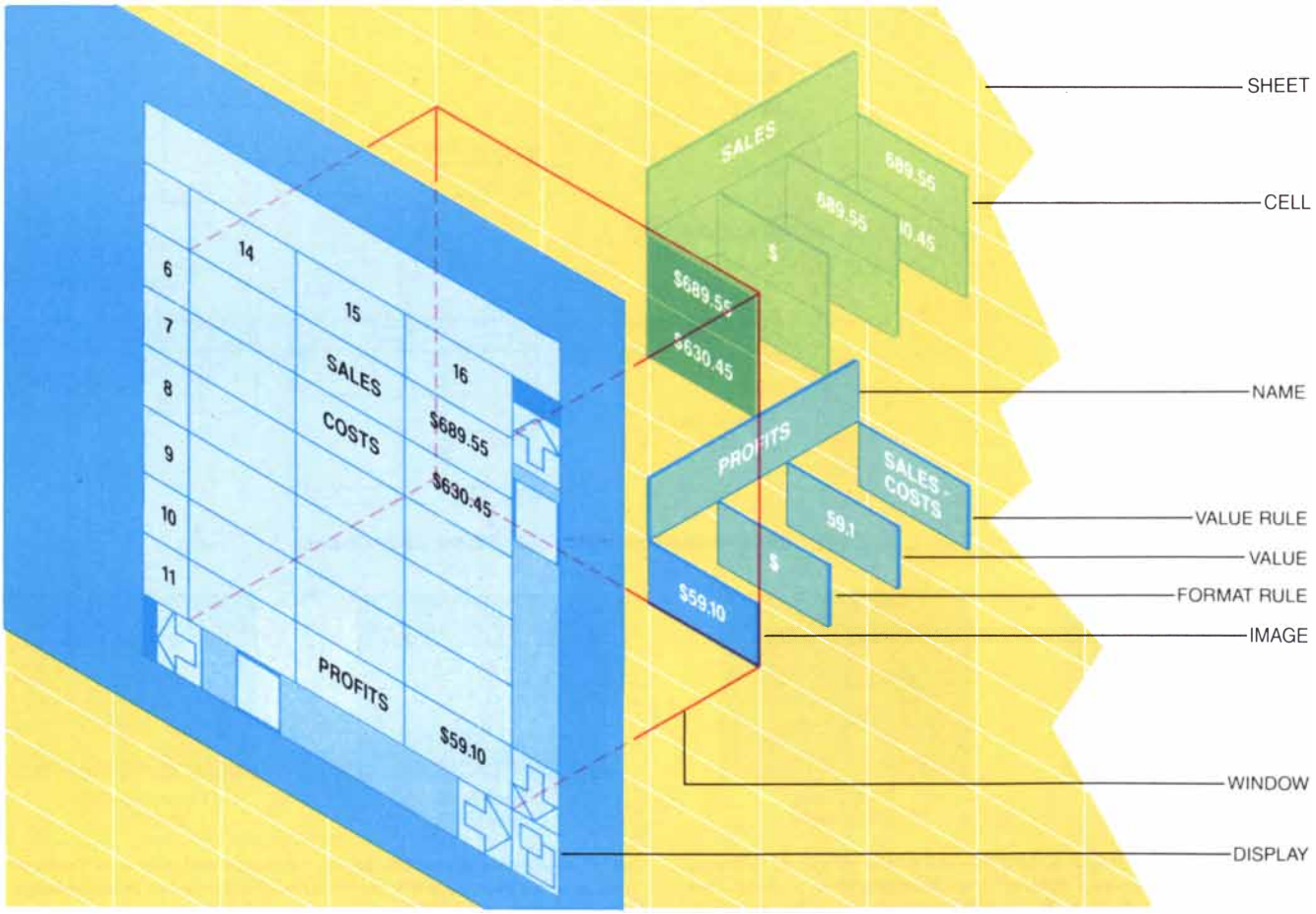
The French mathematician Jacques S. Hadamard found, in a study of 100 leading mathematicians, that the majority of them claimed to make no use of symbols in their thinking but were instead primarily visual in their approach. Some, including Einstein, reached further back into their childhood to depend on "sensations of a kinesthetic or muscular type." The older parts of the brain know what to say; the newer parts know how to say it. The world of the symbolic can be dealt with effectively only when the repetitious aggregation of concrete instances becomes boring enough to motivate exchanging them for a single abstract insight.

In algebra the concept of the variable, which allows an infinity of instances to be represented and dealt with as one idea, was a staggering advance. Metaphor in language usually accentuates the similarities of quite different things as though they were alike. It was a triumph of mathematical thinking to realize that various kinds of self-comparison could be even more powerful. The differential calculus of Newton and Leibniz represents complex ideas by finding ways to say "This part of the idea is like that part, except for...." The designers of computing systems have learned to do the same thing with differential models, for example with programming methods that have the property called inheritance. In recent years models based on the idea of recursion have been formulated in which some of the parts actually are the whole: a description of the entire model is needed to generate the representation of a part. An example is the fractal geometry of Benoit B. Mandelbrot, where each subpart of a structure is similar to every other part. Chaos is captured in law.

Designing the parts to have the same power as the whole is a fundamental technique in contemporary software. One of the most effective applications of the technique is object-oriented design. The computer is divided (conceptually, by capitalizing on its powers of simulation) into a number of smaller computers, or objects, each of which can be given a role like that of an actor in a play. The move to object-oriented design represents a real change in point of view—a change of paradigm—that brings with it an enormous increase in expressive power. There was a similar change when molecular chains floating randomly in a prebiological ocean had their efficiency, robustness and energetic possibilities boosted a billionfold when they were first enclosed within a cell membrane.

The early applications of software objects were attempted in the context of the old metaphor of sequential programming languages, and the objects functioned like colonies of cooperating unicellular organisms. If cells are a good idea, however, they really start to make things happen when the cooperation is close enough for the cells to aggregate into supercells: tissues and organs. Can the endlessly malleable fabric of computer stuff be designed to form a "superobject"?

The dynamic spreadsheet is a good example of such a tissuelike superobject. It is a simulation kit, and it provides a remarkable degree of direct leverage. Spreadsheets at their best combine the



DYNAMIC SPREADSHEET is a simulation kit: an aggregate of software objects called cells that can get values from one another. The window selects a rectangular part of the sheet for display. Each cell can be imagined as having several layers behind the sheet that compute the cell's value and determine the format of the presentation. The cell's name can be typed into an adjoining cell. Each cell has a value rule, which can be the value itself or a way to compute it; the value can also be conditional on the state of cells in other parts of the sheet. The format rule converts the value into a form suitable for display. The image is the formatted value as displayed in the sheet.

genres established in the 1970's (objects, windows, what-you-see-is-what-you-get editing and goal-seeking retrieval) into a "better old thing" that is likely to be one of the "almost new things" for the mainstream designs of the next few years.

A spreadsheet is an aggregate of concurrently active objects, usually organized into a rectangular array of cells similar to the paper spreadsheet used by an accountant. Each cell has a "value rule" specifying how its value is to be determined. Every time a value is changed anywhere in the spreadsheet, all values dependent on it are recomputed instantly and the new values are displayed. A spreadsheet is a simulated pocket universe that continuously maintains its fabric; it is a kit for a surprising range of applications. Here the user illusion is simple, direct and powerful. There are few mystifying surprises because the only way a cell can get a value is by having the cell's own value rule put it there.

Dynamic spreadsheets were invented by Daniel Bricklin and Robert Frankston as a reaction to the frustration Bricklin felt when he had to work with the old ruled-paper versions in business school. They were surprised by the success of the idea and by the fact that most people who bought the first spreadsheet program (VisiCalc) exploited it to forecast the future rather than to account for the past. Seeking to develop a "smart editor," they had created a simulation tool.

Getting a spreadsheet to do one's bidding is simplicity itself. The visual metaphor amplifies one's recognition of situations and strategies. The easy transition from the visual metaphor to the symbolic value rule brings the full power of abstract models to bear almost without notice. One powerful property is the ability to make a solution generic by "painting" a rule in many dozens of cells at once without requiring users to generalize from their original concrete level of thinking.

The simplest kind of value rule makes a cell a static object such as a number or a piece of text. A more complex rule might be an arithmetic combination of other cells' values, derived from their relative or absolute positions or (much better) from names assigned to them. A value rule can test a condition and set its own value according to the result. Advanced versions allow a cell's value to be retrieved by heuristic goal seeking, so that problems for which there is no straightforward method of solution can still be solved by a search process.

The strongest test of any system is not how well its features conform to anticipated needs but how well it performs when one wants to do something the designer did not foresee. It is a question less of possibility than of perspicuity: Can the user see what is to be done and simply go do it?

Suppose one wants to display data as a set of vertical bars whose height is normalized to that of the largest value, and suppose such a bar-chart feature was not programmed into the system. It calls for a messy program even in a high-level programming language; in a spreadsheet it is easy. Cells serve as the "pixels" (picture elements) of the display; a stack of cells constitutes a bar. In a bar displaying one-third of the maximum value, cells in the lowest third of the stack are black and cells in the upper two-thirds are white. Each cell has to decide whether it should be black or white according to its position in the bar: "I'll show black if where I am in the bar is less than the data I am trying to display; otherwise I'll show white" [*see illustration on next page*].

Another spreadsheet example is a sophisticated interactive "browser," a system originally designed by Lawrence G. Tesler, then at the Xerox Palo Alto Research Center. Browsing is a pleasant way to access a hierarchically organized data base by pointing to successive lists. The name of the data base is typed into the first pane of the display, causing the subject areas constituting its immediate branches to be retrieved and displayed in the cells below the name. One of the subject areas can be chosen by pointing to it with a mouse; the chosen area is thereby entered at the head of the next column, causing its branches in turn to be retrieved. So it goes until the desired information is reached [*see illustration on page 59*]. Remarkably, the entire browser can be programmed in the spreadsheet with just three rules.

The intent of these examples is not to get everyone to drop all programming in favor of spreadsheets. Current spreadsheets are not up to it; nor, perhaps, is the spreadsheet metaphor itself. If programming means writing step-by-step recipes as has been done for the past 40 years, however, then for most people it never was relevant and is surely obsolete. Spreadsheets, and particularly extensions to them of the kind I have suggested, give strong hints that much more powerful styles are in the offing for novices and experts alike. Does this mean that what might be called a driver-education approach to computer literacy is all most people will ever need—that one need only learn how to "drive" applications programs and need never learn to program? Certainly not. Users must be able to tailor a system to their wants. Anything less would be as absurd as requiring essays to be formed out of paragraphs that have already been written.

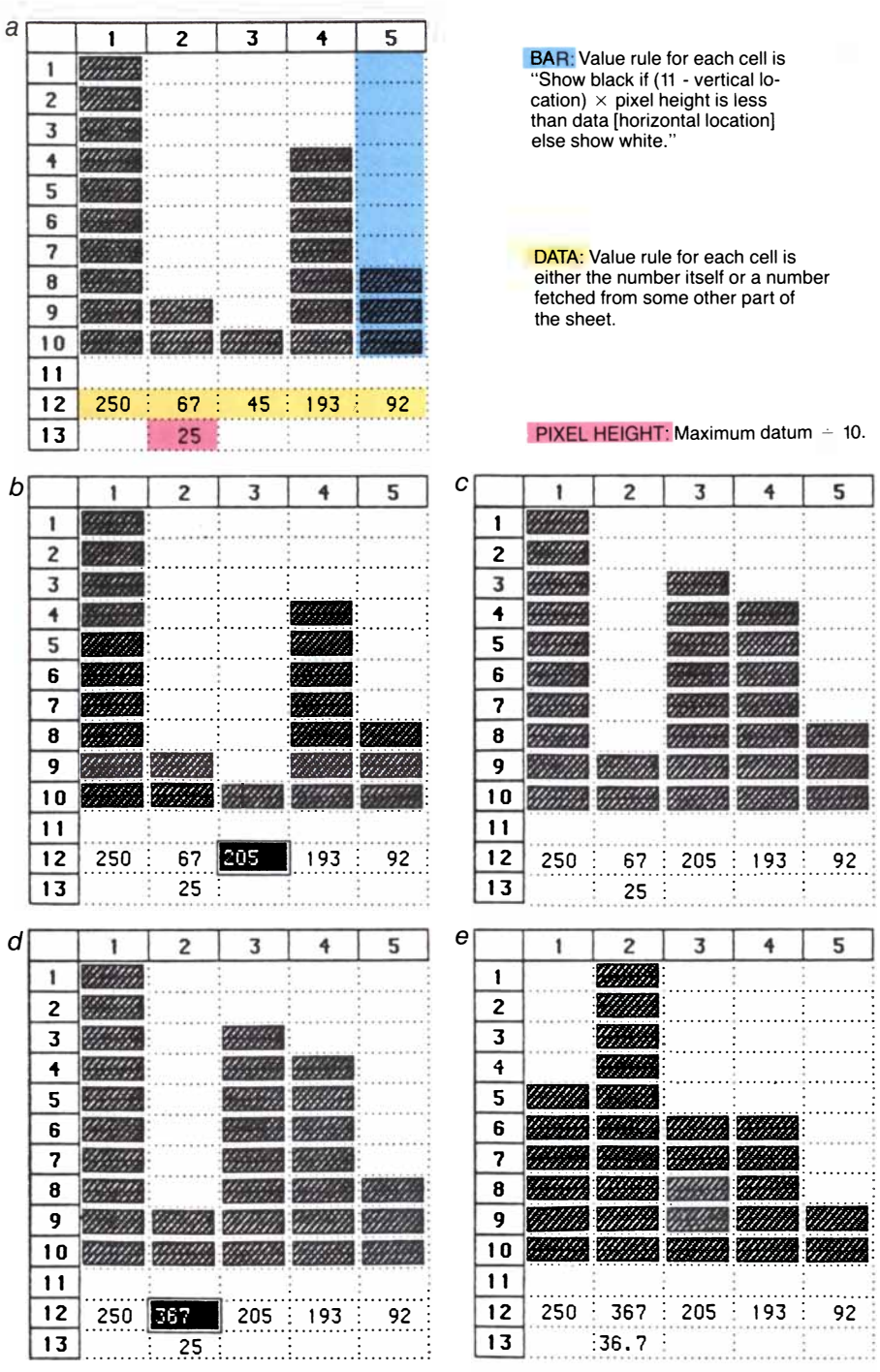
In discussing this most protean of media I have tried to show how effectively design confers leverage, particularly when the medium is to be shaped as a tool for direct leverage. It is clear that in shaping software kits the limitations on design are those of the creator and the user, not those of the medium. The question of software's limitations is brought front and center, however, by my contention that in the future a stronger kind of indirect leverage will be provided by personal agents: extensions of the user's will and purposes, shaped from and embedded in the stuff of the computer. Can material give rise to mentality? Certainly there seems to be nothing mindlike in a mark. How can any combination of marks, even dynamic and reflexive marks, possibly show any properties of mentality?

Atoms also seem quite innocent. Yet biology demonstrates that simple materials can be formed into exceedingly complex organizations that can interpret themselves and change themselves dynamically. Some of them even appear to think! It is therefore hard to deny certain mental possibilities to computer material, since software's strong suit is similarly the kinetic structuring of simple components. Computers "can only do what they are programmed to do," but the same is true of a fertilized egg trying to become a baby. Still, the difficulty of discovering an architecture that generates mentality cannot be overstated. The study of biology had been under way some hundreds of years before the properties of DNA and the mechanisms of its expression were elucidated, revealing the living cell to be an architecture in process. Moreover, molecular biology has the advantage of studying a system already put together and working; for the composer of software the computer is like a bottle of atoms waiting to be shaped by an architecture he must invent and then impress from the outside.

To pursue the biological analogy, evolution can tell the genes very little about the world and the genes can tell the developing brain still less. All levels of mental competence are found in the more than one and a half million surviving species. The range is from behavior so totally hard-wired that learning is neither needed nor possible, to templates that are elaborated by experience, to a spectrum of capabilities so fluid that they require a stable social organization—a culture—if full adult potential is to be realized. (In other words, the gene's way to get a cat to catch mice is to program the cat to play—and let the mice teach the rest.) Workers in artificial intelligence have generally contented themselves with attempting to mimic only the first, hard-wired kind of behavior. The results are often called expert systems, but in a sense they are the designer jeans of computer science. It is not that their inventors are being dishonest; few of them claim for a system more than it can do. Yet the label "expert" calls up a vision that leads to dis-

illusionment when it turns out the systems miss much of what expert (or even competent) behavior is and how it gets that way.

Three developments have very low probabilities for the near future. The first is that a human adult mentality can be constructed. The second is that the mentality of a human infant can be constructed and then "brought up" in an environment capable of turning it into an adult mentality. The third is that current artificial-intelligence techniques contain the seeds of an architecture from which one might construct some kind of mentality that is genuinely able to learn competence. The fact that the probabilities are low emphatically does not mean the task is impossible. The third development is likely to be achieved first. Even before it is there will be systems that look and act somewhat intelligent, and some of them will actually be useful.



BAR CHART can be constructed out of the standard materials of a spreadsheet. A bar is a column of cells, where each cell serves as a pixel, or picture element. One cell associated with each column holds the datum, or value, to be represented by the height of the corresponding bar. Within a bar all the cells are governed by the same rule. The quantity represented by the height of a single pixel is the maximum datum divided by the number of pixels in the longest bar; in chart *a* there are 10 pixels per bar and each pixel represents 25 units. Each cell shows black if its vertical position in the bar multiplied by the number of units per pixel is less than the datum for that bar; otherwise it shows white. When a new datum is entered in a column (*b*), a new bar appears in that column (*c*). If a new datum is larger than the previous maximum (*d*), the set of bars is replotted (*e*) on the basis of the new number of units per pixel, which in this case is 36.7.

What will agents be like in the next few years? The idea of an agent originated with John McCarthy in the mid-1950's, and the term was coined by Oliver G. Selfridge a few years later, when they were both at the Massachusetts Institute of Technology. They had in view a system that, when given a goal, could carry out the details of the appropriate computer operations and could ask for and receive advice, offered in human terms, when it was stuck. An agent would be a "soft robot" living and doing its business within the computer's world.

What might such an agent do? Hundreds of data-retrieval systems are now made available through computer networks. Knowing every system's arcane access procedures is almost impossible. Once access has been gained, browsing can handle no more than perhaps 5,000 entries. An agent acting as a librarian is needed to deal with the sheer magnitude of choices. It might serve as a kind of pilot, threading its way from data base to data base. Even better would be an agent that could present all systems to the user as a single large system, but that is a remarkably hard problem. A persistent "go-fer" that for 24 hours a day looks for things it knows a user is interested in and presents them as a personal magazine would be most welcome.

Agents are almost inescapably anthropomorphic, but they will not be human, nor will they be very competent for some time. They violate many of the principles defining a good user interface, most notably the idea of maintaining the user illusion. Surely users will be disappointed if the projected illusion is that of intelligence but the reality falls far short. This is the main reason for the failure so far of dialogues conducted in ordinary English, except when the context of the dialogue is severely constrained to lessen the possibility of ambiguity.

Context is the key, of course. The user illusion is theater, the ultimate mirror. It is the audience (the user) that is intelligent and can be directed into a particular context. Giving the audience the appropriate cues is the essence of user-interface design. Windows, menus, spreadsheets and so on provide a context that allows the user's intelligence to keep choosing the appropriate next step.

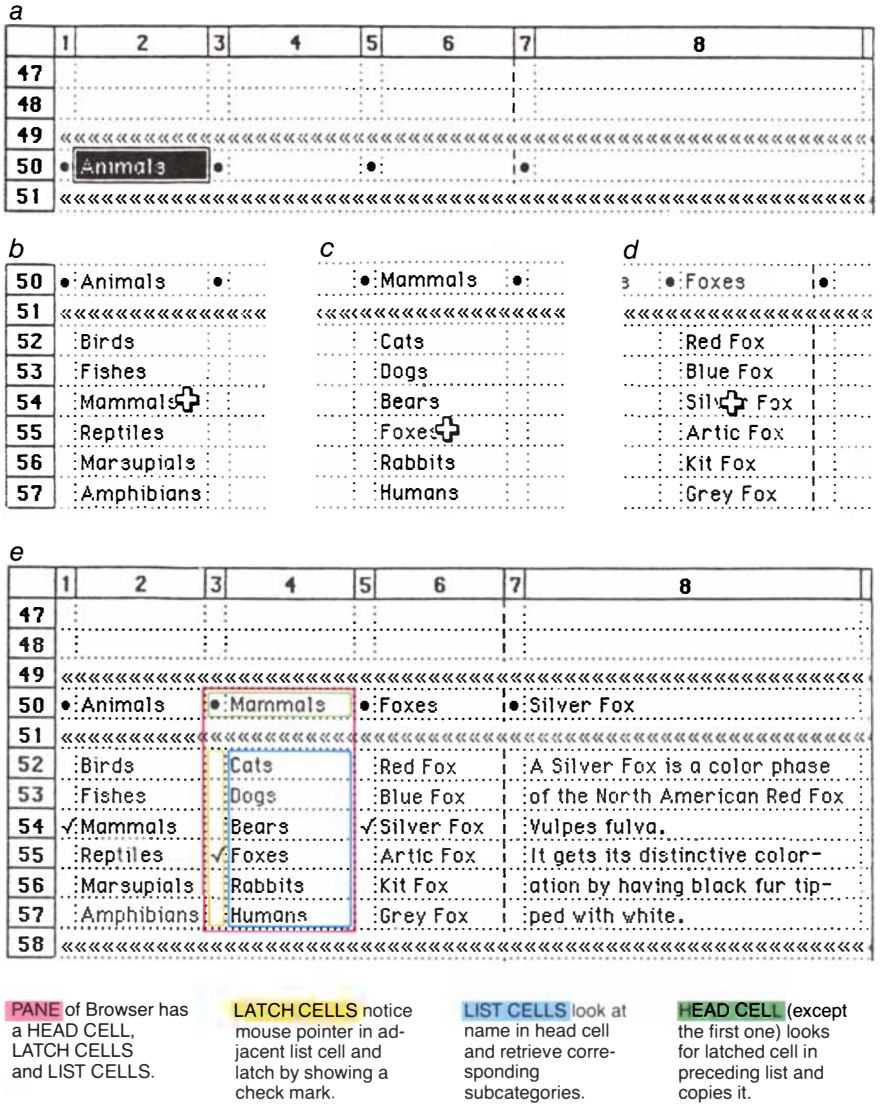
An agent-based system will have to do the same thing, but the creation of an interface with some semblance of human mentality will call for a considerably subtler approach.

Any medium powerful enough to extend man's reach is powerful enough to topple his world. To get the medium's magic to work for one's aims rather than against them is to attain literacy. At its simplest, literacy means fluency. Familiarity (knowing the "grammar") is not enough. People who can recognize a book and its words, a typewriter and its keyboard or a computer and its input-output devices are not literate unless they can spend most of their time dealing with content rather than with the mechanics of form.

Is the computer a car to be driven or an essay to be written? Most of the confusion comes from trying to resolve the question at this level. The protean nature of the computer is such that it can act like a machine or like a language to be shaped and exploited. It is a medium that can dynamically simulate the details of any other medium, including media that cannot exist physically. It is not a tool, although it can act like many tools. It is the first metamedium, and as such it has degrees of freedom for representation and expression never before encountered and as yet barely investigated. Even more important, it is fun, and therefore intrinsically worth doing.

If computers can be cars, then certainly computer literacy at the level of driver-education courses is desirable. Indeed, the attempt is now being made to design user interfaces giving access to the computer's power by way of interactions even easier to learn than driving a car. Integrated programs for word processing, graphics, simulation, information retrieval and person-to-person communication will be the paper and pencil of the near future. The driver-education level of paper-and-pencil literacy is taught, however, in kindergarten and first grade, implying that what can be called mark-making literacy in computers should be attained as early as possible; children should not be made to wait until they can get in a half year of it just before they graduate from high school, as recent reports by educational commissions suggest. Children need informational shoes, bicycles, cars and airplanes from the moment they start to explore the universe of knowledge.

Paper-and-pencil literacy does not stop, moreover, when children know how to manipulate a pencil to make certain kinds of marks on paper. One reason to teach reading and writing is certainly that people need these skills to get through daily life in the 20th century, but there are grander and more critical goals. By reading we hope not only to absorb the facts of our civilization and



DATA-BASE BROWSER allows one to gain access to a hierarchically organized data base by simply pointing to items in successive lists. To learn about the silver fox one types "animals" into the first pane (*a*). Subject areas of the "animals" data base appear in the pane (*b*). Selection of "mammals" causes that subcategory to appear in the next pane (*c*); selection of "foxes" brings up a list of foxes (*d*), and eventually the description of the silver fox is retrieved (*e*).

of those before us but also to encounter the very structure and style of thought and imagination. Writing gets us out of the bleachers and onto the playing field; old and new knowledge becomes truly ours as we shape it directly.

In short, we act as though learning to read and write will help people to think better and differently. We assume that starting with centuries' worth of other people's knowledge is more efficient than starting from scratch and will provide a launch pad for new ideas. We assume that expressing and shaping ideas through metaphor and other forms of rhetoric makes the ideas more fully our own and amplifies our ability to learn from others in turn. (Oliver Wendell Holmes said, "The mind, once expanded to the dimensions of larger ideas, never returns to its original size.") We hold all of this to be important even though reading and writing seem to be quite hard and take years to master. Our society declares that this kind of literacy is not a privilege but a right, not an option but a duty.

What then is computer literacy? It is not learning to manipulate a word processor, a spreadsheet or a modern user interface; those are paper-and-pencil skills. Computer literacy is not even learning to program. That can always be learned, in ways no more uplifting than learning grammar instead of writing.

Computer literacy is a contact with the activity of computing deep enough to make the computational equivalent of reading and writing fluent and enjoyable. As in all the arts, a romance with the material must be well under way. If we value the lifelong learning of arts and letters as a springboard for personal and societal growth, should any less effort be spent to make computing a part of our lives?