SPECIFYING PROCEDURES TO OFFICE SYSTEMS

Ian H. Witten, Bruce A. MacDonald
Saul Greenberg

Research Report No. 87/257/05
January 1987

Cite as:
Witten, I. H., MacDonald, B. A., and Greenberg, S. (1987). Specifying procedures to office systems. In Proceedings of the Automating Systems Development Conference, Leicester. April 14-16.
Earlier version as Report 87-257-05.

© Ian H. Witten, Bruce A. MacDonald and Saul Greenberg
Knowledge Sciences Laboratory
Department of Computer Science
The University of Calgary
2500 University Drive N. W.
Calgary, Alberta T2N 1N4

Introduction

How can end users — especially casual users without knowledge of or aptitude for conventional programming — be given the ability to specify and communicate procedures to a computer system? To the extent that they can, opportunities for system design and implementation are placed where they belong, directly in users' hands. A good domain for examining this question is office computing, where the notion of "casual user" is more palpable and there is evident need to automate procedures that would otherwise have to be executed manually [Raed85]. This paper surveys current practice, research, and future prospects for communicating procedures to office computer systems, placing special emphasis on robustness and suitability for the casual user.

In practice, users of existing systems who have to specify procedures are commonly forced to work through an intermediary who is an experienced programmer. Failing this, they generally resort to learning some kind of command language. Explicit forms programming languages, perhaps based on ideas of logic programming which suppress control structure, offer better prospects, while knowledge-based techniques which utilize a model of office semantics may provide a solution in the more distant future. Programming by example is a promising method for specifying procedures but presents difficulties with editing, conditionals, iteration, nesting, data structures and variables. These can be alleviated by using several example sequences or, possibly, specifying goals, perhaps with the help of an automated assistant. Alternatively, control information may be provided explicitly by the user through a well-engineered interactive interface.

A neglected aspect of the problem of specifying procedures is the need for *robustness*. This concerns the ability of the system to deal with the vagaries of the real world by incorporating methods for handling errors, inconsistencies, environmental changes, and extraneous information; and by accommodating changes in user requirements and expectations. In examining casual-user programming systems, it is expedient to draw a distinction between

- the inherent ability of the *conceptual* system, and
- the ability of an *implemented* system,

to deal with variability. The first concerns the idea and the second the way it is implemented. The difference is important because much of the work we discuss is experimental, and implementations often lack robustness while the ideas underlying them may not.

The next section first introduces office procedures, showing how they pervade routine office work. It then describes two different user interface paradigms, direct manipulation and command languages, highlighting the shortcomings of each for casual-user procedure specification. The second section considers the possibility of grafting procedure specification ability on to other interfaces. It describes four possibilities: the use of a command language in conjunction with a direct-manipulation interface, forms-based programming, specifying procedures using logic, and knowledge-based systems. Thirdly, we introduce the idea of programming by example, and explain four different methods for accomplishing it. In each section we discuss the power of the procedure specification paradigm, as well as its robustness and suitability for the casual user.

Office systems and the procedure problem

Office procedures

Routine tasks such as clearing one's desk, opening and filing mail, and sorting lists of names constitute simple office procedures. More complex jobs such as processing a purchase order are standard institutional procedures often controlled by a specially-designed form. The fields on the form show the information needed to perform the procedure, and the principal actions required. For example, a purchase order may involve financial authorization, tendering, ordering, shipping, customs and duty, consideration of import regulations, and so on. Each participating department will have its own detailed procedure for dealing with the form. For example, expected delivery date may be recorded so that appropriate action can be initiated in the event of non-arrival. Training new employees in the *modus operandi* is a recognized cost of staff turnover. Much of a newly-hired clerk's time and effort is expended in learning the vital office procedures. Specifying *new* tasks will always be important, even to fully trained office workers or sophisticated office computer systems. Whenever regulations, policies, or superiors change, new or modified procedures will arise.

If office procedures could be automated, much of the cost for teaching and executing them could be avoided. Of course, some will be just too complex. There is even controversy about the extent to which office work really does involve executing procedures. In a study of two office tasks, dealing with a missing invoice and using an unfamiliar copying machine, Suchman [Such85, Such83] found the subjects to be *problem-solving*. Once they discovered the solution, subjects rationalized their activity as procedures that "went wrong" in some respects. Much office work is concerned with handling exceptions to the usual routine; so much so that Suchman considers the exceptions normal and the procedures mythical! This paper is not about automating problem-solving in the office, but rather about automating the simple and standard procedures that do occur. Despite the predominance of problem-solving activity, establishing and performing routine tasks is important in offices, as the above examples show.

The simplest office procedures are fixed sequences of actions; the task is performed exactly the same on every occasion. For example, an office user might want to specify a complex sequence for logging on to a large computer from a personal computer [Poun87]. However, many office tasks will be more complex than this. For example, the procedure for admitting students to a University may revolve around several main events, such as receiving preliminary and final applications, reviewing stages, and admittance or rejection. Each individual event can trigger a complex procedure. Late applications must be dealt with; foreign students may be treated differently; notification of missing documents must be mailed, and so on [Kuni82]. Although specification of such an elaborate system is beyond the ken of the average office worker, adding or modifying smaller sub-procedures is not.

Table 1 shows the major constructs required in a system for specifying tasks. Conditionals allow different actions for different situations. Iteration permits part of a task to be repeated until a specified condition is met. Nesting embeds one control structure inside another, to many levels. For example, the Table shows one loop which is inside another. Data structures are needed for conveniently storing and accessing structured information, and might also be nested. Variables allow different items to be processed by the same procedure. Table 1 also shows a simple example procedure which employs all of these constructs. The procedure opens the electronic mail of the user and sorts items into folders by sender name. A new folder is created for each new sender. The variables used are *New-Mail-Item* and *Sender*. An *if ... then* construct is nested inside a *for* loop.

Direct manipulation interfaces

The most effective user interfaces to office systems allow pictorial representations of real office objects (files, folders, documents, text, diagrams, and the like) to be manipulated directly on the screen [Shne83]. This leads naturally to learnability, robustness, and general ease of use [Lee83, Witt85]. The computer system provides a *metaphor* of the office environment. This makes it easy for users to execute procedures manually, step by step [Shne83, Lee83]. Briefly, direct manipulation provides a robust interface suited to the casual user, with these advantages:

- users are enthusiastic
- learning effort is reduced
- the system is predictable
- the interface is concrete, not abstract

and [Witt85]

- you can always see what you are working on
- you see the final form of your work, for example text to be printed out
- you see all options available to you.

However, current systems do not help when it comes to communicating sequences of actions [Myer86]; indeed, their very nature makes it difficult for users to specify procedures [Zloo81]. Certainly a fixed string of operations could be recorded for playback later. However, for any procedure which involves generalizations, conditionals, controlled iteration, complex data structures, editing and debugging, direct manipulation falls apart. These are not simple operations on concrete objects. They are abstractions, which are specified to real office workers using complex natural language constructions. The manipulation is no longer *direct*.

Command language interfaces

The traditional way of specifying procedures to computer systems is by using an imperative language. Standard interactive computers implement an operating system command language in which procedures can be specified, saved in files and executed on demand. These are often clumsily defined and implemented; for example IBM JCL is renowned for its opacity. The UNIX *sh* command interface contains better facilities for procedure definition, but is still far from suitable for a casual office user. In some ways such command languages are *more* difficult for a casual user than modern programming languages, such as PASCAL, C and ADA, because the details of syntax and semantics are tricky and opaque. For example, this statement in UNIX *sh*

```
cat filename | expand | awk '{ printf "%4d %s\n", NR, $0 }'
```

puts line numbers on the file *filename*.

Monolingual programming environments, which give a user the facility of specifying command procedures in a programming language such as LISP (or even BASIC) are more suitable, but presuppose the ability to write programs in the conventional way. There is some debate on whether this will ever become acceptable to the casual users (see [Cuff80] and [Ande80] for opposing views). To program in a conventional language one needs to be able to translate one's own knowledge of the task into a procedural specification suitable for expression in the language. One also needs to be able to construct, test, and debug the implementation.

Command languages provide a powerful ability to specify procedures, since they are general purpose programming languages. However, they are unsuitable for the casual user because of the need to learn tricky details of computer syntax and semantics, and the difficulty of articulating procedures explicitly. Moreover, they lack robustness in that programs depend on technical and system characteristics.

Adding procedure specification

Adding a command language to direct manipulation

Some office systems have added the ability to create procedures by grafting linguistic commands on to a direct-manipulation interface. However, there are clear contradictions between the two paradigms involved. Not only do these systems fail to escape the above-noted drawbacks to command languages, but they conflict with the basic metaphor of direct manipulation too.

These problems will be briefly illustrated using the Xerox Star's CUSP user programming language. This is an excellent example of a command language designed to be used for office work in conjunction with a direct-manipulation interface. Programs can operate on text, icons, table and form fields, record files, and other objects available in the interface [Halb84]. For example, CUSP enables users to express programs such as: [Halb84]

> If *CreditBalance* < 0 Then
> Move The *Document* Whose Name Is "PleasePay" To The *Printer* Whose Name Is "Gutenburg";

Unfortunately, it departs sharply from the desktop metaphor. The user has to learn a *different* method of operating the Star in order to specify procedures. This is amply illustrated by the program (which, its appearance notwithstanding, does *not* contain typographical errors)

> Store Mean[*Families* [*Row* Call It *Parent*
> With *Parent.LastName* = "Smith"].*Children.Age* Into *AverageSmithChild*;

for computing and storing the mean age of the Smith children listed in the *Families* table [Halb84]. The direct-manipulation user, on the other hand, would accomplish the same task using a very different sequence of actions. He would call up the appropriate part of the table on the screen, and transfer the numbers to an iconic calculator to perform the arithmetic involved.

Specifying procedures using forms

Some systems have added the ability to specify procedures by significantly extending a *forms* metaphor (eg OBE [Zloo81], OFS [Tsic82], FORMAL [Shu85]). Again, however, the method for procedure specification conflicts with the original interactive interface. We will discuss two examples.

The QBE — "query-by-example" — database retrieval system enables a user to type examples of relations in a database [Zloo77]. Some example entries in the query are constants, while others stand for variables that are to be retrieved. For instance a query with the variable *N* in two name fields of a form will return entries with names matching in those two fields. OBE, an extension of QBE, is claimed to provide "two dimensional programming" for non-programmers [Zloo81]. The user can write a program to produce a form letter from database entries, using example elements to specify variables to retrieve. Procedures can be automatically triggered when set conditions occur in the database; for instance to acknowledge new information, follow up previous letters, warn of undesirable conditions in the database, replenish inventory as stocks decrease, and so on. Queries can be combined with letters, reports and graphs which the user can program to extract the required data. Automatic triggers, important in office systems, can cause action to be taken under particular circumstances like low stock numbers. TRI(DAILY) PEN < 500 causes a daily trigger to check for the stock of pens being less than 500.

However, in reality the paradigm of OBE becomes muddled. Despite its name, it is not a programming by example system. Actions are specified explicitly rather than by giving examples. The syntax is procedural and resembles that of a rudimentary conventional programming language. To execute something daily one enters

EXECUTE(DAILY), and to update the sum of the expenses of a manager's staff, Update.SUM.ALL.AMounT.

A second experimental forms system, OFS [Tsic82], has procedures with two parts, a *precondition* and an *action*. Both are specified as forms. The precondition is a request to the system to "find a form that looks like this". For example, one might specify an automatically triggered procedure for processing an item order by entering only the item name on an order form. When an order arrives, this precondition is satisfied, and an action, also specified on a form, can be executed. The procedure might fill in the order form, and do follow-up actions such as copying the form to appropriate users, filing a copy, and even mailing resulting requests for parts.

Like OBE, OFS is not really a programming-by-example system. Although preconditions and actions are specified on forms, the office worker must use special directives within its fields to instruct the system. For example, consider an action form with fields for price, quantity, and total. The directive *#mult !price ?quantity* entered in the total field directs the system to calculate its value by multiplying the updated field value of price by the original field value of quantity, and to insert it into the total field.

Specifying procedures using logic

Logic programming languages enable users to specify procedures for database enquiry by stating goals rather than methods. A general control strategy embedded in the language interpreter frees users from having to consider control structure. For example, programs in the PROLOG [Cloc81] language can be specified generally, and, according to its proponents, naturally, by anyone who knows ordinary predicate logic [Enna82].

PROLOG accomplishes goal-directed inference by following a set of "rules" which constitute the program, and using a collection of facts represented in a database. The user formulates his enquiry by filling in known fields of a form-like statement, and PROLOG infers the unknown fields. Any set of fields can be left unfilled, and the system will retrieve all that match the partial specification; or all could be filled and the system will simply check whether the record is in the database. As well as retrieving records given a partial specification, PROLOG permits problems to be broken down into components through statements like

Goal is true if $Subgoal_1$ and $Subgoal_2$ and ... and $Subgoal_n$ are true.

Read declaratively, this is a statement about the relation between goals and subgoals rather than a specification of a procedure. However, the same statement can be interpreted procedurally, as

to execute the procedure *Goal*, execute procedures $Subgoal_1$ and $Subgoal_2$ and ... and $Subgoal_n$

A computation of a logic program amounts to the construction of a proof of an existentially quantified goal from axioms which are in effect the statements of the program. While this may seem a rather abstract way to view a procedure, it frees the user from thinking about the control structure of his program and allows him to concentrate on how the task can be decomposed into successively simpler subgoals.

A raw textual interface to PROLOG is certainly not well suited to the needs of the casual office user. However, the fundamental idea of a goal-based specification language, with control structure supplied automatically by the interpreter, could be disguised as a forms interface. Such a scheme has several advantages. Firstly, user and system would always have an equal opportunity to fill in any slot of any form. It has been persuasively argued that the principle that "everything that can be supplied or demanded by the machine can also be supplied or demanded by the user", is a sound guide to user interface design [Runc86]. Secondly, forms could be "active", dynamically displaying the contents of unknown fields as the user fills in and removes values from known fields. Thirdly, PROLOG's foundation in predicate logic may provide the well-educated casual user with familiar semantics [Enna82]. Finally, a declarative, goal-oriented language may be more natural than the procedural paradigm of conventional programming languages or the *ad hoc*, action-oriented paradigm of the forms languages described earlier.

There are several unresolved questions in the use of logic programming to enable users to specify procedures for database enquiry. Implementations of logic programming include some procedural constructs which undermine the clean declarative semantics. Debugging normally requires a procedural approach too. Nevertheless, logic programming does seem to offer potential for specifying certain types of procedures using goals rather than methods.

Knowledge-based systems

Outside the office, research has shown that "knowledge bases" can sometimes effectively support inference and problem-solving [Haye83]. Rather than having to specify a procedure in response to a new requirement, we may simply be able to consult a knowledge-based system designed for the problem area. This would allow users to accomplish procedures through informal natural-language descriptions. We anticipate the eventual emergence of knowledge-based systems which support the specification of office procedures. However, existing expert systems are confined to narrow, well-defined domains, while office work is varied, wide-ranging, and open-ended. A survey of sixty odd expert systems lists no office-related ones [Geva83].

Systems which embody detailed semantic models of certain areas of office activity have been constructed for some highly constrained domains. For example, Barber [Barb83] proposes to support office work with a knowledge-based system. However, he views office work as problem-solving and is therefore not concerned with specifying procedures. While certainly conceding that problem-solving is an essential component of office activity, we nevertheless expect many problems to be more easily solved by specifying procedures than by having to pose them correctly to a problem-solver. Several examples were given earlier.

Much of the knowledge needed for accomplishing office procedures is also needed to understand natural language questions. For example, Kaczmarek *et al* [Kacz83] describe an experimental natural language interface for interactive computer services such as electronic mail, personal calendar, word processing, and so on. The interface is customized to a particular set of services, and might include knowledge that

- a meeting can have an owner, some participants, a time interval, and so on
- an individual's schedule is composed of a sequence of meetings
- forwarding is an operation valid for messages but not meetings.

A vast collection of such facts would be needed by an "intelligent assistant" which could learn, understand and use office procedures, and significant research is required before we can understand how to organize this kind of detailed knowledge.

Programming by example

A promising method of communicating procedures is programming "by example". The user performs an example of the required procedure, and the system remembers it for later repetition. Simple examples include text editors that remember a sequence of user keystrokes, and industrial robots that can be "programmed" by leading them manually. However, such systems are limited since they can only repeat a fixed sequence. To be more useful programming by example must also permit the sequence to be edited, conditional, iteration and recursive nested control constructs to be added, data structures to be formed, and variables to replace certain objects in the example sequence. Table 1 illustrates these constructs.

Four distinct ways have been proposed to accomplish these requirements:

- inference from several example sequences [Witt81, Gain76,And84a]
- generalization from example sequences based on specific knowledge of the problem domain (analogously to existing methods for concept learning [Mitc82, Samm83, Samm86] and robot programming [And84b,And84c])
- inference from input and output without the intervening trace [Nix83, Nix84]
- explicit elaboration of the example sequence by the casual user [Smit75, Halb81, Halb84].

In each case the user gives one or more examples and the system is expected to *infer* a procedure for performing the task. To be useful, the procedure must work correctly not only for the original examples but in other situations too. In this sense the inference is *inductive* since a general conclusion must be derived from individual cases. The examples must be generalized. Like any inference, such a procedure may be incorrect in that it fails in some situations. It is up to the user to give suitable examples in the right way.

A skilled teacher will select illuminating examples himself and thereby simplify the learner's task. The benefits of carefully constructed examples were appreciated in the earliest research efforts in learning. Winston [Wins75] showed how "near misses" — constructs that differ in just one crucial respect from examples of a concept being taught — could radically diminish the search required for generalization. Confident that its teacher is selecting examples helpfully, a learning system can assume that any difference between an example being shown and its nascent procedure is a critical feature.

Recently, Van Lehn [Lehn83] has formalized this notion of a sympathetic teacher in terms of what he calls "felicity conditions", constraints imposed on or satisfied by a teacher that make learning better than from random examples. One obvious condition is that the teacher should not (intentionally or unintentionally) mislead the student. Another is that the examples given should correctly represent the procedure. If the absence of something is important, the teacher should point it out explicitly. For example, if the absence of a particular document is crucial to an office procedure, the user should always check for it when performing examples of the procedure. More generally, the teacher should show all work and avoid glossing over intermediate results. Examples should not by coincidence include features that might mislead the learner: one should not illustrate a procedure to change the recorded name of newly married women, by using a woman whose original family name happens to coincide with her husband's; one should not illustrate the geometric concept of isosceles triangles with ones that happen to be congruent. Finally, the teacher should introduce one essential new feature per lesson and not try to teach multiple differences at once — a similar condition to Winston's "near miss" approach.

But how easy is it for an office worker to provide good examples to a learning system? Office workers are not trained teachers. When systems expect to be taught through strict felicity conditions, the user requires not only a deep understanding of the concept, but must be capable of selecting effective examples. Second, in practice learning systems requiring more than a single example demand "perfect" ones — they cannot infer effectively from the (typically) noisy traces provided. Third, there is a question of confidence — when does the user feel he can trust the inferred program? Finally, the notion of "ease of use" may mislead the user into expectations that cannot be delivered [Thim86].

The general problem of inducing procedures from examples is rather intractable. Inference of rules from examples has long been studied, is in general very difficult, and requires either the exploration of vast search spaces or the cooperation of the user in augmenting examples with more general control information. Despite these difficulties, however, programming by example fits naturally into the paradigm of direct manipulation in office systems. Indeed, it is hard to see how the potential benefits of direct manipulation can be fully realized in the absence of a method for *showing* the system how to do routine procedures. Following an initial introduction to the easy problem of specifying straight sequences of actions by examples, we report four ways which have been proposed for demonstrating more complex tasks. The systems described are limited and experimental, but nevertheless instructive. Although not all are confined to examples of office procedures, their techniques could be applied to offices in the future.

Macro operations

Programming by example has long been used for specifying procedures that are simply sequences or "macros" of elementary operations. No inference is required since the example is the procedure. For instance, the method of leading industrial robots "by the hand" is a popular way of specifying a movement sequence to a robot [Alla79, MacD84]. Spray-painting robots are taught by being led through the painting sequence, the operator actually painting a sample object as he goes [Vacc82,Hau74a,Hau74b,Prod82,Indu82]. The robot can then repeat the sequence, painting another object, which must be in the same position and orientation. Even though this technique can capture only fixed sequences, it overcomes the difficulty of having to specify continuous painting movements in the six dimensions of robot hand position and orientation.

In another domain, the use of "start-remembering", "stop-remembering", and "do-it" commands [Gosl81] enable a text editor to learn editing sequences by example. Such sequences can be named and filed for later use. One sequence can invoke another, allowing a complex hierarchy of nested sequences to be specified. However, such constructs as iteration, conditionals, variables and data structures are not accommodated. A practical difficulty with having a special mode — remembering mode — for recording a sequence is that one frequently has already started the sequence before deciding to record it, and so must retrace one's steps and begin again. Also, some mechanism should be available to the user for removing errors as they occur or for editing them out afterwards. Otherwise, the error will remain in the trace, possibly rendering it useless.

To specify procedures more complex than a fixed sequence, it is not enough to record just one example. There are two ways to view the problem. On the one hand, we might consider that to create such procedures, variables, conditional and iterative constructs, and perhaps data structures, are required. Moreover, the resulting structure should be editable. On the other, we might regard the given sequence as an example of performance which is to be *generalized* into the desired procedure. The procedure might still contain variables, data structures, conditionals, iteration and nesting, since these are all themselves generalizations of elements in the example.

Inference from traces

Practical systems for inferring procedures from example traces of their execution lie somewhere between two extremes. One pole involves no knowledge of the semantic domain within which procedures are constructed, while the other relies on domain knowledge to guide generalization of the examples. And while one might regard the two poles as similar in principle, differing only in the amount of domain knowledge available, in practice different techniques are employed. This section introduces three systems with little domain knowledge,

- inference of a sorting procedure
- a self-programming calculator
- learning to "count in the head", or internalizing numerical manipulation;

and contains a more extended discussion of one with substantial domain knowledge:

- acquiring robot procedures from examples.

Inference of a sorting procedure. Programming by example can be trivial if the examples are presented in the right way. Gaines [Gain76] showed how a program to sort a list of numbers can be inferred from a trace of execution on a single example. The trace is expressed in a programming language notation. The fragment of Figure 1(a) shows how a particular five element array A can be sorted using a bubble sort in 86 statements, six of which are reproduced. Figure 1(b) explains how conditionals are included by noting the condition that held at the time.

Due to the use of variables, many statements are repeated in the trace. The entire 86 statements include only 15 different ones, six of which appear in the Figure. When these 15 statements are connected into a directed graph according to their position in the trace, a correct and almost complete flowchart for a bubble sort appears! Only a link necessary for sorting an empty array is missing, and an example with the empty array completes the procedure.

While this is an impressive demonstration that programming by example is possible, it does place a heavy burden on the user. He must choose variables judiciously. For instance, it would be easier to give traces in the form shown in Figure 1(c), without variables. Then the method would fail completely because almost every statement in the trace would be different. Generating a trace of the required type for a bubble sort is tedious and error-prone, particularly since variables must be incorporated and mentally updated when the trace is compiled.

The system does assimilate control constructs from examples. Still, the user must make all tests explicit. *If ... then ... else* is performed by "? ... ?" statements which record the results of conditional tests in the example trace. Branching occurs naturally from this once the directed graph is formed.

Self-programming calculator. Another project studied the inference of iterative computations from examples executed on an electronic calculator [Witt81]. People who use interactive computers regularly know that there are many situations in which it is difficult to decide whether to do a minor, but repetitive, task by hand or to write a program to accomplish it. Simple, repetitive, arithmetic operations frequently present this quandary. For example, one may wish to plot $y = xe^{1-x}$ for a dozen or so values of x; should it be done on a hand calculator or by writing a BASIC program? The self-programming calculator watched the keystrokes made by a person calculating $.1e^{1-.1}$ and then $.2e^{1-.2}$, and inferred the sequence by halfway through the second iteration. Figure 2 illustrates this example. Once it has inferred the sequence, the calculator behaves as though it had been explicitly programmed for the job, pausing for input of x and then immediately calculating the corresponding value of y.

The generalization scheme is simple but effective. The calculator must infer that .1 typed by the user is an input variable, but that the 1 in the exponent is not. It does so by waiting for a second confirmation of a constant number before accepting it as part of the predictable sequence. Thus the user needs to enter the "1" in xe^{1-x} twice before the system will incorporate it as a constant into its stored procedure. However, the problem is not so easily solved in general. For example, if the user had entered the sequence given in Figure 2(b), the calculator would never have realized that the two occurrences of x were the same parameter.

The calculator modelled did not have conditionals, so these were not inferred. Only the outer, infinite iteration was inferred. Also, the calculator does not enable the sequence to be edited, and has no facility for such things as single-stepping through the taught program. Although these appear to be implementation considerations, they may also have implications at the conceptual level.

Internalizing numerical manipulation. A scheme has been described which learns to count with the aid of an external counter [And84a]. It can push buttons to clock over the digits of a three-digit counter, then when the counter is removed it can count "in its head", thus internalizing the externally supported counting. The system's teacher takes it action by action through an example sequence. As the interaction proceeds, the system gradually makes more and more decisions for itself, eventually being able to count internally. The learning sequences are too long to reproduce here. The summary of Figure 3 only shows crucial events in a trace fragment; less significant events intervene between all those shown. Actions are for counting "out loud", or for pushing buttons on the counter. The values of counter digits are inputs to the system.

Figure 3(a) shows the teaching of a few counting combinations, using the counters. Of the vast number of possible combinations, only enough are taught to illustrate digit incrementing, names of digits and transfer of carries. The system is taught to count from zero using a counter with three digits, and a button for incrementing each digit. Teaching is done in a cycle of three main steps:

- output current count
- push an increment button on the counter
- see new count digit.

On being presented with a stimulus for hundreds, tens or units — representing a one, ten or one hundred value monetary note — the system is able to count without the aid of the counters, by "imagining" the counters "in its head" (Figure 3(b)).

The system remembers multiple fixed length sequences of events — a multiple context — from the example counting sequence. Later it uses these remembered sequences to predict and perform actions for counting monetary values.

We have now seen three systems which infer procedures from traces using little domain knowledge. They vary widely in suitability for the casual user. The self-programming calculator is easiest to use, while the others require special knowledge of the user when specifying a procedure. The sorting system required the user to use variables explicitly and indicate conditional tests in trace fragments. The counting system requires the user to give all actions in the fragment traces, including some required for the internal operation of the learning system that are not obviously part of the external task. Although the implementations themselves are experimental and probably somewhat fragile, we regard all three systems as conceptually robust because they deal directly with traces from the real world rather than relying on built-in knowledge and its inherent fragility in real situations.

Acquisition of robot procedures. The knowledge-based system NODDY [And84b,And84c] acquires robot procedures, complete with control information which is not explicitly present in the examples. It copes with problems of action sequencing, and also handles real numbers representing angles and distances. It employs an explicit, pre-programmed, generalization hierarchy, and pre-programmed information on about 30 basic mathematical and set-theoretic operators that may be combined to create complex generalizations.

Examples are traces of the desired procedure. The first trace is taken to be the initial version of the procedure. As further traces are seen they are merged with the nascent procedure, generalizing it in various ways. The system cannot reconsider generalizations it has made in the current version of the procedure, and therefore adopts a conservative policy of requiring considerable evidence before generalizing. The elements that make up the traces are called "descriptors".

The principal problem is to meld two example traces of execution of a procedure into one augmented state-transition representation that encompasses them both. In the first stage of generalization, if two descriptors are identical and unique within each example trace, they are assumed to emanate from the same state. In particular each trace begins with a *start* descriptor which forces the initial states of the two sequences to be merged, and ends with *stop* which merges final states. This builds one state model from the two traces. However the parts corresponding to each still remain largely separate, since descriptors are rarely identical (apart from *start* and *stop*). The second stage of generalization examines states that are different but whose predecessor states are the same, or whose successor states are the same, and attempts to unify the descriptors associated with each. If they can be unified, then the states are coalesced. Unification is done through the generalization hierarchy, and succeeds if the two descriptors have a common generalization. Since two states may be unified only if their successors or predecessors have been unified, the process proceeds from states matched at the first stage, propagating both backward and forward. Finally, a third stage examines states that are different but whose predecessors *and* successors are the same, and again attempts to unify the descriptors but this time with a more liberal unification procedure. This uses the same generalization hierarchy as before, but involves synthesizing functions which unify parameters of the descriptors. Synthesis is accomplished by searching a function space, possibly using numbers which have been "mentioned" in nearby descriptors as components of the function. This introduces variables into the state-transition representation, effectively creating a form of augmented state-transition network [Wood70]. It is a very expensive process in terms of search time, and is only undertaken when there is strong evidence (identical predecessor and successor states) that the descriptors match†.

Figure 4 shows an example of NODDY in action. The system is to be taught how to get from the start point (–6, 0) to the end point (0, 0), circumnavigating obstacles in the way by retreating perpendicularly to the collision surface, moving a short distance parallel to it, and then continuing. Figure 4(a) shows three example traces, both as diagrams and in the form in which they are given to the system. Primitive actions are *start*, *stop*, and a relative *move* with polar coordinates. Observations are also given to the system: its current position *at(x, y)*, and any contact with an obstacle *contact* Ω (Ω being the angle of contact). From these three examples, one with no obstacle and two with single obstacles in different orientations, the system can generate the desired procedure shown in Figure 4(c). Note that the procedure is capable of avoiding several obstacles and even feeling its way round a large obstacle of any (convex) shape‡.

Inspection of the procedure shows that it uses an additional primitive not in the example traces. The action *move-until-contact-toward(x, y)* moves towards a point specified in absolute coordinates, stopping on contact with an obstacle. It was introduced through the generalization hierarchy shown in Figure 4(d). This hierarchy conveys what the system knows about the various actions available. (Also known are the transformations between Cartesian and polar coordinates, not shown.)

When the first two traces are merged, the first stage of generalization unifies the *start* and *stop* states. The second stage examines the two states immediately following *start*, and the two immediately preceding *stop*, and tries to merge them. The generalization hierarchy is used at this point. For example, the *move 6@90* will be merged with *move 3@90* into *move-until-contact 6@90* since the shorter move ends with *contact*. The result of this stage of generalization is to merge the middle state of the first trace with both the second and penultimate state of the second trace, forming a single state out of three. This is shown in Figure 4(b) as the state *at(?, ?); move-until-contact-toward(0, 0)*, in other words, from anywhere, move toward the goal until contact is made. As can be seen, this is the point at which the loop in the procedure appears. When the third trace is considered, state merging is prevented by different parameters appearing in the *move* actions. However, there are strong structural reasons for attempting to merge corresponding states, and so NODDY embarks upon the third stage of generalization, namely functional induction. It rationalizes the difference between the *move* parameters because they are functions of the contact angle already observed. This produces the final procedure.

NODDY has an inherent robustness in requiring considerable justification for its generalizations. However, once an incorrect generalization is made the mistake cannot be corrected. Still, NODDY is yet an experimental system.

A problem closely related to programming procedures by example is "concept learning"; for example learning the concept of a royal flush poker hand. In fact the distinction between procedure and concepts is blurred, since descriptions can be executed by modern programming languages such as PROLOG [Witt87]. Established theoretical results for identifying languages show that there is no alternative to searching the space of candidate descriptions [Gold67, Angl83, Witt87]. In general this makes acquiring procedures from examples intractable, because of the huge spaces that would need to be searched. However, practical systems provide methods for limiting the search. Andreae's [And84b,And84c] system effectively uses built-in knowledge and the requirement for justification to limit this search. The version space method stores the finite upper and lower lattice edges of the plausible generalization set, attempting to merge the two as examples are presented [Mitc82]. Thus it enables a search of a finitely wide, but perhaps infinitely deep, lattice of generalizations [Witt87]. MARVIN [Samm83, Samm86] synthesizes general purpose computer programs from examples, requiring the teacher to give considerable guidance about what already known procedures might be used in the new one, thus drastically limiting the search. Witten and MacDonald [Witt87] investigate the practical and fundamental issues of concept learning. While holding great promise for the future, concept learning is not yet a mature technology suitable for practicing knowledge engineers or office system users.

---

† In fact, the procedure does not require both predecessor and successor states to be identical, but is applied to parallel chains of states that begin and end in the same state and, if successful, merges corresponding members of the chains. Consequently the merging is only done if both predecessor and successor states end up being identical.

‡ NODDY can also negotiate some concave objects.

Inference from input and output

All the above systems for programming by example attempt to generalize the user's action sequences into procedures. The user's actions in each case are a *single sequence*, a "trace" of an execution of the desired procedure. But inference need not use traces; instead it could be based solely on the initial and final states exhibited in the examples. One might say that the user shows the system *what* to do rather than how to do it.

The Editing by Example system of Nix [Nix83], embedded within a screen editor, allows users to exemplify a text transformation. The system then attempts to synthesize a procedure for selecting other parts of the file which contain "similar' text and performing the transformation on them too. If only one example is specified, the system will seek a literal match with the input elsewhere in the file and simply replace it. However, other examples serve to show which parts of the text are "constant" and which may vary. A template is constructed which distinguishes constant and variable parts, and an editing transformation is built which re-arranges the variable parts appropriately. The transformation is represented as a "gap grammar", a set of grammatical rules in which boths gaps and strings appear. Figure 5(a) shows sample input and output texts, while (b) gives a gap program which implements the transformation and could be inferred from the examples.

The scheme is interesting in that input/output pairs are used to exemplify a procedure. Information about *how* the user performs the transformation is discarded. While it seems rash to jettison potentially useful information, Nix notes that any reasonably rich editor will offer many ways for performing any given job. For example, a user might search for a string visually and move the cursor to it, or use a search command. Moreover, users make errors and repair them as they go, perhaps accidently deleting text and retyping it. Such operations should not be faithfully repeated whenever the procedure is executed! Analyzing traces would be virtually impossible if users were to define new commands in an extensible editor. Nix concludes that little is lost in discarding such unreliable information.

This may be so in an environment like an editor, where inputs and outputs are well defined. However, the argument is unlikely to apply in general. In ill-defined environments it may well be hopeless to attempt to infer procedures just from input and output. At the very least, traces give clues as to the sequence the procedure might follow.

Improved explicit methods

An experimental "programming by example" interface has been constructed for the Xerox Star office workstation which operates according to the direct manipulation paradigm [Halb81, Halb84]. First, a method for recording sequences of actions was established, with commands for:

- start-remembering
- stop-remembering
- do-it,

similar to the robot and text editor methods described above. Actions were recorded at a level of abstraction which matches the user's conceptualization, as illustrated in Figure 6. Part (a) shows a simple sequence for moving a file to another folder. Note the level of abstraction in "select Report" rather than "move the cursor to point (260,410)".

When an icon is selected on the screen, it is necessary to disambiguate the mode of reference. The significance for the user may be in the icon's name or its position; alternatively there may be no significance in that particular selection (for instance, when iterating through all icons). Which choice is make will radically affect future executions of the procedure. This is the problem of generalization: how does a system infer general descriptions given only some examples? The self-programming calculator system resolved the question by waiting until it had seen a couple of iterations and inferring that items that changed were input. A similar but more sophisticated approach is taken by Mitchell [Mitc82], who assumes a hierarchy for generalization of items, rather than the calculator's "constant-or-variable" dichotomy. However, in the office context this would

require a strong semantic model.

Instead, one can ask users to indicate explicitly how to generalize example icons, an approach pioneered originally by Smith [Smit75]. Halbert's first system [Halb81] had users select the generalization from a pop-up menu. To open a folder and extract the first item, putting it on the desktop and closing the folder, a user might perform the steps shown in Figure 6(b). After each selection he would indicate how to generalize the icon using the choices shown in (d). For example, "same place" after "select *To-Smith*" indicates that the program should choose whatever is in that position on the screen. The procedure formed is shown in CUSP notation in (c), while Figure 7 indicates what the screen might look like when the initial example has been finished. Unfortunately it is all too easy to forget to generalize an item while recording, so [Halb84] had users specify this at the end of the example instead. Users manipulated the program itself, using an interface of icons, menus for generalization and iteration, and so on. For example, when recording the sequence of Figure 6(b), the user would omit the lines "ask for a similar object" and "same place". When he had finished he would be prompted to select a generalization from Figure 6(d) for each item involved (ie *Letters* and *To-Smith*). Halbert's design change reflects his belief that it is hard to denote programming constructs when tracing through an example, but easy to do so afterwards. Whereas CUSP is hard to write, he considers it easy to edit.

Both Halbert's systems represent a considerable improvement over other explicit methods such as command languages. However, they do not avoid the difficulty some users will have in articulating algorithms for complex procedures.

Two systems have recently appeared for programming by example on personal computers: the TEMPO system (Affinity Microsystems Ltd) and AUTOMATOR [Poun87]. Both have limited ability for the casual user to specify control and data structures, sacrificing power for ease of use. TEMPO works within a direct-manipulation interface. The user selects branches and loops explicitly, but conditions are restricted to simple textual comparisons; Figure 8 shows an example. AUTOMATOR is more powerful. It incorporates a general-purpose command language, and thus moves away from the programming by example paradigm. Still, it is claimed that a casual user can easily teach the system such tasks as logging on to a large computer, awaiting a response, retrieving mail, and so on.

Conclusion

Table 2 summarizes the issues raised by our study of current and future prospects for casual-user specification of office procedures. The most effective user interfaces represent real office objects iconically and allow users to manipulate them directly on the screen. However, the direct-manipulation metaphor — at least as presently conceived — does not help when it comes to communicating procedures. Command language systems and monolingual programming environments permit procedures to be specified, but are unsuited to casual users. Even custom-designed command languages such as CUSP fall far short of the ideal for a casual-user interface. Forms-based systems which revolve around a database provide an alternative to icon-oriented systems, and forms programming languages have been defined and implemented. Again, however, these seem to reduce to primitive command languages, lightly disguised. The root problem is that procedural programming in either icon or forms-based systems conflicts with the basic metaphor of the interface. There is some hope that logic programming techniques might be usable within a casual-user forms interface, enabling users to specify procedures for database enquiry by stating goals rather than methods. Another possibility is that knowledge-based systems may eventually allow users to specify some office procedures informally through natural-language descriptions.

A promising method of communicating procedures is programming "by example". To be useful programming by example must permit the specification of conditionals, iteration, nested structures, variables, data structures and editing. Four distinct ways have been proposed to accomplish these requirements, and each has been exemplified by a description of one or more experimental systems. The basic problem is that of

generalization: an example does not by itself provide enough information to unambiguously define the procedure. One way of generalizing is to make inferences from several example sequences. Another is to use specific pre-programmed knowledge of the problem domain. A third is to discard the sequence of steps involved in executing the examples and concentrate on formalizing the input-output transformation that they exhibit. Finally, one can ask the user to elaborate the example sequence explicitly, and provide a convenient interface for him to do so.

The column of Table 2 labelled "robustness" is of particular interest. Systems which provide feedback to the user directly and unambiguously are quite robust since users will seize the opportunity to correct any errors of interpretation. These include direct manipulation, forms interfaces, and explicit programming by example. In all these cases users can see what they are doing. This contrasts with command languages, which present great opportunities for misunderstanding, although in practice human interface technology has developed to the point where the dangers can be anticipated and defused (by such techniques as alert messages and undo commands). Programming by example with only non-branching sequences lacks robustness because small changes in the definition of a problem can easily take it outside the system's capabilities. Programming by example with traces has the potential to be highly robust by dispensing with any *a priori* assumptions and making inferences directly from actual data. Moreover such systems typically retain almost all the information in examples they have seen, so conclusions can always be reconsidered. Knowledge-based systems, on the other hand, are likely to lack robustness at least until knowledge bases are much larger than they tend to be at present.

Although there are no easy solutions to the problem of end-user definition of office procedures, programming by example — with either explicit or implicit generalization — appears to have considerable potential in the near future.

Acknowledgements

This work is supported by the Natural Science and Engineering Research Council of Canada. David Pauli made some useful observations about NODDY and concave objects.

References

[Alla79] Allan, R. (1979) "Busy Robots Spur Productivity" *IEEE Spectrum, 16* (9) 31-6.

[Ande80] Anderson, B. (1980) "Programming in the home of the future" *Int J Man_Machine Studies, 12* (4) 341-365, May.

[And84a] Andreae, J.H. (1984) "Numbers in the head" 5-28, Man-Machine Studies Progress Report UC-DSE/24, Department of Electrical and Elcetronic Engineering, University of Canterbury, Christchurch, New Zealand.

[And84b] Andreae, P.M. (1984) "Constraint limited generalization: acquiring procedures from examples" *Proc American Association on Artificial Intelligence,* Austin, TX, August.

[And84c] Andreae, P.M. (1984) "Justified generalization: acquiring procedures from examples" PhD Thesis, Department of Electrical Engineering and Computer Science, MIT.

[Angl83] Angluin, D. and Smith, C.H. (1983) "Inductive Inference: Theory and Methods" *Computing Surveys, 15* (3) 237-269, September.

[Barb83] Barber, G.R. (1983) "Supporting organizational problem solving with a workstation" *ACM Trans*

Office Information Systems, 1 (1) 45-67, January.

[Cloc81] Clocksin, W.F. and Mellish, C.S. (1981) *Programming in Prolog*. Springer-Verlag, Berlin.

[Cuff80] Cuff, R.N. (1980) "On casual users" *Int J Man-Machine Studies, 12*, 163-187.

[Enna82] Ennals, R. (1982) "Teaching logic as a computer language in schools" February.

[Gain76] Gaines, B.R. (1976) "Behaviour/structure transformations under uncertainty" *Int J Man-Machine Studies, 8*, 337-365.

[Geva83] Gevarter, W.B. (1983) "Expert systems: limited but powerful" *IEEE Spectrum*, 39-45, August.

[Gold67] Gold, E.M. (1967) "Language identification in the limit" *Information and Control, 10*, 447-474.

[Gosl81] Gosling, J.A. (1981) *Unix Emacs manual*. Carnegie-Mellon University.

[Halb81] Halbert, D.C. (1981) "An example of programming by example" Technical Report, Xerox Office Products Division, Palo Alto, California.

[Halb84] Halbert, D.C. (1984) "Programming by example" Technical report, Xerox Office Products Division, Palo Alto, California, December.

[Hau74a] Haugan, K.M. (1974) "Spray Painting Robots: Advanced Paint Shop Automation" *The Industrial Robot*, 270-2, December.

[Hau74b] Haugan, K.M. and Jarvis, D.E. (1974) "Electronically Controlled Manipulator for Spray Gun Applications" *The Industrial Robot*, 119-21, March.

[Haye83] Hayes-Roth, F., Waterman, D.A., and Lenat, D.B. (1983) *Building expert systems*. Addison-Wesley, Reading. Mass., (editors).

[Indu82] (1982) "New products from Trallfa" *Industrial Robot, The*, 78, June.

[Kacz83] Kaczmarek, T., Mark, W., and Sondheimer, N. (1983) "The Consul/CUE interface: an intergrated interactive environment" *Proc ACM CHI 83 Human factors in Computing Systems*, 98-102, Boston, December 12-15.

[Kuni82] Kunin, J.S. (1982) "Analysis and specification of office procedures" PhD Thesis, Department of Electrical Engineering and Computer Science, MIT, February.

[Lee83] Lee, A. and Lochovsky, F.H. (1983) "Enhancing the usability of an office information system through direct manipulation" *Proc ACM CHI 83 Human Factors in Computing System*, 130-134, Boston, December 12-15.

[Lehn83] Van Lehn, K. (1983) "Felicity conditions for human skill acquisition: validating an AI-based theory." Research Report CIS-21, Xerox PARC, Palo Alto, November.

[MacD84] MacDonald, B.A. (1984) "Designing Teachable Robots" PhD thesis, Canterbury University, Christchurch, New Zealand.

[Mitc82] Mitchell, T.M. (1982) "Generalization as search" *Artificial Intelligence, 18*, 203-226.

[Myer86] Myers, B.A (1986) "Visual programming, programming by example, and program visualization: a taxonomy" *Proc ACM CHI 86 Human Factors in Computing Systems*, 59-66, Boston, MA, April 13-17.

[Nix83] Nix, R. (1983) "Editing by example" PhD Dissertation, Computer Science Department, Yale University, New Haven, CT.

[Nix84] Nix, R. (1984) "Editing by example" *Proc 11th ACM Symposium on Principles of Programming Languages*, 186-195, Salt Lake City, Utah.

[Poun87] Pountain, D. (1987) "The Software Robot" *Byte, January*, 383-390.

[Prod82] (1982) "Painting robots Halve Cycle Times for Subcontractor" *Production Engineer, The*, 50-51, May.

[Raed85] Raeder, G. (1985) "A survey of current graphical programming techniques" *IEEE Computer, 18* (8) 11-25.

[Runc86] Runciman, C. and Thimbleby, H. (1986) "Equal opportunity interactive systems" Report, Computer Science Department, University of York, York .

[Samm83] Sammut, C. and Banerji, R. (1983) "Hierarchical memories: an aid to concept learning" *Proc International Machine Learning Workshop*, 74-80, Allerton House, Monticello, IL, June 22-24.

[Samm86] Sammut, C. and Banerji, R. (1986) "Learning concepts by asking questions" in *Machine learning Volume 2*, edited by R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, pp 167-191. Morgan Kaufmann Inc, Los Altos, CA.

[Shne83] Shneiderman, B. (1983) "Direct manipulation: a step beyond programming languages" *IEEE*

Computer, 16 (8) 57-69, August.

[Shu85] Shu, C.S. (1985) "FORMAL: A forms-oriented visual-directed application development system" *IEEE Computer, 18* (8) 38-49.

[Smit75] Smith, D.C. (1975) "Pygmalion: A computer program to model and stimulate creative thought" PhD thesis, Department of Computer Science, Stanford University.

[Such83] Suchman, L.A. (1983) "Office procedure as practical action: models of work and system design" *ACM Trans on Office Information Systems, 1* (4) 320-328.

[Such85] Suchman, L.A. (1985) "Plans and situated actions: the problem of human-machine communication" PhD Thesis, Xerox PARC, Palo Alto, CA.

[Thim86] Thimbleby, H. (1986) "Ease of use — the ultimate deception" in *People and Computers: Designing for Usability (Proceeding of the Second Conference of the British Computer Society Human Computer Ineraction Specialist Group)*, edited by M.D. Harrison and A.F. Monk. University of York, September 23-26.

[Tsic82] Tsichritzis, D. (1982) "Form management" *Communications of the Association for Computing Machinery, 25* (7) 453-478, July.

[Vacc82] Vaccari, J.A. (1982) "Robots that paint create jobs" *American Machinist*, 131-134, January.

[Wins75] Winston, P.H. (1975) "Learning structural descriptions from examples" in *The psychology of computer vision*, edited by P.H.Winston. McGraw Hill, New York, NY.

[Witt81] Witten, I.H. (1981) "Programming by example for the casual user: a case study" *Proc. Canadian Man-Computer Communication Conference*, 105-113, Waterloo, Ontario, June.

[Witt85] Witten, I.H. and Greenberg, S. (1985) "User interfaces for office systems" *Oxford Surveys in Information Technology, 2*, 69-104.

[Witt87] Witten, I.H and MacDonald, B.A. (1987) "Concept Learning: A Practical Tool for Knowledge Acquisition?" *submitted to Workshop on Expert Systems and Their Application*, Avignon, also available as report ???? Computer Science dept., University of Calgary..

[Wood70] Woods, W.A. (1970) "Transition network grammars for natural language analysis" *Communications of the Association for Computing Machinery, 13* (10) 591-606, October.

[Zloo77] Zloof, M.M. (1977) "Query-by-example: a data base language" *IBM Systems J, 4*, 324-343.

[Zloo81] Zloof, M.M. (1981) "QBE/OBE: A language for office and business automation" *IEEE Computer*, 13-22, May.

Table 2: Summary of methods for specifying procedures to office systems

Method	Robustness	Suitability for the casual user	Procedure specification	Systems
Direct Manipulation	good	excellent	no	Macintosh, Star
Command Languages	no (idea) yes (implementation)	no	powerful	Unix sh, IBM JCL, BASIC, Lisp, Pascal, Ada, ...
Forms	good	good in limited domains	muddled	QBE, OBE, OFS, Prolog
Knowledge-based	experimental	experimental	pose problem in narrow domain	Barber, Kaczmarek
PBE† — sequences	no	good	limited	Emacs, industrial robots
PBE — traces with little domain knowledge	good (has all information recorded in traces)	experimental	good	Gaines, Witten, J.Andreae
PBE — traces with domain knowledge	experimental	experimental	limited to narrow domains	P.Andreae
PBE — from input/output	yes	yes	limited to well-defined domains such as text transformation	Nix
PBE — explicit	good	yes, so long as the procedure does not become complex	good	Halbert

† PBE stands for Programming-by-example

Figure 1: Inference of a sorting procedure [Gain76]

(a) Fragment showing six of 86 statements in a trace of sorting a particular five element array
(b) Format of the conditionals in the fragment.
(c) A fragment of a more reasonable trace for a user to give. Unfortunately the method would fail completely with this trace since almost every statement in the trace would be different.

(a)

Trace	Notes
...	
? A[i+1] < A[i] ?	*notes that the next two elements are out of order*
x = A[i]	*swaps them, using temporary variable x*
A[i] = A[i+1]	
A[i+1] = x	
i = i + 1	
? i < t-1 ?	*notes that the end of the array has not yet been reached*
...	

(b)

? A[i+1] < A[i] ?

means

note that for the present value of i (say 1) and array A's present contents (say A[1] = 20 and A[2] = 15), A[i+1] is less than A[i]

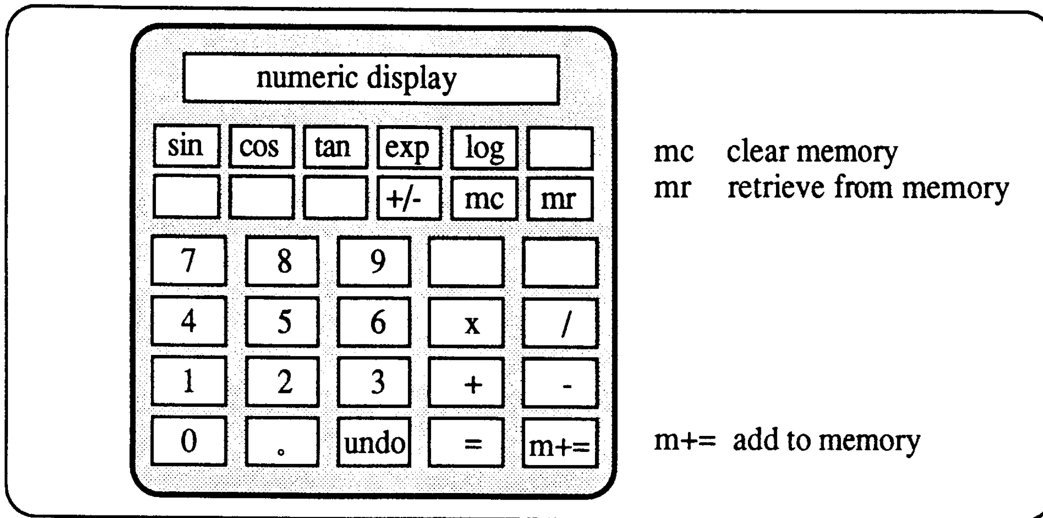
(c)

Trace	Notes
...	
? A[2] < A[1] ?	*notes that the next two elements are out of order*
x = A[1]	*swaps them, using temporary variable x*
A[1] = A[2]	
A[2] = x	
2 < 5	*notes that the end of the array has not yet been reached*
...	

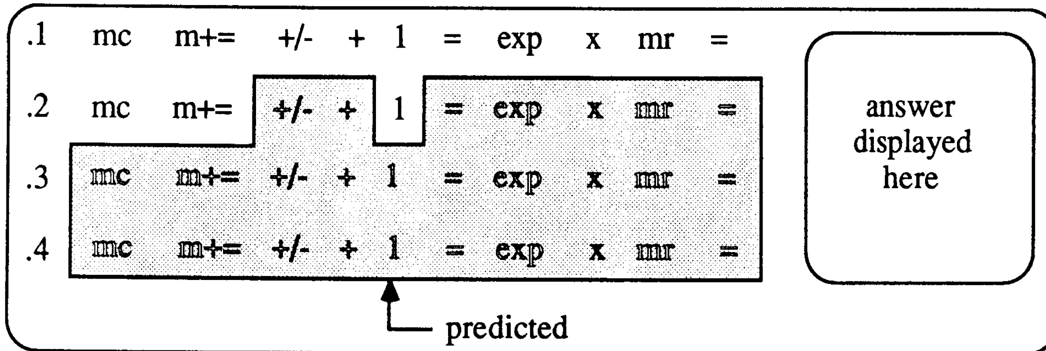
Figure 2: Self-programming calculator [Witt81]

(a) The calculator attempts to predict and push the next key itself. An *undo* allows the user to correct.
(b) Learning $y=xe^{1-x}$. X is .1, .2, .3, then .4. The shaded area shows the operations the calculator performs.
(c) If the user performs the calculation without using the memory, then the calculator will never realize that the two different inputs for the same variable are the same.

(a)



(b)



(c)

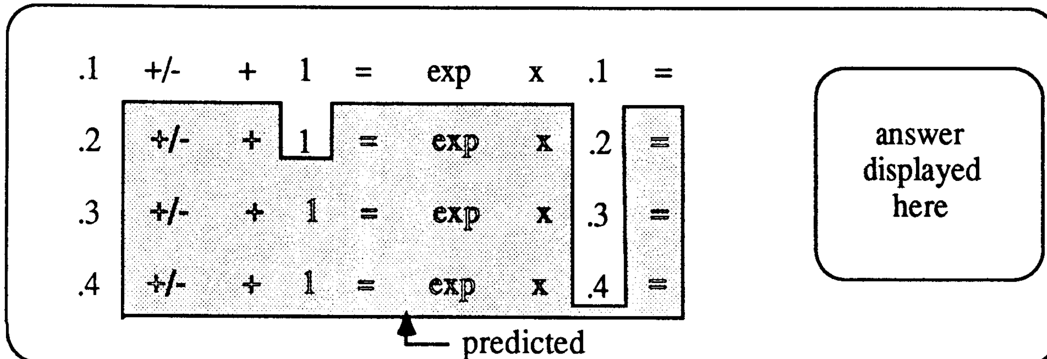


Figure 3: Internalizing numerical manipulation [Andr84]

(a) Firstly the system is taught to count from zero.
(b) The counters are removed. The system counts monetary values "in its head".

(a)

Action	*Input*
NO HUNDREDS	-
NO TENS	-
NO UNITS	-
push units button	see units digit "1"
NO HUNDREDS	-
NO TENS	-
ONE UNIT	-
push units button	see units digit "2"
NO HUNDREDS	-
NO TENS	-
TWO UNITS	-
...	
(the teaching continues on to teach other combinations, but it is not necessary to teach every one possible)	

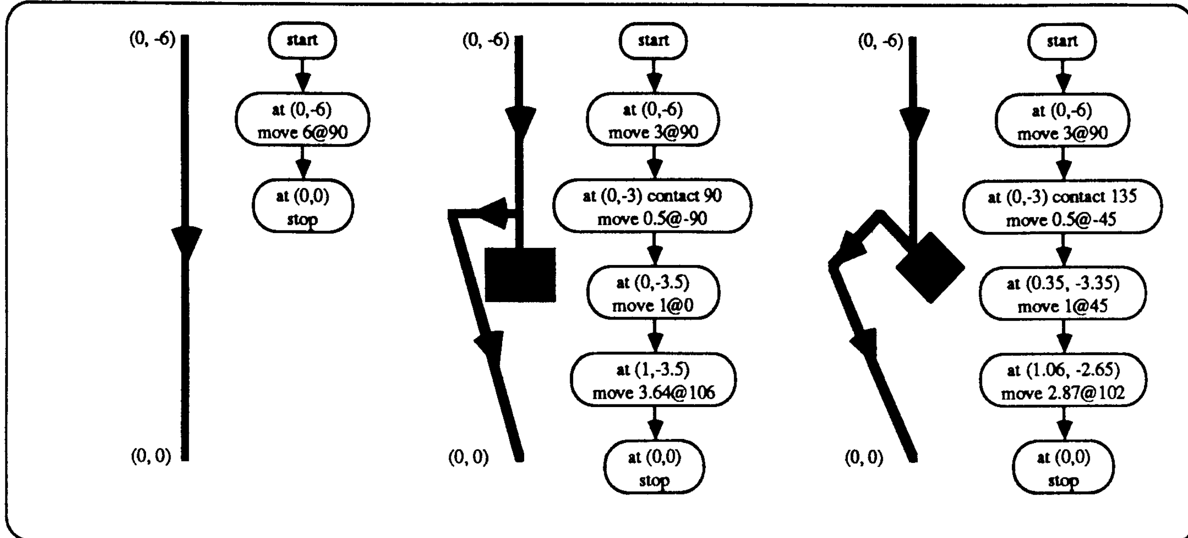
(b)

Actions	*Inputs*
(the count is presently at 120)	
...	
ONE HUNDREDS	-
TWO TENS	-
NO UNITS	see hundred unit note
TWO HUNDREDS	-
TWO TENS	-
NO UNITS	-
...	
(and so on)	

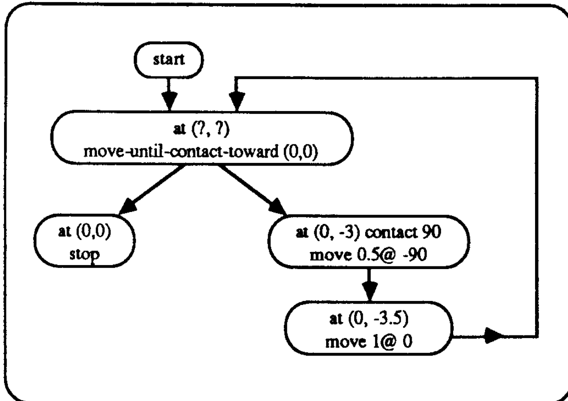
Figure 4: Programming a robot by example [And84b].

(a) Three example robot traces
(b) Partially complete procedure
(c) Final, desired procedure
(d) Generalization hierarchy

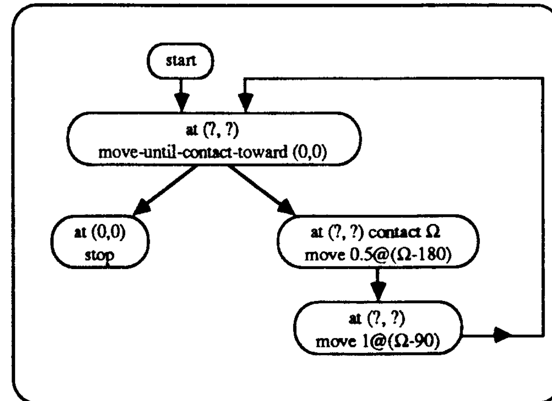
(a)



(b)



(c)



(d)

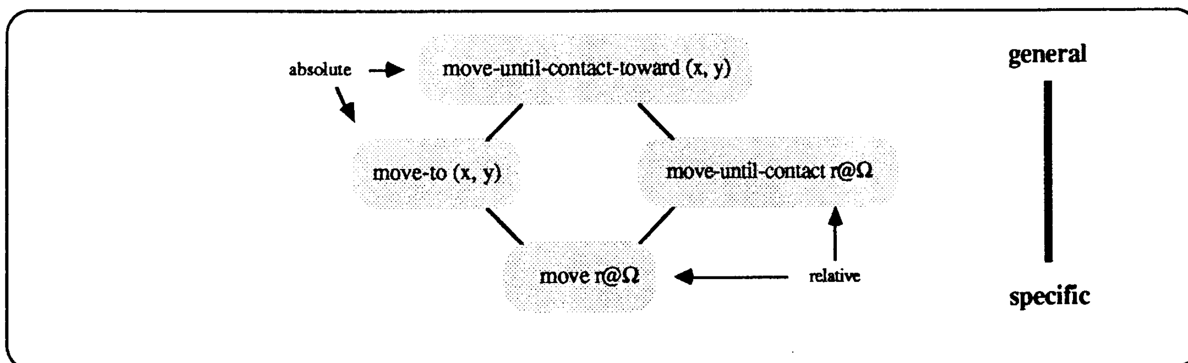
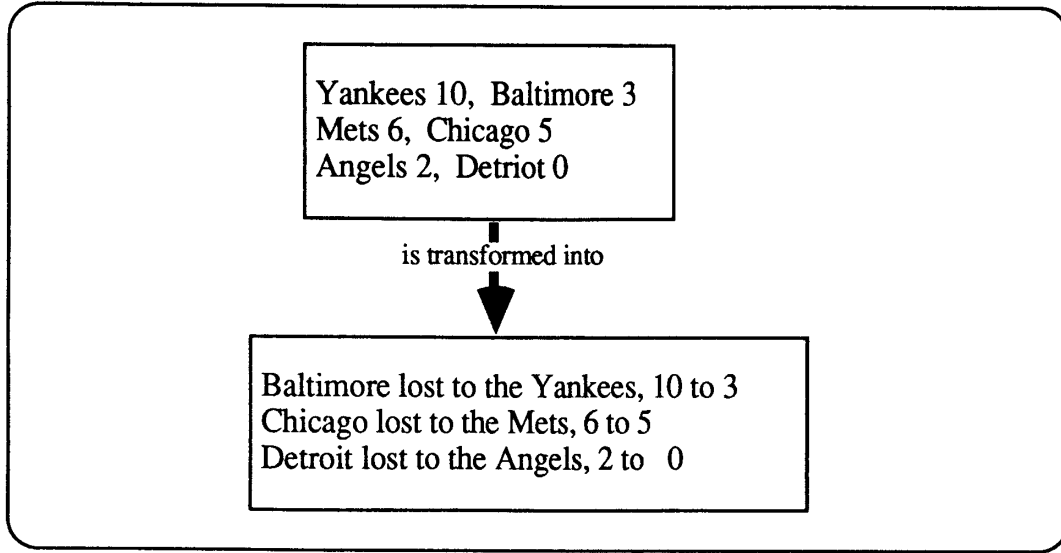


Figure 5: Editing by example [Nix83, Nix84]

(a) Example text transformation
(b) A gap program for (a). □ indicates a gap.

(a)



(b)

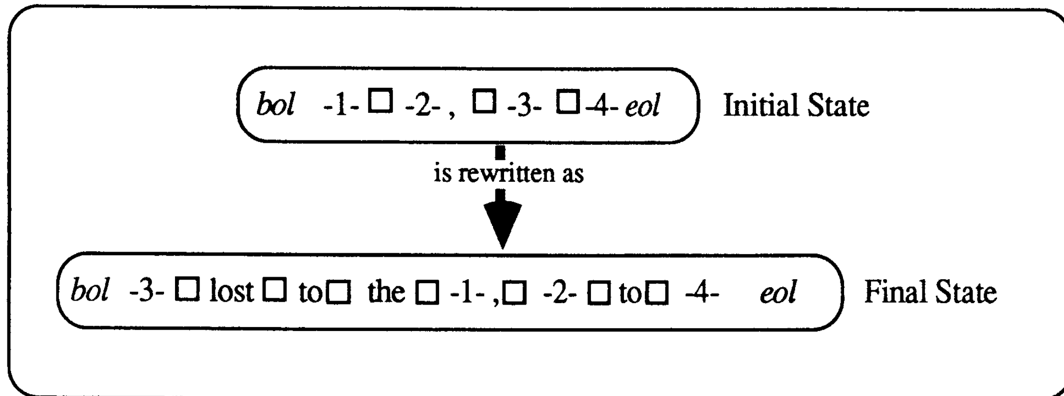


Figure 6: The Xerox Star programming-by-example interface [Halb81,Halb84]

(a) Actions recorded as a straight sequence, as the user performs the task using a mouse.
(b) Specifying generalization during recording. Again these actions are recorded as the user manipulates a mouse, windows, menus, and function keys (see also Figure 7).
(c) CUSP procedure formed for (b) [Halb81]
(d) The generalizations possible

(a)

Select Report Move Report to Reports-Folder

(b)

Action	Generalization
start recording	
select *Letters*	ask for a similar object
open† *Letters*	
select *To-Smith*	same place
move†	
set down cursor on the desktop	
close† *Letters*	
stop recording	

†These generic system actions are selected by function keys

(c)

```
Define A to be the icon asked for by ''Select a Folder, then resume.'';
Open A;
Define B to be the icon in row 1 of A;
Move B to the Desktop;
Close A;
```

(d)

same name *same place* *ask for a similar object*

Figure 7: A mock-up of Halbert's [Halb81] example task given in Figure 6

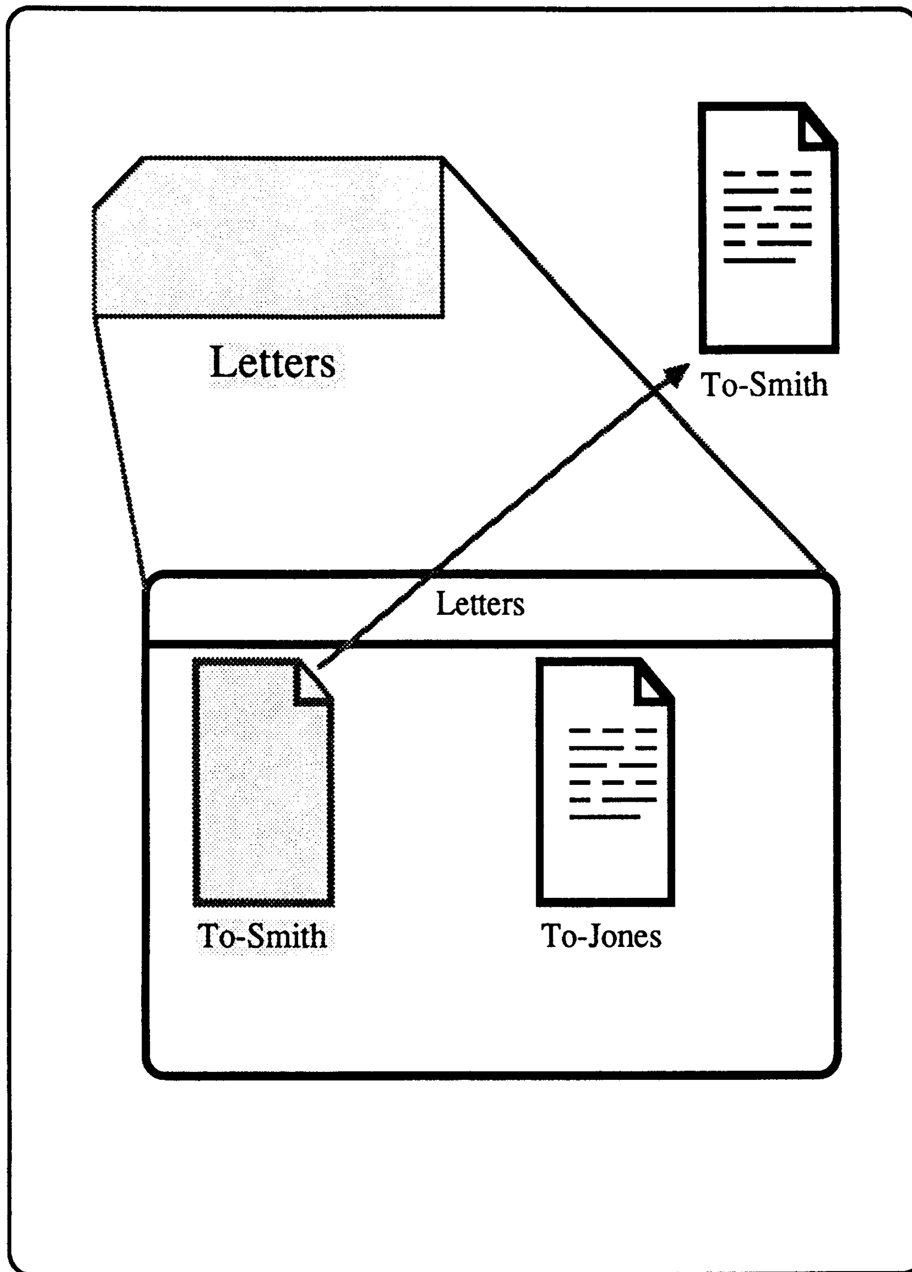


Figure 8: Specifying an iteration in Tempo

Tempo is recording user actions in the direct manipulation interface to the Apple MacIntosh. The user has explicitly selected the option "Loop If", and selected "=" as the test between the clipboard contents and the text "Applications". The clipboard is shown at top right with that same text in it. So in this case the loop would continue.