# Toward Intelligent Interfaces for Graphic Design Applications

**Henry Lieberman**

Visible Language Workshop
Media Laboratory
Massachusetts Institute of Technology
Cambridge, Mass. 02139 USA

Conference on Fifth Generation Computer Systems, Tokyo, November 1988

Abstract

Progress in designing intelligent interfaces for graphic design applications such as electronic publishing and illustration will depend crucially on the application of symbolic programming techniques from artificial intelligence. But the traditional "expert systems" methodology breaks down when applied in fields where visual problem solving, as opposed to verbal or symbolic problem solving, is paramount. Expert graphic designers are fluent in the generation and critique of visual examples rather than the articulation of such abstract principles as `IF-THEN` rules.

A machine learning technique that holds promise for capturing the expertise of skilled problem solvers in visual domains is "programming by example". The designer constructs and edits examples using an interactive graphical interface, such as a graphic editor for illustration or page layout. The system simultaneously records the designer's actions, using a symbolic procedural representation. The designer may then converse with the system about how to generalize the actions to apply to future examples.

This paper will describe work underway at the Visible Language Workshop of MIT's Media Laboratory, where artificial intelligence programmers and graphic designers are collaborating to develop graphical interfaces that can bridge the gap between the "hands-on" world of the designer, and the more abstract, symbolic world of the programmer.

**AI should study intelligent problem solving in visual design domains**

Artificial Intelligence has had, in recent years, several successes in creating problem solving programs and tools to aid people in solving problems in such fields as medicine, business and engineering. Such successes all depend crucially on the ability to analyze the knowledge involved in solving problems in these fields and making it explicit enough to be embodied in a symbolic computer language. The benefits of doing so are twofold: First, it enables the creation of programs which help professionals in these fields in their daily work, which is of great practical importance. Second, and more abstractly, it yields insight into the nature of knowledge in these fields and increases our understanding of how people achieve competence in these fields.

To date, though, the application of Artificial Intelligence techniques has been largely confined to domains in which thinking seems linguistic and symbolically oriented. People have been more reluctant to attack areas in which visual and aesthetic competence plays an important role. Perhaps this is because of a common feeling that these areas are so fundamentally different that computational methods will prove ineffective. But this prejudice has never been sufficiently tested experimentally.

The time seems ripe now for exploring the application of heuristic problem solving techniques to areas like the visual arts and music, for several reasons. First, technology in applications of artificial intelligence to the "harder" domains has reached a sufficient state of maturity that transplanting it to domains far removed from its original application no longer seems out of the question. Second, the proliferation of computer graphics and sound capabilities on inexpensive personal computers has made these applications commercially important.

Problems in graphic design seem like a good testbed for exploring the cognitive processes involved in visual and aesthetic design. Excellence in graphic design is crucially dependent on making good aesthetic and stylistic choices about arrangement of visual elements. Yet the problem is not so difficult and underconstrained as that of artistic expression in the fine arts. It is easier to judge the success of a program which attempts to perform automatic layout of a magazine than to judge a program that tries to produce a beautiful abstract painting.

The success of the phenomenon of "desktop publishing" is one example of how computer tools for visual and aesthetic design are becoming important to large communities of users. The effectiveness of communication in computer-generated print media is dependent upon the quality of aesthetic decisions made during its production. Current generation layout and illustration tools are "dumb"; they represent only the geometric relationships between visual elements. With the introduction of knowledge representation techniques from artificial intelligence, decisions can also be based on the function of visual components in communicating ideas, and on the denotation relations between visual elements and what they represent. These techniques hold out the promise of making "smart" tools that free designers from tedious and repetitive layout chores.

**How does knowledge engineering in visual domains differ from verbal domains?**

The first paradigm for Artificial Intelligence applications, and the one most prevalent in the commercial world, is the *knowledge engineering* approach [Buchanan and Shortliffe 84]. A knowledge engineer interviews acknowledged experts in the field, gets the expert to communicate how problems are solved, and tries to codify that knowledge in the form of `IF-THEN` rules which can later be used by the computer to reproduce the expert's behavior. The great advantages of this approach are that it does not require much preconceived analysis of the target domain, and uses relatively simple computational mechanisms. It has not yet been applied in earnest in the field of graphic design, but holds promise since we have ready access to experts in graphic design and examples of their work.

My suspicion is that in the long run, the knowledge engineering approach will prove limited in the domains of visual and aesthetic design. Knowledge engineering works best in domains that are filled with enormous numbers of facts and involve relatively shallow processing, whereas I suspect aesthetics involves relatively little concrete knowledge but deep conceptualization. Also, designers tend to be relatively inarticulate in expressing their ideas in purely verbal form. This is not to meant detract from the importance of doing experiments applying knowledge engineering to graphic design, since they are likely to yield short-term benefits, but only to suggest that this approach alone may not be sufficient.

**Programming by example is an alternative to traditional programming for communicating procedures**

In addition to asking designers to share their knowledge verbally, a more effective approach may be to *watch* them in the process of design, and equip a system with the ability to learn directly from a designer's graphical and gestural actions. Artificial Intelligence contributes the techniques of *programming by example* [Lieberman 82], [Lieberman 84] and *learning by analogy* [Mitchell 83], [Lieberman 85], which may be very well suited to the task of capturing design knowledge in computational form.

The designer presents a concrete example of a design problem to the machine, then demonstrates the steps of solving it, using graphical and gestural commands. The designer also presents some indication of *why* each step is taken, and how it can be generalized to further examples. The system displays the effect of the actions in the concrete example, and also remembers the steps for later use. Subsequently, the computer can apply an "analogous" procedure to any example sufficiently similar to the ones on which it was taught. Programming by example is a powerful technique that can bridge the cultural gap between the "hands-on" world of the designer, and the more abstract and symbolic world of the programmer.

We have deliberately chosen to keep the learning procedure used quite simple. We are avoiding sophisticated inductive inference procedures so that the designer can easily grasp the generalizations performed by the system. The cost is requiring more numerous examples and more verbose explanations. Later, more complex inductive procedures could be substituted, while retaining the interface. The learning procedure employed [essentially the same as in [Lieberman 82] and [Lieberman 84]] is similar to what is usually called *explanation-based generalization* [Mitchell 83] except that the order in which the steps take place is a bit different. In classical explanation-based generalization the explanation for a given generalization is presented all at once. Here, bits of explanation are provided incrementally, in order to cause the system to come up with a desired generalization, which may change in the course of the explanation. This might be termed *generalization-based explanation*.

**The road to design starts with simple procedures, gradually made more flexible**

The aesthetic decisions made by human designers are wonderfully creative and complex. People draw on a wealth of real-world knowledge and the enormous computational power of the human visual system. Short of solving the machine vision problem and achieving human-level performance in a real-world knowledge base, we cannot expect to reproduce human competence in graphic design anytime soon.

Rather, the near-term goal of this research is to produce computer systems that can usefully act as assistants to human designers. A person who is just beginning as an assistant to an experienced designer might start out by simply executing procedures whose content was already determined by the mentor. The expert designer would gradually communicate the reasons for the various steps performed, and illustrate them in a variety of concrete situations. As the student learns, the procedures become more flexible, and knowledge can be applied in less obvious situations. Eventually, the assistant becomes capable of solving analogous problems independently.

We are interested in explicitly modelling this process, with the computer playing the role of the design assistant. In a companion paper [Lieberman 88] I argue that there is evidence that design knowledge is actually taught by the presentation of visual examples, together with advice that directs the student's interpretive process for generalizing from the examples.

Like the beginning design assistant, a learning system must start out with simple procedures explicitly directed by a human expert. The role of the learner is initially limited to making "obvious" applications of the expert's procedures. Gradually, as more examples, and more sophisticated generalizations are accumulated, the relation between the original examples and the new problems becomes less obvious, and more in the nature of analogy. And, as many authors have observed, analogy is an essential component of creativity.

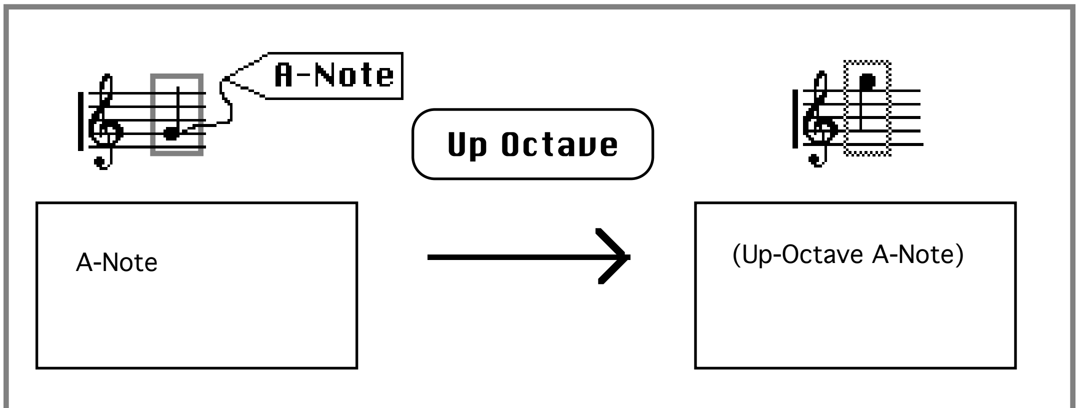
At this stage, our systems are still at the fairly literal level of learning graphic procedures by explicit direction. Later in this paper, we will present a detailed example of how a simple graphical procedure can be learned by explained example. The system contains some simple mechanisms for generalizing the procedure shown so it can later be applied in "analogous" cases. This will illustrate the approach.

**Knowledge acquisition from graphical interfaces through programming by example**

The key to making visual communication of design procedures work is establishing a correspondence between demonstrative actions like direct manipulation of graphical objects, and procedural concepts like functions, conditionals and recursion. This correspondence allows the designer to perform actions in the visual domain and use them as parts of procedures that capture knowledge about intelligent design. The technique of programming by example provides a bridge from the essentially concrete nature of pictures and the abstract nature of procedures.

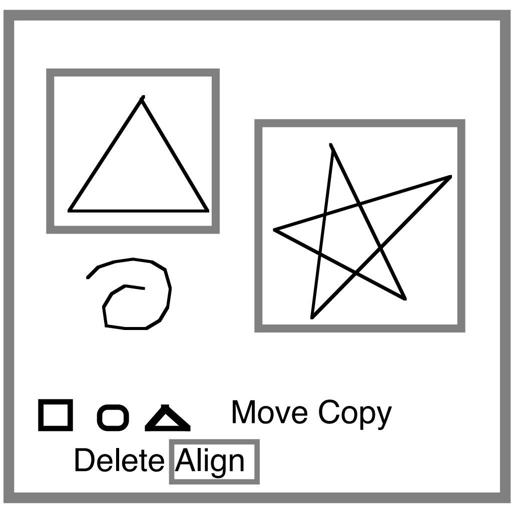
In [Lieberman 85] I described a graphical interface interpreter which exploited an analogy between menu-driven direct manipulation interfaces and procedural programming languages. Commands selected from menus act like functions do in a programming language; they are the means of accessing specific named aspects of a system's functionality. Graphical objects selected with a pointing device such as the mouse play the role of arguments to these functions. In a circuit design system, for example, pointing at a visually displayed circuit element and invoking a menu operation that copies the element is analogous to calling a copying function with an argument of a data structure representing the circuit element. Entire graphical interfaces can be easily specified by definitions of the relations of functions and their arguments, analogous to the way an interpreter for a language like Lisp is driven by such definitions.

We construct a graphical manipulation system in which each graphical object carries with it its own procedural semantics -- a history of the code that gave rise to the object. We provide ways of attaching procedural semantics to graphical objects, and arrange that if operations are performed on an object, the result carries the history of the operation and its arguments. Naming of graphical objects can be used to introduce variables into graphical procedures. The history of operations is recorded with the name of the object, and next time the procedure is run, a different object can be bound to the same name.

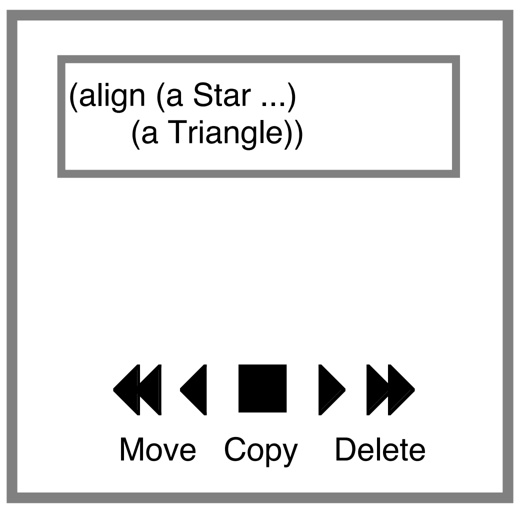



Correspondence between menu actions and function invocations

Code generated by actions recorded through the graphical interface can be edited with an *event editor*, which keeps a complete history of the computation. The history can be run forward or backward using tape recorder-like controls, and individual events or sequences can be edited with cut and paste operations.




Graphical interface




Reversible event editor

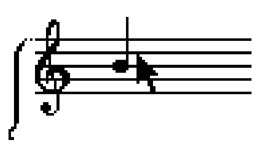
**A design procedure by example: Gourlay's note stem heuristic**

As an example of a design procedure, let's consider a problem in layout of music notation. My apologies are extended to those who are not familiar with Western music notation, but my example does not require actually interpreting the notation. It is sufficient to know that musical tones are represented by large dots, with lines extending from them that may point up or down. The choice of whether to place the line up or down does not affect the sound of the music, but musicians usually agree on the "right" way to do it for a given piece of music.

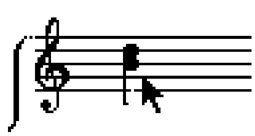
A musician places notes on a score by indicating their positions on the staff with a mouse. We would like the system to automatically make the decision about whether to point the stem of each chord up or down. Musicians typically make the decision reliably "by eye" -- few can articulate in words the precise criteria they use to discriminate these two cases.

This is an example, albeit a simple one, of an aesthetic procedure. The intent is to achieve a certain visual *balance* in the appearance of the music. Balance is itself an important concept that appears in many graphic design problems. The goal is for the designer to explain the procedure for achieving this balance to the machine in terms of the mundane pixel-manipulating operations of the graphical editor.

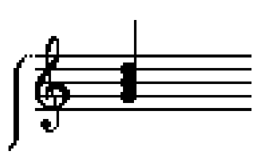
Gourlay [Gourlay 86] reports a heuristic for making the decision in an automatic music typesetting program. The following illustration shows the effect that a similar heuristic would have if implemented in an interactive music notation editor. As each new note is added, the system makes a decision as to which way the stem for the chord of which it is a part should point.



The user places a B on the staff.
The note stem points upward.
Then, a D is added to the chord.



The D forces the note stem of the chord to point downward.



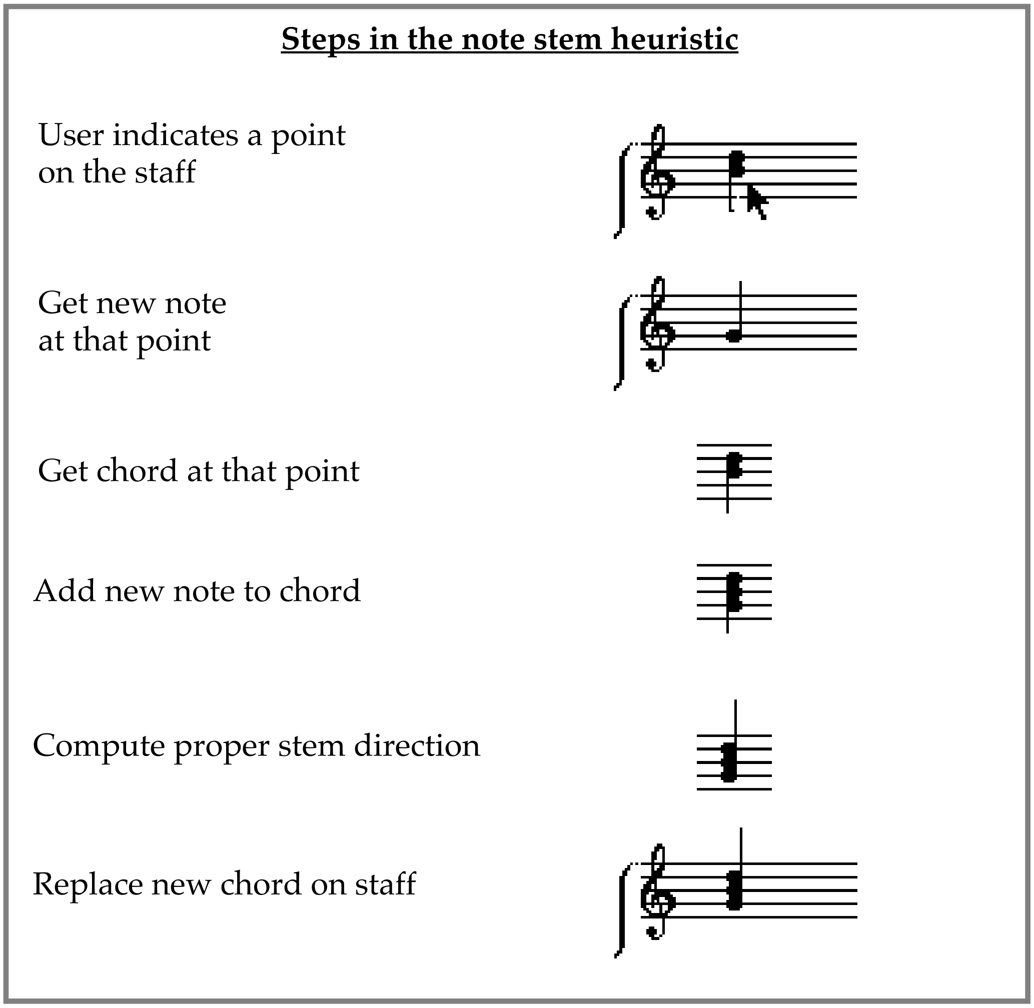
Adding a G to the chord changes the stem to point upward.

One way of explaining the trick is to consider the center line of the staff [the B pitch on the treble clef] as a "fulcrum" which "balances" the notes of the chord. If the "center of gravity" of the notes lines below the center line, the stem points upward to balance the chord. If the weight of the chord lies above, the stem points down.

[This analogy between musical notes and physical weight is itself interesting. It might be amenable to description in terms of structure-mapping [Gentner 1983], so that a goal for a future system might be to accept an explanation of the note-stem heuristic in terms of the physical analogy. This is beyond the scope of the present paper, however.]

Imagine you are a designer of an interactive music typesetting program, and wish to communicate this heuristic to the system. We will show how this procedure can be taught to the system graphically, through presenting examples and explaining how to make the decision for each case.

Implementing the note stem heuristic consists of the following steps. The heuristic is a procedure that accepts as input a point where the user clicked on the staff to insert a new note, and returns an object representing a chord with the stem pointing in the appropriate direction. Given the point, we must find which pitch on the staff the point is indicating, retrieve the chord which lies at that position on the staff, then add the new note to the chord. Then we perform the heuristic test of computing the center of gravity of the new chord and comparing its coordinates to the center line of the staff. The actual details of computing the coordinate arithmetic comparison are uninteresting from our present perspective. What we are really interested in is how the manipulations of the underlying musical representations are communicated by interaction with their graphical representations.




**Steps in the note stem heuristic**

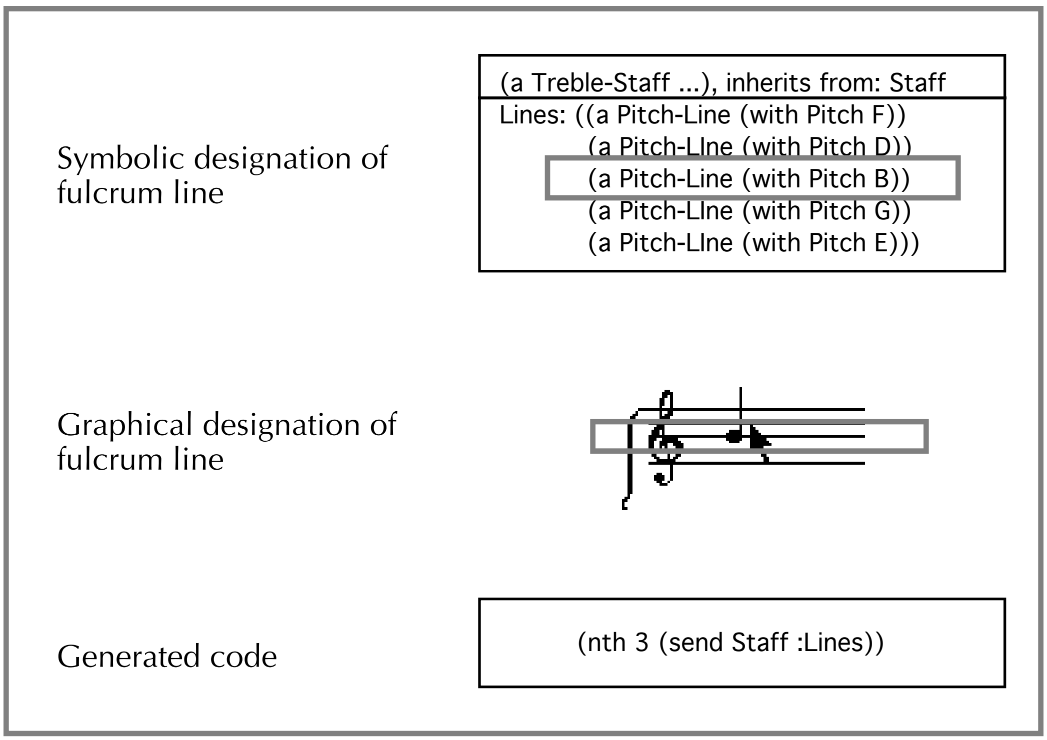
**Graphics code is generated both by pointing to graphical objects and textual representations**

The music editor uses an *object-oriented* representation which provides connections between data structures representing musical objects [staves, notes, chords, clefs] and graphical representations of these objects [lines representing pitches, ovals representing note heads]. Interactively, the user can only point to the graphical representations. We therefore need some way of making the musical objects accessible through their graphical representations.

We illustrate the technique for accessing objects by showing how we can pick out the line which is at the center of the staff, and the focal point for our balancing act. This illustrates that aspect of the programming process which involves *access paths* to objects in the system.

As mentioned earlier, each graphical object that appears on the screen carries with it associated procedural semantics. This consists of either a *name* given to it explicitly by the programmer to refer to it symbolically, or an *expression* in a language like Lisp which encodes the history of computation which gave rise to that object. Selecting the object by pointing with the mouse during the recording of a procedure has the same effect as typing the code would if programming in the conventional manner.

Not all objects of interest in the system are always directly accessible by pointing to them. Normally, the center line of the staff is not mouse-sensitive, since the musician will seldom have reason to indicate this line explicitly. However, the graphical object representing the line is accessible as a component of the graphical object representing the entire staff. This object is also accessible through expressions in the programming language.




We provide a variety of equivalent techniques, both graphical and textual, for indicating objects in the system. To access the component parts of a graphically displayed object, we provide a *browser* or *inspector* facility. Traditional inspectors deal only in textual representations of objects. A window is displayed listing the constituent parts of the object, and the user can indicate a sub-object by pointing at the displayed text corresponding to that part. The inspector can then be recursively called on one of the sub-objects.

We augment the traditional approach by linking the text oriented inspector to the graphical representation of objects. When the inspector is invoked on a graphical object, two things happen. The traditional textual inspector listing the parts of the object appears. In addition, all those parts of the object that themselves correspond to displayed graphical objects are *sensitized* -- that is, made available for selection by pointing with the mouse. Pointing at a sub-object selects that part both graphically, and simultaneously selects its textual representation in the text inspector. Pointing at the textual representation of a subpart also selects its graphical representaton.

In the example, inspecting the object representing the staff brings up a textual inspector detailing the components of that staff, which include the five lines representing pitches. The graphical representations of the pitch lines themselves become sensitized, so that they can be selected with the mouse. Pointing at the text corresponding to a pitch line in the inspector window highlights the corresponding line of the staff. In either case, the selection generates code which specifies the access path to that component. In the case of an object-oriented data structure, the code contains the name of the message used to access that component, here the `lines` instance variable of the staff. In the case of a list, the code extracts the appropriate element.

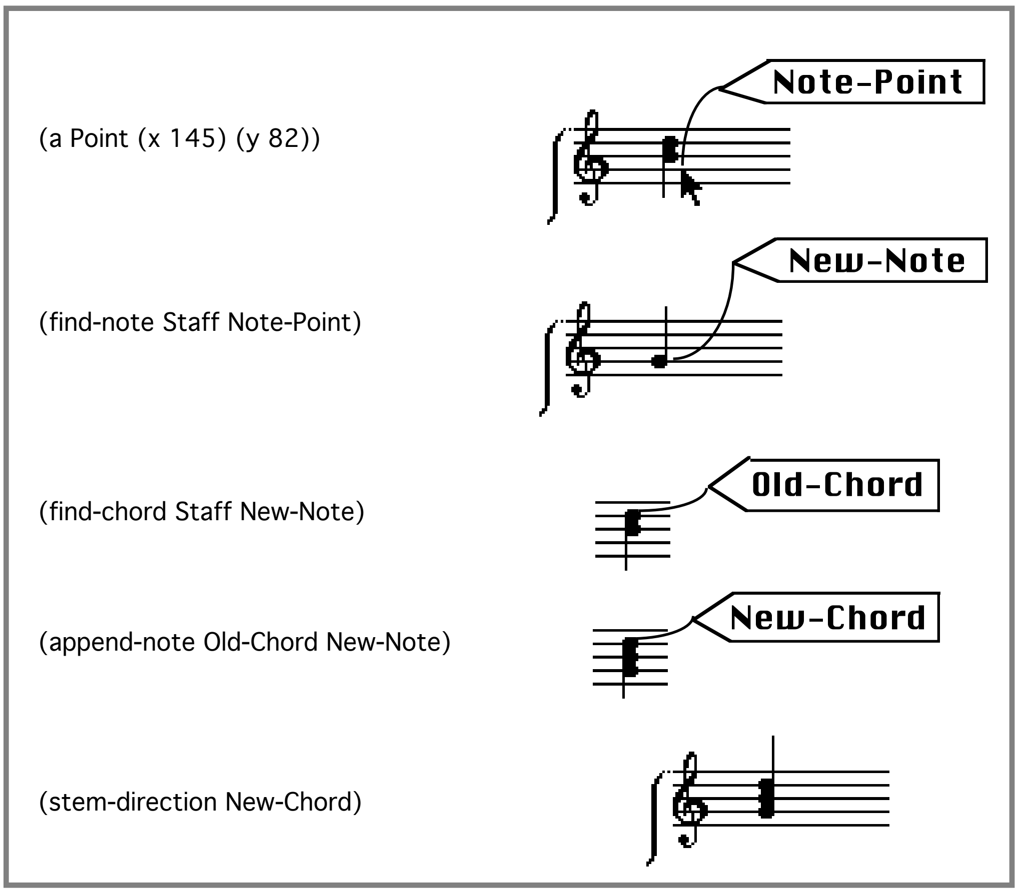
**Direct manipulation of graphical objects yields code for manipulating the underlying semantic objects**

In addition to merely accessing graphical objects and their components, we must be able to perform operations upon them. Just as we give both graphical and textual representations to the objects in our musical domain, we also give both graphical and textual semantics to operations as well.

Direct manipulation systems have always been treated as worlds in which every operation acts solely via *side effects*. Menu and iconic operations are considered to work by global changes to the state of the screen. Certainly, some direct manipulation operations are best modeled in this way, but this view loses the crucial ability to have the analogue of *functional programming* in the graphical world. The lack of a function oriented view is what limited previous programming-by-example systems such as [Smith 78] and [Halbert 84] to simply recording straight-line programs. We provide graphical functions which can accept arguments and return values. These values may be used in further computation, leading to the nesting of expressions.

Previously defined functions can be specified textually, by typing in the name of the function. Function *templates* display the argument lists to these functions, with the names of each argument mouse-sensitive. An argument can be filled in by pointing to its textual or graphic representation, or by typing. Some arguments are typed, so that only the objects which satisfy the type check are available for interactive selection to fill in that argument. Inspecting an object can offer a menu of appropriate functions on that object. Functions can also be referred to by icons as well.

The illustration below recaps the steps in the definition of the note-stem heuristic, showing the code generated at each step. The name attached to a graphical object at each step can be used in further expressions to refer to the result of that step.




**Reifying action: Connecting gesture with intent**

We can't manipulate objects we can't see; therefore an essential part of making a system capable of manipulating actions in a visual world is to have a visual representation for these actions. Since we may view actions in more than one way during the process of programming, we must have multiple representations for actions that can capture the intent of the procedures we wish to convey to the machine.

The first split in point in view is between that of the user and that of the system implementor. The user sees a graphical world in which gestures invoke commands and manipulate graphical objects. The implementor must also see, behind the scenes, a more complex world in which the gestural actions of the user can be manipulated.

The learning engine attached to the graphical interface records these steps, generating the following Lisp procedure. It is parameterized by the point where the user clicked the mouse for the new note, so that the next time it is used, a new note can be given.

---

```
(defun Add-Note (Note-Point)
 (let* ((Staff (staff Note-Point))
        (New-Note (find-note Staff Note-Point))
        (Old-Chord (find-chord Staff New-Note))
        (New-Chord (append-note Old-Chord New-Note)))
       (setf (stem New-Chord) (stem-direction New-Chord)))

(defun Stem-Direction (Chord)
 (cond ((< (center-of-gravity Chord)
           (note-y (fulcrum-line Staff)))
        (stem-down Chord))
       ((stem-up Chord))))

(defun Center-of-Gravity (Chord)
 (/ (sum (mapcar #'(lambda (Note) (note-y Note)))
                 (notes Chord))
    (length (notes Chord))))
```

---

The determination of the direction of the stem requires, unavoidably, some numerical computation. The stem direction procedure is computed by graphically inspecting the objects representing the staff and chord, retrieving the numerical values of the coordinates. The system then operates in a "calculator" mode, and the numerical average of the chord's notes Y coordinates are compared against the Y coordinate of the fulcrum line.

As described in [Lieberman 84], the learning engine is capable of integrating multiple examples. In this case, the demonstration of the stem direction procedure requires two examples: one indicating the case where adding a note results in flipping the direction of the stem, one where the stem direction remains unchanged. This is the mechanism for generation of conditionals. If the target language were a rule-based formalism instead of Lisp, multiple examples would correspond to multiple rules [though not necessarily in a one-to-one mapping].

**Summary: Programming by example integrates machine learning with interactive graphical interfaces**

Graphic designers are visual thinkers. They can neither be expected to write programs directly in rule- or frame- based languages, nor express their expertise verbally to knowledge engineers. However, their intelligent problem solving behavior can be captured by a system which records their actions in using an interactive graphical editor to solve particular concrete design problems. If the actions are recorded in a sufficiently high-level symbolic form, they can form the basis for a machine learning engine to synthesize intelligent procedures. Through the use of multiple examples and interactive graphical feedback, the designer can have a conversation with the machine about how such examples are to be generalized to work on examples other than those that were originally shown. Furthermore, the resulting procedure is then smoothly integrated back into the original graphical interface. Programming by example is the key technique for accomplishing this synthesis.

**Status**

Implementation of the music editing scenario is not yet complete as of this writing. A programming-by-example learning engine, a graphical interface interpreter, and an editor for a reversible procedure recording system, all necessary components, have been implemented. Suguru Ishizaki has implemented a preliminary version of a graphical object inspector/editor that allows editing in the graphical or program domains as described.

**Acknolwedgments**

Muriel Cooper, Suguru Ishizaki, Fred Lakin, Suzanne West, Laurie Vertelney, Michael Arent, Scott Kim and Walter Lieberman gave me valuable insights into graphic design. I would like to thank Muriel Cooper and Nicholas Negroponte for their support of my efforts. The Visible Language Workshop is supported in part by grants and equipment from HP, IBM, DARPA, Apple, NYNEX and Xerox.

**Bibliography**

[Buchanan and Shortliffe] Bruce Buchanan and Edward Shortliffe, Computer-based Medical Consultation: The MYCIN Experiments at Stanford. American Elsevier, New York 1976.

[Gentner 83] Structure-Mapping: A Theoretical Framework for Analogy. Cognitive Science, Volume 7, No. 2, 1983.

[Gourlay 86] John Gourlay, Interactive Typesetting of Music. Ohio State University, 1986.

[Halbert 84] Daniel Halbert, Programming by Example. PhD Thesis, U. of California, Berkeley, 1984.

[Lieberman 82] Henry Lieberman, Designing Interactive Systems From the User's Viewpoint, in Integrated Interactive Computer Systems, P. Degano and E. Sandewall, eds. North-Holland, 1982.

[Lieberman 84] Henry Lieberman, Seeing What Your Programs Are Doing, International Journal of Man-Machine Studies, July 1984.

[Lieberman 85] Henry Lieberman, There's More to Menu Systems Than Meets the Screen, SigGraph '85, San Francisco.

[Lieberman 88] Henry Lieberman, Communication of Expert Knowledge in Graphic Design, MIT Media Lab, forthcoming.

[Mitchell 83] Tom Mitchell, P. Utgoff, R. Bannerji, Learning by Experimentation: Acquiring and Refuting Problem Solving Heuristics. In Machine Learning: An AI Approach, Tioga, 1983.

[Smith 78] David C. Smith, Pygmalion, a Creative Programming Environment. Birkhauser-Verlag, 1978.

[White 88] Jan V. White, Graphic Design for the Electronic Age, Xerox Press, 1988.