Art Imitates Life: Programming by Example as an Imitation Game

HENRY LIEBERMAN lieber@media.mit.edu
Media Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139 USA

Introduction

Having the computer imitate recorded human actions is the basis for an experimental technology for programming, variously called "Programming by Example" or "Programming by Demonstration". This is an under-appreciated technology that holds the promise of revolutionizing programming and making it more accessible, especially to non-expert programmers. Because imitation is a natural learning strategy for people, it can help alleviate the barriers of abstraction and lack of short-term memory that makes programming difficult for people. Much past work has focused on how to represent the recorded actions, and how to generalize the resulting procedures so that they can be applicable to examples analogous to those on which the system is taught. This chapter will survey past work in the field and speculate on how the mechanics of imitative behavior might inform future developments, especially the feedback loop of verifying that imitative behavior has the desired result in new situations.

Monkey see, monkey do

Imitation can be a fundamental basis of learning, in both animals and human beings. Early on, children learn by repeating what adults do, and gradually learn more and more sophisticated ways of incorporating aspects of the observed behavior of others, into their own behavior. We do not yet know to what extent we can ascribe imitative behavior in animals to an instinct to copy others or to some form of learning. It is clear, however, that observation and imitation are central capabilities for many forms of learning.

Computers are funny things. They are not people, but since they are designed and used by people, we can't help but incorporate some aspects of our cognitive processing into their operation. We can certainly make them imitate certain aspects of our own behavior, more or less closely. And just as we have trouble drawing the line between simple instinctive copying and true learning in humans and animals, we also have trouble drawing the line between simple repetition and true learning in computers. Clearly, as with animals, our interaction with computers can affect their behavior. Those changes in behavior can subsequently be observed in new situations that resemble the previous ones in which the imitation first occurred. Whether or not we call it learning.

Programs are the instincts of computers. When we program computers, we are giving them behavior that then becomes automatic for the computer in the same way that instincts are automatic in animals. Certain animals, such as ourselves, may be endowed with an instinct to imitate observed behaviors, that forms the basis for learning. If we want machines to learn, can we endow them also with an instinct to imitate? Can a program that observes and imitates behavior of a human teacher enable a computer to learn? This is the question we will explore in this chapter.

Computer see, computer do

Programming by Example, also often referred to as Programming by Demonstration, is a technique that enables a user to teach new procedures to a computer. It consists of a learning program, or *agent*, attached to a conventional interactive graphical interface, such as a text editor, graphical editor, spreadsheet, or Web browser. The learning agent records the actions performed the user in the user interface. The user demonstrates a concrete example of how to perform a procedure, using sample data objects that are meaningful to the user. The system then records the steps to form a program that can be used in situations that are analogous to, but not necessarily exactly the same as, those on which it was taught.

Programming by Example (PBE) can be seen as a way of trying to get the computer to imitate the user. By demonstrating a sequence of steps, the user is trying to get the computer to learn those steps so that the computer can in the future imitate what the user would have done, thereby saving

the user time. Programming by Example also establishes a "show and tell" kind of interaction that is similar to the way people teach each other tasks, leading to a natural style of interaction. Because we all experience imitation as young children, computer novices can easily understand the idea of trying to get the computer to imitate them as a metaphor for the programming task.

Programming by Example was developed in response to a perceived difficulty in the skill of computer programming. Programming in a conventional programming language requires giving the computer an abstract description of steps it is to follow. This abstraction is difficult for people in all but the simplest of situations. It places high demands on the user's short-term memory. For most users who are not expert programmers, it is difficult to foresee the effect of a given programming language statement or expression, and to understand the consequences of a given program for the concrete cases of interest. With Programming by Example, the user can always see the direct consequences of an action, at least for the demonstrated examples, which makes it easier to decide what the next operation should be.

Many conventional applications already provide the ability to simply play back exactly a sequence of previously recorded user actions, also called a *macro*. Macros can be also be seen as the simplest form of imitation. But macros tend to be brittle. If any minor aspect of the interface changes: positions of icons, slightly different data, different file names, etc. the procedure may fail. In Programming by Example, the sequence of steps can be *generalized* by the agent. Some of the specific details of the sample data objects can be removed, or concrete actions generalized to a broader class of actions, so that the procedure then becomes useful in a wider variety of situations.

Generalization can take any of several forms -- the agent can ask the user how to generalize the program and present lists of options; or the agent can use heuristics, perhaps dependent on the specific domain, to guess what the best generalizations are for each example and action.

The first true Programming by Example system was David Smith's Pygmalion (Smith 1993), in 1978. Pygmalion also introduced graphical programming, using icons to represent program elements and values, and using a graphic editor to manipulate the program. This was followed by a

wide variety of systems in several domains, including desktop file systems, general Lisp programming, text editing, and several kinds of graphic editing. The state of the art in the early 90's, including a general introduction to the topic, glossary, history, and other tutorial material is admirably covered in (Cypher, ed. 1993). The current state of the art as of this writing is presented in (Lieberman, ed. 2000).

As an example, Mondrian (Lieberman 1993) is typical of many Programming by Example systems. Mondrian is a graphical editor with an agent that records graphical procedures using programming by example. The user demonstrates a procedure using the graphical editing operations on concrete graphical objects, indicating which ones are to be taken as examples.

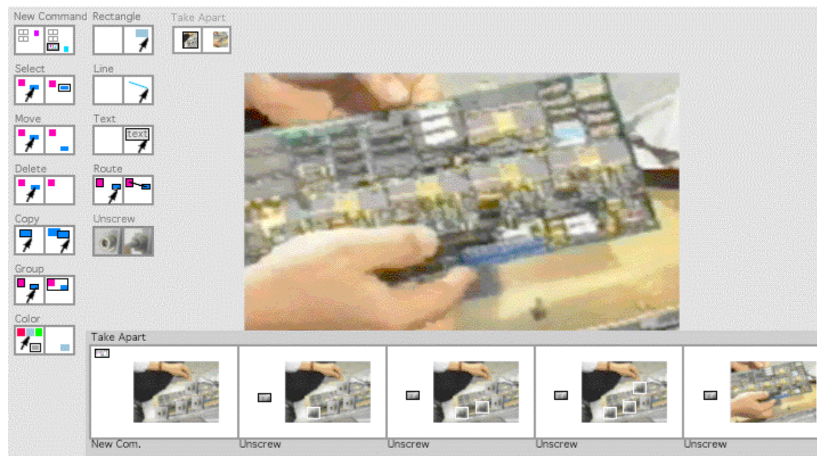

Figure 1. Mondrian is a graphical editor that learns new procedures through Programming by Example. Here, we teach the system how to take apart a circuit board

:.

The system then records a procedure that can be used later with new objects in place of the examples. Mondrian's learning procedure is similar to what is called in the literature *explanation-based generalization*, where a tree of dependencies among operations is constructed and the generalizations are propagated through this dependency tree (Lieberman, 1993).

Mondrian also possesses the ability to learn declaratively as well as procedurally. The user could put graphical annotations on images or video frames to create a visual representation of objects that can be manipulated by the graphical editor. These graphical annotations name objects and establish a graphical part-whole hierarchy. The user could then operate on the graphical objects and the procedures would be recorded in terms of the relations described by the graphical annotations. In (Lieberman and Maulsby, 1996), the user can teach the agent how to take apart a motor by annotating a video of a human performing the same procedure in the real world. By asking the user to graphically annotate objects, there was no need to have computer vision procedures for recognizing objects in the scene. Graphical annotation represents a method for the user to communicate the meaning of demonstrated actions for the computer to imitate.

Imitation in interface agents

Imitation also plays a role as a metaphor in the movement toward *interface agents* in computer-human interaction (Bradshaw, 1997). In contrast to traditional direct-manipulation icon-and-menu based software, the idea of an interface agent is to cast the computer in the role of a human assistant, such as a secretary or travel agent. This vision is related to that of PBE, but unlike PBE, where the user is explicitly trying to teach the machine a procedure, in many interface agent projects the user may not be explicitly trying to teach the machine, but simply performing tasks for other reasons. The agent then learns from more passive observation. The role of the user is then to provide feedback on the agent's actions, and perhaps also to provide advice that affects the agent's actions.

In some ways, this might be closer to the kind of imitation-based learning that may be occurring in some animals, in the sense that it usually occurs without explicit instruction or even feedback, from the teacher, as we did with Programming by Example. Communication in the other direction, from the agent to the user, is usually also limited, to avoid bothering the user with too many explicit requests for information.

Letizia (Lieberman, 1997) is an example of this kind of agent. It is an agent that records selections of Web pages in a browser, and compiles a profile of the user's interests, without any explicit declaration of interest on the user's part. The agent then independently performs a breadth-first

search surrounding the user's page, and filters the candidate pages through the user's profile. It produces a continuous display of recommendations of pages that the user may be interested in, representing the progress of the search in real time. Thus Letizia's search seeks to imitate the user's browsing, producing a prospective evaluation of pages that saves the user time which would be otherwise be wasted in looking at irrelevant pages.

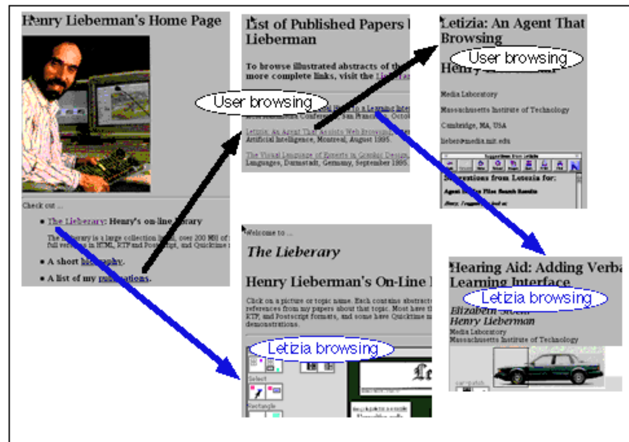

Figure 2. Letizia's breadth-first search doesn't precisely imitate, but rather *complements* the user's depth-first search

However, Letizia specifically chooses *not* to imitate exactly the control structure of user browsing. Most Web browsers encourage a depth-first search of the Web space, since it's always easier to go "down" in a browser by following a link, than to go "sideways" and visit sibling links to the page. The problem, which we observed in practice, is that users are tempted to go too deeply, following ultimately unfruitful "garden paths". So Letizia tries to adopt a complementary control structure, pursuing a breadth-first strategy to complement the user's presumed depth-first search, finding what would otherwise be ignored. This is an important lesson for learning – sometimes the goal is not to imitate the teacher exactly, but to understand what the teacher is doing in order to do something else complementary to it. This is clearly related to imitation, but there's no single word for it. I do not know whether or not this kind of imitative behavior is exhibited by animals or has been studied.

Can Programming by Example imitate imitation?

We can analyze the structure of many kinds of imitative behavior in the following steps:

- The learner *observes* behavior of the teacher.
- The learner *interprets* the teacher's behavior
- The learner tries to *execute* their interpretation of the behavior
- The learner receives *feedback* from the teacher or from the environment

Analysis of these stages -- observation, interpretation, execution and feedback -- can also be applied to Programming by Example. Observation amounts to the recording of the user actions in the interface. There is always the issue of at what granularity the actions are recorded. At the lowest level in an interactive graphical interface, the actions can be recorded as mouse movements, mouse clicks and typed characters. At a higher level, the actions can be recorded as selection of menu operations, icons, ranges of text, spreadsheet cells, graphical objects, etc. If the granularity is too small, recorded histories will get overwhelmed with data and place too much burden on the interpretive processes to follow. For example, we might record every mouse movement in an interface, but in most interfaces, not every mouse movement is significant, and recording all such movements produces prodigious amounts of data. If the granularity is too large, we might miss some significant events. For example, in the Applescript language on the Mac, each application must declare what operations are recorded by the user interface recording facility, and many fail to declare enough operations to enable the computer to fully reproduce the user's actions. The granularity issue is discussed further in (Lieberman, 1998).

The issue of how to *generalize* a recorded program in Programming by Example is the step of interpreting the user's behavior. Generalization in Programming by Example typically means replacing constants in the program by variables to be filled in each time the program is subsequently executed, and also replacing descriptions of data and actions by more general descriptions that will have wider application in subsequent executions. When talking about data selected by the user, this is also sometimes referred to in (Cypher, 1993) as the *data description problem*.

Among the systems described in (Cypher, 1993), we can see a variety of approaches to the generalization problem. Some systems take the approach of asking the user to choose generalizations explicitly. Some ask the user to choose from a list of plausible generalizations computed heuristically in order to cut down the space of possible generalizations, which can grow very large when combinations of generalization characteristics are possible. Some choose generalizations automatically based on some domain knowledge, without user intervention. Some will accept user advice that alters how the generalization is chosen. None of these approaches is universally better, and Lieberman and Maulsby (1996) use a "spectrum of instructibility" to classify the dimensions of the unavoidable tradeoff between convenience and control.

Key to the generalization problem is the related problem of *context*. Understanding which actions and objects are dependent upon the details of their context says whether or not they can be carried over to other contexts, and how. Traditional hardware and software design overlooks context because it conceptualizes systems as input-output functions. Systems take input explicitly given to them by a human, act upon that input alone and produce explicit output. But this view is too restrictive. Smart computers, intelligent agent software, and digital devices of the future will also have to operate on data that they observe or gather for themselves. They may have to sense their environment, decide which aspects of a situation are really important, and infer the user's intention from concrete actions. The system's actions may be dependent on time, place, or the history of interaction. In other words, dependent upon context. The impact and implications of the context problem are discussed further in Lieberman and Selker (2000).

The step of execution of the imitative behavior consists of the learner repeating the behavior of the teacher. Of course, there is hardly ever any such thing as exact repetition. What the learner does in imitation of the teacher will of course be dependent on the previous steps of what the learner has observed and interpreted in the teacher's behavior.

And if the teacher's behavior in question consists of multiple steps, then there may be interaction between the teacher and the learner at each step. The teacher may correct the student or give feedback, verbally or behaviorally, indicating approval or disapproval. The learner may observe directly his or her own success or failure by effects of the behavior on the

environment. Thus the phases of execution and obtaining feedback may be interleaved rather than occur one after the other.

In Programming by Example, there might be interaction between the user and the system during the execution and feedback phases. The user may be able to "step through" the recorded and generalized program to monitor execution and verify that it is having the intended effect. The user may be able to accept or reject various moves made by the system, or perhaps give it advice or critique that change its behavior, either during the demonstration, or subsequently.

One system that perhaps went the farthest in providing feedback and monitoring of execution was Cypher's Eager (Cypher, 1993). Eager was a Programming by Example system for Hypercard that recorded actions, and a prominent feature was its recognition of looping behavior. When it noticed actions that tended to repeat a sequence of earlier actions, it would provide *anticipatory feedback*. Every time the user took a step that led on a path that the agent considered to be repetitive of past actions, the agent [represented by a green cat] would predict the next action, and unobtrusively highlight it on the screen in green. The user was free to confirm the prediction by choosing the action predicted, or to perform some other action, in which case the agent would abandon its prediction. If the prediction was confirmed, the next action would be predicted, until the entire loop was performed a second time. At that time, the user would be given the option of repeating the whole procedure, either a single time or until the data was exhausted. Here the cat was explicitly imitating the user's past actions, and the user was cast in the role of "animal trainer" to provide positive or negative feedback.

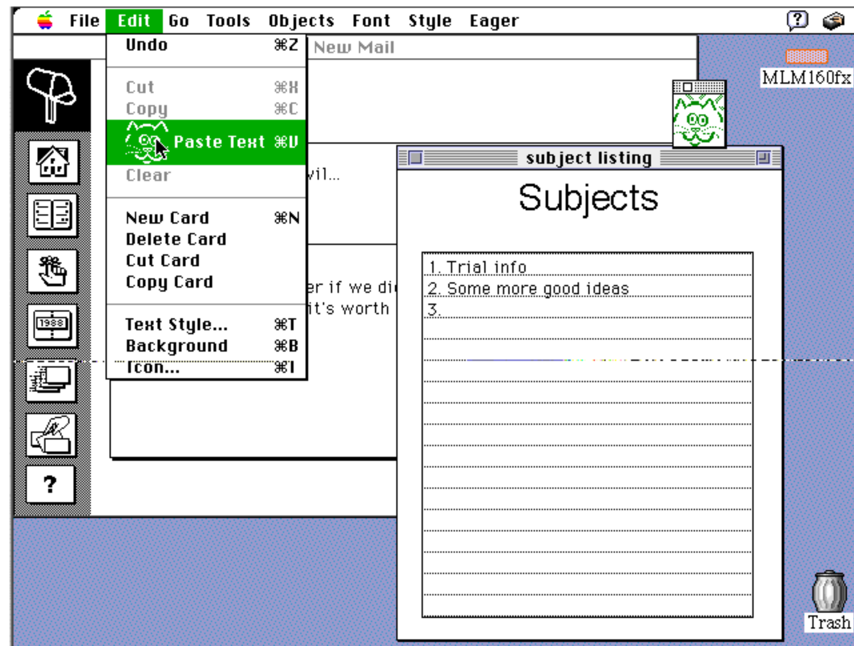

Figure 3. Eager's cat allows step-by-step confirmation by the user of the pattern of actions recorded and generalized by the system.

It is in these last phases, execution monitoring and feedback, that the connection between Programming by Example and imitative behavior in humans and animals needs to be developed in greater depth. Programming by Example often has the problem of debugging and editing programs developed in this style, and enabling the user to visualize the process of recording, generalization and execution. Imitation in humans and animals has produced highly developed strategies for dealing with these issues. Deeper understanding of these strategies might produce inspiration for better providing monitoring, editing and debugging capabilities in our Programming by Example systems.

Can a parrot learn how to program?

We are about to embark on an unusual test of the relationship between imitative behavior and programming. Cognitive studies on the higher primates have shown that they can be made to show simple

communication and learning capabilities, although there still exists considerable controversy about the significance of these results.

Recently, however, surprising findings have surfaced about the communicative and cognitive abilities of African gray parrots. The imitative vocalizations of parrots are, of course, well known, but Irene Pepperberg (1999) has shown that parrots are capable, at least under some circumstances and with careful long-term training, of using a language-like code based on human speech for communication and problem solving. This despite their relatively small brain size! See also the Pepperberg chapter in this volume.

In particular, such parrots are able to demonstrate some specific cognitive capabilities. They can remember and invoke sequences of events [up to a length limit imposed by short-term memory constraints, as is the case with humans]. They can also understand, to some extent, categories of objects. They can, for example, be shown a yellow toy key, a yellow ball, and other yellow objects, and learn the category yellow such that they can respond “yellow” to a question “What color?” when shown a previously unseen yellow object. They can be taught other properties of an object, such as shape or material, so if you show them a blue key, they can say “key”, knowing that the shape property does not refer to a specific object, and that shape, color and material are independent properties of objects. This shows that they are somewhat capable of generalizing objects to categories.

It is our intuition that these two fundamental cognitive capabilities -- sequencing and generalization -- are at the root of the cognitive task of programming. Sequencing and generalization also happen to be the essential ingredients of a programming by example system. So we propose to try to teach a parrot how to program, using a programming by example system!

Media Lab student Ben Resner is building an experimental apparatus with which we can test these hypotheses. The apparatus consists of a set of stages of boxes containing a food reward, and a series of valves that allow the passage of food from one box to another, and eventually, accessible to the parrot. The box at each stage is controlled by a series of buttons, so that if the parrot presses a correct sequence of buttons, it can eventually obtain the food. The idea is that that the parrot could be taught to write a

program to obtain the food, even if the sequence of buttons changes. The parrot will be taught to generalize over properties of the sequence.

Figure 4. Parrot researcher Irene Pepperberg and parrot Wart play with an apparatus designed by Ben Resner for the parrot programming experiment.

An early experiment by Radia Perlman (1976) at the Logo group of the MIT Artificial Intelligence Laboratory gives us some hope that this could be accomplished. The aim of the group was to teach programming to young children, and Perlman set out to discover the minimum age at which children could be taught programming skills. She built a Button Box that consisted of a series of buttons that each launched a command to a small robot "turtle".

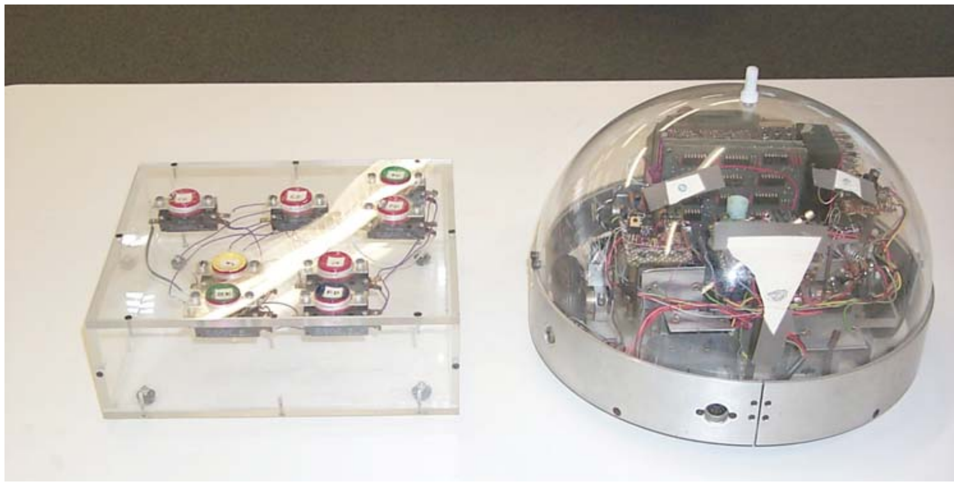

Figure 5. Radia Perlman's 1974 Button Box (left) and Logo Turtle (right).

The programming operation consisted of a “Start Remembering” button that memorized a sequence of events that could subsequently be launched by a button that executed the stored procedure. Similar ideas were subsequently realized in the Slot Machine, where computer-readable cards inserted into slots represented the programming operations, and Instant Logo, a screen interface for programming macros consisting of Logo commands. The generalization operations of these systems were absent or quite limited, but the principle of programming was demonstrated. Children as young as four years old could learn to program the Button Box, well before they could learn to read. Thus literacy skills were shown not to be essential to the fundamental task of creating procedures. They could well understand the idea of getting the computer to imitate the series of operations it was taught. Parrots can be shown to have performance on certain problem solving tasks equivalent to 3-or-4 year old children.

Teaching a parrot to program might sound a bit crazy, but we feel that the attempt should at least lead to some interesting results. To our knowledge, no one has ever tried to teach a non-human animal to exhibit any kind of programming activity. So, if it succeeds, we feel this will be a major result in the study of the cognitive capabilities of animals. And, in keeping with the theme of this book, it will provide evidence of the centrality of imitation in purposeful behavior by both animals and humans.

References

Bradshaw, Jeffrey, ed. *Software Agents*. MIT Press, 1997.

Cypher, A., ed., *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA, USA, 1993.

Cypher, A., Eager: Learning Repetitive Tasks by Demonstration, in (Cypher, ed. 1993).

Lieberman, H., Tinker: A Programming by Demonstration System for Beginning Programmers, in (Cypher, 1993).

Lieberman, H. and David Maulsby, Instructible Agents: Software That Just Keeps Getting Better, *IBM Systems Journal*, Volume 35, Nos. 3 & 4, 1996.

Perlman, Radia, Using Computer Technology to Provide a Creative Learning Environment for Preschool Children, AI Memo #360, Logo Memo 24, May 1976.

Lieberman, H., Integrating User Interface Agents with Conventional Applications*, ACM Conference on Intelligent User Interfaces*, San Francisco, January 1998. Revised version to appear in *Knowledge Systems Journal*.

Lieberman, H., ed. *Your Wish is My Command*, Morgan Kauffman, San Francisco, 2000.

Lieberman, H. and T. Selker, Out of Context: Computer Systems that Learn From, and Adapt To, Context. IBM Systems Jounral, to appear, 2000.

Pepperberg, Irene. *The Alex Studies: Cognitive and Communicative Abilities of African Grey Parrots*. Harvard University Press, 1999.