REPRINTED FROM IBM SYSTEMS JOURNAL, VOL28, NO 4, 1989; © 1989, 1999

Visual programming: Perspectives and approaches

by N. C. Shu

Visual programming tackles the problem of bringing computing facilities to people who do not have extensive computer training by using visual (i.e., nonlinear) representations in the programming process. In this paper, we first define visual programming and briefly discuss its many facets. The purpose is to lay a conceptual background so that common understanding can be established and various aspects of visual programming can be focused on and examined. We then concentrate on visual programming languages, namely, languages that enable the users to "program" with visual expressions. Examples are used to illustrate three fundamentally different approaches: diagrammatic, iconic, and form based. Finally, we show that FORMAL, a system developed and implemented at the IBM Los Angeles Scientific Center, not only captures the spirit of visual programming languages but also has the capability to automate a wide variety of common data processing applications.

Programming can be defined as specifying a method for doing something a computer can do in terms the computer can interpret. In order to get something done on a computer, there is a division of labor: a human to specify to a computer; the computer to interpret and execute. The traditional programming approach requires a great deal of human effort because programming languages were designed primarily for efficient interpretation and execution by computers. Until recently, very little attention was paid to whether it was easy for end users to do the specification because it was commonly assumed that only those having highly trained programming skills would write programs.

The situation has dramatically changed in the last few years. The rapid decline of computing costs, coupled with the sharp increase in the number of personal computers, has expanded substantially the population of the user community and its drive for computerized applications. However, to many people, the usefulness of a computer is bound by the usefulness of the canned application software available. Those who wish to use the computer to do something beyond the capabilities of the canned programs discover that they will have to "program."

Learning to program, unfortunately, is a time-consuming and often frustrating endeavor. Moreover, even after the skill is learned, writing and testing a program is usually a labor-intensive chore. Many people shy away from it simply because they cannot afford the time and effort required. The challenge is to bring computer capabilities simply and usefully to people whose work can benefit from programming, but who are not programmers by profession.

Visual programming represents a revolutionary approach to meet this challenge. It is stimulated by the premises that: (1) pictures can convey more meaning in a more concise unit of expression; (2) pictures can help understanding and remembering; (3) pictures

can make programming more interesting; and (4) when properly designed, pictures can be understood by people regardless of what language they speak. The field is young and fast moving. There are many interesting proposals and developments within it, but common understanding is lacking.

This paper is tutorial in nature. Its purpose is manyfold: to present a framework so that a common understanding can be established and various aspects of visual programming can be focused on and examined; to bring together a wide array of research efforts within this framework; to illuminate the fundamental differences underlying three major categories of visual programming languages; and to present a "programming paradigm" (represented by FORMAL) that focuses not only on "the ability to communicate" (which is the predominant driving force behind all visual programming languages), but also on "the ability to automate" (which is lacking in most programming languages).

Although one of the purposes of this paper is to pull together the multiplicity of ideas that have shaped the current state of the art in visual programming, it is not our intention to present a complete survey. Obviously, within the space of this paper, we can neither include all the interesting work reported in the literature, nor cover all aspects of any selected work. Instead of a comprehensive survey that portrays a labyrinth, we strive to present a guide through the labyrinth. Readers who are interested in more detail may find the extensive list of references helpful. With this in mind, we are now ready to begin our exposition.

Many facets of visual programming—A framework

Let us begin with the definition of visual programming. Myers[1] classified programming systems into eight categories using the orthogonal criteria of (a) visual programming or not, (b) example-based programming or not, and (c) interpretive or compiled. According to Myers' definitions, "*Visual Programming* refers to any system that allows the user to specify a program in a two- (or more) dimensional fashion," and "*Example-Based Programming* refers to systems that allow the programmer to use examples of input and output data during the programming process."[1]

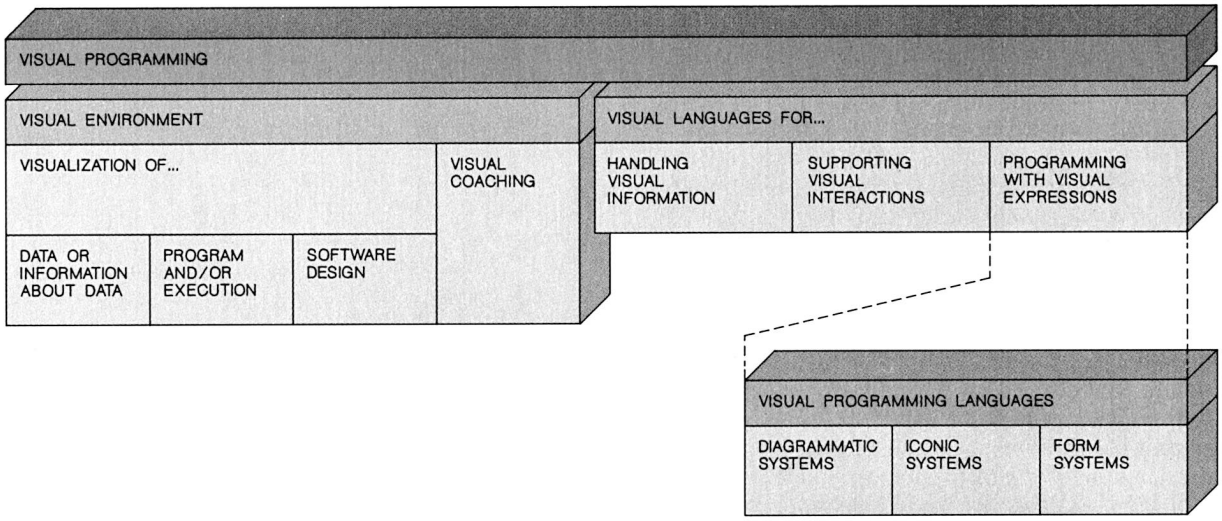
Looking at the subject from a broader perspective, Shu[2] uses the term *visual programming* to mean "the use of meaningful graphical representations in the process of programming." Work on visual programming explores the use of graphical (i.e., nonlinear) representations for all phases of the programming process. Progress is made along two major directions, depending on the primary focuses. Along one direction, graphical techniques and pointing devices are used to provide a *visual environment* for program understanding, construction, and debugging, for information retrieval and presentation, and for software design and maintenance. Along another direction, *languages* are designed to handle visual information, to support visual interaction, and to allow programming with visual expressions. The last category can more aptly be called the "visual programming languages" and is the primary interest of this paper. Figure 1 (taken from Shu[2]) categorizes the many facets of visual programming and is used as the conceptual framework for the following exposition.

Visualization of data or information about data. In the visual environment, the first category of work concentrates on visualization of data or information about data (e.g., data structures, database schemas). Two distinct motivations serve as the driving force. The first is to fulfill the needs of people who want to access information in data management systems but are not trained in the use of such systems. Typically, data are stored internally in conventional databases but expressed and presented to the user in graphical forms. Users can access data via graphical forms or zoom into the data to obtain greater detail with a joystick or a pointing device. This approach permits many types of questions to be answered without the need for a keyboard or the necessity of learning a query language.

The second motivation is to enhance the understanding of people who have to deal with the intricacies of programming. Since data structures and database schemas play a prominent role in programming, and since graphical depictions found in every good textbook are known to be extremely helpful, it is natural that the visualization of data structures and database schemas is pursued.

Regardless of the motivations, systems of this category are mostly devoted to using "direct manipulation"[3] as a means of information browsing and using graphical views for visualization of the information retrieved. Systems such as SDMS,[4] VGQF,[5] KAESTLE,[6] ISIS,[7] AMETHYST,[8] IBS,[9] the Metaphor system,[10] and "A graphical entity-relationship database browser"[11] represent some of the work falling into this category.

Figure 1 Categorization of visual programming



Visualization of a program and its execution. Another category in the visual environment provides graphical support for the visualization of programs and their run-time states and results. Any person who writes, tests, debugs, changes, or maintains a program needs to know why the program solves the given problem. Visualization of programs and execution helps to manifest what the programs do, how they work, why they work, and what are their behavior and effects.

Activities in this area span a wide spectrum, ranging from (a) pretty-printing the source code; through (b) displaying traditional programs in diagrams; and (c) showing multiple views of a program and its execution states in well-known programming terms (e.g., syntax tree, symbol tables, variables, control flow, execution stacks); to (d) watching execution of a program in animated forms. The SEE compiler,[12] a visual syntax editor for Lisp,[13] "An icon-based design method for Prolog,"[14] CEDAR,[15] PECAN,[16] GARDEN,[17] PROVIDE,[18] "VIPS: A visual debugger,"[19] "Visible Pascal,"[20] "Animating programs using Smalltalk,"[21] BALSA,[22] and BALSA-II[23] represent some diversive activities in this area.

Visualization of software design. The third category is aimed at providing a visual environment for the development and understanding of "programming in the large" (as opposed to detailed algorithms in a program). As Lampson[24] pointed out, "Designing a computer system is very different from designing an algorithm: the external interface—that is, the requirement—is less precisely defined, more complex, and more subject to change; the system has much more internal structure—hence, many internal interfaces; and the measure of success is much less clear."

Perhaps because of the problems inherited from the complexity of large systems, work in this area is represented by two contrasting approaches. The contrast reflects two often competing concerns: providing an effective environment and maintaining mathematical rigor. For example, the PV system,[25] software through pictures,[26] and many CASE (Computer Aided Software Engineering) tools[27] focus on the former: the users are provided with a set of graphical tools for creating views of their own choosing. The PegaSys system,[28] in contrast, concentrates on the latter: formalism is introduced into graphical representations with the intention that the consistency between a formal diagram and a program can be proved or disproved by the system.

Visual coaching. The fourth category of efforts explores visual programming with the intention of narrowing the gap between the mental process and the programming process of solving a problem. In the environment provided by these systems, a user does not mentally visualize the effects of his or her instructions. The effects take place on the screen before the user's eyes. The programming process relies almost completely on interactions. The style of interaction, in general, "mimics the informal way we explain programs, by showing pictures of the data

and defining the computation on them by pointing sequences similar to hand waving."[29] We therefore use the term "visual coaching" to characterize this category.

Almost all visual coaching systems are inspired by the programming by *examples* or programming by *demonstration* approach, which has been a field of study within the arena of artificial intelligence.[30] From the user's point of view, the underlying idea is quite simple. In essence, the user "writes" a program (i.e., makes a specification) by giving examples or demonstrating to the computer what he or she wants the program to do. The system records, and hopefully also generalizes, what has been shown.

Since most people are much better at dealing with specific, concrete objects than with abstract concepts, specification by demonstration can indeed be natural and potentially easy for the user to formulate. However, from the system standpoint, program synthesis from examples involves many difficulties.[31]

First of all, the examples given may be ambiguous. The system must be able to determine whether the user's specification is consistent and whether the system's "model" of what the user wants is indeed the right program. Furthermore, specification by example is rarely complete, since a few examples will not fully describe the behavior of the desired program in all cases. The system must be able to work with partial or fragmented information and must have the ability to do inductive inferencing in order to synthesize a program that covers more than the examples given. In addition, when the process is attempted in the general case, the system must enumerate the set of all possible programs, the cost of which could be very high.

These difficulties can be circumvented by restricting the problem domain to a relatively small area, where the ability of the system to "infer" the general case is not required or is not crucial. The difficulties can be further reduced by providing an interactive visual environment which encourages cooperation between the user and the system and leads to an increase in both the clarity and the amount of information that a user conveys to the system. It is here that visual coaching comes in.

Visual coaching represents various attempts to couple the power of examples with the benefits of working in an interactive visual environment. Representative examples include AUTOPROGRAMMER,[32] Pygmalion,[33] "Programming by Abstract Demonstration,"[34] Smallstar,[35] ThinkPad,[36] PIP,[29] Tinker,[37] "Programming by rehearsal,"[38] ALEX,[39] Peridot,[40] "Visual programming with objects and relations,"[41] and Basil.[42]

To make the problems tractable, most of the visual coaching systems today (with few exceptions, of which AUTOPROGRAMMER[32] is an example) choose to limit their applications to a relatively small and precise domain and attempt to do limited or no inductive inferencing. At the current state of the art, visual coaching offers an intuitive environment which works well when the objects being manipulated have obvious and concrete representations but the problem domain of each system is rather restricted.

Contrasts between visual environment and languages

Before we go on, note that the first three categories of visual programming differ from each other because they focus on visualization of three distinct classes of objects, and the problems that they attempt to solve relate to three different aspects of programming. The fourth category (visual coaching) goes beyond visualization by using visual signaling as a means of program construction. Thus, on a close look, these four represent four distinct categories of visual programming.

Looking from a higher level, however, we see that it is clear that all four have the following characteristics in common: (a) All of them provide a *visual environment* for humans to interact with the computer, where *showing* is the primary means of communication. From the user's point of view, visualization means "Show me what I have" (in terms of program or data or system design) or "Show me what is going on" (in terms of execution states), whereas visual coaching can be characterized as "Do as I show you." (b) None of them provide anything new in terms of an approach to the *language* aspects of the programming process. In other words, the emphasis of these four categories is on interaction in a visual environment, not on languages. This characteristic marks the sharp distinction between the two major areas of visual programming: visual environment and visual languages.

In contrast to using "showing" as the primary means of communication, languages are used in *telling* the computer what to do. Depending on their objectives,

visual languages can be classified into three categories: those for handling visual information, those for supporting visual interactions, and those for programming with visual representations. The distinctions among these three categories become clear when we take a closer look at the representative systems.

Many query languages for pictorial data are implemented as augmented database query languages.

Languages for handling visual information. The development of this category has historical interest. In the early 1970s, the image processing systems designed for the handling of pictorial data and the database management systems designed for the storage, retrieval, and management of alphanumeric data were developed in parallel at different camps. Image processing systems were mostly designed for geographical, medical, scientific, or engineering applications. Each system was tailored for a specific purpose in a special environment, which makes the sharing of pictorial data very difficult. Database management systems, in contrast, were designed for business applications. Although they allow effective accessing and sharing of alphanumeric data, the handling of pictorial data was not part of their domain. It was difficult, if not impossible, to use the conventional query languages to express or manipulate spatial relationships.

By the late 1970s, the technologies of pictorial data processing and database management began to merge. The growing amount of pictorial data being generated and analyzed and the ever-increasing need to share and to make pictorial data readily accessible have motivated the development of "generalized" systems. One relatively straightforward approach is to incorporate picture-processing capabilities into the conventional database query languages. Consequently, many query languages for pictorial data are implemented as augmented (or extended) database query languages. For example, GRAIN (the Graphics-oriented Relational Algebraic INterpreter)[43] is an extension of RAIN,[44] GEO-QUEL[45] is a geographical extension of QUEL,[46] and PSQL (Pictorial Structured Query Language),[47] ISQL (Image SQL),[48] and IDMS[49] are extensions of SQL (Structured Query Language).[50]

These languages are often referred to as "pictorial query languages"—a term which may be misleading, since in most cases the query languages themselves are not pictorial, even though the data objects that they deal with are.

Languages for supporting visual interactions. Graphical displays and pointing devices play an important role in visual interactions. The traditional approach to creating a graphical display is to write a program that accepts parameterized input, accesses a database, and then calls a graphics subroutine to create the desired display. The principal impediment to this approach is the use of traditional languages which were not designed for visual interactions.

Thus, as visual interactions are gaining ground, languages are developed to support the visual interactions. ICDL (the Icon-Class Description Language) of SDMS,[4] HI-VISUAL ("A Language Supporting Visual Interaction in Programming"),[51] Squeak ("A language for communicating with mice"),[52] Coral (Constraint-based Object-oriented Relations And Languages),[53] and "An interface description language for graph editors"[54] are examples of this category. They support various forms of visual interaction, but the languages themselves are textual.

Visual programming languages. The third category of visual languages, visual programming languages, allows users to actually program with graphical expressions. Informally, a *visual programming language* can be defined as a language which uses some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language.

Note that this definition imposes no restrictions on the type of data or information. It is immaterial whether the object being operated on or being presented to a user is textual, numeric, pictorial, audio, or a combination of these forms. What is important is that, in order to be considered a visual programming language, *the language itself must have some meaningful* (i.e., not merely decorative) *visual representations as a means of programming.* Programs using visual programming languages are constructed

from the language components in the traditional sense. The *language primitives* (e.g., icons, lines, boxes, arrows, form constructs) have well-defined *syntax* and *semantics*. The "sentences" expressed in these languages (e.g., icons connected with flow paths, nodes connected by arrows, structured charts, stylized forms) can be "parsed" and "interpreted."

Based on the principles of design, most of the visual programming languages reported in the literature fall into three broad categories. At one end of the spectrum, flowcharts and diagrams that are already in use on paper are either incorporated into programming constructs as extensions to conventional programming languages, or made into machine interpretable units to be used in conjunction with conventional programming languages. They can be characterized as diagrammatic programming languages.

Some aspects of programming can best be represented in diagrammatic forms.

At the other end, icons or graphical symbols are deliberately designed to be the programming language primitives. The primary objective is to teach and to carry out programming concepts by pictorial representations. The rationale behind the iconic approach is the premise that pictures provide an incentive to learn. Challenge, fantasy, and curiosity (the three most important factors that make computer games so captivating)[55] are all there when we deal with pictorial systems. These languages can be characterized as iconic programming languages. They are often misleadingly equated with visual programming languages, but strictly speaking, they represent only one type of the latter.

Between the two extremes (i.e., chart or diagrams, and iconic systems) we have visual programming languages of the third kind: the forms-oriented languages. In this category, like all other visual programming languages, graphical representations (typically, stylized form constructs) are designed as an integral part of the language. However, unlike the icons in the iconic systems, these graphical representations are not designed to induce a sense of excitement. They are adopted mainly because they represent familiar notions. Yet, unlike the diagrammatic systems, these languages are not attempts to make the "paper and pencil aids" executable.

We now discuss these three types of visual programming languages in more depth.

Diagrammatic programming languages

For many years, charts, graphs and diagrams of various sorts have been used as visual aids for the illustration or documentation of one or more aspects of programs. But these graphical aids, by and large, did not comprise the executable programs themselves. The high cost of graphic terminals and the large data storage needed for graphic representations have kept the graphing and diagramming techniques on paper and on blackboards. Only recently have efforts been made to make the paper and pencil tools executable. The motivation behind these efforts is not hard to understand if we look at the conventional process of programming. Traditionally, programming involves several distinct phases: problem analysis, charting (i.e., using some kind of diagrammatic depictions for program abstraction), coding, translation (compiling/interpreting), and testing. And, more often than not, these processes would have to be reiterated at various points. A serious problem with this approach has to do with the need to keep *both* the charts and the code (which are basically two representations of the same program) *up to date.* It is not surprising that somewhere along the process, the chart (which is also part of the documentation) no longer represents the actual code that gets executed.

Making charts executable is an attempt to collapse the two separate processes (program abstraction and coding) into one, thus not only making programs easier to comprehend, but easier to document and maintain.

In addition, some aspects of programming can best be represented in diagrammatic forms. For example, state transition diagrams are found to be suitable for the user interface description.[56] By giving the transition rules for each state, they make explicit what the user can do at each state and what the effects will be. As another example, the interrelationship be-

tween concurrent processes can be more vividly described with diagrams than with text. Thus, graphics is exploited for those aspects of programming where diagrammatic forms are appealing. The basic notations for these diagrammatic forms follow fairly widely used conventions. They include data flow diagrams (e.g., FGL,[57] GPL[58]), state transition diagrams (e.g., USE,[59] Jacob's state transition diagram language[56]), and Petri Nets (e.g., VERDI,[60] GSDL,[61] PFG[62]), as well as various forms of flowcharts (for example, FPL,[63] Pascal/HSD,[64] GAL,[65] "A graphics-based programming support system,"[66] PIGS,[67] and Pigsty/I-PIGS[68]).

To give some of the flavor of these diagrammatic languages and to show the trend in which some of them are evolving, we discuss "A graphics-based programming support system" (hereafter referred to as the Programming Support System),[66] PIGS,[67] and Pigsty/I-PIGS[68] in more depth. These three systems comprise a series of efforts that evolved over the last ten years.

The Programming Support System was a pioneering work in using charts as the graphical extension of a conventional programming language. In order to make charts executable, the Programming Support System extended the structured diagrams proposed by Nassi and Shneiderman[69] to include "headers" and called the extended forms *NSDs*.

Each NSD consists of two parts. The (declarative) header part includes the diagram name, a comment about its function, and definitions of its local variables and parameters. The (imperative) body part consists of NSD constructs (SIMPLE, i.e., sequential, IF, CASE, and DO-LOOP) with embedded base language statements (in this case, a subset of PL/I). The NSD constructs specify the control flow, and the base language statements specify the operations to be performed. Figure 2 shows an example of an NSD.

Drawing or modifying charts is normally a cumbersome process and is often a deterrent to using charting techniques. To make the task easier, the Programming Support System provides an interactive graphic editor to serve as a charting device. Editing in general involves pointing to a location on a screen and typing in a single letter to indicate the desired action. For example, typing the letter "S," "I," "C," or "L" causes the system to embed a SIMPLE, IF, CASE or DO-LOOP construct at the pointed location. Typing the letter "T" allows text to be entered within a pointed NSD construct.

To execute an NSD program, a preprocessor is invoked to translate the charts into PL/I source programs, which are then compiled by a regular PL/I compiler and executed.

The extra layer of software imposed by the preprocessor approach prevents interactive execution which is useful at the program development stage. The desire to provide a more flexible environment led to the development of PIGS, which stands for "Programming with interactive graphical support."[67] It is a direct descendent of the Programming Support System. A PIGS program is built with NSDs developed by the Programming Support System as the executable chart form, but uses a subset of Pascal (instead of PL/I) as the base language.

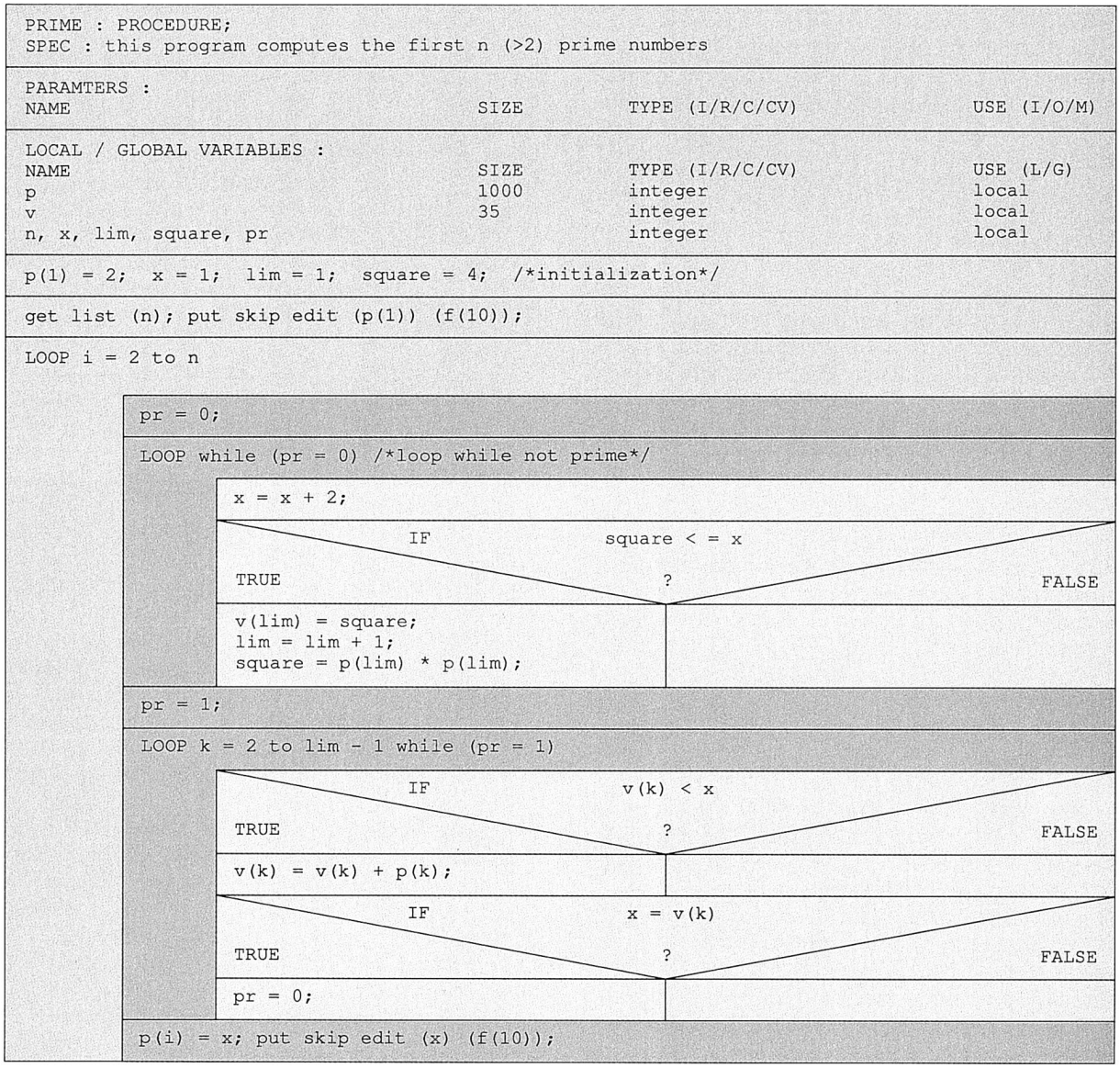
The emphasis of PIGS is on the interactive support for testing and debugging at execution. An interpreter, rather than a compiler, is used to allow the user to interact with the program and make changes during execution. As execution proceeds, the user can watch and follow the logic flow of an NSD program. The construct outline and the embedded text are displayed and brightened at the graphics terminal when an NSD construct is executed.

More recently, the concept of PIGS was extended to support concurrent programming in Pigsty/I-PIGS.[68] Pigsty is the language based on Pascal and Communicating Sequential Processes (CSP),[70] and I-PIGS is the programming environment that supports Pigsty.

Like PIGS, Pigsty uses a combination of text and diagrams to represent a program. The sequential part of Pigsty is based on Pascal. The control constructs are represented in chart form. Dijkstra's alternative IF-FI and repetitive DO-OD guarded commands[71] become the ALT-/ALT and the *ALT-/*ALT, respectively. A Pigsty program consists of one or more sequential processes, each represented by a box. Processes may communicate with each other via one-directional links. The communication and synchronization mechanism between Pigsty processes is the same as the CSP communication mechanism. I-PIGS can execute the chart programs via a simulated concurrent execution mechanism and detects deadlock during execution of a concurrent program.

These three examples show the evolution of a species of diagrammatic systems—a species that has its root in the structured diagram of Nassi-Shneiderman. The evolution has widened the horizon but has not changed the fundamental characteristics of diagram-

Figure 2 An NSD program



matic languages: Diagrammatic depictions that are useful on paper are made into machine interpretable units and used as extensions of or in conjunction with conventional programming languages.

Iconic programming languages

Iconic systems use icons (pictograms) to represent objects and actions. Xerox's Star system[72] is often credited as the forerunner of the iconic systems. Star uses icons and pointing devices as a means to communicate with the computer. Every user's initial view of Star is the "desktop" on the screen. Documents, folders, file drawers, in and out baskets, etc. are displayed as small pictures (or icons) on the desktop. A user can "open" an icon by selecting it (with a mouse) and pushing the OPEN key on the keyboard. When opened, an icon expands into a larger form called a "window." Contents of an icon are displayed in the window, enabling a user to read

documents, inspect the contents of folders and file drawers, send and receive mail, etc., by "seeing and pointing" versus "remembering and typing." Star has a powerful editor for document creation on the screen. The much talked about WYSIWYG (What You See Is What You Get) refers to the situation in which the display screen portrays a rendition of a printed page.

As far as the computational capabilities are concerned, "calculators" (modeled after pocket calculators) exist to let a user perform arithmetic calculations. Arithmetic computations for "records processing" (i.e., traditional data processing)[73] can be embodied in the "fill-in rules," specified as a property of fields. Figure 3 shows an open property sheet for a field with a fill-in rule.

For more complicated computations, users would have to "escape" to a conventional language called CUSP (for CUStomer Programming). The Star designers recognized that "the complexity of user applications is essentially unbounded, which makes some sort of programming language virtually mandatory."[73] Unfortunately, what is envisioned is: "Eventually, CUSP will become a full programming language, with procedures, variables, parameters, and a programming environment."[73]

In short, Star was primarily designed for office professionals who create, retrieve, and distribute documents. The main contribution of Star is making the system seem friendly by simplifying the human–machine interface for office workers. It is evident that its iconic "desktop metaphor" has widespread

Figure 3 A field with a fill-in rule

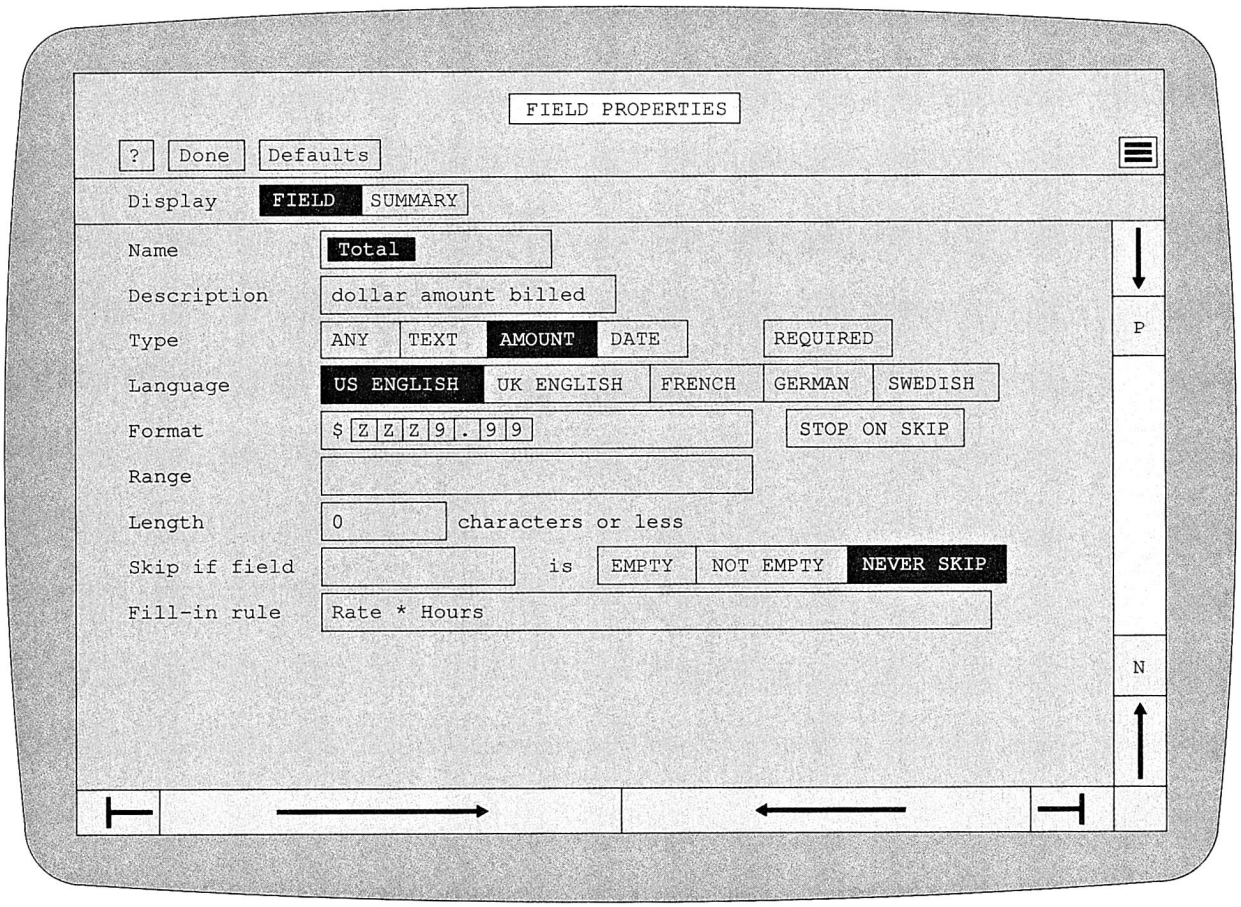
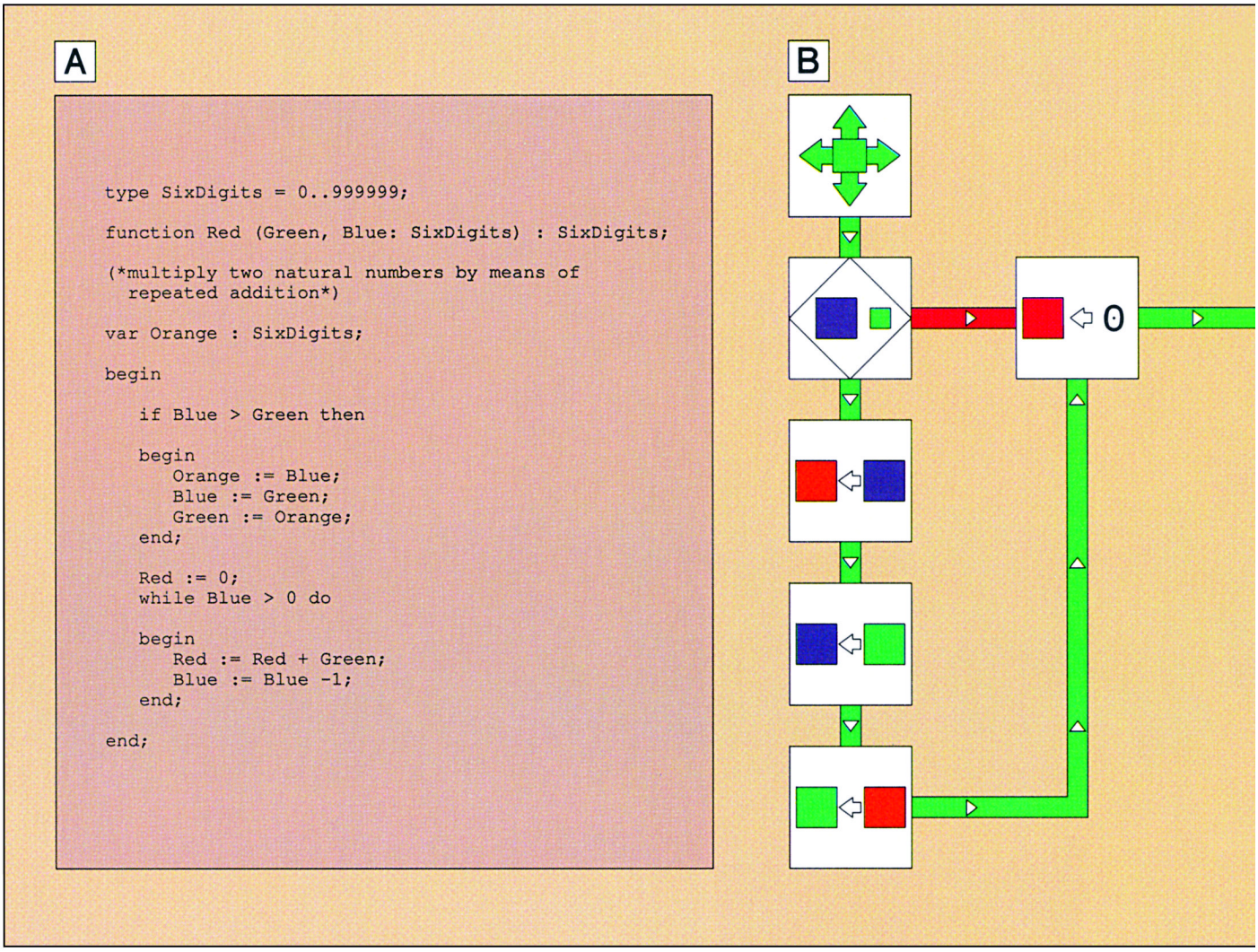


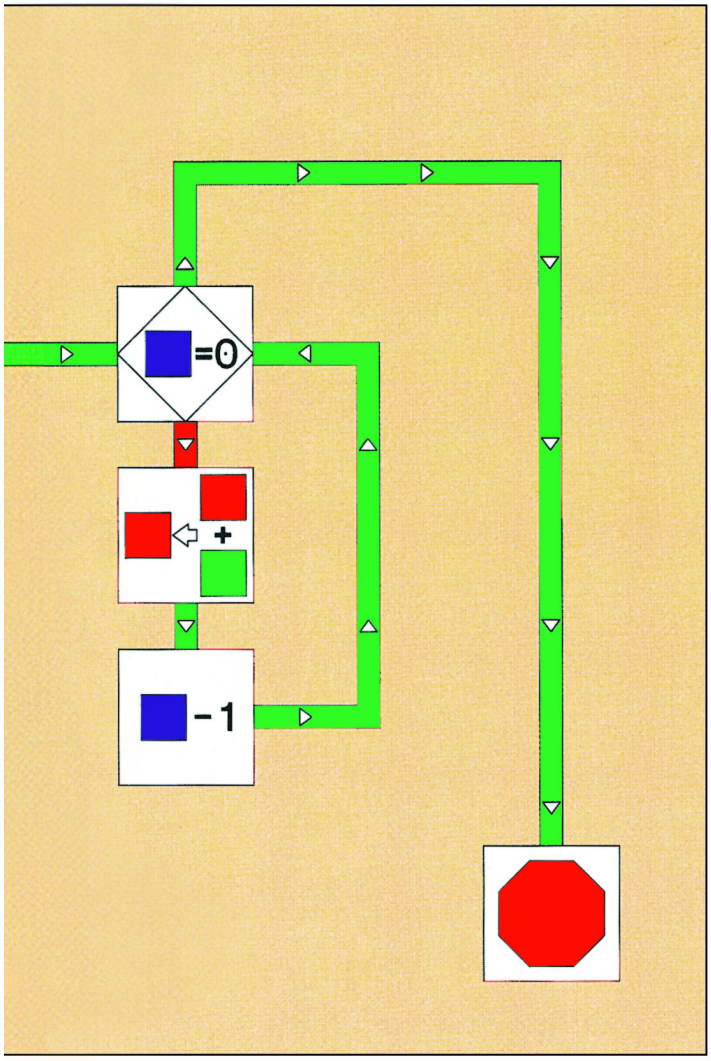
Figure 4 Multiplication by repeated addition: (A) a Pascal program; (B) a Pict program



influence on many of the later commercial products. User interfaces using icons and windows are now commonplace in many systems.

The function of iconic programming languages goes beyond that of iconic user interfaces, since the latter deals with communication with a computer at the "command" language level, whereas the former attempts to achieve what the traditional programming languages are capable of doing.

In the last few years, a significant number of iconic programming languages have been reported. Among them, VennLISP,[74] ICONLISP,[75] and Tinkertoy[76] are examples of executable graphics based on Lisp; Dialog.I[77] is an iconic programming language based on logic programming; IDEOSY[78] and Clara[79] support a formal semantics based on Milner's Calculus of Communication Systems (CCS);[80] Pict[81] and BLOX[82] express algorithmic languages, such as Pascal and C, in graphics; "Show and Tell"[83] treats computation problems as the completion of puzzles; PROGRAPH,[84] the extended HI-VISUAL,[85] and the G language of LabVIEW[86] fortify data flow concepts with iteration, CASE, IF (or "SWITCH"), and other control structures; PROGRAPH2[87] combines data flow and object orien-



tation; InterCONS[88] implements data flow concepts with direct manipulation techniques; and BridgeTalk[89] proposes an approach that is based on how novices learn to program.

It should be clear, from the above citations (which by no means cover all of the iconic programming languages that have surfaced), that there are diversified approaches to iconic programming languages. Nevertheless, in spite of the variations in the backbones of these languages, most of them are designed to entice novices or end users into the world of programming or to make the learning of programming fun. We use Pict[81] and the G language of LabVIEW[86] as illustrative examples.

Pict represents one of the earlier efforts in using icons for purposes beyond the user interface at the operating system (or command language) level. It is designed to aid program implementation. Traditional programming concepts such as "(sub)program names and parameter passing modes, data structures, variables, and program operations are represented by icons of various sorts, whereas control structures such as the Pascal REPEAT-UNTIL or WHILE are represented by colored, directed paths that can actually be seen."[81]

In writing a program, icons denoting the various operations needed to perform the calculations are selected from a predefined set and placed in the program area of the screen. These icons are then connected by paths to denote the flow of control. As an example, Figure 4A shows a Pascal program for multiplying integers by means of repeated addition, and Figure 4B illustrates the essence of a Pict program that performs the same function. The correspondence between these two is obvious.

At execution time, Pict uses simple forms of animation to make the drawing "come to life." The run-time stack height is shown at both edges of the main area of the user program. A white box moves along a flow-of-control path, showing the progress through the program as it is run.

The Pict system was intended to serve as an experimental prototype capable of supporting the implementation of programs similar to those often assigned to students in introductory programming courses. As a programming language, Pict is at a language level similar to that of BASIC or simple Pascal. Pict allows recursive and arbitrary chains of subroutine calls and was found to be suitable for initiating novices into the world of computer programming. However, because of the very restricted sizes of the user program modules, the limited set of data types and language constructs, and the extremely small number of variables to which a module can refer, the applicability of Pict beyond the classroom is quite limited. The authors acknowledged that "To woo the expert user, we must significantly expand the system's capabilities."[81]

In contrast, the G language of LabVIEW[86] has been put to use to solve some real-world problems (for example, mathematical modeling by physiologists[90]).

Labview is an acronym for Laboratory Virtual Instrument Engineering Workbench. It is a software construction system for scientific applications involving instrument control, data acquisition, analysis, computation, and display. Its intended users are engineers and scientists with little programming experience. A Labview program can be thought of metaphorically as a "virtual instrument" consisting of two parts: a front panel and an executable block diagram. The front panel contains the graphical representations (i.e., icons) of the input and output controls (e.g., switches, dials, knobs, digital or analog meters, strip-charts). The block diagram contains the program that the instrument executes. From various menus the user obtains the desired icons and places them in the front panel or in the block diagram. "Wires" connecting the icons define the path of data flow from one icon (or node) to the next. A simple example[86] is shown in Figure 5. In Figure 5A, the left half contains the front panel for the scale number virtual instrument. It has two input variables (or controls) and one output control. The right half is the executable block diagram that uses these controls. The program logic of this block diagram is based on Figure 5B, which shows a way to scale a number from the range (-1,1) to the range (0,s). The execution of a virtual instrument is inherently parallel and data-driven.

The language used to build the block diagrams is called the graphical language G. G is based on the data flow model of programming with extensions that overcome the difficulties of specifying conditional and iterative operations in the pure data flow paradigm. Four "flow of control" structures are provided: (1) a sequence structure that allows the user to specify a strict sequence of execution steps (to override the inherent parallelism); (2) an iterative (or FOR) loop; (3) a case selection structure; and (4) an infinite (or WHILE) loop. In addition, a set of shift registers can be attached to the boundaries of the iterative and indefinite looping structures to enable the result of one iteration to be used as input to a subsequent iteration, thus permitting recursive calculations.

The G language of Labview supports four basic data types (i.e., real numbers and arrays of reals, Booleans and arrays of Booleans, strings and arrays of strings, and structures) and a rich set of built-in functions (e.g., array arithmetic, matrix and vector algebra, statistical functions, and signal-processing routines). Built-in functions and user's virtual instruments appear as icons and can be incorporated into the block diagrams of a larger instrument and executed like a subroutine.

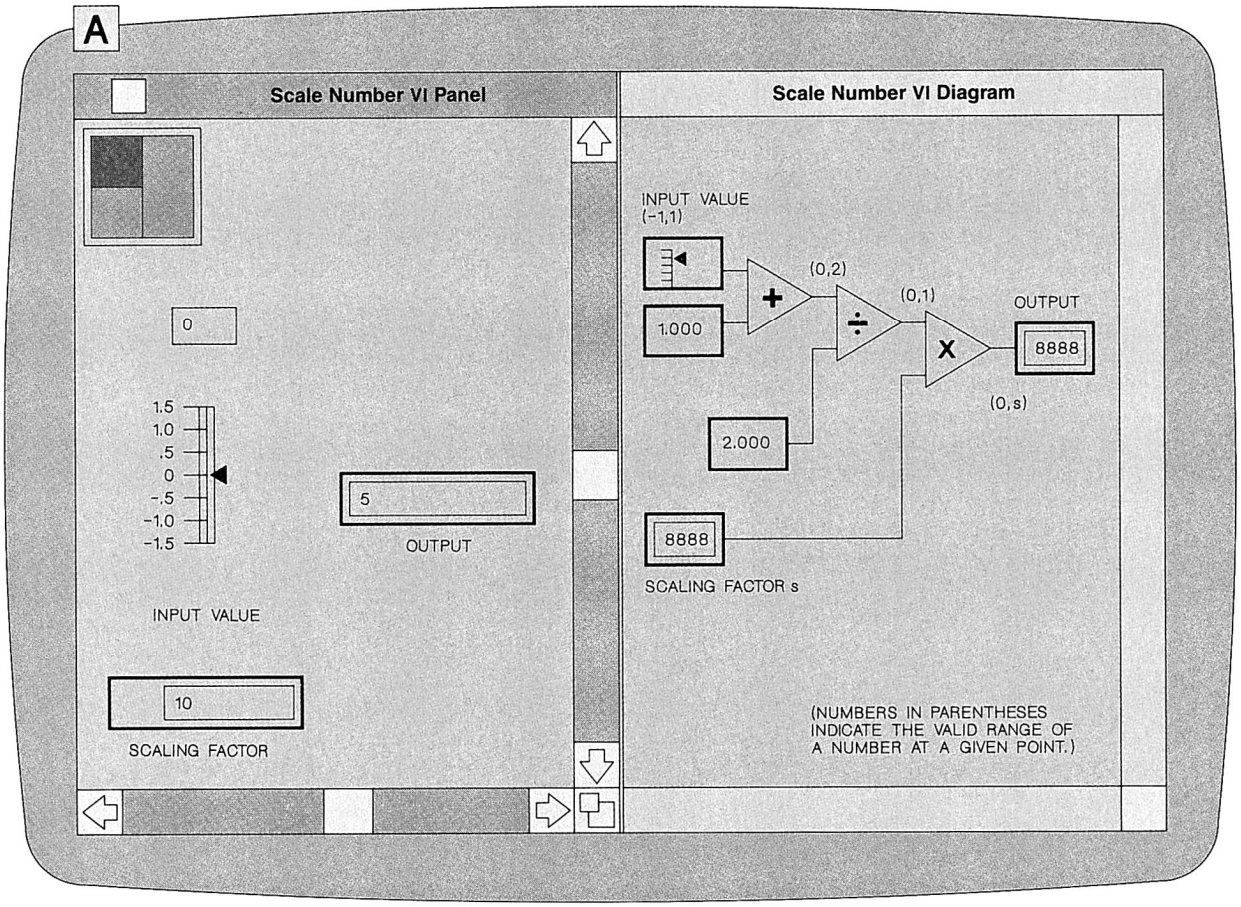
According to Kiel and Shepherd,[90] "Before using it, we had the impression that Labview would make mathematical modeling possible for all life scientists who are not computer programmers. In most respects, Labview lived up to this expectation, but to some extent it did not. Certainly, the many useful features of Labview, such as the math routines and the graphic displays, can easily be used by the unsophisticated programmer. Similarly, simple diagrams are indeed self-explanatory programs. In our opinion, an inexperienced programmer could construct a much more sophisticated model in Labview than in other high-level languages. However, more elaborate Labview programs can be as difficult to decipher and debug as those in other languages. Similarly, like other computer languages, Labview can demand arcane programming tricks, and the methods that have to be used to store variable values and to branch on a particular condition are often not straightforward. Nevertheless, even the accomplished programmer will find much to admire and use in Labview . . ."

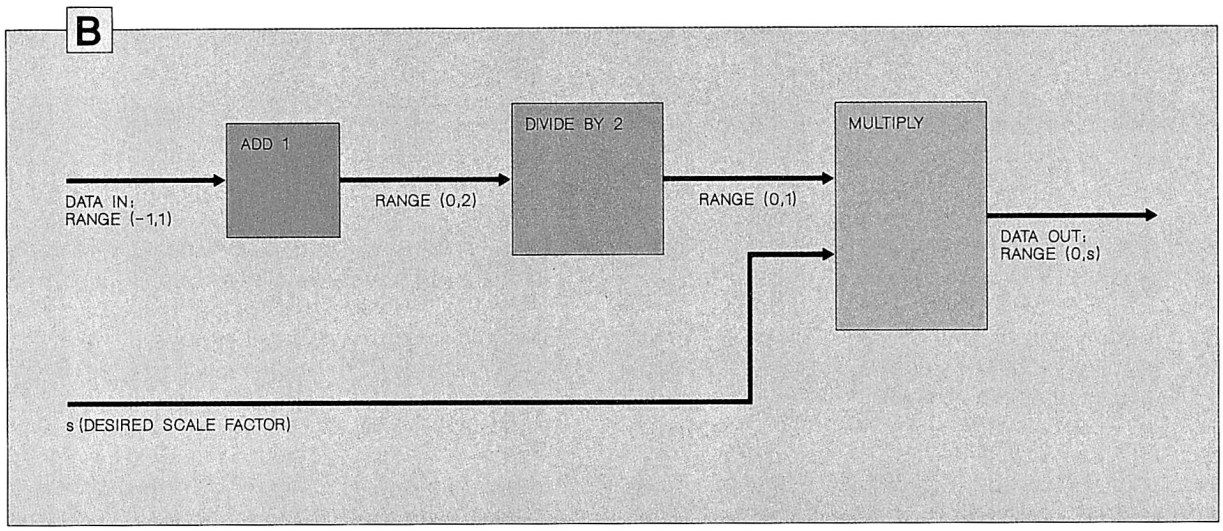
Forms are considered to be a natural interface between a human and a computer.

Forms-oriented programming languages

Forms are considered to be a natural interface between a human and a computer because a large number of users are familiar with forms. The success story of spreadsheet programs testifies to the appeal of the table- or form-oriented approach. Spreadsheets are designed for special purposes. They have succeeded admirably in letting users do the table-oriented computations, but their functions are limited. As Rich and Waters[91] put it, "a spreadsheet is a concise domain-specific interface that makes it remarkably easy to write certain kinds of programs and startlingly hard to write other kinds of pro-

Figure 5 **The scale number virtual instrument** (A is from G.M. Vose and G. Williams, "LabVIEW: Laboratory Virtual Instrument Engineering Workbench," *BYTE*, 11, No. 9, September 1986. Reprinted with permission.)





grams." Exploring the strength and the weakness of spreadsheets is an interesting topic in itself and is outside the scope of this paper.

Other applications that take a forms-oriented approach generally involve data entry and display, as well as database query and maintenance. Forms-oriented languages that support these applications include QBE,[92] QBE/OBE,[93] QBE/PC,[94] QPE,[95] FORMANAGER,[96] IDEAL,[97] "Fill-in-the-form programming,"[98] FILLIN,[99] Forms,[100] PICQUERY,[101] and FORMAL,[102] to name a few. Most of these languages are designed to support relational databases. In contrast, FORMAL, a system developed and implemented at the IBM Los Angeles Scientific Center, supports hierarchical data structures of arbitrary complexity, and thus can perform more complex data manipulations than operations on the flat tables. More importantly, FORMAL differs from other forms-oriented languages in another respect: It offers users a greater degree of flexibility for a much wider range of applications. Its focus is on *data processing* applications of a broad scope. Data entry, display, and query are merely side benefits. In the following, we discuss FORMAL in more depth.

FORMAL: A nonprocedural visual programming language

FORMAL[102] is an acronym for Forms ORiented MAnipulation Language. It is implemented at the IBM Los Angeles Scientific Center as an experimental application development system for end users. The language is not designed to teach the would-be programmer the concepts of traditional programming, nor is it aimed at incorporating the existing "paper and pencil tools" as graphical extensions to conventional programming languages. Rather, it is designed for people to computerize many fairly complex data processing applications without having to learn or to labor over the intricacies of "programming."

FORMAL capitalizes on the user's familiarity with forms in several ways: First, stylized *form headings* are used as visual representations of data structures (which could be quite complex). As an example, Figure 6 shows the form heading and a few instances of PERSON data.

Note that the form heading serves as an unambiguous description of a data structure. The form name is placed on the top line of a form heading. The names of the first-level components (i.e., fields or groups) are shown in columns under the form name. The names of components of groups, in turn, are placed under the associated group names. Groups can be either repeating or nonrepeating. A nonrepeating group refers to a collection of consecutive fields (e.g., DATE as a nonrepeating group over MONTH, DAY, and YEAR). A repeating group may have multiple instances and is denoted by a pair of parentheses over the group name. Repeating groups can be nested (representing several levels of a branch) or in parallel (representing several branches of a tree). A double line signals the end of a form heading. To view the data, values are displayed under the form heading. The compactness of the form heading enables the visualization of many instances at a time.

Second, data processing activities are viewed as *form processes* (or a series of form processes), where each form process takes one or two forms as input and produces another form as output. For example, a Christmas party for employees' children is being planned. Gifts will be given to children under 11 years of age and whose parent works either at 'LA' or 'SF'. Different gifts are planned for each age group. The party organizer wishes to list by age for each location, the names of the children (KIDS) who are eligible for gifts (along with their parents' names). Figure 7A shows the desired GIFTLIST form. The process of transforming the existing information in the PERSON form into GIFTLIST is viewed as a form process.

Third and most important, *programs are also represented in forms.* Since data processing activities are viewed as form processes, a program, then, is a user's way of telling the computer what is desired as the result of a form process. In using FORMAL, one starts with the visual representation of an output form and completes a program by specifying a few relevant properties (SOURCE, MATCH, CONDITION, ORDER) under the form heading.

Briefly, SOURCE defines where to obtain the relevant data for the operation. MATCH specifies the fields to be matched if the output instances are constructed from two input sources. If specified, CONDITION describes the criteria for selecting instances from the input, and ORDER depicts the desired sequencing of instances within a form or within a group.

For example, a FORMAL program used to create the GIFTLIST form is shown in Figure 7B. The desired output (GIFTLIST) data structure is depicted in the form heading. SOURCE specifies that the instances of LOC, AGE, and KNAME are to be taken from the

Figure 6 A PERSON form

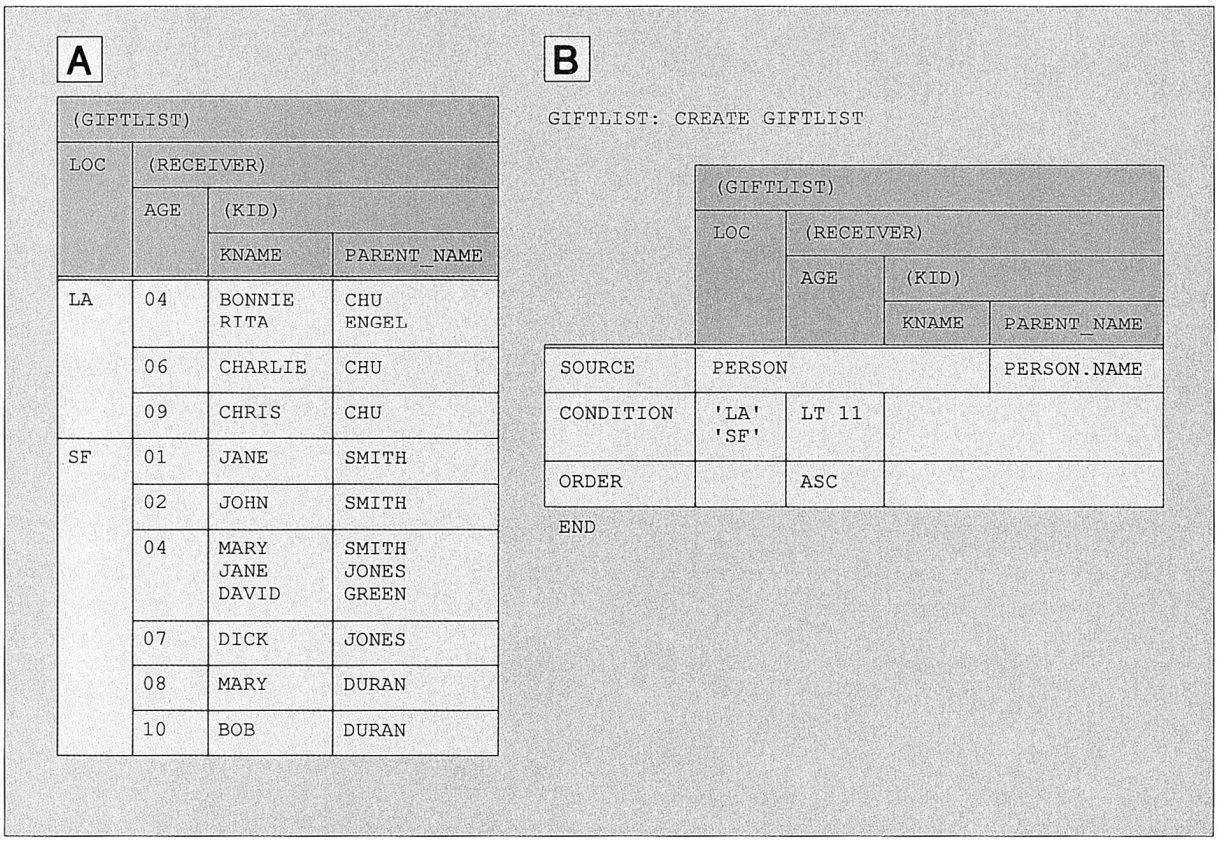
(PERSON)											
ENO	DNO	NAME	PHONE	JC	(KIDS)		(SCHOOL)			SEX	LOC
					KNAME	AGE	SNAME	(ENROLL)			
								YEARIN	YEAROUT		
05	D1	SMITH	5555	05	JOHN MARY	02 04	PRINCETON	1966 1972	70 76	F	SF
05	D1	SMITH	5555	05	JANE	01				F	SF
07	D1	JONES	5555	05	DICK JANE	07 04	SJS	1960	65	F	SF
							BERKELEY	1965	69		
11	D1	ENGEL	2568	05	RITA	01	UCLA	1970	74	F	LA
12	D1	DURAN	7610	05	MARY BOB JOHN	08 10 12				M	SF
19	D1	HOPE	3150	07	MARYLOU MARYANN	10 07				M	SJ
02	D2	GREEN	1111	01	RON DAVID	15 04				M	SF
20	D2	CHU	3348	10	CHARLIE CHRIS BONNIE	06 09 04	HONGKONG	1962	66	F	LA
							STANFORD	1967 1972	69 75		
21	D2	DWAN	3535	12			USC	1970	74	F	SJ
43	D2	JACOB	4643	09	PAULA PAUL	07 16	BERKELEY	1962	66	M	SJ
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.				
.	.	.	.	.	.	.					

corresponding fields of the PERSON form, and PARENT_NAME taken from the NAME field of the PERSON form. As a rule, CONDITIONs specified in two or more rows under the same field (column) are "ORed." Conditions specified under different fields are then "ANDed." In this example, only those having LOC equal to 'LA' or 'SF', and AGE less than 11 will be extracted, restructured, and placed in the output.

Note that the basic concepts underlying FORMAL are so simple that its power and scope of applications may not be immediately obvious. For example, writing a program equivalent to the GIFTLIST program in a procedural language would reveal that, in addition to data extraction and selection, algorithms must be developed, coded, and debugged for the required data restructuring. Similarly, a wide range of data processing applications of a much more complicated nature can be programmed in FORMAL. In most cases, the applications seem simple because data restructuring, often an integral but nontrivial part of an application, is *implied* in the differences in the output and input form headings, and *executable code* for the desired processing is *automatically* generated by the compiler.

Figure 8 shows a program that creates a DEPTMENT form from the PERSON and PROJECT forms. For documentation purposes, the structures of the two inputs are included in the program as comments.

Figure 7 GIFTLIST form: (A) desired output; (B) a FORMAL program to create GIFTLIST from PERSON



A

(GIFTLIST)			
LOC	(RECEIVER)		
	AGE	(KID)	
		KNAME	PARENT_NAME
LA	04	BONNIE RITA	CHU ENGEL
	06	CHARLIE	CHU
	09	CHRIS	CHU
SF	01	JANE	SMITH
	02	JOHN	SMITH
	04	MARY JANE DAVID	SMITH JONES GREEN
	07	DICK	JONES
	08	MARY	DURAN
	10	BOB	DURAN

B

```
GIFTLIST: CREATE GIFTLIST
```

	(GIFTLIST)		
	LOC	(RECEIVER)	
		AGE	(KID)
			KNAME PARENT_NAME
SOURCE	PERSON		PERSON.NAME
CONDITION	'LA' 'SF'	LT 11	
ORDER		ASC	

```
END
```

Normally, an instance of DEPTMENT will be produced only when a MATCH of input instances (where PERSON.DNO equals PROJECT.DNO) is found. However, when the PREVAIL option is specified (as in this example), an instance of output will be produced for each instance in the prevailing form, regardless of whether there is a matching instance or not. In case of no match, "NULL" values will be assigned to the missing values. Unlike comments, <*n*> (where *n* denotes a digit) is a *note* meaningful to the compiler. Following the notation <1> used in Figure 8, we observe that BUDGET in DEPTMENT is to be derived by PROJECT.PROJ.COST TIMES 1.5 (for cases where PROJ.COST is less than 50 000), or by PROJ.COST TIMES 1.6 (for other cases).

An experienced programmer would realize that the creation of the DEPTMENT form from the PERSON and PROJECTs forms requires not only extensive data restructuring as shown in Figure 9 (which involves "projection," "restriction," and "outer-join" of hierarchical data, and increasing hierarchical levels along one branch of a two-branch tree) but also the case-by-case assignments of new values, sorting of form instances within a form, and sorting of group instances within parent instance. Traditionally, algorithms for this rather complicated process must be carefully developed and tested by application programmers.

In summary, the data processing capabilities supported by FORMAL include: (1) data restructuring implied by the differences in the input and output form headings, (2) automatic iteration, (3) arithmetic and string operations, (4) case-by-case assignments, (5) supplying of field values by the user at execution time, (6) sorting of instances within a form or within a parent instance, and (7) aggregation (COUNT, SUM, AVG, MAX, MIN) progressively up the hierarchical path.

Figure 8 Program to produce DEPTMENT from PERSON and PROJECT forms

```
/****
```

(PROJECT)					
DNO	MGR	(PROJ)			
		PJNO	(EQUIP)		COST
			NAME	USAGE	

(PERSON)										
ENO	DNO	NAME	PHONE	JC	(KIDS)		(SCHOOL)		SEX	LOC
					KNAME	AGE	SNAME	(ENROLL)		
								YEARIN YEAROUT		

```
****/
```

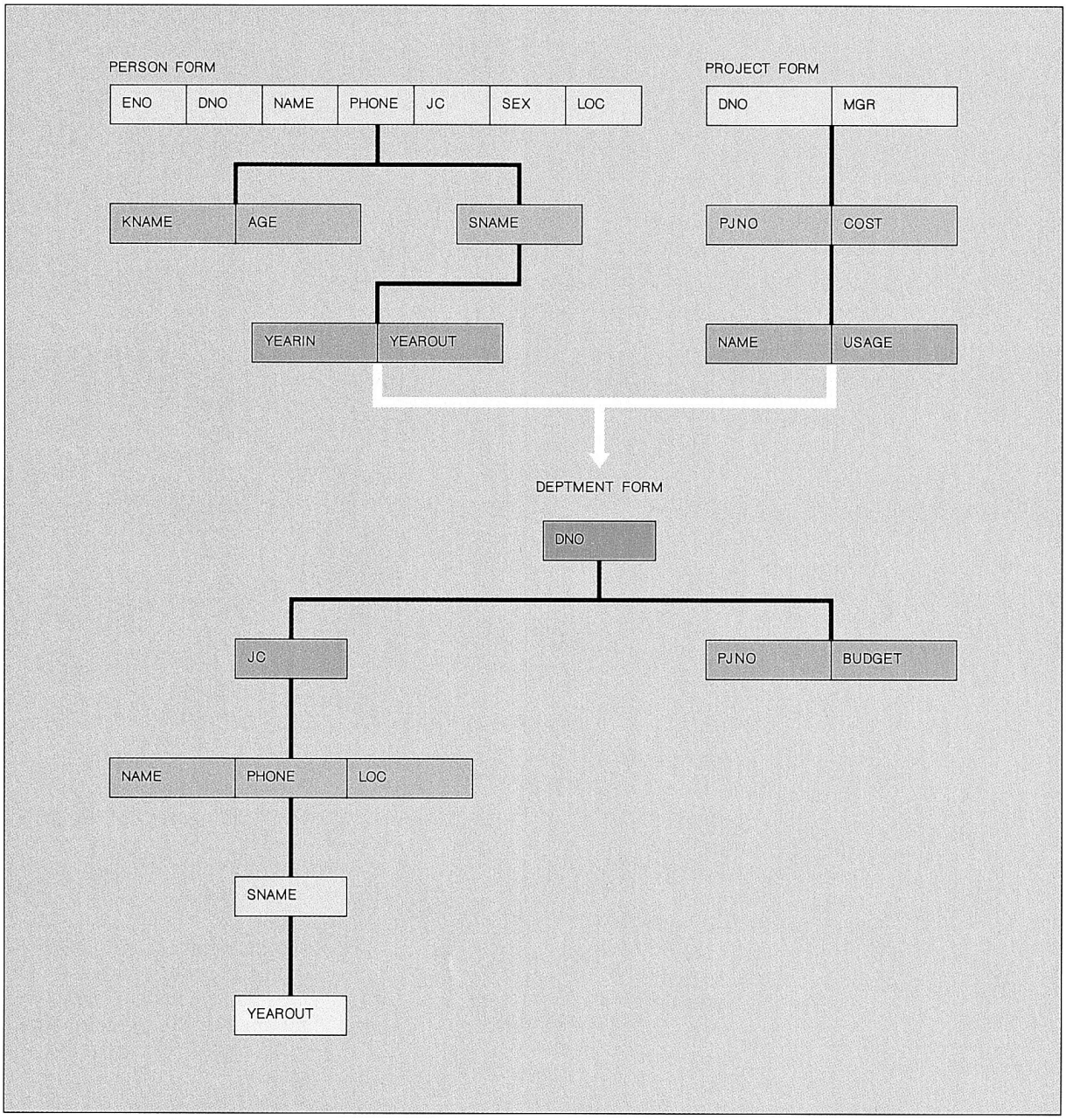
```
DEPTMENT: CREATE DEPTMENT
```

(DEPTMENT)								
DNO	(RESOURCE)						(PROJ)	
	JC	(EMPLOYEE)					PJNO	BUDGET
		NAME	PHONE	LOC	(SCHOOL)			
					SNAME	(ENROLL)		
						YEAROUT		

SOURCE	PERSON					PROJECT	< 1 >
< 1 >	PROJECT.PROJ.COST TIMES 1.5 WHERE PROJ.COST LT 50000 PROJECT.PROJ.COST TIMES 1.6 OTHERWISE						
MATCH	PERSON.DNO, PROJECT.DNO ELSE PERSON, PROJECT PREVAIL						
CONDITION		GE '05'		'LA' 'SJ'			
ORDER	ASC	DES	ASC				

```
END
```

Figure 9 Structural transformations associated with producing DEPTMENT from PERSON and PROJECT



With these capabilities, a wide range of fairly complex data processing applications can be accomplished. Yet, FORMAL is nonprocedural. There are no prescriptive or control constructs in FORMAL. Users do not tell the computer what steps to follow in order to achieve the results. As far as users are concerned, no algorithms are to be designed and coded. Results are obtained *automatically* by running the compiler-generated code. This is made possible because the FORMAL compiler is able to take over, from the user, the "thinking and coding" process normally associated with writing algorithms for

data restructuring and data manipulation operations.[103]

The "thinking and coding" process carried out by the compiler is accomplished in two stages. In the first stage, the differences in the input and output data structures are recognized, and the applicability of various transformation rules are analyzed. The result is a plan for mapping the specified input to the desired output. In the second stage, construction begins. Embedded knowledge on the target language and the run-time efficiency considerations are utilized to implement the plan. The result is an executable program tailored for the task at hand.

FORMAL has succeeded in doing automatic programming for a wide range of fairly complex applications.

In short, FORMAL has succeeded in doing automatic programming for a wide range of fairly complex data processing applications. Its success is made possible because of the visual expressions (form outlines) incorporated as an integral part of the language. The natural tendency of people to draw pictures to show what they want is exploited in this approach. The result is a new style of programming that combines the "naturalness" of visual programming languages with the power of automatic programming techniques: "*What you sketch is what you get!*"

Summary

The programming process is a many-phased endeavor. Work on visual programming explores the use of visual representations for all facets of the programming process. In this paper, we have first shown the diversity of activities in this rapidly growing area and categorized the activities according to their prominent characteristics. We then focused our attention on one facet of visual programming, namely, visual programming languages.

The work on visual programming languages has gained momentum in recent years primarily because the falling cost of graphics-related hardware and software has made it possible to employ visual expressions as a means of programming. However, aside from the fact that they all use some sort of visual expression as language components, visual programming languages differ significantly from each other in their goals, their approaches, their design philosophies, and their appearances. The diversity is not really surprising since the examples of visual programming languages have come into existence from various backgrounds and were designed for different audiences and purposes.

Generally speaking, diagrammatic systems use charts and diagrams either as graphical extensions to conventional programming languages, or as machine-interpretable units used in conjunction with conventional programming languages. The diagrammatic forms used by these systems are based on the kinds of program abstractions already used on blackboard or paper as visual aids for some aspects of programming concepts which can be better expressed in diagrams than in text (for example, flow of control, flow of data, time dependence of concurrent systems, transitions of states, etc.).

With iconic programming languages, icons are specifically designed to be the essential language elements playing a central role. A programming process essentially involves selecting or composing icons, or both, placing them in proper juxtaposition on the screen, and connecting the icons by paths to indicate the desired flow of control. Iconic systems appear to be more fun. They provide a more interesting way for novices to learn programming than conventional, text-based programming languages.

It is interesting to note that in spite of the significant differences in the goals and approaches taken by the diagrammatic and the iconic programming languages, they have some things in common. Conceptually they both are engaged in *algorithmic programming*. To use these languages other than for play, one must understand the basic concepts of traditional programming languages such as variables, operations, flow of data, flow of control, iterations, subprograms, recursions, run-time stacks, parameter passing, time dependencies, etc. And, one must work out the procedural aspects of programming. In order to produce an executable program, a user must develop and specify the algorithms in sufficient detail.

In contrast, the form-based language FORMAL is designed for people who need to computerize their applications but have no desire to learn to program or no time to program in step-by-step instructions. There are no pretty pictures in FORMAL, only simple, stylized form outlines. Consequently, FORMAL does not appear to be as exciting as iconic languages. But with FORMAL, a user can accomplish a wide range of data processing applications without the need to understand any programming concepts, and without the necessity to develop algorithms and make detailed specifications. With FORMAL, automatic programming plays a vital supporting role to visual programming.

Obviously, however innovative visual programming may be, it is by no means a panacea, let alone the best solution for *all* programming problems. It has advantages in some areas and limitations in others. The framework established by the categorization enables us to look at the advantages and limitations in a more focused fashion.

Historically, major new developments have come from the intersection of a multiplicity of ideas. As a new field, visual programming shows a lot of potential, but we are just at the beginning. In the last few years, we have seen rapid and diversified growth. It is hoped that in the next few years we will witness a new type of quest: maturation with scientific and engineering discipline and cross-fertilization with other areas of learning.

Acknowledgment

The author wishes to thank John Kepler and Jim Jordan for their management support.

Cited references

1. B. A. Myers, *The State of the Art in Visual Programming and Program Visualization*, Technical Report CMU-CS-88-114, Carnegie Mellon University, Pittsburgh, PA (February 1988).
2. N. C. Shu, *Visual Programming*, Van Nostrand Reinhold Company, New York (1988).
3. B. Shneiderman, "Direct manipulation: A step beyond programming languages," *Computer* **19**, No. 8, 57–69 (August 1983).
4. C. F. Herot, "Spatial management of data," *ACM Transactions on Database Systems* **5**, No. 4, 493–514 (December 1980).
5. N. H. McDonald, "A multi media approach to the user interface," in *Human Factors and Interactive Computer Systems*, Y. Vissiliou, Editor, Ablex Publishing Corp., Norwood, NJ (1984), pp. 105–116.
6. H.-D. Boecker, G. Fischer, and H. Nieper, "The enhancement of understanding through visual representations," *Proceedings of CHI'86 Conference, Human Factors in Computing Systems* (August 1986), pp. 44–50.
7. K. J. Goldman, S. A. Goldman, P. C. Kanellakis, and S. B. Zdonik, "ISIS: Interface for a Semantic Information System," *Proceedings of ACM SIGMOD International Conference on the Management of Data* (May 1985), pp. 328–342.
8. B. A. Myers, R. Chandhok, and A. Sareen, "Automatic data visualization for novice Pascal programmers," *Proceedings of the 1988 IEEE Workshop on Visual Languages* (October 1988), pp. 192–198.
9. M. Erradi and C. Frasson, "Interaction with IBS: An icon-based system," *Proceedings of Computer Graphics Conference*, Tokyo (1986), pp. 159–171.
10. P. B. Sevbold, "Metaphor computer systems: A quiet revolution," *Office Computing Report* **11**, No. 8, 1–15 (August 1988).
11. L. M. Burns, J. L. Archibald, and A. Malhotra, "A graphical entity-relationship database browser," *Proceedings of the 21st Annual Hawaii International Conference on Systems Sciences* **2**, 694–704 (1988).
12. R. Baecker and A. Marcus, "Design principles for the enhanced presentation of computer program source text," *Proceedings of CHI'86, Human Factors in Computing Systems* (April 1986), pp. 51–58.
13. R. Levien, "Visual programming," *BYTE* **11**, No. 2, 135–144 (February 1986).
14. G. M. Karam, "An icon-based design method for Prolog," *IEEE Software*, 51–65 (July 1988).
15. W. Teitelman, "A tour through CEDAR," *IEEE Transactions on Software Engineering* **SE-11**, No. 3, 285–302 (March 1985).
16. S. P. Reiss, "PECAN: Program development systems that support multiple views," *IEEE Transactions on Software Engineering* **SE-11**, No. 3, 276–285 (March 1985).
17. S. P. Reiss, "Working in the GARDEN environment for conceptual programming," *IEEE Software* **4**, No. 6, 17–27 (November 1987).
18. T. G. Moher, "PROVIDE: A Process Visualization and Debugging Environment," *IEEE Transactions on Software Engineering* **14**, No. 6, 849–857 (June 1988).
19. S. Isoda, T. Shimomura, and Y. Ono, "VIPS: A visual debugger," *IEEE Software* **4**, No. 3, 8–19 (May 1987).
20. C. D. Hughes and J. M. Moshell, "Visible Pascal: A graphics-based learning environment," *Proceedings of Computer Graphics 86*, National Computer Graphics Association (May 1986), pp. 401–411.
21. R. L. London and R. A. Duisberg, "Animating programs using Smalltalk," *Computer* **18**, No. 8, 61–71 (August 1985).
22. M. H. Brown and R. Sedgewick, "Techniques for algorithm animation," *IEEE Software* **2**, No. 1, 28–39 (January 1985).
23. M. H. Brown, "Exploring algorithms using Balsa-II," *Computer* **21**, No. 5, 14–36 (May 1988).
24. B. W. Lampson, "Hints for computer system design," *IEEE Software* **1**, No. 1, 11–28 (January 1984).
25. G. P. Brown, R. T. Carling, C. F. Herot, D. A. Kramlich, and P. Souza, "Program Visualization: Graphical support for software development," *Computer* **18**, No. 8, 27–35 (August 1985).
26. A. Wasserman and P. Pircher, "A graphical, extensible integrated environment for software development," *ACM SIGPLAN Notices* **22**, No. 1, 131–142 (January 1987).
27. V. Vemuri, Editor, *Software Development: Computer Aided SE (CASE)*, Technology Series, IEEE Computer Society Press, Los Angeles (1988).

28. M. Moriconi and D. F. Hare, "Visualizing program designs through PegaSys," *Computer* **18**, No. 8, 72–85 (August 1985).
29. G. Raeder, *Programming in Pictures*, Ph.D. dissertation, Department of Computer Science, University of Southern California, Los Angeles (November 1984).
30. A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, Vol. 2, William Kaufmann, Inc., Los Altos, CA (1982), Chapter X.
31. A. W. Biermann, "Automatic programming: A tutorial on formal methodologies," *Journal of Symbolic Computation* **1**, 119–142 (1985).
32. A. W. Biermann and R. Krishnaswamy, "Constructing programs from example computations," *IEEE Transactions on Software Engineering* **SE-2**, No. 3, 141–153 (September 1976).
33. D. C. Smith, *Pygmalion: A Creative Programming Environment*, Ph.D. dissertation, Technical Report STAN-CS-75-499, Department of Computer Science, Stanford University, Stanford, CA (1975).
34. G. A. Curry, *Programming by Abstract Demonstration*, Ph.D. dissertation, Technical Report 78-03-02, University of Washington, Seattle, WA (March 1978).
35. D. C. Halbert, *Programming by Example*, Ph.D. dissertation, Computer Science Division, University of California, Berkeley (1984).
36. R. V. Rubin, E. J. Golin, and S. O. Reiss, "ThinkPad: A graphical system for programming by demonstration," *IEEE Software* **2**, No. 2, 73–79 (March 1985).
37. H. Lieberman, "An example-based environment for beginning programmers," *Instructional Science* **14**, 277–292 (1986).
38. W. Finzer and L. Gould, "Programming by rehearsal," *BYTE* **9**, No. 6, 187–210 (June 1984).
39. D. Kozen, T. Teitelbaum, W. Chen, J. Field, W. Pugh, and B. V. Zanden, "ALEX—An alexical programming language," *Proceedings of the 1987 IEEE Workshop on Visual Languages* (August 1987), pp. 315–329.
40. B. A. Myers, "Creating dynamic interaction techniques by Demonstration," *CHI + GI 1987 Conference Proceedings* (April 1987), pp. 271–278.
41. G. Rogers, "Visual programming with objects and relations," *Proceedings of the 1988 IEEE Workshop on Visual Languages* (October 1988), pp. 29–36.
42. D. L. Maulsby and I. H. Witten, "Inducing programs in a direct-manipulation environment," *Proceedings of CHI'89, Human Factors in Computing Systems* (1989), pp. 57–62.
43. S. K. Chang, J. Reuss, and B. H. McCormick, "Design considerations of a pictorial database system," *International Journal on Policy Analysis and Information Systems* **1**, No. 2, 49–70 (January 1978).
44. S. K. Chang, M. O'Brien, J. Read, R. Borovec, W. H. Cheng, and J. S. Ke, "Design considerations of a database system in a clinical network environment," *Proceedings of the National Computer Conference* (1976), pp. 277–286.
45. R. Berman and M. Stonebraker, "GEO-QUEL: A system for the manipulation and display of geographic data," *Computer Graphics* **11**, No. 2, 186–191 (Summer 1977).
46. G. D. Held, M. Stonebraker, and E. Wong, "INGRES: A relational database system," *Proceedings of AFIPS, National Computer Conference* **44**, 409–416 (1975).
47. N. Roussopoulos and D. Leifker, "An introduction to PSQL: A Pictorial Structured Query Language," *Proceedings of the 1984 IEEE Computer Society Workshop on Visual Languages*, Hiroshima, Japan (1984), pp. 77–87.
48. K. Assmann, R. Venema, and K. H. Hohne, "The ISQL language: A software tool for the development of pictorial information systems in medicine," in *Visual Languages*, S. K. Chang, et al., Editors, Plenum Publishing Corporation, New York (1986), pp. 261–284.
49. G. Y. Tang, "A logical data organization for the integrated database of pictures and alphanumerical data," *Proceedings of the IEEE Workshop on Picture Data Description and Management* (August 1980), pp. 158–166.
50. D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Loroe, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A unified approach to data definition, manipulation, and control," *IBM Journal of Research and Development* **20**, No. 6, 560–575 (1976).
51. M. Hirakawa, N. Monden, I. Yoshimoto, M. Tanaka, and T. Ichikawa, "HI-VISUAL: A Language Supporting Visual Interaction in Programming," in *Visual Languages*, S. K. Chang, et al., Editors, Plenum Publishing Corporation, New York (1986), pp. 233–259.
52. L. Cardelli and R. Pike, "Squeak: A language for communicating with mice," *Proceedings of ACM SIGGRAPH '85* (July 1985), pp. 199–204.
53. P. A. Szekely and B. A. Myers, "A user interface toolkit based on graphical objects and constraints," *Proceedings of OOPSLA '88 Conference* (September 1988), pp. 36–45. (Also special issue of *ACM SIGPLAN Notices* **23**, No. 11, November 1988.)
54. F. J. Newbery, "An interface description language for graph editors," *Proceedings of the 1988 IEEE Workshop on Visual Languages* (October 1988), pp. 144–149.
55. T. W. Malone, "Heuristics for designing enjoyable user interfaces: Lessons from computer games," in *Human Factors in Computer Systems*, Thomas and Schneider, Editors, Ablex Publishing Corp., Norwood, NJ (1984).
56. R. J. K. Jacob, "A state transition diagram language for visual programming," *Computer* **18**, No. 8, 51–59 (August 1985).
57. R. M. Keller and W-C. J. Yen, "A graphical approach to software development using function graphs," *Digest of Papers*, Compcon Spring 81 (February 1981), pp. 156–161.
58. A. L. Davis and S. A. Lowder, "A sample management application program in a graphical data driven programming language," *Digest of Papers*, Compcon Spring 81 (February 1981), pp. 162–167.
59. A. I. Wasserman, P. A. Pircher, D. T. Shewmake, and M. L. Kersten, "Developing interactive information systems with the USE software engineering methodology," *IEEE Transactions on Software Engineering* **SE-12**, No. 2, 326–345 (February 1986).
60. M. Graf, "A visual environment for the design of distributed systems," *Proceedings of the 1987 IEEE Workshop on Visual Languages* (August 1987), pp. 330–344.
61. S. Coote, J. Gallagher, J. Mariani, T. Rodden, A. Scott, and D. Shepherd, "Graphical and iconic programming languages for distributed process control," *Proceedings of the 1988 IEEE Workshop on Visual Languages* (October 1988), pp. 183–190.
62. P. D. Stotts, "The PFG environment: Parallel programming with Petri Net semantics," *Proceedings of the 21st Hawaii International Conference on System Sciences* (January 1988), pp. 630–638.
63. N. Cunniff, R. P. Taylor, and J. B. Black, "Does programming language affect the type of conceptual bugs in beginners' programs? A comparison of FPL and Pascal," *Proceedings of CHI'86, Human Factors in Computing Systems* (April 1986), pp. 175–182.
64. J. L. Diaz-Herrera and R. C. Flude, "Pascal/HSD: A graphical programming system," *Proceedings of IEEE COMPSAC*

'80 (1980), pp. 723–728.
65. M. B. Albizuri-Romero, "GRASE—A graphical syntax directed editor for structured programming," *ACM SIGPLAN Notices* **19,** No. 2, 28–37 (February 1984).
66. H. P. Frei, D. L. Weller, and R. Williams, "A graphics-based programming support system," *Proceedings of ACM SIGGRAPH '78* (August 1978), pp. 43–49.
67. M. C. Pong and N. Ng, "PIGS—A system for programming with interactive graphical support," *Software Practice and Experience* **13** (1983), pp. 847–855.
68. M. C. Pong, "A graphical language for concurrent programming," *Proceedings of the 1986 IEEE Workshop on Visual Languages* (June 1986), pp. 26–33.
69. I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *ACM SIGPLAN Notices* **8,** No. 8, 12–26 (August 1973).
70. C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM* **21,** No. 8, 666–677 (August 1978).
71. W. W. Dijkstra, "Guarded commands, nondeterminancy, and formal derivation of programs," *Communications of the ACM* **18,** No. 8, 453–457 (August 1975).
72. D. C. Smith, C. Irby, and R. Kimball, "The Star user interface: An overview," *Proceedings of the National Computer Conference* (1982), pp. 515–528.
73. R. Purvy, J. Farrell, and P. Klose, "The design of Star's records processing: Data processing for the noncomputer professional," *ACM Transactions on Office Information Systems* **1,** No. 1, 3–24 (January 1983).
74. F. Lakin, "Spatial parsing for visual languages," in *Visual Languages*, S. K. Chang, et al., Editors, Plenum Publishing Corp., New York (1986).
75. G. Cattaneo, A. Guercio, S. Lavialdi, and A. Tortora, "ICONLISP: An example of a visual programming language," *Proceedings of the 1986 IEEE Workshop on Visual Languages* (June 1986), pp. 22–25.
76. M. Edel, "The Tinkertoy graphical programming environment," *IEEE Transactions on Software Engineering* **14,** No. 8, 1110–1115 (August 1988).
77. T. Kurita and K. Tamura, "Dialog.I: An iconic programming system based on logic programming," *Bulletin of the Electrotechnical Laboratory* (Japan) **48,** No. 12, 966–975 (1984).
78. A. Giacalone, M. C. Rinard, and T. W. Doeppner, Jr., "IDEOSY: An ideographic and interactive program description system," *ACM SIGPLAN Notices* **19,** No. 5, 15–20 (May 1984).
79. A. Giacalone and S. A. Smolka, "Integrated environments for formally well-founded design and simulation of concurrent systems," *IEEE Transactions on Software Engineering* **14,** No. 6, 787–802 (June 1988).
80. R. Milner, *A Calculus of Communication Systems*, Springer-Verlag, New York (1980).
81. E. P. Glinert and S. L. Tanimoto, "Pict: An interactive graphical programming environment," *Computer* **17,** No. 11, 7–25 (November 1984).
82. E. P. Glinert, "Towards 'second generation' interactive graphical programming environments," *Proceedings of the 1986 IEEE Workshop on Visual Languages* (June 1986), pp. 61–70.
83. T. D. Kimura, J. W. Choi, and J. M. Mack, *A Visual Language for Keyboardless Programming*, Technical Report WUCS-86-6, Washington University, St. Louis, MO (June 1986).
84. S. Matwin and T. Pietrzykowski, "PROGRAPH: A preliminary report," *Computer Languages* **10,** No. 2, 91–126 (1985).
85. M. Hirakawa, S. Iwata, I. Yoshimoto, M. Tanaka, and T. Ichikawa, "HI-VISUAL iconic programming," *Proceedings of the 1987 IEEE Workshop on Visual Languages* (August 1987), pp. 305–314.
86. G. M. Vose and G. Williams, "LabVIEW: Laboratory Virtual Instrument Engineering Workbench," *BYTE* **11,** No. 9, 84–92 (September 1986).
87. P. T. Cox and T. Pietrzykowski, *A Very High-Level, Pictorial, Object-Oriented Programming Language for Computer Science Education,* School of Computer Science, Technical Report TR-1-1988 (September 1988).
88. D. N. Smith, "Visual programming in the interface construction set," *Proceedings of the 1988 IEEE Workshop on Visual Languages* (October 1988), pp. 109–120.
89. J. G. Bonar and B. W. Liffick, *A Visual Programming Language for Novices,* Technical Report LSP-5, University of Pittsburgh, PA (September 1987).
90. J. W. Kiel and A. P. Shepherd, "A graphic computer language for physiology simulations," *Computers in Life Science Education* **5,** No. 7, 49–56 (July 1988).
91. C. Rich and R. C. Waters, "Automatic programming: Myths and prospects," *Computer* **21,** No. 8, 40–51 (August 1988).
92. M. M. Zloof, "Query-by-example," *AFIPS Conference Proceedings, National Computer Conference* (1975), pp. 431–438.
93. M. M. Zloof, "QBE/OBE: A language for office and business automation," *Computer* **14,** No. 5, 13–22 (May 1981).
94. K. T. Huang, A. Bolmarcich, S. Katz, and R. Li, "QBE/PC: The design of an integrated software system for a personal computer," *Proceedings of 1986 ACM SIGSMALL/PC Symposium on Small Systems* (December 1986), pp. 92–100.
95. N. S. Chang and K. S. Fu, "Picture query languages for pictorial data-base systems," *Computer* **14,** No. 11, 23–33 (November 1981).
96. S. B. Yao, A. R. Hevner, Z. Shi, and D. Luo, "FORMANAGER: An office forms management system," *ACM Transactions of Office Information Systems* **2,** No. 3, 235–262 (July 1984).
97. N. A. Rin, "An interactive applications development system and support environment," in *Automated Tools for Information System Design*, H. J. Schneider and A. I. Wasserman, Editors, North-Holland Publishing Company, Amsterdam (1982), pp. 177–213.
98. L. A. Rowe, "'Fill-in-the-form' programming," *Proceedings of VLDB85* (August 1985), pp. 394–404.
99. S. P. Wartik and M. H. Penedo, "FILLIN: A reusable tool for form-oriented software," *IEEE Software* **6,** No. 3, 61–69 (March 1986).
100. A. L. Ambler, "Forms: Expanding the visualness of sheet languages," *Proceedings of the 1987 IEEE Workshop on Visual Languages* (August 1987), pp. 105–117.
101. T. Joseph and A. F. Cardenas, "PICQUERY: A high level query language for pictorial database management," *IEEE Transactions on Software Engineering* **14,** No. 5, 630–638 (May 1988).
102. N. C. Shu, "FORMAL: A forms oriented visual directed application development system," *Computer* **18,** No. 8, 38–49 (August 1985).
103. N. C. Shu, "Automatic data transformation and restructuring," *Proceedings of the IEEE Third International Conference on Data Engineering* (February 1987), pp. 173–180.

Nan C. Shu *IBM Los Angeles Scientific Center, 2525 Colorado Avenue, Santa Monica, California 90404.* Ms. Shu joined the IBM Thomas J. Watson Research Center in 1964 as a member of an

experimental programming group responsible for the development of the first IBM virtual memory system. In 1972 she joined the IBM San Jose research laboratory where she was active in data translation, database design, and office automation. She is currently a senior scientific staff member at the Los Angeles Scientific Center, where she is responsible for the design and implementation of a forms-oriented and visual-directed application development system for end users. A patent on an Automatic Data Restructurer (based on her work) was issued by the U.S. Patent and Trademark Office. Ms. Shu has received several IBM awards and has been the author of numerous published technical papers as well as a book on visual programming. She is a senior member of the IEEE.