1. Introduction

A human centred approach to computing seems at odds with the goal of artificial intelligence (AI) to replicate, supplement and perhaps eventually supplant human intelligence. However, there are signs that AI may be able to make an important contribution to human centred computing. First, the recent rapid growth of research activity in machine learning signals a new interest in methods of learning from human users, as well as from the environment in general. Second, modern techniques of human computer interaction allow a high degree of coupling and two way information flow between user and computer, providing a rich source of information about human behaviour.

Taken together, these two trends are beginning to provide a foundation for systems that learn from their human users and help to automate predictable parts of their tasks. They are built around the paradigm of "equal opportunity," in that both system and user have the ability to alter the current state of the task (Shneiderman & Maulsby, 1986). However, the user must retain the authority and the responsibility to reject inappropriate actions suggested by the system, and to undo actions performed by either party. Systems help out when they can, leaving control and creativity to their human users (Smith, 1975; Halbert, 1984; 1986).

The aim of "programming by example" is to create systems that learn how to perform tasks from their human users by being shown examples of what is to be done. Just as the user creates a learning environment for the system, so the system provides a teaching opportunity for the user, and emphasis is placed as much on facilitating successful teaching as on underlying techniques of machine learning. These systems attempt to capitalize on the interactive situation to increase the limited power of automatic methods of induction. The purpose of this article is to portray the current state of the art in this area.

Present techniques of programming by example are most effective for simple, but repetitive, tasks. People who employ internal (computer) systems are frequently confronted with such tasks. Indeed, the advent of computer technology seems to have added to our problems by providing another source of tedious, repetitive tasks. These can be frustrating to perform manually, yet are too variable and intellectually insignificant to automate by a custom built program.

Areas where repetitive tasks occur include

- technical drawing;
- text editing;
- office tasks;
- robot assembly tasks.

We have created prototype systems in each of these areas, and these are described below. First, however, a number of principles that have emerged from our work are presented. Following an account of the four systems, common threads are identified by returning to the principles and relating them to features of the systems. Finally we introduce the notion of "felicity conditions," which describe the mutual expectations of teacher and learner and prove to be a convenient and productive way of characterizing their interactions.

2. Principles

> I hear and I forget
> I see and I remember
> I do and I understand
>
> Chinese saying, quoted by Cooley (1987)

Programming by example is based upon *doing*. The human user begins a task and the machine attempts to complete it, or perform a similar task, right away. As far as users are concerned, they are placed in a teaching situation. But this is teaching "on the job," not in the classroom—the system is a kind of apprentice who learns the teacher's craft by watching and helping. This is important because it means that users are not diverted from their primary tasks and compelled to undertake a different kind of activity (as they are, for example, when they decide to write a conventional program to accomplish the task). Teaching takes place during normal task performance, and with minimal disruption to it.

This section begins by investigating the teaching metaphor and exploring its consequences for system design. The emphasis on action pervades the learner too, and we go on to discuss its impact on the learning paradigm.

2.1 Teaching metaphor

Just as the desktop metaphor facilitates manually performed tasks, users can be encouraged to adopt a teaching stance by employing a suitable metaphor through which they can conceptualize the target of teaching. Users who have a clear idea of the capabilities of the learner will automatically adapt their exposition to capitalize on its strengths and compensate for its weaknesses. The success of the metaphor will determine how well the user's teaching matches the system's ability to learn.

The use of a teaching metaphor has a number of important consequences. First, there must be an identifiable "student" to teach—users should conceptualize the system as a persona. Second, users will be inclined to adopt an intentional stance to the system and explain its actions by attributing to it knowledge and even purpose ("it's trying to find another ... but it can't because it doesn't realize that ..."). If teaching is to be effective, the system must meet these expectations. Third, users will be prepared to follow rules for successful teaching, and the system may take advantage of this. Fourth, users can be encouraged to provide additional information such as constructions to make the constraints of the task explicit. Finally, the metaphor should be removable: users may need an opportunity to practice the task before teaching it, to avoid confusing the learner with their own mistakes. Each of these points sets up expectations in the user that should be met by the system design, and these are discussed below.

PERSONIFICATION OF THE LEARNER

We have found it helpful to represent, or personify, the system as a visible entity that reacts to events around it. Teachers might move an icon around so that it can "touch" objects, "pick them up" and "carry them around," or they might enter into a dialogue with an animated, talking face. The teacher should be given as accurate a conception of the capabilities of the learner as possible. For example, if it is oblivious to part of its environment it can be described as "blind" or as having an appropriately restricted sense of sight. Focusing the system's attention is an essential prerequisite for successful induction. The focus should indicate what the system should attend to and provide visible feedback to the user, perhaps by highlighting objects on the screen, gesturally by moving its hand or eyes, or verbally by synthetic speech or cartoon speech bubbles.

ATTRIBUTING INTENTIONALITY

Users attribute intentionality to complex programs (Dennett, 1981, 1987). In programming by example, the teacher is executing, creating and debugging the program, and it is important to simplify the interaction by having the system behave consistently with the intentions attributed to it. This is facilitated by a rich representation for the learner that reveals its "knowledge," "attention," and "purpose" as clearly and accurately as possible. Dennett claims that effective human communication *requires* participants to attribute intentions, so the system must meet these human expectations.

CONSTRUCTIVE APPROACH

It is important that teachers make hidden features of a task explicit. It is easy to find situations where a completely intractable induction is made very simple by providing an appropriate hint. Such hints constitute a kind of commentary on the actions that comprise the task, and can take different forms. For example, in a geometric domain the user may teach the system to create temporary constructions and erase them when its work is done. As another example, a finite-state machine can be made into a universal computing device by the addition of an external notepad and the ability to make notes (Minsky, 1967): users can be empowered to teach the system how to create and use such a commentary (Andreae & Cleary, 1976).

If the system expects users to adopt a constructive approach, then when a "surprising" (underconstrained) action is encountered, it may notify the user and point out that a construction may have been omitted.

PRIVATE PRACTICE

We have observed that users may initially perform interactive tasks by trial and error, and settle on a suitable procedure only after some experimentation. It is worth providing a practice mode to allow a suitable procedure to be discovered. This could simply take the form of a command that resets the environment to its original state, to be invoked after each practice attempt. However, if the programming-by-example system is designed to interject with its own suggestions during performance of the task, this may well disturb users who are practicing, and it should be possible to turn off the apprentice so that a user can practice privately.

2.2 The learning paradigm

The learner learns by doing. Its model will be action-oriented or "procedural." However, a fully procedural orientation makes it difficult to incorporate domain knowledge to guide the system towards appropriate generalizations.

PROCEDURAL REPRESENTATION OF TASKS

The apprenticeship model of learning by doing encourages a procedural approach to programming by example, where a sequential trace of user actions is generalized into an explicit procedural representation.

A possible alternative is to capture just the input and output to the program and attempt to infer the demonstrated mapping. The problem of inducing a function from examples of its input and output cannot be solved in general without an exponentially growing search over the space of all possible functions (Phan & Witten, 1990). By ignoring the trace of user actions, this strategy effectively discards information that could be used to reduce the search space. The argument for doing so is that the trace may contain noise, in the form of free variation or unwarranted actions, that would confuse a system that attempted procedural induction (Nix, 1985). However, we opt to use the trace information and attempt to suppress noise by other means.

A second alternative is a knowledge-intensive approach where domain knowledge and plan inference techniques combine to infer the user's goals so that they can be accomplished automatically. This presupposes a comprehensive description of possible tasks, including the goals that they accomplish and preconditions that they depend on. Goal inference techniques (e.g. Cohen *et al.*, 1990; Carberry, 1990) attempt to relate a user's actions to the goal that is being pursued. The drawback to this approach is the amount of knowledge needed and the difficulty of tracking inevitable changes in task requirements and user goals.

CONSTRAINTS

It is infeasible to infer a fully procedural description of most tasks from examples of their performance without the use of some knowledge of the domain. We have found it useful to augment actions with "constraints" or conditions that govern when the actions can be performed. These provide a way to take advantage of domain knowledge when

constructing the system. For example, important features in a graphical environment include end-points of lines, corners and centres of boxes, and intersection of objects; relations on these can be expressed as constraints. Important features in a text-editing environment include the beginning and end of words, lines, and paragraphs, logical attributes of text blocks, and so on; these can participate in relations that constrain editing actions. Robot domain features include object geometry and other characteristics. Office task features include window attributes, textual fields, and other visible properties of a desktop metaphor.

Procedural descriptions, augmented with the ability to condition actions by constraints that embody domain knowledge, provide a rich representation of tasks that we have found useful in a variety of domains. It is also one that can easily be understood by teachers and communicated to them—the system may be able to provide feedback on operative constraints and have the teacher check and modify them explicitly.

ATTENTION

Induction of constraints from examples is intractable in reasonably rich environments. In order to reduce the space through which the system searches for plausible generalizations, it is necessary to focus its attention on relevant aspects (Heise & MacDonald, 1991). Initially, relevance may be determined by physical proximity: objects close to the current action are the ones most likely to participate in relations that constrain it. However, there are many exceptions to this, so it may be useful for the user to be able to see, and alter, the system's focus of attention, drawing its attention to distant features that are also relevant.

SIMPLE ABSTRACT MODEL OF THE TASK

Machine learning of procedures is difficult. In order to make it possible at all, we have found it necessary to distil the essential features of the task environment and produce a clean conceptual model for the learner to work in. Interactive systems often provide a vast number of features designed to increase their convenience to the user, but which are inessential for basic operation of the system. Any attempt to incorporate these features is likely to overwhelm the learning mechanism. Instead, it should work with a clean, stripped-down subset of functionally essential features.

In fact, interactive interfaces often provide features that programming by example renders inessential. For example, a global search-and-replace facility in a text editor is unnecessary

once one can program by example, for the user need only give one or two examples and allow the system to continue the operation itself.

INDUCTIVE TECHNIQUES

The essence of learning from examples is inductive inference—reasoning from the specific to the general. It is possible for non-inductive learners to make deductions from a comprehensive knowledge base, deductions that can be viewed as operationalizing knowledge that was previously implicit (Ellman, 1989). However, systems that learn procedures in real domains cannot avoid induction, partly because a comprehensive knowledge base is not feasible, and partly because they are required to learn genuinely new tasks. An ever-present danger in programming by example is that induction will become intractable—this can happen *even in toy problems* because procedures are inherently far more complex than static concepts. The key to success is a judicious marriage of induction with interaction. The teacher is encouraged to limit combinatorial explosion by guiding the learning process.

UNDO

Programming by example is dangerous. Reducing redundancy of communication between user and machine inevitably increases the likelihood of error. Amplifying the user's actions also amplifies the potentially disastrous consequences of error. In order to encourage users to create programs by example in real situations, it is essential to provide a means of damage control. One simple and natural mechanism which is of wide applicability is a comprehensive "undo" facility.

INTERACTION CAN REDUCE NOISE

The problem of noise plagues any inductive process. In a procedural setting, variation in an action sequence is likely to confound attempts to learn the procedure. However, the interactive situation allows immediate feedback from the system that guides the user into a consistent action sequence. By offering predictions as early as possible the system encourages consistency.

3. Examples

In order to illustrate the operation of these principles, four rather different examples of interactive programming-by-example systems are presented. It is not feasible to include

full details of the working of each system—references are given to more complete descriptions.

3.1 BASIL: an interactive drafting assistant

BASIL is a user interface agent that learns repetitive, constraint-based editing tasks within a drawing program (Maulsby *et al.*, 1989). The system was designed to minimize the difference between programming a task and doing it. The user instructs BASIL by demonstrating the procedure, now and then issuing simple instructions to focus attention and correct mistaken inferences. The teaching metaphor is embodied in a turtle icon that crawls about the display screen, moves to wherever the user clicks with the mouse, and highlights constraints it has observed. "Constraint" in this system means a point of contact between two objects. Thus to show BASIL some spatial relation, the user might have to construct a sequence of touch constraints, by drawing one or more objects connecting the related ones. Teaching an action therefore may involve more work than performing it, but the user can opt out by indicating that a given step is manual. Construction is a viable approach nonetheless, because people find it natural to express themselves procedurally—they envisage spatial relations with imaginary lines and motions and convey them by gestures (Maulsby, 1988).

The learning problem tackled by BASIL is to identify, from examples, the constraints on selecting and positioning primitive 2-D geometrical objects. In both cases, the only features considered are:

- the object type (line, box) and pre-defined parts (edges, vertices, centres);
- whether the target object is already known or must be selected by search;
- the general direction of search over a set of objects (up, down, left, right);
- touch relationships that constrain the action's destination point (e.g. that the end of the line being moved must meet the corner of the target box).

BASIL uses a kind of explanation-based generalization to identify the relevant features (Maulsby *et al.*, 1990). In the absence of sufficient constraints, as when no touch relations result, an action is assumed to be manual (an input). BASIL matches actions to infer loops and branches. In demonstrating multiple examples (iterations), the user might accidentally re-order operations or alter the method. To minimize such irrelevant variation that makes sequence matching harder, the program predicts actions immediately after a match is

found. In this mode the user's role is to accept or reject the actions that BASIL performs, thereby debugging the procedure.

The system uses a constraint solver based on a fast clipping algorithm. If the constraints have no solution (for instance, because a pool of objects for selection has been exhausted), it forms a branch conditional on those constraints. This is how it creates the exit from a loop. If the user rejects a prediction, BASIL tries alternatives. If the user rejects all (by answering "no" or by doing the desired action himself), a branch is formed in which the user's new action is the highest priority choice. This is how programs are debugged, but also how unlearnable conditionals are modelled. A severe limitation of the current system is that it does not try to derive conditionals from preconditions.

The user can correct mistaken inferences by direct instructions given with the mouse. The system highlights touch relations with "tack" icons having two states: black and pushed in means that the touch is a constraint; white and lying on its back means that it is incidental. The user can toggle the state by clicking on the tack. Similarly, the system indicates a search direction by heading BASIL that way; the user can turn BASIL to the desired path. When the system infers an input, it presents a menu of alternative hypotheses—that the relative or absolute position is a program constant, or that the position should have been constructed.

Figure 1 illustrates a session with BASIL in which the user's task is to move a set of boxes to an input line. An additional constraint is that each box is to remain at the same vertical coordinate after it is transformed. The user begins the lesson by drawing the guideline (Figure 1a); BASIL (the turtle) follows the user's cursor (the arrowhead). Since the guideline's endpoints are not constrained by touch relations, a dialog box appears for each one, stating BASIL's default hypothesis that it should be set by the user (in other words, it is an input parameter). The user may accept this or select another option. To enforce the vertical constraint the user draws a horizontal sweepline whose endpoints are program constants (Figure 1b); the user corrects BASIL's default hypothesis. BASIL notes that the sweepline crosses the guideline, but that this does not constrain its endpoint, so he marks the touch with a white tack that seems to be lying on its back.

In demonstrating the program's main loop, the user selects the sweepline, drags it up to a box, and then selects and drags the box to the point where the two lines cross (Figure 1c–e). When the user rolls the sweepline up to the first rectangle (Figure 1d), the path is displayed by rotating BASIL: a dialog box allows the user to disagree with this constraint

either by stating that the path does not matter or by clicking on the turtle to direct him as desired. The system infers a variable for the box, used in touch relations that govern where the sweepline stops. Associated with the variable is a selector function to get its value: *nearest-object-on-path(typeBox, pathUpward)*. In this implementation the user has no access to the selector function itself. The relevant constraints inferred by explanation-based generalization are shown as black tacks that seem to be pressed into the picture. In Figure 1e, after the box is dragged to where the sweep- and guidelines cross, two black tacks indicate that the corner must touch both lines. When the user moves the cursor onto a tack, the objects it joins are highlighted.

When the user selects the sweepline again (Figure 1f), BASIL matches this action with an existing program step (Figure 1c) and asks if he might take over execution. With the user's consent, BASIL performs the second iteration of the loop (Figure 1g–i). At each step the system asks that the user confirm its prediction or demonstrate the correct action; path and touch markers are also editable whenever they appear so that the user can debug by issuing direct instructions as well.

Upon entering the fourth iteration (Figure 1k), BASIL is supposed to drag the sweepline up to another box, but *nearest-object-on-path(typeBox, pathUpward)* returns nil and so the constraint solver fails. The system creates a branch to be executed when this step fails; this is the loop's exit. BASIL asks the user to demonstrate actions to perform in this case; the user deletes the sweep- and guidelines and tells BASIL the lesson is over. The task can be saved for recall by name.

3.2 TELS: help with repetitive text editing

People who employ interactive text editors are frequently confronted with simple, but repetitive, editing tasks. These can be frustrating to perform manually, too variable and individually not significant enough to automate by a custom-built program, and difficult to address with structured editors because the text lacks formal structure. One such task—reformatting a textual database of addresses—is illustrated in Figure 2. The TELS system, whose operation is sketched below, addresses such tasks by providing interactive assistance to the user. More details, including an evaluation of its performance on different editing tasks, can be found in Mo & Witten (1990).

The conceptual editing model used assumes the notion of current text position and the following primitive actions (summarized in Table 1):

- *insert* places text at the current position, replacing selected text if any;
- *locate* allows the user to specify the current position;
- *select* takes a stretch of text and makes it the current position;
- *delete* deletes the currently-selected text.

Programming by example takes place in three stages: recording a trace, generalizing it into a program, and executing and extending it. Following a brief practice session in which they settle on an editing strategy for the task at hand, users perform normal editing operations on the first block of text. These are silently recorded. Once the first block is finished, the user signals the system to enter the second stage, creating an initial program based on the trace that has been recorded. The system then begins to execute the program on the next block of text, generating predictions which the user may accept or reject. When a prediction is rejected the user must indicate the correct action, and the program is automatically extended to accommodate it.

Editing traces are recorded as sequences of actions and attributes. The four primitives above correspond to four types of action. In practice these operations may not be primitive as far as the user is concerned—for example, locating a position may require many *next-line* and *next-character* commands; selecting text may require *mark* and *copy-region* commands—however, they are mapped into actions by the recording process. *Insert* and *select* have a parameter that records the text string involved. *Locate* and *select* require positional information, and with them are recorded attributes that characterize the place at which the action occurred. Three basic kinds of attribute have been identified, and are summarized in Table 1.

Figure 3 shows a sequence of attributes and actions that constitute an editing trace. Actions include typing a string of characters, clicking to reposition the cursor, selecting a stretch of text (or a single word by double-clicking), cutting (either by menu selection or the delete key), and pasting. File operations are not considered to be actions. The system ignores extraneous actions such as consecutive mouse-clicks when recording a trace. The typing and selecting actions have a parameter that records the text string.

Programs are created by generalizing a trace—which is in effect a straight-line program—into a procedure that includes both variables and control structures such as loops and branches. The idea is to identify steps that are to be merged into a single procedure step, and generalize their attributes. The program-creation mechanism generalizes the trace of Figure 3 into the procedure of Figure 4. Nodes are considered to be mergeable whenever

their actions (and parameters, where appropriate) are identical. For example, T1, T3, and T5 are mergeable, as are T2, T4 and T6. Whenever nodes are merged, loops are formed. However, mergeable nodes are only actually merged so long as this does not create a branch into the body of an already-formed loop. For example, step T6 of the trace in Figure 3 is not merged with state S2 of Figure 4, since this would mean branching into a loop.

For select actions, the attributes are discarded. This assumes that the text selected is the only clue to the location of the action—an oversimplification for some tasks, but one that sidesteps the difficult problem of balancing constraints on the context against constraints on the selection parameter when executing such actions. Thus only for click actions do attributes actually need to be generalized; Mo & Witten (1990) describe how this is done.

Once a program has been created from the trace of actions on the first block of text, it is executed on subsequent blocks. This is easiest to explain in terms of an example. Consider applying the program of Figure 4 to the second block (in this case, line) of the input in Figure 2. State S1 finds the first comma (after the name) and S2 suggests a RETURN character, which is accepted by the user. A position satisfying S1 (after "`Suite¤1,¤`") is found before one satisfying S3, and so S2 suggests another RETURN , also accepted. Now S1 finds the comma after "`Banff¤Blvd.`" and suggests a RETURN here. This is not what the user intended. He rejects the prediction and is invited to position the cursor correctly.

This is an over-generalization bug. These are handled by storing with each node "negative examples" (or patterns generalized from several negative examples) that indicate a mismatch with the node. Also, the context of the correct position indicated by the user is merged with that specified in the state to ensure that the correct position will indeed be chosen in the future. In this case the position's prior and posterior contexts are "`N.W.,¤`" and "`Calgary,¤`" which already match the patterns that are stored with the state.

Continuing with the example, the program correctly predicts the RETURN character following "`Calgary,¤`". This leaves it in state S2. Now it fails to identify a position for either of S2's successors, states S1 and S3. Because the pattern in S3, "`284-4983`", is specific, it is generalized to "`[digit][op][digit]`" and a match is sought again. This finds the telephone number at the end of the second line of the task, and so the program predicts that this item will be selected. When the user accepts this prediction, the select action of state S3 is generalized to "`2[digit]-4[digit]`" to accommodate the two patterns it has actually been applied to.

On the third and subsequent blocks of text up to the fifth, the system predicts all actions correctly. However, on the fifth line the telephone number has a different format, and the system proposed to cut just part of it. The teacher rejects this and selects the whole string, which re-invokes the string generalization algorithm to generalize the pattern.

3.3 The CLERK: automating routine office tasks

The direct manipulation, desktop metaphor does not provide an appropriate method of programming—the user must drop back to the level of the development system when writing sequences of instructions. To provide a programming capability, this system constitutes an "office clerk" that is a metaphor of a real office assistant. The current prototype of CLERK, depicted in Figure 5, allows the user to manipulate electronic mail messages and instruct CLERK to undertake related tasks (Campbell, 1990; MacDonald & Witten, 1987). Instructions include giving examples and pointing to important mail header fields.

While the user guides CLERK through a task, mouse and menu operations are recorded. The user may instruct him to focus on a particular visible attribute, and internally this causes a nested conditional branch to be formed. The task can be interrupted, and debugged by stepping through the recorded actions. This also happens automatically when a conditional does not match, allowing the user to guide an additional branch. Window parameters are recorded during instruction and may be generalized as more examples are given, so that window objects become task variables. The learning is interactive and the teacher confirms the inductive generalizations.

CLERK's domain is a front-end interface for electronic mail, comprising a menu-driven, three-level mail system. The top level contains all second-level mailboxes, each of which stores read-only mail messages (the third level), or allows outgoing messages to be created. The mail system provides a simple domain in which to demonstrate CLERK, and allows a variety of possible user-specific tasks for managing daily incoming mail messages. The primitive menu actions are shown in Table 2. The top and message levels have only one menu, while the mailbox level has two. Incoming mail automatically appears in the "inbox" mailbox, and the user can move messages from there to other mailboxes, delete them, rearrange existing messages, create new mailboxes, or compose new mail. Mail could be organized into prioritized mailboxes, saved by fields such as sender or date. The other primitive actions are mouse movements to select, move and re-

size windows, and to select text in messages or entries from a list of messages or mailboxes.

CLERK enables a teacher to automate these operations. As depicted in Figure 5, the teacher interacts with it via a speech emulation. Potential responses are collated into a speech bubble from which the teacher is able to select one. To show a new task, the teacher selects "Learn a new task," CLERK responds with "Lead me through the steps"—as in Figure 5—and the teacher proceeds to perform an example task sequence manually, such as opening the incoming mailbox and storing the first message in a folder named for the sender. During this leading, the teacher may also ask for a previously taught task to be inserted as a sub-task, stop the recording mechanism to rearrange the desktop, name the task, or focus on an important visible aspect of the desktop. Focusing is a metaphor for providing a conditional branch in the acquired procedure, although the user need not know this.

Rather than pay the computational price of induction to discover branches in the procedure from multiple examples, CLERK uses the focusing metaphor to allow non-programmers to specify conditional sequences in a single example. The user selects the response "Focus on this ..." and then, guided by CLERK's responses, selects a window and a field in the window to be used as a conditional test. All actions after a focusing operation are remembered as conditional on the selected field value, until the user selects "Stop focusing on the last thing." A visible reminder of the current focus is provided by a small icon of CLERK placed directly over the selected field. To remind the user of the current task state, all other active foci are shown as inverted icons.

Once a task has been taught and named, the teacher can ask for it to be repeated. Each example sequence will demonstrate only one branch in a conditional. When a task is performed and an unexpected condition holds, CLERK automatically shifts into learning mode and asks the user to demonstrate the new branch. It would be too rigid to expect the desktop to be in the same state whenever a task is executed, and window descriptions are generalized to provide some flexibility in their arrangement. The user may be asked to confirm window generalizations during execution, as CLERK builds up a full description. When no window matches the task description, CLERK asks the user to open the appropriate one, and extends its description of the window required at that point of the task.

Window descriptions are learned using the "version space" or "candidate elimination" method (Mitchell, 1982). For each window in a task, CLERK maintains an upper and a lower boundary in the partial order of possible window descriptions. The more the task is executed the narrower the gap between the boundaries becomes, finally converging on the intended description. Window concepts have features of window type (mailbox or message), parent window, index in mailbox list, and name.

The CLERK prototype has been evaluated in a small study of one experienced desktop user. Two tasks were successfully taught: sorting mail by sender, and sorting by sender while deleting unimportant mail. The key was setting up a conditional on the mail sender. Future extensions will include: more window attributes, loop generation, more powerful variable induction, new domains, an undo mechanism, and the use of task parameters. CLERK gives the user an opportunity to give instructions to something tangible, rather than the mysterious internals of machines that programmers and command-language script writers must address.

3.4 ETAR: robot programming by example

Robots live in the real world, so teaching is direct rather than metaphorical. To show an example robot task the teacher simply grabs the robot's hand and forces it through the required motion sequence. This is a common method in industry, but produces fixed, inflexible, low-level sequences, rather than the richly represented procedures needed for more interesting tasks. ETAR learns robot assembly procedures—with loops, branches, and variables—from user-guided examples (Heise, 1989; Heise & MacDonald, 1990). An automatic focusing mechanism limits the computation required to learn these.

Initially ETAR is given a set of primitive robot motions, a set of arithmetic functions, and a workspace model. There are five primitive motions: *translate*, *rotate*, and *translate-rotate* provide accurate, interpolated movements, *move-to* is a ballistic move, and *grip* controls the robot hand. The arithmetic functions (+, –, ×, /) are used to compose expressions for motion command parameters. The workspace model is a frame hierarchy giving the characteristics of objects in the robot's environment. The model is indexed by position, emulating visual information.

Given the two examples shown in Figure 6, ETAR learns the procedure of Figure 7. Teaching begins with a statement of the task name, and a reference to the important

objects, such as `stack block531 on block892`, after which the teacher leads the robot through the task. ETAR is passive during leading, leaving the teacher completely free to perform the desired task *with the robot's arm.* The teacher must lead a few different examples of the task, showing various repetitions and branches. Then ETAR generalizes the recorded examples to form a procedure for the task such as the one in Figure 7.

The robot joints and gripper are sampled at about 10 Hz, recording a stream of numerical data during each example. This data is transformed from joint to world coordinates. The positions are used to index the workspace model and annotate the sequence with information about the focus of attention: that is, about the objects near the hand at each data point. The numerical robot data is then partitioned into primitive robot motions, using the focus of attention changes as a guide to the important manipulations in the sequence. Each step of an example trace now comprises an outer-level symbolic primitive, with an inner-level set of numeric arguments (giving commands such as `moveto 1,2,3,90,90,-45`), and is annotated with focus information about nearby objects. As well as indexing the object frames, the annotation identifies objects explicitly mentioned in the initial task statement. Finally redundant subsequences are reduced to single primitives using heuristic rewrite rules and the focusing annotations.

The induction engine in ETAR first takes this symbolic sequence and generalizes it at the outer symbolic level to find repeated structures for loops and to find differences for branches. This is accomplished by both intra- and inter-trace matching. Repeated sequences are discovered by attempting to match fragments of a sequence with other fragments of itself. Conditional branches fall out when matching between sequences fails. The matcher operates on groups of motions, allowing a match when the same key movements (e.g. grasp or release) occur and the corresponding focus objects have the same roles. Figure 8 shows the state of the generalization process after this first stage.

The second induction step generates the low-level arguments for symbolic robot actions, attempting to merge parallel sequences where symbolic motions match. However, the arguments may be expressions that are composed of the initial arithmetic functions applied to characteristics of the manipulated objects—for example `moveto p+q` where `p` and `q` are positional characteristics of nearby objects, or simple constants. The computation in enumerating the enormous space of possible expressions is drastically reduced by the focusing mechanism, which allows only a few objects to be referenced in each expression. Finally, a symbolic concept learning stage finds the conditions for loop exits and branches, using the A^q method (Michalski, 1983) and the focus information. A branch

(together with the alternative paths) and a loop (together with the termination) provide positive (and negative) examples for A^q.

ETAR has successfully learned several assembly tasks in addition to the stacking task, such as obtaining the first block with a particular characteristic from a stack, and sorting objects as they come off a conveyor.

4. Synthesis

Relating the four example systems to the principles set out above sheds further light on the design options and trade-offs in programming by example. This allows us to pull together common threads from the example systems, showing how they relate and in what respects they differ, apart from the superficial disparities caused by their task domains. We address each of the principles in turn.

PERSONIFICATION OF THE LEARNER

Two of the systems place great stress on personification of the learner. BASIL is presented to prospective users through the bio-sheet shown in Figure 9. This is intended to convey the extent of his perceptual facilities so that users will have some idea what they can expect him to observe. CLERK, as Figure 5 shows, is a cartoon figure that communicates with the user through a speech bubble. Both of these systems are designed to give the impression that, to teach tasks successfully, the user should strive to "help the system out" by focusing its attention appropriately or by inventing construction tools to make relations explicit. ETAR is an off-line procedure in which active personification is inappropriate, but a feeling of presence is generated when a teacher leads the robot through task examples. This is an important benefit of the direct teaching methods such as leading. A metaphor is unnecessary in robot teaching, as the teacher directly interacts with the robot. TELS generates a degree of personification by employing the direct leading method as the user demonstrates tasks and through interaction with the user during debugging.

ATTRIBUTING INTENTIONALITY

Both BASIL and CLERK actively encourage the user to attribute intentions to them. BASIL's teacher can assume that BASIL intends to help as soon as he can, achieve the teacher's (implicitly inferred) goal, and display all important conditions. The CLERK's teacher can assume intentions to: carry on with the sequential task unless the teacher

explicitly interrupts, watch the teacher's moves and record them exactly, try to find suitable windows to perform actions in, and tell the user when new conditions occur in a task. By watching the position of the editing cursor during a demonstration and interacting when there is an error, TELS apparently intends to discover correct conditions for actions.

CONSTRUCTIVE APPROACH

Constructions take the form of auxiliary parts of a task; parts that are unnecessary for performance but make the essential aspects more explicit or otherwise easier to learn. Examples are the focusing actions used to help CLERK, and the auxiliary objects used in BASIL's constructive geometric tasks.

Three questions to consider are:

- Does the system enable the teacher to include auxiliary actions and objects?
- Does the system use them when performing the task or only when learning it?
- Does the system notify the teacher when construction may be necessary, i.e. when constraints cannot be inferred?

BASIL uses boxes and lines as auxiliary objects both in learning and performing, and since they are remembered as individuals it is easy to teach BASIL to remove them after use. This is the only system that notifies the teacher when construction might be needed, although CLERK always presents the opportunity for focusing. TELS could use inserted text to mark positions, but since it does not remember text objects it would confuse them with similar markings when removing them. ETAR infers position constraints and conditionals involving objects brought into its focus of attention. The CLERK allows auxiliary objects to be brought into the task, such as strings for focusing, to form conditional branches in the task.

PRIVATE PRACTICE

All four systems allow practice while the learner is not watching. Formally, this is nothing more than a "suspension" feature added to the implementation of the learner. It would be very inconvenient for the teacher if the learner could not be stopped to allow readjustment of the environment—for example, if the required parts are not available during a robot assembly task.

PROCEDURAL REPRESENTATION OF TASKS

A central tenet of our work is that the procedural approach is a viable way to enable the user to assist the system perform computationally infeasible feats of induction. All four systems use a strong sequential, procedural representation of tasks. However, an explicit state-transition representation of procedures, such as the one illustrated in Figure 4, can become unwieldy. It is a low level of representation which is difficult to modify and extend reliably.

For this reason, one of the examples—BASIL—uses a production-system representation of procedures. This seems a promising approach because ideally, separate productions are independent and allow chunks of functionality to be added (and, perhaps, subtracted) individually. However, it is not clear whether this extends to more complex tasks. A further possibility, which has not yet been investigated, is to use structured state-transition nets, or structured programs, to represent procedures. These would require limits to be set on the state-merging process that forms the basis of induction, so that poor program structures (such as branches out of loops) could not be created. It would be attractive to be able to provide facilities for the system to reason about the effects of such programs, through some kind of formal semantics.

CONSTRAINTS

BASIL represents an action as one primitive operator (e.g. select or move) governed by constraints to represent the implicit relationships, and selects parameters using a constraint-solver. Touch relations are used to constrain an action's destination, and "selector functions" to constrain the choice of objects in those relations. A weak domain theory orders the significance of observed touches; the most significant are chosen as constraints. In TELS the criteria for selecting a string or cursor location are the contents of the string selected or of strings up to a system-specified length on either side of it. TELS also uses a weak domain theory; for instance, certain neighboring characters—such as end-of-line—are assumed to be especially constraining. CLERK infers constraints on windows used during a task, based on their features. ETAR infers functional relationships between robot motion parameters and information in the current focus of attention. It also infers conditional branches and loop termination conditions.

BASIL and TELS represent constraints as postconditions, and CLERK represents them as preconditions, while ETAR uses preconditions in control flow and numerical functions for parameters.

ATTENTION

Three of the systems have a static bias, a fixed set of attributes that influence the learned task description. In BASIL this is the primitive objects and their vertices, edges, and centres; in TELS it includes words and characters; while in ETAR it is the set of features used to describe the workspace and objects therein. In order to help make the induction process feasible, three of the systems have a dynamic bias: BASIL selects features from objects being touched but allows the user to override this attentional focus; CLERK performs induction only when the user specifically indicates that a portion of the screen must be focused on; ETAR automatically varies the focus of attention during each task example and between task examples. ETAR always selects those objects that are near the robot's hand in a particular example and limits the features to those that are common between examples.

SIMPLE ABSTRACT MODEL OF THE TASK

Three of the systems are based on a simple abstract model of the task, in order to minimize their complexity (which is already very high). There are differences in how this model is described and used, however. CLERK allows users to perform *any* action permitted by the Macintosh user interface, and is able to record and play back any sequence of operations. However, it can perform generalization only on a small subset of window attributes that it knows about. This provides an important kind of upward compatibility. Unlike the other systems, ETAR is built around a model of its task which is *formal*, and is described by a simple grammar. Although this does not have any direct bearing on useability since it is inaccessible at the user level, it proves to be of great assistance during system development by providing a clear, formal, and consistent description of the task domain.

INDUCTION

BASIL, TELS and ETAR possess complex induction algorithms. BASIL induces productions with constraints, while the others induce procedures that include abstractions (such as conditionals and loops). CLERK has a minimal inductive component, instead relying on the teacher to give abstractions using a natural method of specification in a

metaphorical interface. CLERK does induce window descriptions, but there is no induction on the sequence. In practice ETAR, TELS and BASIL require a few examples of a task, which are then merged and generalized to form a procedure. BASIL and TELS will generalize even a single trace, while ETAR is more conservative, making the first "procedure" exactly the initial trace, and waiting for new traces to force generalization.

UNDO

Since BASIL represents its target procedure as a series of productions, a step is undone simply by removing the corresponding node. The other three systems use a procedural representation in which steps are merged into a general description, so an undo facility requires previous states to be saved. Furthermore, undo mechanisms pose special problems for systems that allow complex and irreversible actions, such as ETAR, which operates in the real world. A simple step reversal may not be sufficient, since an action may affect multiple objects, as when a robot knocks over a pile of blocks. In this case, undoing requires a planner to determine a series of steps which returns the environment into its previous state. At other times, as when an object is transformed (for example by having a hole drilled in it), the state modification may be permanent and irrevocable.

INTERACTION CAN REDUCE NOISE

BASIL begins to predict actions as soon as possible. In this way, the user sees the correctness of the learned procedure and, when necessary, changes technique to maintain consistency. TELS and CLERK wait for a complete example before attempting a task. However, they allow procedural steps to be corrected through a debugger. ETAR handles some noise automatically, while sampling and modelling a robot example path. The attention mechanism removes unnecessary steps (where no object is being manipulated), while a simple rule removes the redundancy resulting from precision "fiddling." There is a tradeoff: a system that is eager to predict must ask the user whether its predictions are acceptable, and this may *introduce* noise by asking too many questions. One has to be careful that the number of questions is not a rapidly growing function of the task length and complexity. As yet, none of the systems are able to handle extreme noise, such as learning when a task example does not achieve the desired goal.

5. Felicity conditions

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 concept learning. Winston (1975) showed how "near misses"—constructs which differ in just one crucial respect from a concept being taught—can radically diminish the search required for generalization. Confident that its teacher is selecting examples helpfully, a learning system can assume that any difference between a shown example and its nascent concept is in fact a critical feature.

The notion of a sympathetic teacher has been formalized in terms of "felicity conditions," constraints imposed on or satisfied by a teacher that improve on the random selection of examples (Van Lehn, 1990). Table 3 shows the felicity conditions we have adopted for programming-by-example systems. There is a dual aspect to the notion of felicity conditions in learning systems: on the one hand, they are rules a teacher should follow; on the other, they are assumptions that a learning system makes about the instructions it receives. Their purpose is to reduce the complexity of inference needed to build an appropriate model. The rules formulated below address inference problems encountered by machines with perfect memories, hence they differ somewhat from Van Lehn's. Moreover, we have operationalized them in terms of specific inference problems.

Task learning systems induce a representation of a procedural task from the examples, attention focusing and constructions provided by the teacher. Explicitly or implicitly this involves a search over a subset of all procedures describable using the domain's primitives. Information in the teacher's trace helps to reduce the search space explicitly. The assumption that felicity conditions are met by the teacher reduces it implicitly.

Meeting felicity conditions can be difficult—most people are not gifted teachers. It is important that the learning system help the teacher follow the rules of good instruction, and that it fail gracefully when they are violated. The systems we have described introduce conventions and techniques of interaction that promote good teaching tactics or restrict the user's agenda so that unhelpful methods are prevented.. In the following subsections we examine six rules of teaching practice, the consequences of violations, and the methods used by the example systems to help the teacher avoid violations. The rules are somewhat overstated in the sense that satisfying them perfectly would eliminate the need for inference

altogether—in fact, the intention is to minimize inference to make it feasible, without demanding too much of the teacher.

SHOW ALL STEPS

When demonstrating a procedure, the teacher should show all the steps of a task. Ideally this means decomposing it into steps simple enough that the system could execute each one without further teaching, but in practice a teacher will not know enough about the learner's internals to judge this. It follows that the system should be prepared to accommodate the teacher's interpretations of "all the task steps." Violations of this rule occur when the teacher performs computations mentally and presents only the results. This forces the system into the computational nightmare of inducing implicit constraints without any help in reducing the search space.

Systems described here support this felicity condition by focusing the teacher's attention on action granularity. This is done through system design, the wording of prompts, and matching the granularity of interface feedback to that of observable actions.

SHOW ABSENCES

If the absence of something is important, the teacher should point it out explicitly. (The classic example is the gap that is necessary between the columns of an arch.) Not only must all steps be shown, but important omissions must be transformed into concrete events in the demonstration.

ONE DISJUNCT PER LESSON

The teacher should not try to teach multiple differences at once. If a system can assume that only one important new feature exists, it can more easily identify this aspect of the example by eliminating the ones it already knows. If several new features are present it cannot so easily determine which are important.

CORRECTNESS

By default, the teacher's inputs are assumed correct. Violating this rule creates contradictions that a system may not be able to resolve. It can cause the search space of potential procedures to collapse, so that concepts and procedures are unlearnable. In programming by example, an error in the teacher's action trace may introduce a faulty

program step. When the error constitutes a difference between traces, the system will try to construct an inappropriate branch, whose condition may be unlearnable.

Systems described in this paper employ three techniques to support correctness: an "undo" facility, eager prediction, and private practice. A common convention for identifying an error is by explicitly "undoing" an action to restore the previous state. Eager prediction reduces user errors over multiple examples by pre-empting execution control as soon as possible (say, after one repeated event). Private practice allows the teacher to become familiar with the data and sequencing of actions.

FOCUS ATTENTION

An extension of van Lehn's "show all steps" felicity condition is that important aspects of a task should not only be shown but also distinguished by focusing the systems attention on them as key parts of the demonstration. A strong form of this is that all relevant objects appear either in the system's perceptual focus of attention *during the most recent action*, or in explicit focusing instructions given by the teacher. Adhering to this condition can be difficult, particularly when the constraint is the absence of some object or property. When selecting, inserting, or altering the properties of objects during a demonstration, the teacher may direct the system's attention to any constraints on the parameters and resulting values of the action.

Systems described in this paper adopt a variety of techniques to help teachers meet this condition. Automatic focusing mechanisms alleviate the need for teacher focusing. Teachers may be able to give an explicit "focus" instruction or make a declaration of objects in the focus of attention. Teachers can be asked to construct a relation when "easy" inference fails. Objects or constraints in the system's focus can be highlighted. Constraints can be made into first-class objects for direct manipulation by the teacher—for example, they might be shown on the screen as connecting lines, or box handles, enabling the teacher to manipulate them directly).

When a learning system chooses the wrong action, the teacher should indicate the criterion for choosing the right one (i.e. its context). This may be found in the current state of the actor's "world," in some past state of the world, or in the structure of the procedure. Violating this felicity condition forces the system to search a potentially huge state space. Some of the example systems help the teacher meet this condition by requiring that explicit

focusing instructions, which specify conditionals, precede the demonstration of the conditional action sequences.

MINIMIZE VARIATION

Usually there are many ways of accomplishing the same goal, and being able to apply a variety of approaches is an important form of generalization. If a variation is useful, then by definition there is some context that selects it as opposed to another (assuming that the important state variables of a task are observable by the learner). Other variations cannot be distinguished by context and should be avoided. The ideal modelling system would recognize free variation by observing cumulative changes of state across a sequence of actions, tracking references to variables, or otherwise reasoning about the semantics of actions. In general, however, these methods are computationally expensive and violating this rule will result in the creation of unnecessary conditional branches whose conditions may be impossible to induce.

To avert unnecessary variation in the teacher's demonstrations, a learning system can take control of execution after observing a minimal number of examples. Useful variations are thus taught only when the system fails to solve a problem. The convention of private practice also supports this aim by allowing teachers to settle on a consistent method.

5. Conclusions

Interactive computer users often find themselves repeatedly performing similar tasks. Frustratingly, improvements in interactive technology such as the desktop metaphor drive the user further from tools for automating simple, but variable, tasks. This paper has presented principles empirically derived from experience in building prototype learners in areas including technical drawing, text editing, office tasks, and robot assembly.

A teaching metaphor enables the user to demonstrate a task by performing it manually. The metaphor encourages human users' natural tendency to personify interactive systems and uses this to advantage in explaining the learner's abilities. It must meet the need of users' to attribute intentionality. Hidden features of a task must be made explicit so that the learner need not entertain, and search, all possible missing steps. Finally, the metaphor must allow the user to practice the task, so that a suitable, consistent procedure is developed for demonstration.

Our paradigm for learning includes procedural representations of tasks and constraints which make procedure induction tractable. Automatic or user-driven attention focusing mechanisms are essential in distinguishing relevant task characteristics for the learner. The performance component of an interactive system must normally be stripped down to a primitive set so that the learner's model of task performance is simple enough. Inductive inference from demonstrations to procedure descriptions is essential if the learner is to generalize its tasks. However, we stress the need to minimize induction because of its vast computational cost. An "undo" mechanism is necessary during both demonstration and automatic task performance. The interactive context of programming-by-example allows the user and system to form a feedback loop, thereby reducing the effects of noise.

Four illustrative systems were described. BASIL is an interactive drawing assistant that eagerly predicts actions for a user, and allows direct manipulation of task constraints. TELS helps a user teach repetitive text editing tasks. CLERK performs little induction but provides an appropriate metaphor for users to explicitly focus on important task aspects. ETAR induces robot assembly tasks from demonstration traces; it incorporates an automatic focusing mechanism and requires a minimum of additional help from the user.

Principles that govern the teacher–learner interaction can usefully be summarized as felicity conditions, which help the learner by guaranteeing more explicit, consistent information in demonstrations.

Systems that are programmed by human demonstrations can capitalize on interactive methods to boost the computational limitations of inductive inference. Ideally, proper interactive devices provide the human user with the opportunity to directly manipulate procedures, facilitating automation of tasks that can already be undertaken manually using direct manipulation. A combination of a teaching metaphor, procedurally-oriented learner, strong felicity conditions, and well-controlled induction can permit robust learning of tasks from examples.

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada and by Apple Computer Inc.

References

Andreae, J.H. and Cleary, J.G. (1976). A new mechanism for a brain, *Int J Man-Machine Studies*, **8**. 89–119.

Campbell, B. (1990). Office Clerk II. Project report, Computer Science Department, University of Calgary, April.

Carberry, S. (1990). *Plan recognition in natural language dialog*. Bradford Books, MIT Press, Cambridge, Massachusetts.

Cohen, P.R., Morgan, J. and Pollack, M.E. (eds.) (1990). *Intentions in communication*. Bradford Books, MIT Press, Cambridge, Massachusetts.

Cooley, M. (1987). *Architect or bee—the human price of technology*. Hogarth Press, London.

Dennett, D.C. (1981). *Brainstorms*. Harvester Press Brighton, Sussex

Dennett, D.C. (1987). *The intentional stance*. MIT Press, Cambridge, Massachusetts.

Ellman, T. (1989). Explanation-based learning: a survey of programs and perspectives, *Computing Surveys*, **21**(2): 163–221; June.

Fisher, D. (1987). Knowledge acquisition via conceptual clustering, *Machine Learning*, **2**. 139–172.

Halbert, D.C. (1981). An example of programming by example. Technical Report, Xerox PARC (Office Products Division), Palo Alto, CA.

Halbert, D.C. (1984). Programming by example. Technical Report, Xerox PARC (Office Products Division), Palo Alto, CA; December.

Heise, R. (1989). *Demonstration instead of programming,* MSc Thesis, Department of Computer Science, University of Calgary.

Heise, R. and MacDonald, B.A. (1990). Robot program construction from examples. Proceedings of AI / CS '89, British Computer Society Workshops, Springer–Verlag.

Heise, R. and MacDonald, B.A. (1991). Dynamic bias is necessary in real world learners. Research Report 91/428/12, Department of Computer Science, University of Calgary.

Laird, J.E., Rosenbloom, P.S. and Newell, A. (1986). Chunking in SOAR: the anatomy of a general learning mechanism, *Machine Learning*, **1**(1): 11–46.

Maulsby, D.L. (1988). Inducing procedures interactively. MSc thesis. Research Report 88/335/47. Department of Computer Science, University of Calgary.

Maulsby, D.L., Kittlitz, K.A. and Witten, I.H. (1989). Metamouse: specifying graphical procedures by example, *Computer Graphics*, **23**(3): 127–136.

Maulsby, D.L., Witten, I.H., Kittlitz, K.A. and Franceschin, V.G. (1990). Inferring graphical procedures: the compleat Metamouse. Research Report 90/388/12, Department of Computer Science, University of Calgary.

MacDonald, B.A. and Witten, I.H. (1987). Programming computer controlled systems by non-experts. *Proc IEEE Systems, Man and Cybernetics Annual Conference*, Virginia, pp. 432–437; October.

Michalski, R.S. (1983). A theory and methodology of inductive learning, *Artificial Intelligence*, **20**. 111–161.

Minsky, M. (1967). *Computation: finite and infinite machines*. Prentice Hall, Englewood Cliffs, NJ.

Mitchell, T.M. (1982). Generalization as search, *Artificial Intelligence*, **18**. 203–226.

Mo, D.H. and Witten, I.H. (1990). Learning text editing tasks from examples: a procedural approach. Research Report 90/394/18, Department of Computer Science, University of Calgary.

Nix, R. (1985). Editing by example, *ACM Trans Programming Languages and Systems*, **7**(4): 600–621.

Phan, T.H. and Witten, I.H. (1990). Accelerating search in function induction, *J Experimental and Theoretical Artificial Intelligence*, **2**(2): 131–150; April–June.

Runciman, C. and Thimbleby, H.W. (1986). Equal opportunity interactive systems, *Int J Man–Machine Studies*, **25**(4): 439–451.

Smith, D.C. (1975). Pygmalion: A computer program to model and stimulate creative thought. PhD Thesis, Stanford University.

Van Lehn, K. (1990). *Mind bugs: the origins of procedural misconceptions*. Bradford Books, MIT Press, Cambridge, Massachusetts.

Winston, P.H. (1975). Learning structural descriptions from examples. In P.H.Winston (ed.) *The psychology of computer vision*. McGraw Hill, New York, NY.

Captions for Tables and Figures

Primitive actions		**Parameter**	**Positional information**
	insert	<text>	
	locate		<attributes>
	select	<text>	<attributes>
	delete		
Attributes		**Details**	
	context	before, after	
	lexical	beginning middle of end	file paragraph line
	relative distance	characters words lines paragraphs	

Table 1 Summary of actions and attributes in the text editing model

(a)

Mail	
`Close`	Close top level window
`Open Mbox`	Open/display a selected mailbox
`Delete Mbox`	Remove a selected (and empty) mailbox
`Send Message`	Compose a message to send
`Quit`	Quit the mail system

File	
`Close`	Close the mailbox window
`Open Msg`	Open/display the selected mail message
`Show Mbox List`	Display the top level window
`Send Message`	Compose a message to send
`Quit`	Quit the mail system

(b)

Edit mailbox	
`Cut`	Remove the selected message and save it in a buffer
`Copy`	Save the selected message in a buffer
`Paste`	Insert the buffer message into the mailbox
`Clear`	Delete the selected message

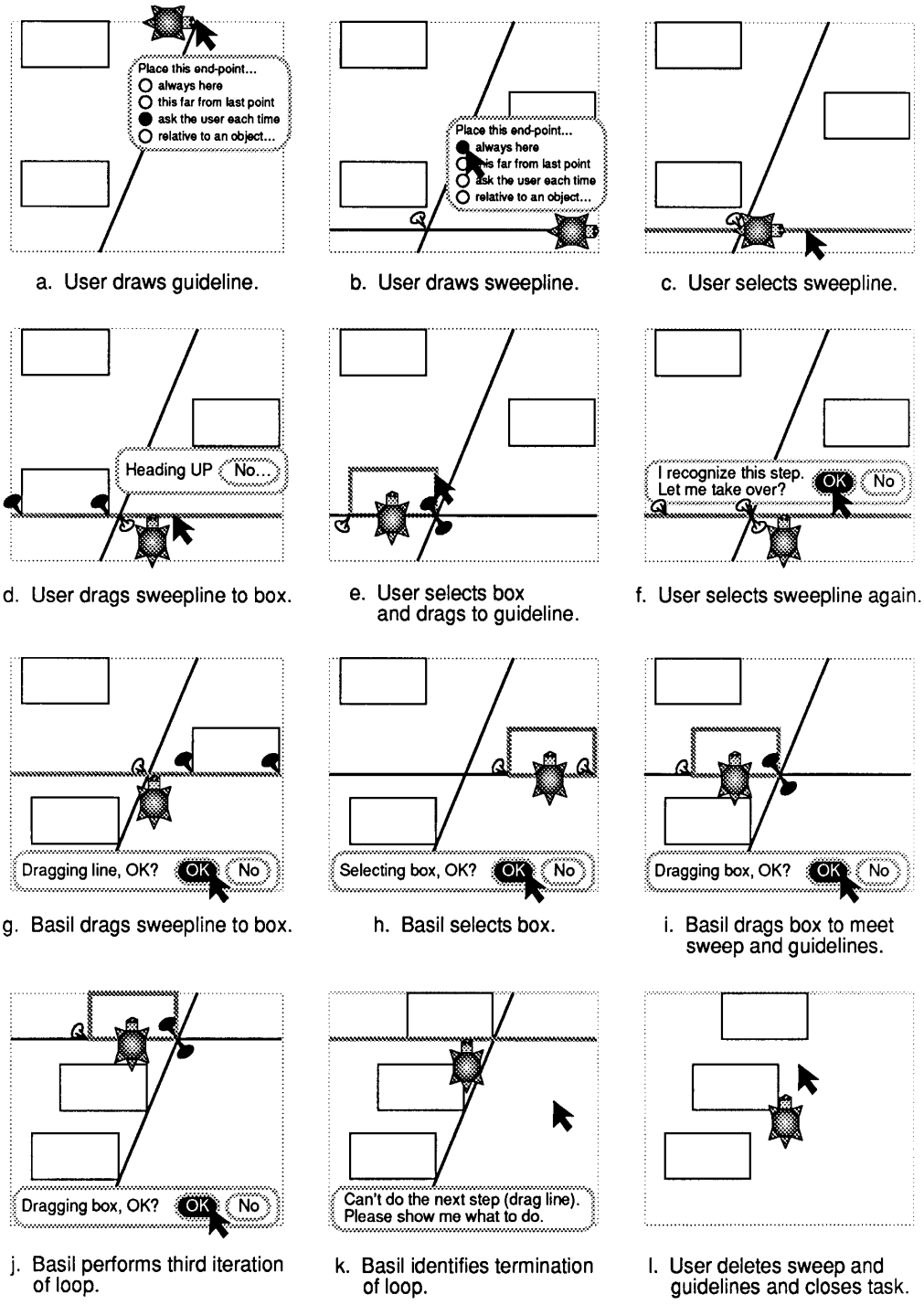
(c)

Message	
`Close`	Close the message window
`Show Mbox List`	Display the top level window
`Show Mail Box`	Display the parent mailbox window
`Send Message`	Compose a message to send
`Quit`	Quit the mail system

Table 2 Menus for the CLERK
(a) Top-level menu
(b) Mailbox-level menus
(c) Message-level menu

Felicity condition	Description
Show all steps	The teacher must not omit steps of the task during demonstration
Show absences	Important absences must be explicitly pointed out
One disjunct per lesson	Only one new feature should be introduced per lesson (similar to Winston's "near-miss")
Correctness	Negative instances, errors and mistakes must be explicitly identified. The teacher must not mislead the learner
Focus attention	Important aspects of a task should be distinguished
Minimize variation	The teacher should be consistent about the order of events and choice of operators in repeated parts of the task

Table 3 Felicity conditions



a. User draws guideline.
b. User draws sweepline.
c. User selects sweepline.
d. User drags sweepline to box.
e. User selects box and drags to guideline.
f. User selects sweepline again.
g. Basil drags sweepline to box.
h. Basil selects box.
i. Basil drags box to meet sweep and guidelines.
j. Basil performs third iteration of loop.
k. Basil identifies termination of loop.
l. User deletes sweep and guidelines and closes task.

Figure 1 Working with BASIL to align a set of boxes

Input

```
John Bix, 2416 22 St., N.W., Calgary, T2M 3Y7. 284-4983
Tom Bryce, Suite 1, 2741 Banff Blvd., N.W., Calgary, T2L 1J4. 229-4567
Brent Little, 2429 Cherokee Dr., N.W., Calgary, T2L 2J6. 289-5678
Mike Hermann, 3604 Centre Street, N.W., Calgary, T2M 3X7. 234-0001
Helen Binnie, 2416 22 St., Vancouver, E2D R4T. (405)220-6578
Mark Willianms, 456 45Ave., S.E., London, F6E Y3R, (678)234-9876
Gorden Scott, Apt. 201, 3023 Blakiston Dr., N.W., Calgary, T2L 1L7. 289-8880
Phil Gee, 1124 Brentwood Dr., N.W., Calgary, T2L 1L4. 286-7680
```

Output

```
John Bix,
2416 22 St., N.W.,
Calgary,
T2M 3Y7.

Tom Bryce,
Suite 1,
2741 Banff Blvd., N.W.,
Calgary,
T2L 1J4.

Brent Little,
2429 Cherokee Dr., N.W.,
Calgary,
T2L 2J6.

Mike Hermann,
3604 Centre Street, N.W.,
Calgary,
T2M 3X7.

Helen Binnie,
2416 22 St.,
Vancouver,
E2D R4T.

Mark Willianms,
456 45Ave., S.E.,
London,
F6E Y3R.

Gorden Scott,
Apt. 201,
3023 Blakiston Dr., N.W.,
Calgary,
T2L 1L7.

Phil Gee,
1124 Brentwood Dr., N.W.,
Calgary,
T2L 1L4.
```

Figure 2 Example task for TELS

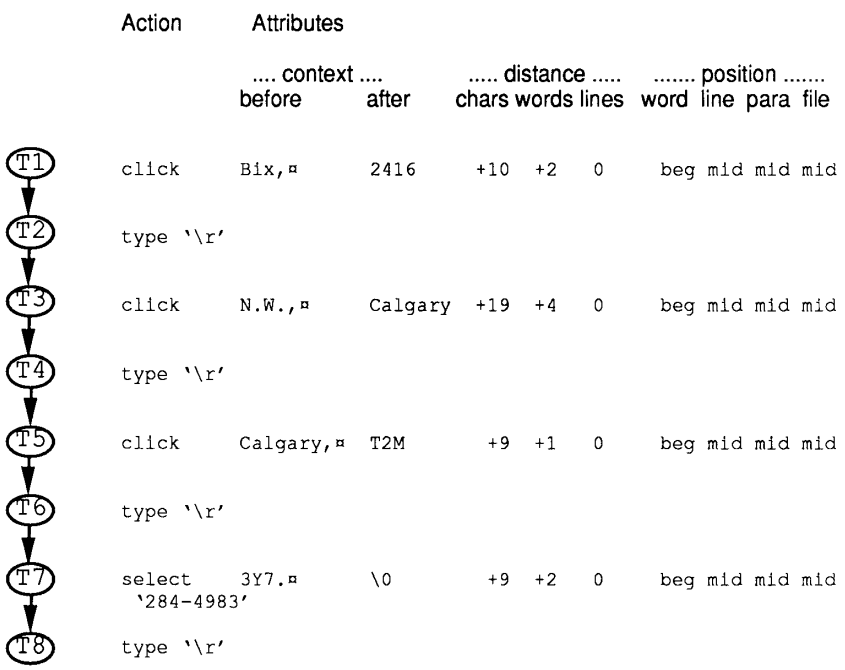


Figure 3 Trace for first block of example task

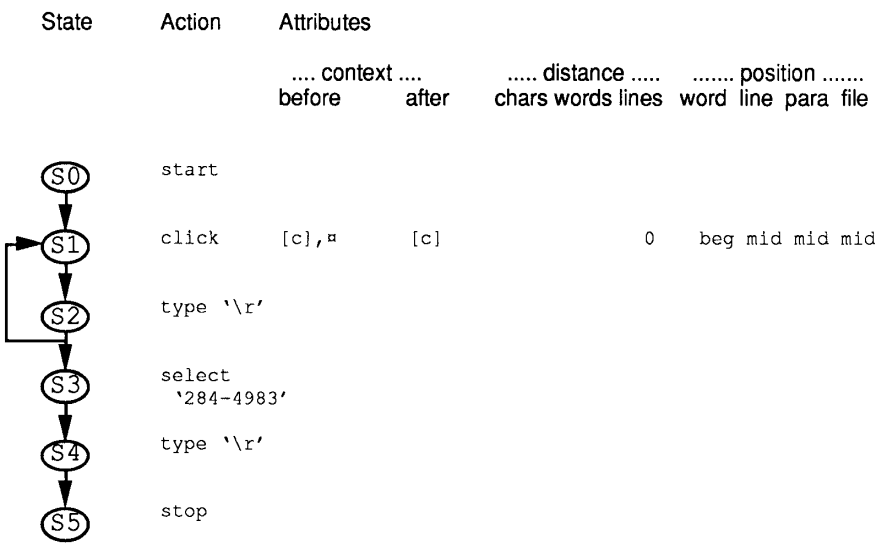


Figure 4 Program created from the trace of Figure 3

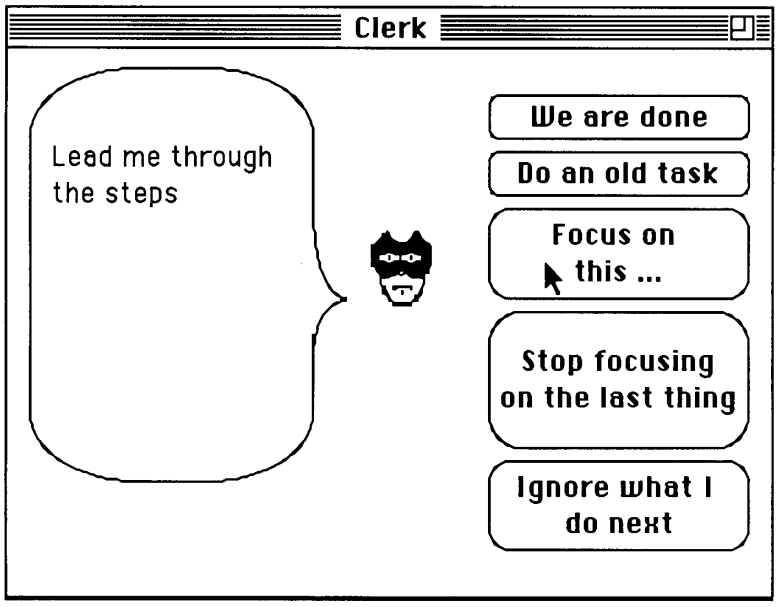
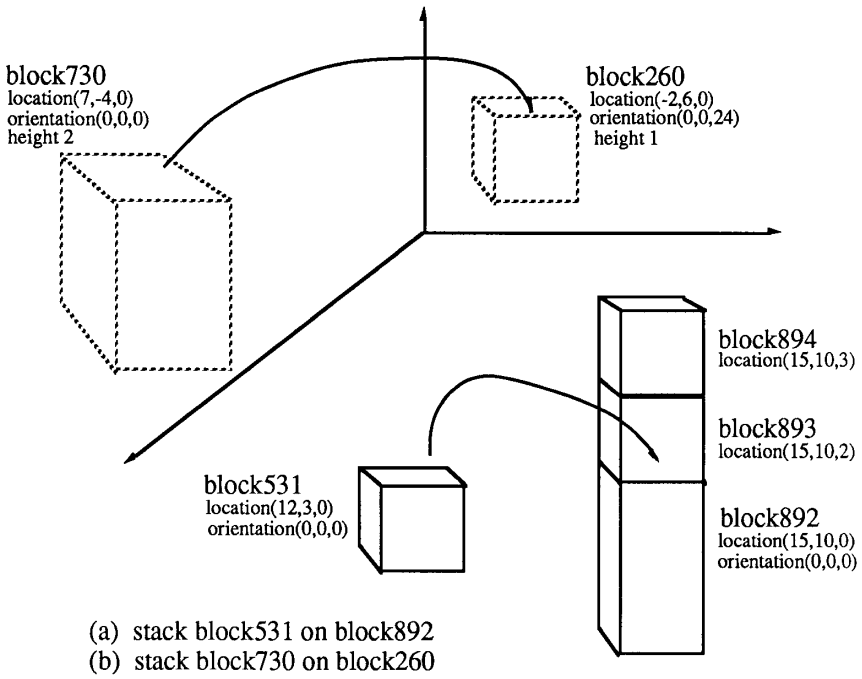


Figure 5 The CLERK metaphor



(a) stack block531 on block892
(b) stack block730 on block260

Figure 6 Examples of a stacking task

stack block1 onto **block2**

UNTIL (block2 under nothing) — remove any blocks from block2

focus$_1$ = highest object on top of block2

approach and pick up top block

moveto(15, 10, $z_{focus1} + height_{focus1} + approachDist_{robot}$, ?, ?, ?)
rotate(0, 0, 0)
translate(15, 10, $z_{focus1} + height_{focus1}$)
grip(*close*)
translate(15, 10, $z_{focus1} + height_{focus1} + approachDist_{robot}$)

move it away

moveto(?, ?, 1, ?, 0, 0)
grip(*open*)
translate(?, ?, 2)

pick up block1

focus$_1$ = block1
moveto(x_{block1}, y_{block1}, $height_{block1} + approachDist_{robot}$, ?, ?, ?)
rotate($roll_{block1}$, 0, 0)
translate(x_{block1}, y_{block1}, $height_{block1}$)
grip(*close*)
translate(x_{block1}, y_{block1}, $height_{block1} + approachDist_{robot}$)

place block1 on top of block2

focus$_{1,2}$ = block2, block1
moveto(x_{block2}, y_{block2}, $height_{block1} + height_{block2} + approachDist_{robot}$, ?, ?, ?)
rotate($roll_{block2}$, 0, 0)
translate(x_{block2}, y_{block2}, $height_{block1} + height_{block2}$)
grip(*open*)
translate(x_{block2}, y_{block2}, $height_{block1} + height_{block2} + approachDist_{robot}$)

Figure 7 Program generated from the stacking examples

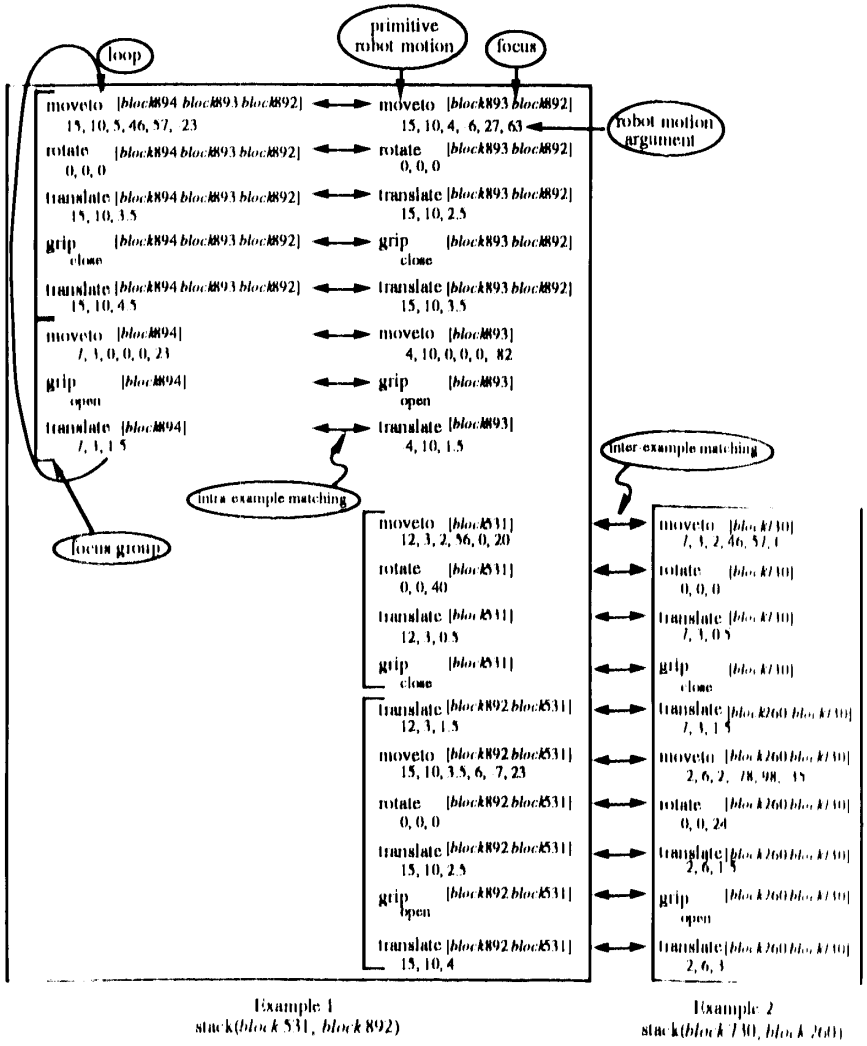
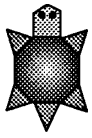
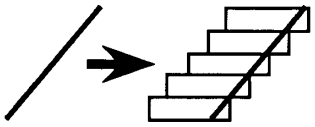


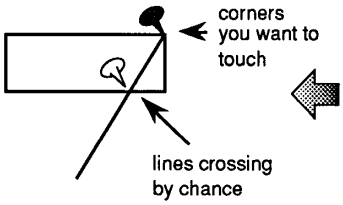
Figure 8 Stacking task generalization



My name is Basil and as you can see I'm a turtle. I'm here to help you draw. You teach me repetitive and finicky tasks — like evenly spacing a row of boxes or reconnecting a group of lines when one is moved. I learn by acting as your apprentice — I follow you till I think I know what you'll do next, then I try to do it for you. If I guessed wrong, give me a gentle tap so I'll undo it and wait for you to show me what's right.

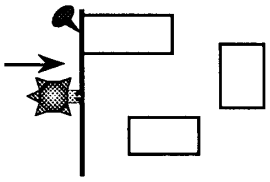
I can draw lines and boxes and carry them by their handles (which I grasp with my jaws). You can teach me to make tools for a task — for example to build a staircase of boxes, use a diagonal line. When done with a tool, just delete it.

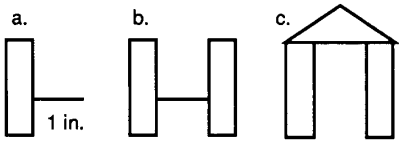




Because I crawl around a video screen I see things almost edge on — that makes it hard to spot patterns. Instead I work mainly by feel. I remember how things fit together, which parts — corners, centers and lines — are connected. When you show me an edit, I put black tacks where I think you want objects to touch, and leave white tacks beside other touches I don't think matter. If you disagree, click on a tack and it changes color — as if you'd pushed it in or pulled it out.

I'm touch-sensitive only at my snout but I can sense contact between what I'm grasping and anything else it touches. If I have to find, say a box, I set off in the general direction you've taught me (up, down, left, right) until I bump into one. It doesn't have to be dead ahead. If you want me to be more selective, give me a tool to carry and teach me to move until it touches something. I'll remember exactly how it touched.





Now, this is very important. I can't learn directly how things should not touch — I mean how they should be separated. You should give me tools to separate them. Say you're drawing an arch and want the columns an inch apart. Draw a one-inch horizontal line, then place the columns at either end of it, as shown on the left.

When you want to teach me, choose "Time for a lesson!" from the Basil menu, and "End of lesson" when you're done. If you want to interrupt the lesson for something else, like working out a method before showing me, just say "Take a nap" — then "Wake up, Basil!" when you're ready. When you don't agree with what I do, just tap me and I'll undo it. In any case when I don't know what to do next, I'll ask you to show me.

So in general you teach me by doing the task yourself, using some extra tools to help me see patterns by feel. As soon as I can predict what to do, I'll take over the task, but I'm always ready to learn something new.

Figure 9 Description of BASIL given to users