CONTENTS

I INTRODUCTION 1
- A Intelligence . . . 1
- B Autonomy . . . 1

II INSTRUCTABILITY 2
- A Instructability—a neglected area . . . 2
- B Requirements for instructability . . . 2

III LEARNING IN THE USER INTERFACE 3
- A Teaching *vs* Programming . . . 3
- B The programming problem . . . 3
- C Constructs . . . 4
- D The user interface . . . 4

IV TECHNIQUES FOR LEARNING FROM EXAMPLES 5
- A Representation . . . 5
- B Bias . . . 7
  - 1 Background knowledge . . . 7
  - 2 Constraining induction . . . 8

V HOW THE TEACHER INTERACTS 9
- A Theoretical foundations . . . 9
- B Felicity conditions . . . 10

VI COMPETITIVE LEARNING ENVIRONMENTS 10

VII EXAMPLES OF INSTRUCTABLE SYSTEMS 11
- A Online verbal task editing . . . 11
  - 1 Editing led sequences . . . 11
  - 2 The Verbal Correction Scheme . . . 12
  - 3 Experimental work . . . 12
  - 4 Learning coordinate transforms by example . . . 12
  - 5 Learning in frames using function induction . . . 13
- B An automated office clerk . . . 13
  - 1 Sequences . . . 14
  - 2 Selection . . . 14
  - 3 Variables, Iteration, Data Structures, Editing must be acquirable by the clerk . . . 14
  - 4 A demonstration of an automated clerk . . . 15
  - 5 Constructive graphics editor . . . 15

VIII CONCLUSION 15

I INTRODUCTION

It will be essential for robots and other artificial systems to acquire knowledge and skills while interacting in the real world, without the help of programmers and without the benefit of special knowledge priming. If new tasks are to be assimilated and changing conditions accommodated outside carefully controlled and supervised environments, then systems must learn while immersed in the ongoing activity and richness of the world.

This paper examines the notion of *instructability* as a way of achieving a significant degree of autonomy. We presuppose the existence of more basic forms of autonomy, such as power supply, communication, and computational equipment. But we do not address the question of full autonomy in the form of self-motivation. Indeed, a central tenet is that in general, self-motivation will not even be desirable (although we acknowledge that under certain circumstances—for example in deep space and under the sea—it may be essential). Instructability cannot be provided merely by an internal "learning module," but must be carefully engineered into the human-machine interface and pervade the system as a whole. Autonomous learning occurs in the normal mode of execution, in the "performance" interface, rather than through a privileged communication channel designed specially for programmers.

A Intelligence

Our aim is to build autonomous intelligent learning machines. While "intelligence" defies formal description [96], most would agree that such machines must acquire and perform tasks under a variety of realistic conditions, interacting at times with humans and each other. Machine intelligence will involve complex inter-communication, dextrous manipulation, the ability to represent, acquire and manipulate real-world knowledge, handling the rich information available from sophisticated sensors, recognizing and coping with errors and other unexpected conditions, self-preservation (such as by robustness, fault-tolerance, graceful degradation, self-maintenance, and basic built-in survival goals), and the ability to choose one's own goals. Intelligent robots will need versatile, robust bodies, capable of acting and sensing in realistic environments. While present sensors and effectors fall far short of the mark, the state of the art is advancing rapidly [52]. A more fundamental impediment is our limited understanding of how to employ robot sensors and effectors to enable them to learn to perform tasks [15]. This paper addresses the difficult problem of learning in the real world.

B Autonomy

"Autonomy" is as slippery a concept as intelligence. One can identify several components or degrees, summarized in Figure 1. A fully autonomous system would include them all, an instructable one all but autonomy of motivation.

Dictionary definitions of autonomy emphasize free will [22], suggesting that we might concentrate on how a machine could set and achieve its own goals [102]. A secondary concern is survival, so that free will can continue to be exercised in difficult and hostile real world situations. However, we argue that for many artificial autonomous systems, self-motivation would be counter-productive, as human users will want the exclusive right to set high-level goals. This paper complements work on performance autonomy by examining autonomy in knowledge and skill acquisition.

Instructability endows machines with the ability to act without reprogramming (i.e. autonomously) in new situations and tasks. Existing machines are unable to acquire and perform a range of tasks under a wide variety of circumstances. To make them instructable requires more than adding a learning facility (even though that in itself is a demanding enough undertaking). Furnishing systems with a "teach" button is uncomfortably like adding an ultra-high-level programming capability. Rather, the ultimate goal of instructability is the graceful integration of task acquisition and task performance. Instructability, in this sense, will be essential to a fully autonomous machine with self-motivation and the ability to survive (although the converse is not the case).

The next section introduces instructability, citing previous work and identifying the main aspects: user interface, inductive learning techniques, teaching constraints, and the potentially competitive nature of learning in lifelike multi-agent settings. Subsequent sections detail each of these aspects individually, while the final section illustrates instructable systems with examples from current research.

II INSTRUCTABILITY

A *Instructability—a neglected area*

Current research in autonomy concentrates on the practical aspects of task planning, sensor interpretation, terrain modeling, dynamics and control, specialized computer architecture, algorithms for concurrent computation, path planning and navigation, and coordinated manipulation [19,20,24,37,65,75]. These relate to the performance of the system—its ability to carry out prescribed tasks in diverse and unexpected situations—rather than to the acquisition of new capabilities.

On the other hand the technique of physically leading a robot "by the hand" through a task—although severely limited in current implementations—fulfils the basic requirements for instructability quite well. As the teacher manipulates the arm to perform a task, movements are recorded for later playback. Teaching is accomplished through the normal interface with the system—its arm—with no need for either special teacher training or priming with domain-specific knowledge. Leading is the most common method used in industry to teach robots, but only when the task is to be repeated unerringly in exactly the same environment [52,54].

There are more general techniques for programming by example [7,13,14,41,68,73,74,82,86,101], but the technology is not yet ready for practical use [99]. One demonstration system enabled a user to teach a mobile robot to move around arbitrary convex obstacles in a 2D environment [13,14]. Current research seeks to improve robot programming languages so that the powerful abstractions of programming may be combined with the naturalness of leading. Our work also aims to enhance user interaction in order that systems can be taught to handle real-world complexities, but differs in tactics by beginning with the leading method and striving to enrich its power. In this way we extend performance autonomy to instructability.

Acquisition and learning of complex tasks is hardly mentioned in work on intelligent robots, which instead concentrates on the technology of sensors and effectors, planning systems, learning in the control mechanisms, learning of particular predefined parts of a task, and high-level programming languages [21,24,37,65,67,79,94]. For example [19] is concerned with autonomy in mobility and robot control systems, [20] concentrates on technological aspects of autonomy such as computation and performance, [75] addresses the learning of navigation information for a predetermined task, and [91] is concerned with robot programming.

Some recognise the importance of instruction, but generally in the guise of programming—"A robot is only as useful as what it can be programmed to do" [52]—although [46] discusses a 3D sensor system for teaching and [88] emphasize teaching rather than programming. Some recognise the primacy of learning and aim at eventual machine autonomy and learning in the real world [5,6,8,9,10,11,12,55,57,58,59]. However, in general the literature devotes little space to the question of instructing robots. Instead it is implicitly assumed that very high level programming languages, planning and reasoning techniques will eventually provide adequate solutions.

Why has autonomy in learning been neglected? Perhaps because it is just too difficult. We still have little idea of how humans interact with their environment, especially when it changes. There is also a common misconception that sophisticated autonomous machines will not need to learn. This is clearly false: there will always be new knowledge and skills to acquire. Despite a recent revival of interest, machine learning techniques have not matured enough to contribute to practical learning in the user interface, although their importance has been noted ([5,6,13,14,40,41,59,62,63,86,101]).

B *Requirements for instructability*

Figure 2 summarizes the four main areas of concern. A teacher interacts with an instructable system through its user interface, so the important features include the teacher's abilities, the nature of the interface, and the system's learning methods. Other agents may make the environment competitive. New knowledge and skills must be acquired through the normal, natural channels of communication with the user, not in any special programmer's or teacher's interface. An instructable system comprises machine learning techniques properly matched with a user interface. The interface provides for the normal performance of the system, so learning must employ actual performance data, that is, examples. General felicity conditions [90] must be met to overcome the intractability of inductive learning. The environment may not always be benign and so the system must have control over whether it learns or performs.

III LEARNING IN THE USER INTERFACE

If autonomous systems are to learn in the world outside, they must be able to learn from non-programming, casual, non-expert users. Therefore they should be taught, not programmed. Anything that is learned must be communicated through the machine's regular interface to the world. This section discusses teaching and programming, the programming problem and why it will not go away, shortcomings of the popular fixed sequence "leading" method of robot control "by example," the constructs that a more sophisticated instructable system must be able to acquire, and the user interface. Later sections give examples of systems in which a casual user is able to achieve some measure of improvement over the "leading method," which represents the current state of the art.

A *Teaching* vs *Programming*

A system's practical utility depends critically on how well the teacher is able to communicate new knowledge and tasks. Teaching differs from programming in that a teacher does not have a detailed formal model of the learner. Teachers are always uncertain about exactly how their pupils learn. A programmer normally expects to know precisely how his instructions are going to be interpreted. In Dennett's [26,27] terms, a teacher adopts the *intentional stance* while the programmer adopts the *design stance.* In the former case intentions are ascribed to the learner, and its behavior is modeled in terms of those intentions; while in the latter a detailed formal model of the system is available which reflects its behavior *by design.* In the design stance, if by any chance there is a mismatch between the formal model and the system's actual behavior, this indicates a malfunctioning; the design has not been realized correctly at the physical level. In the intentional stance, mismatches do not indicate any fault in the learner but are corrected by the teacher modifying his model.

In principle, a teacher should need to know nothing about the design or physical implementation of the learner, nor about the representation it uses for concepts; although good teachers certainly make and use models at the intentional level. Learning even in the absence of a good teacher is an important skill for human and machine learners, enabling them to acquire knowledge on their own. But as human society has found, unassisted learning can be very difficult and is often impractical.

Teachers do not expect to give explicit analytic descriptions to learners in the way that programmers create explicit analytic specifications in their programs. Rather, they use pragmatically established, intention-based techniques, coupled with a carefully selected set of examples, to engender understanding in the learner. Furthermore, domain experts, robot users, and others who expect to employ concept learning techniques for transferring knowledge and skills to machines, are not necessarily skilled teachers. They are unlikely to have a clear analytical understanding of the task domain; otherwise more explicit methods of knowledge transfer would have proven more effective. In general, people find it difficult to translate their own expertise into explicit descriptions. Consequently instructable systems should be able to work with the kind of examples that a user finds it natural to provide, ordered in a way that seems natural to him. For example, welding robots allow teachers to demonstrate tasks *by using the robot's welding arm to make a weld.* This is the most natural mode of operation for a welder. The user interface is more than the modes and devices of interaction; it is also the mental and physical reactions engendered in the user.

B *The programming problem*

The traditional way of specifying procedures to computer systems is through 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. 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 an office worker. In many ways these command languages are *more* difficult for casual users 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 ability to specify command procedures in a 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 casual users (see [4,25] for opposing views). To program in a conventional language it is necessary 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.

Two popular methods of teaching fixed sequences to computer controllers for robots are leading, where the robot is physically led through the sequence, and guiding, where the robot is remotely guided through the sequence [3,17,18,42,51,54,64,88]. The sequence is always recorded and repeated as taught.[1] Thus even without the benefit of programming expertise, a user can specify a fixed sequence to be automated. A similar method is used by some text editors, such as EMACS [36], allowing users to record a sequence of key pushes for later repetition.

However, for more complex tasks conditionals, iterations, hierarchical control, and structured data will be needed, along with editing facilities to fix mistakes and make updates. Casual users will be unable to supply these constructs explicitly; nor will they have the editing skill required. It is hard for non-programmers to articulate such constructs, for at least five reasons.

Specification: Existing systems need a description *in a programming language.* A rough English outline of the task is insufficient [18,51,54,64]. Non-programmers do not find it easy to translate their natural knowledge of a task into an algorithm, nor to communicate the algorithm in a programming language. Their communication skills are interpersonal. A spray-painter knows how to spray-paint, and perhaps how to describe it to a human, but not how to write a spray-painting program.

Irrelevant detail: Explicit programming languages are powerful vehicles for expressing procedures, but are unsuited to casual users because of the need to learn tricky details of computer syntax, semantics, and internal technical characteristics [101].

Inherent difficulty: Programming is difficult even for expert programmers. The complexity of declarations about our 3D world may explode even in high-level languages [42]. Multi-dimensional simultaneous robot programming may be unmanageable for humans [3].

Sensory information: Robots lack the ability to use sensory information [80]—as do other computer controlled systems—so it is difficult to teach them sensory tasks. Existing robots are not adaptable [72] and can perform manual tasks only in a limited range of situations [17]. Advanced robot systems are limited to specific tasks and their associated environments, and may be difficult to extend [92].

Expertise: A robot programmer must be expert at robotics, algorithm design and programming, *as well as at the task to be taught* [51,54,64].

C Constructs

Table 1 shows the major constructs that an instructable system must be able to acquire from a casual user. Conditionals allow different actions in different situations. Iteration permits part of a task to be repeated until a specified condition is met. Nesting embeds one control structure inside another.

Data structures allow information to be stored and accessed conveniently, and might also be nested. Those shown in Table 1 are simple PASCAL-like records. More complex descriptions will sometimes be required, for example ones in which different parts are related. However, simple structures are adequate for many applications. Variables allow different items to be processed by the same procedure.

Table 1 also shows a simple example procedure which employs many of these constructs. It opens the user's electronic mail and sorts items into folders by sender. 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.

D The user interface

The user interface must be suitable for casual, non-expert users. Otherwise not only will learning be impeded, but so will the performance of the system in its other interactions.

The most effective interfaces to office systems provide pictorial representations of real office objects (files, folders, documents, text, diagrams, and the like) which can be manipulated directly on the screen [84]. This leads naturally to learnability, robustness, and general simplicity, [47] enhancing the usability of an office information system through *direct manipulation* [98]. The computer system provides a *metaphor* of the office environment, making it easy for

users to execute procedures manually, step by step [47,84]. Direct manipulation provides a robust interface suited to the casual user, with the advantages of user enthusiasm, reduced learning effort, a predictable system, and a concrete (not abstract) interface. Users can always see both their current work in its final form, and all options that are available to them [98].

However, direct manipulation does not help communicate sequences of actions [69]; indeed, its very nature makes it difficult for users to specify procedures [103]. Although a fixed sequence of operations could be recorded for later playback, generalization, conditionals, controlled iteration, complex data structures, editing and debugging, are not provided. These are abstractions, not visualizable operations on concrete objects, and are specified to humans using natural language constructions rather than mere examples. The manipulation is no longer *direct.*

Traditional programming environments distinguish between the user's and programmer's interfaces.[2] When this distinction blurs the user generally has to move into the programmer's realm. (For example, UNIX users become system programmers by improving their repertoire of commands and their understanding of how the system works.) Our aim is quite the opposite, to bring programming into the user's realm rather than *vice versa.*

As section IV reveals, machine learning techniques are immature and only simple methods can be employed in practice. However, sophisticated learning systems can be produced by combining a suitable interface with existing techniques for learning from examples, as demonstrated in section VII. Halbert's [40,41] programming-by-example scheme allows users to program in a direct-manipulation interface, including conditional branches, variables, and so on. Simple techniques of generalization are blended with the interface. For example, the user specifies variable ranges explicitly, but appropriately, on request from the system. Whenever an item is selected on the desktop, the user is asked to indicate whether this means any item in that desk position, any item in that folder, or an item named by the user. Smith [86] employs similar methods to enable users to program in a graphical interface, as does a constructive graphics editor in a graphical instruction environment [62,63].

IV TECHNIQUES FOR LEARNING FROM EXAMPLES

Learning techniques have been reported which claim to learn by *analogy, being told, debugging, discovery, doing, examples, experimentation, exploration, imitation, instruction, observation, rote,* and *taking advice.* However, the apparent richness and variety of these approaches may give a misleading impression of a field teeming with fruitful techniques, with a selection of well-defined methods for tackling any given problem. Currently, the most important method is concept learning, the acquisition of structural descriptions from examples of what is being described. In recent years the study of machine learning has successfully elucidated some basic techniques for generalizing concrete examples to more abstract descriptions. These include heuristics for generalizing particular data types, candidate-elimination algorithms, methods for generating decision-trees and rule-sets, back propagation of constraints through an explanation tree, function induction, and synthesis of procedures from execution traces [60].

In this paper we emphasize the potential to learn new tasks, and so *inductive* (rather than deductive) learning is paramount. What is learned in deductive learning from an example or examples could have been deduced from primed—and usually domain-specific—background knowledge [60]. An example is used to guide the deductive process so that inappropriate descriptions are not considered.

While the notion of learning from examples may conjure up an alluring promise of the magic of human intelligence, this is misguided, and the techniques are rooted firmly in the reality of practical algorithms. The essence is a constrained search of what is invariably an astronomically large space of possible descriptions. There are two important issues for concept learning techniques: the representation of concepts and examples, and the way the search is biased to avoid inappropriate concepts.

A Representation

We seek suitable general forms of representation that allow examples and induced concepts to be expressed. Separate (but related) representations are required for examples and concepts. No one representation encompasses the broad application of concept learning techniques. Instead we propose three: *first order predicate calculus, expressions composed of functions,* and *procedures* [60]. These reflect fundamental formulations of computing which have been realized in logic, functional and imperative programming styles. Although equivalent in expressive power, the different representations are more or less appropriate for particular concept learning problems, depending on the nature of the examples, background knowledge, the way the complexity of concepts is measured, and the style of

interaction with the teacher. For example, decision trees are naturally represented as logical expressions, polynomials as functions and robot tasks as procedures. Functional representations incorporate the powerful mathematics available for numbers. Procedures embody the notions of sequencing, side effects and determinism normally required in executing tasks.

Logic Many learning methods apply to examples that can be expressed as vectors of attributes in a form equivalent to *propositional calculus*. This represents logical statements using predicates on constant terms, with connectives for disjunction, conjunction, negation and implication [71]. The values an attribute can assume may be *nominal*, *linear*, or *tree-structured* [66]. A *nominal* attribute is one whose values form a set with no further structure; for example the set of primary colors. A *linear* attribute is one whose values are totally ordered; for example natural numbers. Ranges of values may be employed in descriptions. A *tree-structured* attribute is one whose values are ordered hierarchically. Only values associated with leaf nodes are observable in actual examples; concept descriptions, however, can employ internal node names where necessary. For example, the attribute *shape* might be hierarchical, having subsets such as *polygon* and *oval*.

Attribute vectors (propositional calculus) are not powerful enough to describe situations where each example comprises a "scene" containing several objects. First order *predicate calculus* allows variable terms in logical statements, and quantification over those variables [71]. The classic example is Winston's [95] *arch* concept, defined as three blocks, two having the attributes required of columns and the third having those required of a lintel, with the columns supporting the lintel and set apart from one another. Objects are characterized by their attributes, and pairwise (or more complex) relations may exist between them. This means that variables must be introduced to stand for objects in various relations. Such relations can be described by predicates which, like attributes, may be nominal, linear, or tree-structured. For example, a predicate `relative-position` might relate two objects with a pair of linear values `x-distance, y-distance`.

Functions Typical functional expressions include many natural laws, as well as relationships between quantities sensed by a robot and parameters of a subsequent movement. Functional representations are appropriate for nested and recursive numeric or non-numeric expressions. Any functional relationship can be represented in logic, but in a framework for induction, it is preferable to treat functional expressions separately and omit the additional explicit quantifiers required. An important difference is that functional representations of concepts must be single-valued, while logical relations need not be; this greatly affects the search space involved.

A more suitable form of representation may be the lambda calculus, or some incarnation of it in pure LISP or other functional programming languages [33,44,77]. These are suitable representations in which to expressly prohibit aspects that distinguish functional from procedural representations in the framework, namely side effects (e.g. variable assignment) and reliance on sequential execution. Work on programming language semantics, also partly based on lambda calculus and often coupled with functional programming languages (e.g. [33]), may suggest appropriate forms of representation. Existing function induction systems are specially designed for particular domains with little attention to more general forms.

For example, the synthesis of LISP functions from a few example input-output pairs of list structures has been extensively studied [87]. Quite restrictive assumptions are placed on the examples. All atoms in the input list structure must be unique; every atom that occurs in the output must also occur in the input; and there must be a total order relation corresponding to "simplicity" on the inputs. Traces of possible functions are formed directly from the input-output pairs and used to create a suitable recursive function. Simple but powerful forms for possible functions are assumed, comprising an overall LISP conditional with function calls and basic list primitives forming the branches of the conditional. Recurrence relations among the input-output pairs enable recursive calls to be induced.

Procedures Typical concepts in this category include robot procedures for assembly, welding, *etc*, and standard office procedures such as sorting mail [57,59,101]. The formalism is suitable for representing sequential execution where side effects, such as variable assignment and potentially irreversible outputs like robot movements, make it vital to execute the procedure in the correct order. To describe a procedural language formally, a binding environment model of execution is needed instead of the simpler substitution model which suffices for purely functional representations [1]. Representations must be deterministic to be useful procedural concepts. Each of these decisions—order-dependence, determinism—have massive effects on the search space covered by an inductive search. Work on imperative programming language semantics (e.g. [35]) may provide ideas for simple, expressively adequate, consistent representations.

For example, NODDY acquires robot procedures, complete with control information which is not explicitly present in the examples [13,14]. 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.

Note that the generalization of execution traces into a procedure is not reducible to an equivalent problem involving the generalizating non-sequential input-output pairs [16], as such a reduction will lose information about sequential changes in state.

Example representation is closely related to concept representation, as examples direct the search. Differences arise because examples are specific descriptions while concepts are general. In particular:

- Variables may appear in concepts but not in examples. Thus for functions, examples are input-output tuples of constants; for relations they are tuples comprising either attribute values or relations on constants, and for procedures they are execution traces with no variables identified.
- If there are optional elements in the concept, only one option occurs in each example.
- Procedural control structures do not occur in examples, so example execution traces are single sequences without branches and loops.

B Bias

The search involved in inductive learning is intractable unless it is strongly *biased*, as the space of possibilities is too large. In effect, any search is biased towards discovering some things rather than others by the way the space is constructed. Search is biased by background knowledge and assumptions, and in the way the induction process is constrained.

1 Background knowledge How much knowledge does a system need before it can infer concepts from examples? There is an oft-quoted aphorism that in order to learn something, you must nearly know it already. The amount of knowledge already possessed about a problem domain critically affects the kind of things one can expect to be learned. Concept learning techniques can be classified as *knowledge-sparse* and *knowledge-rich*, not so much on the basis of the amount of knowledge they embody as on the extent to which that knowledge is represented explicitly. Even the first kind embodies substantial prior knowledge in the form of languages in which examples and concepts are expressed. An instructable system must *learn* most of what it needs to know; otherwise its instructability will be limited to the tasks for which it already has the background knowledge. Unfortunately existing concept learning systems are able to learn very little, if any, of the background knowledge needed for them to work.[3]

Knowledge-sparse methods are of most interest in this context, as rich built-in knowledge may not help in learning new tasks. There are three categories.

Parameter learning optimizes the numerical parameters of a pre-specified model. Such systems are often designed to detect regularities in noisy situations, and use knowledge in the form of statistical techniques. The structure of the model is not represented explicitly, and the system cannot modify it or reason about it. Generally, large numbers of examples are used to allow incremental, hill-climbing-style optimization and to overcome the effects of noise. Parameter learning is a marginal case of concept learning with a well-developed arsenal of statistical and other model-fitting techniques; we do not address it further here.

Similarity-based learning delineates the space of possible concept descriptions in advance, and searches for ones which best characterize the structural similarities and/or differences between the examples presented [60,99,100]. The space is often finite (but very large; too large for simple enumeration to be feasible). One key issue is whether learning proceeds incrementally, using each example to further refine a generalized concept description, or all-at-once, when the examples are batched together for processing. Another is whether an approximate or exact match of concept to examples is sought, in other words whether the examples may be contaminated by noise.

Hierarchical learning augments the description language as each concept is learned [60,99,100]. This allows new descriptions to build upon old ones; so that background knowledge grows. Such learners are open-ended in that concepts of arbitrary complexity may be constructed. Search is controlled by apposite choice of order in which concepts are learned; thus the teacher plays a crucial role in directing learning.

2 Constraining induction In existing systems the designer constrains the search by ruling out possible forms of concept. Ideally, a mechanism would be provided whereby an instructable system could learn about the constraints that operate in a particular domain, but the general problem of learning constraints has yet to be tackled.[4] Note that it is not amenable to a purely formal treatment, for biasing constraints can only be expressed in logic as properties of predicates, so reasoning about bias entails the use of higher-order logics (which are undecidable). This section examines three different ways in which induction can be constrained.

Conceptual bias limits the vocabulary for expressing concepts. It applies equally well to predicate calculus, functional, and procedural representations. The vocabulary in question is not the entire set of symbols from which concepts are constructed, but only those that appear in examples. In logical representations of playing-card concepts, for instance, values might be represented by elements of the set {`2,3,4,5,6,7,8,10,J,K,Q,A`}, or just by the descriptors {`face-card, non-face-card`}. However, conceptual bias does not restrict the use of non-domain symbols such as $\wedge$, $\vee$ and $\Longrightarrow$.

In functional representation, conceptual bias determines the inputs (variables) and outputs (results) of expressions. For example, when learning functional expressions embedded in robot arm procedures, inputs might be robot joint angles, or the hand position in world coordinates. In procedural representations conceptual bias determines which domain objects can be referenced, in other words what the variables can be. For example, robot procedures might be able to refer to a variable giving the angle of contact between the arm and an obstacle, or alternatively to variables which record the touch pattern and forces at the points of contact.

A search can be conceptually biased to the extent that the system designer knows what components are needed in concepts. Conceptual bias is implicitly present in any formulation of a problem. For instance, the fact that physical attributes like `dog-eared` are excluded in a playing-card domain is a form of bias.

Composition bias constrains how the allowed vocabulary can be used to construct concepts. It generalizes the notion of *logical bias* [32] to functional and procedural representations. It limits the connectives used when forming expressions from vocabulary terms, thereby restricting the language in which concepts may be couched. For example, disjunctions might be prohibited so that only conjunctive descriptions such as `Spade` $\wedge$ `Face-card` are permitted. At any rate, unconstrained disjunctive combinations should be discouraged, to stop a concept degenerating into a list of all examples seen.

In functional representations, composition bias determines the vocabulary of built-in functions and operators, and the syntax by which concepts may be constructed. For example, in searching for a functional expression we might consider combining terms with operators from the set $\{\times, \div, +, -, \circ\}$ (where $\circ$ is function composition). In both functional and procedural representations, compositional constraints can be used to govern how terms match, ensuring that data types and physical quantities are combined appropriately. Composition bias in procedural representations is reflected in the use of a limited repertoire of control constructs, like `if...then...else...` and `while...do ...`; and even in the presupposition of sequential execution. In future there may be greater use of parallel procedural representations, where a construct like `sequence` is necessary to indicate sequentiality explicitly. Such constraints reflect at a rather deep level the style of computation envisaged.

There is an interesting relationship between conceptual and composition bias, and formal grammars. A grammar has a terminal vocabulary, the set of all symbols in the language it generates, and a set of production rules for sentence formation. To any concept space there corresponds a grammar which generates possible concepts. Conceptual biasing determines a vocabulary which is a *subset* of that grammar's terminal symbol vocabulary. It specifies only *content* symbols (those that refer to things in the domain) and not the non-content symbols such as logical connectives. Composition bias determines both the allowable non-content symbols *and* the production rules that may be used to form concepts.

For example, consider a grammar which generates concepts in the playing-card domain like `Spade` $\wedge$ `Face-card`. `Spade`, `Face-card`, and $\wedge$ are terminal symbols for the grammar. `Spade` and `Face-card` are content symbols belonging to the vocabulary determined by conceptual bias. $\wedge$ and a production rule allowing it to combine as an infix operator are both determined by composition bias. `Spade` and `Face-card` are aspects of the domain, whereas $\wedge$ and the production rule are aspects of concept composition.

Preference ordering. Creative science proposes hypotheses that explain observations. There is no formal way to choose between competing hypotheses, so long as they explain the examples given and can successfully predict crucial examples not in the original set [78]. More informally, hypotheses are preferred if they are elegant, simple, parsimonious, *etc.* These are patently subjective qualities, hard to capture formally. Concept learning systems must often choose between competing concepts but cannot afford the luxury of ill-defined measures. Methods for determining a preference order inevitably depend on the syntax of the language used to express concepts.

A straightforward approach is to apply Occam's razor by measuring the brevity of concept descriptions. One concept is simpler than another if it is shorter when written down. Length can be measured in various ways, including

- number of disjuncts in logical expressions
- number of function calls in functional expressions
- number of function calls plus total number of function arguments
- length of function expressions represented as character strings
- number of loops and conditionals in procedures
- number of statements in procedures
- number of states or transitions in a state-diagram representation [31].

A different approach, model-maximization [32], uses the extension rather than the intension of a concept. One concept is preferred over another if its extension is a superset of the other's. The instructable system must know how to order concepts according to their extensions; the ordering will be partial as extensions may overlap. For example the concept `Spade` $\wedge$ `Face-card` would be preferred to `Spade` $\wedge$ `Face-card` $\wedge$ `Room-temperature(30 degrees)` as it includes all of the extension of the latter concept, and more. Neither would be ordered with respect to `Black` $\wedge$ `Queen`.

Unfortunately, these machine learning techniques have contributed little to practical real world systems such as robots, because learning task descriptions from recorded robot actions requires detailed knowledge of robot and object geometry, and the physics of force interactions between objects [52], and because of the difficulty of learning to use sensory information.

V HOW THE TEACHER INTERACTS

Even when the interface for learning is very good, a teacher must meet certain conditions if the knowledge or skill is to be acquired. In this section we examine some of the general teaching issues for instructable systems.

A Theoretical foundations

The paradigm of generalization by enumerating and searching through candidate descriptions highlights the importance of the sequence of examples presented and the role of the "teacher" who selects them [16,34,60]. It is obvious that learning can be prevented by showing only a subset of examples. However, a teacher can have the power to inhibit learning by pernicious choice of the example sequence, even when all examples are eventually presented [34].

There is a basic distinction between presenting positive examples only, presenting both positive and negative (or counter-) examples, and allowing the learning system to choose examples itself and have them classified by an informant. In the first case, we assume that all positive examples are included eventually. In the second, we assume that every member of some universe which contains all possible descriptions is shown eventually; this allows an informant to be simulated by simply waiting until a selected example occurs in the presentation sequence. Gold [34] proved unequivocally what one expects informally, namely that the second and third methods are more powerful than the first. They permit primitive recursive languages, which include the context sensitive, context free, and regular languages, to be learned eventually, whereas with positive presentations only, as noted above, the inclusion of even one infinite language in the set of potential concepts can destroy learnability.

Gold [34] also showed a curious theoretical result about the example sequence: descriptions normally unlearnable from positive and negative examples—e.g. recursively enumerable languages—*may* be learned *if the presentation sequence is sufficiently regular.* Moreover, positive examples alone suffice! In essence the learner can identify the algorithm used to select the examples, thereby identifying the set from which they are drawn. For example, it is much harder to recognize the Fibonacci numbers from the presentation 34, 5, 13, 1, 8, 1, 21, 3, 2, 55 ... than from the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Here the importance of the teacher's selected example sequence is clearly reflected at the theoretical level. A good sequence of examples does more than speed the search, it makes learnable the otherwise unlearnable. This regular patterning of examples is a vital skill in teaching humans and it will be important for an instructable system to make use of it; experienced learners expect the order of example presentation to be carefully selected to give the maximum assistance to learning.

B 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 [95] showed how "near misses"—constructs which 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 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 make learning better than from random examples [90]. One obvious condition is that the teacher should correctly classify examples as positive or negative, and not (intentionally or unintentionally) mislead the student. Another is that if the absence of something is important, the teacher should point it out explicitly. If in the classic example of the concept of "arch" it is necessary that a gap be left between the pillars, the gap should be explicitly labeled in examples. 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 the concept of married *vs* maiden names using a lady 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 equilateral. Finally, the teacher should introduce one essentially new feature per lesson and not try to teach multiple differences at once—a similar condition to Winston's "near miss" approach.

Intermediate results, and other implicit parts of examples, are difficult for a system to learn since they necessarily involve a large search. The system must consider everything that *might* be absent, and the numerous relationships it may have with other parts of the description. Andreae's [13,14] "justified generalization" addresses this problem by using the amount of justification associated with a conjecture to determine which of three different levels of search to undertake. When the evidence is sufficiently strong, it will expend considerable resources seeking implicit relationships between parts of a description.

Although it does not increase formal learning power, the possibility of a system constructing its own examples and having them classified by an informant has considerable potential to speed up learning and reduce dependence on the skill of the teacher. This potential can only be realized if the system is able to synthesize "crucial examples" which discriminate effectively between alternative hypotheses consistent with the information seen so far. This ability seems to be one of the chief distinguishing characteristics of people who are good learners. To take the generation of examples one step further, if there is an automatic way of classifying examples as positive or negative, a system can attempt to learn autonomously and without the teacher. This paradigm of "learning by discovery" was exploited most effectively in pioneering work by Lenat [48,49,50] on automating the theory formation process in mathematics.

VI COMPETITIVE LEARNING ENVIRONMENTS

Real world learning environments are not the simple, one-on-one teacher-machine situations commonly used to exemplify concept learning systems. There will be other humans and machines to interact with, some at least will be indifferent and perhaps hostile. Just as you would want to lock a parked car, you would want to prevent your instructable robot from being given goals, or taught tasks, by anyone else. One can imagine voice prints and other personal locking devices being used.

However, in more complex competitive environments there is the probable danger of the *teacher* accidently corrupting a previously taught task, by teaching something else in a similar situation. Worse, when the tasks to be taught are sophisticated and involve complex dynamic manipulation, a teacher will be unable to give good examples. In holding a robot's hand, the teacher himself significantly disturbs the statics and dynamics of the arm. The robot can only attempt to reproduce the teacher's led *movements*, and learn from the results [54]. The instructor will be unable to tell whether to have the robot learning or performing. In this case the system should perform when *it* can choose actions, since it will likely learn more from the mistakes it makes than from the teacher's help.

For these reasons systems in competitive environments must not have a "teach switch" for external agents to toggle the system's mode between learning and execution [54]. Such a switch would allow acquired knowledge and skills to be deliberately or accidently corrupted, without regard to the consequences, nor to how well the task had been learned. One need only push the button and lead the robot through incorrect movements. To change incorrect learning requires other less direct methods, such as teacher-supplied indications of success or failure at a task. The learning system must alter its performance in response to these gross indications, but must not simply overwrite existing strategies. Reinforcement learning systems [70] such as some neural nets have addressed this problem, but concept learning has not.

However, further complexity requires self-motivation as well as instructability. An instructable robot is not equipped to learn in situations where a teacher cannot reliably provide it with high level goals. There is little point in allowing the teacher to set the goals; the system must treat all users as potentially unreliable, and choose its own goals[5] .

VII EXAMPLES OF INSTRUCTABLE SYSTEMS

This section exemplifies instructability by presenting two prototypes, one robot based and the other for office environments, and mentions systems under development which extend these two. All these systems show characteristics of instructability since they learn in the user interface from examples provided by the user.

A Online verbal task editing

This system demonstrates instructability via a physical manipulation device — a robot arm — and in particular illustrates two subtle aspects not present in typical keyboard/screen interfaces. First, robots are dynamic and we want to edit rapid robot movements on the fly, at normal robot speed. Second, the examples provided by the teacher cannot be perfect, as explained shortly, since they can usefully contain only movements, but not disturbance compensating forces. Still, forces must be applied in some tasks. Verbal task editing corrects movements on the fly, and provides compensating forces.

The scheme allows a teacher to correct a robot verbally as it executes a sequence, using his or her own well-practiced ability to verbally correct humans. Instructability is achieved through natural speech interaction. Corrections may be used to compensate for external disturbing forces applied to the arm, or to rectify badly led movements. The corrections invoked by verbal commands are themselves taught using the leading method.

While verbal correcting has no learning component and enables only one final sequence to be taught, the system can be augmented to allow the user to give a sequence of *conditional action corrections* without having to explicitly "program in" the conditionals. The teacher corrects the robot as appropriate in the present situation, and the robot can then choose its own corrections by sensing the conditions. This improvement, which is implemented by a production system of corrections, is described in detail in [54,55]. A simple learning, goal-seeking system can also be added to enable a teacher to teach individual actions and set goals [54,55]. The robot can be shown sensory-motor goals and selects its own path of actions to achieve a goal, given the sensed conditions. It can learn a repertoire of simple movements, and seek the goals itself along paths of movements in a network of movement actions, conditions and goals. The network may be retained from task to task, enabling the robot to build up a large repertoire of movements.

We first explain why it must always be possible to edit led sequences, no matter how carefully they have been taught. Following that, the verbal correction method is presented. Finally we describe both an implementation of the scheme and experiments with it.

1 Editing led sequences Led sequences *cannot* always be correct. When a movement, say M, is led, it causes a robot movement command, say C, to be stored for later execution. Now C is the command that will cause M, *so long as no uncompensated forces disturb the arm.* Such forces would cause a different movement to be produced. In that case the command C must be corrected to a command C^*, which does produce M. A robot control system can compensate only for certain disturbing forces regardless of its complexity. For example, a simple controller might not compensate for the slower movement caused by a heavy load, while a complex controller might not compensate for a swinging an axe. It is important to remember that when leading a robot arm, the teacher is taking the weight, compensating for the disturbing forces himself; and so led movements cannot themselves show the robot how to exert forces[6] .

2 The Verbal Correction Scheme The verbal correcting scheme changes both the teacher-robot interaction and the internal task representation of the normal led robot.

Teacher-robot interaction:

1. The teacher can set up verbal correction pairs, *before* and *while* he verbally corrects the robot. He does this by speaking the desired commands—for example "UP"—and leading the desired movement—for example an upward acceleration of 1 unit—during a special training mode;
2. The teacher can invoke the previously taught corrections by speaking the previously taught words; for example by saying "UP."

Internal representation: Correction pairs are accumulated in a list in the robot controller's internal representation of a task, for example < "CREEP UP", $(0,0,1,0,0,0)$ > might be the way the "UP" correction is stored. The correction is in Cartesian coordinates and hand orientation angles.

For example, Figure 3 depicts the setting up of the correction pair (UP, $+1^z$), where the led arm movement is one unit in the upward, z dimension. Then later, if the teacher says "UP," then the next command sent to the robot will have $+1$ added to its z component. The new value will immediately be executed, and in addition will be stored in the recorded sequence replacing the previous command. Figure 4 depicts this process. Perhaps the teacher would set up three different sized "UP" corrections in this way; say "CREEP UP," "UP," and "ZOOM UP." Sachs and Leifer [81] have used a full set of verbal corrections for controlling a robot. In addition, two special words are preprogrammed into the robot to enable a teacher to *scale* his taught corrections. Saying "LARGER" before a correction causes that set of corrections to be scaled up, say doubled, so that "LARGER UP" causes the corrections of the pairs for the word "UP" to be doubled in size. "SMALLER" might cause them to be halved. These verbal commands combine to form a suitably natural and flexible repertoire.

We expect verbal correcting to be useful for correcting led robots' sequences for several reasons [54,57]. The teacher may use his or her own natural ability at verbally correcting other humans, so long as the robot does not exceed human speed. A human teacher may anticipate errors in rapid movements [61] enabling him to correct the errors in spite of the delay between the onset of error and the teacher making a correction. The delay should be roughly one second.[7] At any time the teacher can stop the robot, put it into correction training mode and teach new corrections. So the teacher can easily teach new corrections appropriate to a particular part of a particular task. Current techniques enable short phrases from a limited vocabulary to be recognized [29,72,81,85]. Arguments for the theoretical stability and convergence of successive verbal correcting are given in [54,57]. A simple trajectory correcting experiment indicates that huge errors may be corrected in less than 15 successive corrections [54]. Smaller corrections can be made in fewer steps. Prototypes of verbal correcting have been implemented on both an experimental and an industrial robot [38,54,57]. As well as correcting for uncompensated disturbing forces, verbal correcting can also be used to correct errors made by the teacher in the original leading.

3 Experimental work The verbal correcting system has been implemented on a large industrial robot [38,59]. Table 2 shows the verbal repertoire programmed into the system. Three experiments have been performed:

1. Verbal correction of a robot sequence to avoid obstacles not present during initial leading,[8]
2. Using verbal commands to have the robot write on a flat surface.
3. Using verbal corrections to align the arm after one sequence, before a second insertion sequence. Here verbal corrections fine tune the intervening position.

These experiments indicate that verbal command and correction provides viable instructability for existing robots. Instructability is feasible in a complex dynamic setting. The next two sections summarize two current projects which aim at combining instructability with machine learning techniques.

4 Learning coordinate transforms by example Techniques for instructing systems are not yet able to provide for the complex data structures and algorithms required in real robots. This research project [76] takes an existing robot programming language and will eliminate programming aspects in a number of stages. The first stage has been discussed before [89] but the learning interface was not fully tested. The system is being reimplemented and leading will be used to show the robot where objects are during a task execution so that this information need not be programmed. Then the robot controller is able to work out coordinate transforms that were left undefined in the program. In factories much object data can be provided by the CAD system that designed the objects, but the

positions of parts during robot manipulation cannot be known in this way. For example, in assembling a bracket and beam combination, all that is unknown is the positions of objects, and — if there are left and right handed version of the assembly — the orientation of the bracket on the beam. It is worthwhile providing an instructable system — leading — for acquiring this information.

The second stage is to enable the grasp points on objects to be easily learned, as these too may not be provided by a CAD system. This will be accomplished by having the teacher lead the robot to grasp each object before or during the task. However, a teacher will have difficulty specifying when a transformation should be learned, and how many examples are required to calculate it [76]. Our third stage is to have the system learn both. Finally the program would be eliminated altogether, instead employing a techniques for merging example procedure traces into a general procedure, so the robot is fully instructable.

5 Learning in frames using function induction This learning oriented project [43] aims to provide instructability for assembly robots. Robot tasks are represented as a hierarchy of frames, with much of the information in the frames being learned from examples. Features of function induction, hierarchical information structures, concept learning [60], and automatic program construction [87] are combined. The principle learning technique is function induction, as robot movements are functions of previous movements, sensed conditions and nearby object characteristics. All this information is available in frame slots. Examples produced by a teacher's leading guide the function induction process. Control structures such as loops and conditionals may require the use of techniques for merging several example traces into a graph [13,14,87]. The raw recorded traces are processed and partitioned into precise and imprecise movements, which in turn are converted into symbolic robot motions. Learning techniques are applied to the symbolic representation, attempting to find general functional descriptions of the parameters of a movement — internal generalization — and general movement structures — external generalization — so that a general functional representation is formed.

B An automated office clerk

The automated office clerk [59] is a simple example of an instructable office assistant. In this project our aim is primarily to provide instructability, then to add more powerful learning techniques as they are required. The burden of communicating constructs such as conditional branches, is placed on the user. She indicates the components of a branch by direct manipulation rather than expecting the learner to induce the conditional.

The "desktop metaphor" provides a popular direct manipulation interface technique for office systems. Properly implemented, it creates and sustains the illusion that the user is operating on familiar reality, instead of on an entirely artificial, simulated world [98].

A dictionary defines a metaphor as the application of a name or description to an object or action to which it is not literally applicable [22]. A recent treatment [53] emphasizes that both analogies and disanalogies must exist between the attributes of semantic referents in a metaphor. In other words, if X is metaphorical on Y then it is the juxtaposition of X and Y that makes the metaphor; we know they are not the same, but the similarities are "suggestive." This is precisely the nature of the desktop metaphor; we know a "document" icon is not a document, but the metaphor is suggestive since we can compose and print the "document."

The desktop does not provide a metaphor for programming [101]. In existing systems, users must give instructions for new tasks *directly* to the system, rather than dealing with the metaphorical interface. Halbert [41] emphasizes the importance of having casual users program *in the interface of the system.* However, this does not go far enough. A great deal is gained by having a suitable metaphor for the act of programming itself. This metaphor will not circumvent all the difficulties of casual user programming. However, it does provide an easily-understood interface vehicle. Humans instruct human subordinates in office procedures—why not instruct office computers as well?

Consequently we have added an instructable "clerk" to the desktop metaphor. Clearly it is infeasible to make it understand the complex natural language instructions that office supervisors give their human clerks. However, supervisors sometimes "walk" their clerks through new office procedures, and this is just how they instruct the metaphorical clerk. For example, a supervisor might walk a clerk through the process of opening mail and filing it according to sender. (This example applies equally to human and metaphorical clerk.) "Walk-through" is tantamount to programming by example, and parallels the leading method for robots and the method of specifying macros to text editors, both mentioned above.

How does the casual user enable the constructs of Table 1 to be acquired by the clerk? The following sections discuss this question, paying particular attention to conditional branches (section 2), which existing leading systems

find hard to accommodate naturally.

1 Sequences Current programming-by-example systems (e.g. [41]) permit straight sequences to be taught in the desktop metaphor. The user simply instructs the system to start remembering, executes the command sequence herself, and tells it to stop recording. The system can replay the sequence to repeat the task. The lack of a suitable metaphor is not damaging because only three simple operations are required: start remembering, stop remembering, and execute the sequence. This simple technique can be elaborated to give taught sequences names, enable one sequence to invoke another, and so on.

2 Selection A user must be able to specify that different sequences are to be executed in different situations. For example, in the mail sorting task, the clerk must perform a different sequence for each different mail item sender. The TEMPO system [2] allows users to branch on particular text items, with a separate dialogue being used to specify the branch. But this dialogue departs sharply from the user interface metaphor into the realm of conventional programming.

In the automated clerk system, the user specifies conditional branches by drawing the clerk's attention to the significant item. Figure 5 depicts the task of opening mail items and file them by sender, putting mail from "Ian" in the folder "Ian's Mail." This is accomplished by

1. Selecting the "?" button over the clerk icon in the clerk window,
2. Selecting the item whose value is to be used for the conditional branch (the "From" field whose value is "Ian")
3. Proceeding to lead the conditional sequence.

In the figure the user has just selected the "?" button and is about to select the text "Ian" (the clerk cursor can be seen immediately to the right of the text), then proceed to file the mail.

Other conditionals—for example, other senders—are taught in the same way[9] . When a new condition occurs for which there is no taught action the clerk stops, informs the user, and awaits the appropriate sequence.

3 Variables, Iteration, Data Structures, Editing must be acquirable by the clerk

Variables are required in many tasks. For example, mail sorting is more general if the clerk can be instructed to open or create a file X, where X is the sender's name. In Halbert's [40,41] scheme (section III.D), variable ranges are specified explicitly by the user, through a dialogue box. This is not entirely satisfactory, since the user makes her selection not in the desktop metaphor, but through an interactive dialogue with the system. She should be telling the *clerk* what kind of variable it is. An improvement would be to have the system indicate which aspects of the desktop metaphor were candidates for variable ranges, such as groups of icons, individual icons, places on the desktop, icon names, *etc.* The user would select something appropriate. Better still, the system might induce this for itself, given a number of examples. Concept learning techniques [99]—learning descriptions from examples—are ideally suited to this induction task, so long as it is posed simply with only a few possible descriptions (like those Halbert accommodates).

Iteration of the entire sequence is easily specified. It is the default in robot leading, tasks being repeated until the user stops the robot. However, such an indefinite loop is trivially simple.

How should a user specify controlled loops? She must be able to indicate to the clerk both scope and termination condition. We have already seen how to communicate conditional execution. To create arbitrary loops, so that only part of a sequence is repeated, and allow nesting, the system must determine start and end points. In general, users are not be as sure of the exact start of a loop as they are of its end. One simple solution—adopted by the TEMPO [2] system—is to allow loops only around an entire sequence, but let one sequence invoke another—possibly iterative—named sequence. Another is to attempt to infer loop structure automatically from two or more iterations by comparing execution traces [13,14,30,97].

With the office clerk, users specify iteration by first leading the system through the loop two or more times, until the system itself—all the while seeking the loop structure—takes over. The teacher interrupts the system near the end of the loop, completes it, and specifies the exit condition as in section 2.

Data Structures are best created by an interactive form generation system. The user could specify fields and their contents interactively, in much the same way that user interfaces can be built by an interactive construction tool [28]. While this method is suitable for data structures comprising a series of nested fields and values, it may not accommodate general structured descriptions, where data items can comprise a number of parts with various relations between them.

Editing The user must be able to change taught procedures, to make modifications for new requirements and correct errors. Also, copying-and-editing existing tasks—"this task is like that one except that ...."—is likely to be as popular a methodology for generating new tasks as it is in conventional programming environments. The user should not be asked to edit a program of the traditional form—whether in a procedural or a fourth-generation language—for that would destroy the point of the metaphor. A taught program does not having a conventional listing; the nearest equivalent is a dynamic playback of the program, like a movie [86]. Editing should be in the same interface and using the same metaphors as the original teaching. For example, in TEMPO [2] the user plays back a taught sequence in a special edit mode, and can stop it to insert and delete actions. Ideally this would be done with verbal input, as described earlier for robot sequences.

4 A demonstration of an automated clerk A prototype demonstration of an automated clerk interface has been constructed to illustrate the main point, that a programming metaphor provides instructability which by enabling the casual user to specify procedures in the desktop interface. The clerk implements sequence recording and playback, with the option of making parts of the sequence conditional. The system is depicted in Figure 5, for the mail sorting task. It is an object based system, and enables all messages and appropriate system calls for the interface objects to be recorded and played back.

5 Constructive graphics editor A constructive graphics editor, modeled after an instructable draftsman, extends the interface by using domain-specific heuristics to generalize example traces provided by the teacher, forming more general actions and control structures such as loops and conditionals [62,63]. The metaphor is an instructable turtle, which moves around on the drawing sheet. As suggested in section VI the system automatically performs its own actions during teaching, once it can predict what might follow. The teacher is able to undo these actions, and so the system does not really meet the requirements of competitive learning in that section—but features such as undo will be essential in systems that are insufficiently autonomous in learning.

VIII CONCLUSION

This paper advocates the idea of instructability as a useful and practical kind of autonomy, in which the system learns and performs in the same environment, interacting with its teacher as it does with the rest of the world. Our work on autonomous learning complements on the one hand research in systems that perform known tasks autonomously, and on the other research on technical aspects of machine learning algorithms. Instructability, when combined with performance autonomy, creates systems that are autonomous in every respect but overall goals and motivation. Humans will not want all machines to be self-motivated, but they will want to teach their machines new tasks.

Conventional programming techniques are unsuited to casual users, so knowledge and skills must be transferred through a process more akin to teaching. Existing direct manipulation interfaces supply a suitable metaphor for performance, but not for teaching. Although current commercial systems enable example sequences to be memorized, conditions, iteration, data structures, variables and editing facilities are yet to be satisfactorily incorporated. New metaphors are required to support teaching properly.

For learning to take place, the system must employ data available in the interface—examples—to drive an inductive process of concept acquisition. Induction is generally intractable and the teacher must fulfil certain felicity conditions to render learning feasible. Current machine learning techniques enable general descriptions to be induced from examples, and these can form the basis of instructable systems provided that suitable representations and search bias are chosen.

The real world may be a competitive learning environment. This at least partially undermines the teacher's normally privileged position, for the system must protect itself against its knowledge and skills being corrupted by choosing when it will learn and when it will perform. Of course, this creates a basic conflict with the teacher's desire to supply the system's goals. Instructability will be insufficient in highly complex, competitive environments; self-motivation will then be essential.

Prototypes for verbally editing robot movements and an automated office clerk demonstrate the feasibility of instructable systems. Work has begun on the next step, incorporating more powerful learning techniques.

REFERENCES

[1] Abelson, H. and Sussman, G.J. with Sussman, J. (1985) *Structure and Interpretation of Computer Programs.* MIT Press.

[2] Affinity Microsystems Ltd (1988) *Tempo Version II manual.* Boulder, CO.

[3] Ambler, A.P., Popplestone, R.J. and Kempf, K.G. (1982) "An Experiment with the Offline Programming of Robots" *Proc. 12th Int. Symp. on Industrial Robots; 6th Int. Conf. on Industrial Robot technology*: 491-504. Paris, France. June.

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

[5] Andreae, J.H. (1977) *Thinking with the Teachable Machine.* Academic Press.

[6] Andreae, J.H. (1972-1987) Man-Machine Studies. Progress Reports nos. UC-DSE/1-29 to the Defence Scientific Establishment. Dept of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand.

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

[8] Andreae, J.H. (1986) "Purposeful Computer Systems" *Proc.Int.Conf. on Future Advances in Computing*: 5-28. Christchurch, New Zealand.

[9] Andreae, J.H. and Andreae, P.M. (1979) "Machine learning with a multiple context" *Proc.9th IEEE Int.Conf.on Cybernetics and Society*: 5-28. Denver. October.

[10] Andreae, P.M. and Andreae, J.H. (1978) "A teachable machine in the real world" *Int. J. Man-Machine Studies, 10*: 301-12.

[11] Andreae, J.H. and Cleary, (1976) J.G. "A New Mechanism for a Brain" *Int. J. Man-Machine Studies, 8*: 89-119.

[12] Andreae, J.H. and MacDonald, B.A. (1987) "Expert control for a robot body" Research Report 87/286/34, Department of Computer Science, University of Calgary.

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

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

[15] Andrew, A.M. (1987) Editorial for special AI issue. *Robotica 5*(2): 87-88.

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

[17] Benati, M., Gaglio, S., Morasso, P., Tagliasco, V., and Zaccaria, R., (1980) "Anthropomorphic Robotics, Parts I and II" *Biol.Cybern., vol. 38*: 125-150.

[18] Bonner, S. and Shin, K.G., (1982) "A Comparative Study of Robot Languages" *Computer, 15* (12): 82-96.

[19] Brooks, R.A. (1987) "Autonomous mobile robots." In [39], pp. 343-363.

[20] Burks, B.L., de Saussure, G., Weisbin, C.R., Jones, J.P. and Hamel, W.R. (1987) "Autonomous navigation, exploration, and recognition using the HERMIES-IIB robot" *IEEE Expert 2*(4): 18-27.

[21] Cohen, A. and Erikson, J.D. (1987) "Future uses of machine intelligence and robotics for the space station and implications for the U.S. economy" *IEEE Robotics and Automation, RA-1*(3):117-123.

[22] *Concise Oxford Dictionary* (1976) Sixth edition, Oxford University Press.

[23] Crossman E.R.F.W. (1953) "Entropy and Choice Time: The Effect of Frequency Unbalance on Choice — Response" *Quarterly J. of Experimental Psychology, V*(2): 41-51.

[24] Crowley, J.L. (1987) "Coordination of action and perception in a surveillance robot" *IEEE Expert 2*(4): 32-43.

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

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

[27] Dennett, D.C. (1987) *The intentional stance.* MIT Press.

[28] Expertelligence Inc. (1987) *The Exper Interface Builder manual.* Santa Barbara, CA.

[29] Flanagan, J.L. (1982) "Talking with Computers: synthesis and recognition of speech by machines" *IEEE Trans. Biomedical Engineering, BME-29*(4): 223-32.

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

[31] Gaines, B.R. (1977) "System identification, approximation and complexity" *Int. J. General Systems, 3*: 145-174.

[32] Genesereth, M.R. and Nilsson, N.J. (1987) *Logical foundations of artificial intelligence.* Morgan Kauffmann.

[33] Glasser, H., Hankin, C. and Till, D. (1984) *Principles of functional programming.* Prentice-Hall.

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

[35] Gordon, M.J.C. (1979) *The denotational description of programming languages.* Springer-Verlag.

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

[37] Goto, Y. and Stenz, A. (1987) "Mobile robot navigation: the CMU system." *IEEE Expert 2*(4):44-54.

[38] Grant, D (1987) "Report on Robbie" Research Report 87-275-23, Department of Computer Science, University of Calgary.

[39] Grimson, W.E. and Patil, R.S. (1987) (editors) *AI in the 1980s and Beyond.* MIT Press.

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

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

[42] Hasegawa, T., (1982) "An Interactive System for Modeling and Monitoring a Manipulation Environment" *IEEE Trans. SMC, vol. SMC-12*(3): 250-8.

[43] Heise, R. (in preparation) "Robots acquiring tasks through learning from examples" Research report, Department of Computer Science, University of Calgary.

[44] Henderson, P. (1980) *Functional programming.* Prentice-Hall.

[45] Hyman, R. (1953) Stimulus "Information as a Determinant of Reaction Time" *J. of Experimental Psychology, 45*,(3): 188-96.

[46] Ishii, M., Sakane, S., Kakikura, M. and Mikami, Y. (1987) "A 3-D sensor system for teaching robot paths and environments" *Int. J. Robotics Research 6*(2): 45-59.

[47] 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.

[48] Lenat, D.B. (1978) "The ubiquity of discovery" *Artificial Intelligence, 9*: 257-285.

[49] Lenat, D.B. (1983) "EURISKO: a program that learns new heuristics and domain concepts" *Artificial Intelligence, 21*: 61-98.

[50] Lenat, D.B. and Brown, J.S. (1984) "Why AM and EURISKO appear to work" *Artificial Intelligence, 23*: 269-294.

[51] Lozano-Perez, T., (1983) "Robot Programming" *IEEE Proc, vol. 71*(7): 821-41.

[52] Lozano-Perez, T. (1987) "Robot programming and artificial intelligence." In [39], pp. 281-315.

[53] MacCormack, E.R. (1985) *A Cognitive Theory of Metaphor.* MIT Press.

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

[55] MacDonald,B.A. (1987) "Improved robot design" *Trans. IPENZ Elec/Mech/Chem section, 14*(1/EMCh): 33-48.

[56] MacDonald, B.A. (1987) "Connecting to the past" Collected papers of *IEEE Conf. on Neural Information Processing Systems,* Denver, Co., November. American Institute of Physics.

[57] MacDonald, B.A. (1987) "Programming computer controllers by giving examples" *Proc. IEEE Wescanex. 1st Conf. on Programmable Control Systems*: 49-53. Edmonton. September.

[58] MacDonald, B.A. and Andreae, J.H. (1981) "The Competence of a Multiple Context Learning System" *Int. J. Gen. Systems, 7*: 123-37.

[59] MacDonald, B.A. and Witten, I.H. (1987) "Programming Computer Controlled Systems by Non-experts" *Proc IEEE Systems, Man and Cybernetics Annual Conf.*: 432-437. Alexandria, Virginia. October.

[60] MacDonald, B.A. and Witten, I.H. (in press) "A framework for knowledge acquisition through techniques of concept learning" *IEEE Trans. Systems, Man and Cybernetics* special issue on knowledge engineering.

[61] Marteniuk, R.G. (1976) *Information Processing in Motor Skills.* Holt, Rhinehart and Winston.

[62] Maulsby, D.L. and Witten, I.H. (1988) "Inducing programs in a direct-manipulation environment" Research Report 88/320/32, Department of Computer Science, University of Calgary.

[63] Maulsby, D.L. and Witten, I.H. (1988) "Acquiring graphical know-how: an apprenticeship model" *Proc. European Knowledge Acquisition Workshop 88,* Bonn, West Germany; June.

[64] McLaughlin, J.R., (1982) "TRIG: An Interactive Robotic Teach System" Working Paper 234, MIT AI Lab. June.

[65] McTamaney, L.S. (1987) "Mobile robots: real-time intelligent control" *IEEE Expert 2*(4): 55-68.

[66] Michalski, R.S. (1983) "A theory and methodology of inductive learning" In *Machine learning,* edited by R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, pp 83-134. Tioga, Palo Alto, CA.

[67] Miller, W.T. , Glanz, F.H. and Kraft, L.G. (1987) "Application of a general learning algorithm to the control of robotic manipulators" *Int J. Robotics Research 6*(2): 84-98.

[68] Mitchell, T.M. (1982) "Generalization as search" *Artificial Intelligence, 18*: 203-206.

[69] 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.

[70] Narendra, K.S. and Thathachar, M.A.L. (1974) "Learning automata—a survey" *Proc IEEE Trans Systems, Man and Cybernetics, SMC-4*(4): 323-224.

[71] Nilsson, N.J. (1980) *Principles of artificial intelligence.* Tioga.

[72] Nitzan, D. (1979) "Flexible Automation Program at SRI" *Proc JACC*: 754-9. Denver; June

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

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

[75] Oommen, B.J., Iyengar, S.S., Rao, N.S.V. and Kashyap, R.L. (1987) "Robot navigation in unknown terrains using learned visibility graphs. Part I: the disjoint convex obstacle case" *IEEE Robotics and Automation, RA-3*(6): 672-681.

[76] Pauli, D. (in preparation) "Improved robot programming through learning from examples" Research report, Department of Computer Science, University of Calgary.

[77] Peyton-Jones, S.L. (1987) *The implementation of functional programming languages.* Prentice-Hall.

[78] Popper, K.R. (1968) *The logic of scientific discovery.* Hutchinson, London.

[79] Robotica (1987) Special issue on artificial intelligence. *vol. 5*(2).

[80] Rosen, C.A., (1979) "Machine Vision and Robotics: Industrial Requirements." In *Computer Vision and Sensory-Based Robots,* ed. G.G. Dodd and L. Rossol, pp. 3-20. Plenum, N.Y.

[81] Sachs,J. and Leifer, L. (1979) "Voice Command of a Six-Degree-of-Freedom Manipulator" *Proc. J. Auto. Contr. Conf*: 783-789. Denver; June.

[82] 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, Los Altos, CA.

[83] Sheridan, T.B. and Ferrel, W.R. (1981) *Man-Machine Systems. Information, Control, and Decision Models of Human Performance.* MIT Press, 2nd edition.

[84] Shneiderman, B. (1983) "Direct manipulation: a step beyond programming languages" *IEEE Computer, 16*(8): 57-69; August.

[85] Simons, G.L. (1980) "Robots in industry" National Computing Centre, Manchester, England.

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

[87] Smith, D.R. (1984) "The synthesis of LISP programs from examples: a survey." In *Automatic Program Construction Techniques* edited by A.W. Biermann, G. Guiho and Y.Kodratoff, pp. 307-324. Macmillan.

[88] Summers, P.D. and Grossman, D.D., (1984) "XPROBE: an experimental system for programming robots by example" *Int J Robotics Research, vol. 3*(1): 25-39; Spring.

[89] Takase, K., Paul, R.P., and Berg, E.J., (1981) "A Structured Approach to Robot Programming and Teaching" *IEEE, SMC-11*(4): 274-89.

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

[91] Voltz, R.A. (1988) "Report of the Robot programming language working group: NATO workshop on robot programming languages" *IEEE Robotics and Automation, RA-4*(1): 86-90.

[92] Waltz, D., (1982) "Artificial Intelligence" *Scientific American*: 78-83; October.

[93] Welford, A.T. (1980) "Choice Reactions Time: Basic Concepts." Chapter 3 of *Reaction Times*, edited by A.T. Welford. Academic Press.

[94] Weisbin, C.R. (1987) "Real-time control: a significant test of AI technologies" Guest editor's introduction. Special issue on Intelligent systems and their applications. *IEEE Expert* 2(4): 16-17.

[95] Winston, P.H. (1975) "Learning structural descriptions from examples." In *The psychology of computer vision*, edited by P.H. Winston. McGraw Hill.

[96] Winston, P.H. and Brady, M. (1987) Series foreword in [39].

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

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

[99] Witten, I.H. and MacDonald, B.A. (1987) "Concept learning: a practical tool for knowledge acquisition?" *Proc. 4th Intl. Workshop on expert systems and their applications*: 1535-1555. Avignon, France, May.

[100] Witten, I.H. and MacDonald, B.A. (in press) "Using Concept Learning for Knowledge Acquisition" *Int. J. Man Machine Studies*. Presented at *AAAI Knowledge Acquisition for Knowledge Based Systems Workshop*. Banff, Canada, October.

[101] Witten, I.H., MacDonald, B.A. and Greenberg, S. (1987) "Specifying Procedures to Office Systems" *Proc. Conf. on Automating Systems Development*, Leicester, England; April.

[102] Witten, I.H. (1987) "In search of 'autonomy'" *Future Computing Systems*, 2: 35-53.

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

motivation	choosing your own goals
learning	being able to learn by interacting in a world of other autonomous systems, without a teacher having direct access to your "internals"
performance	dealing with unexpected situations, noise, changing conditions, rich real world, fault-tolerance, etc
reproductive	able to make your species autonomous by self-reproduction
survival	in hostile and dangerous situations, and in competition with other autonomous systems
computational	on board computation; no dependence on communication links [20]
communications	controls own communications circuits (not a communications slave)
power source	need not rely on anyone to plug it in

Figure 1: Types of autonomy. A practical instructable system would include all but motivational autonomy.

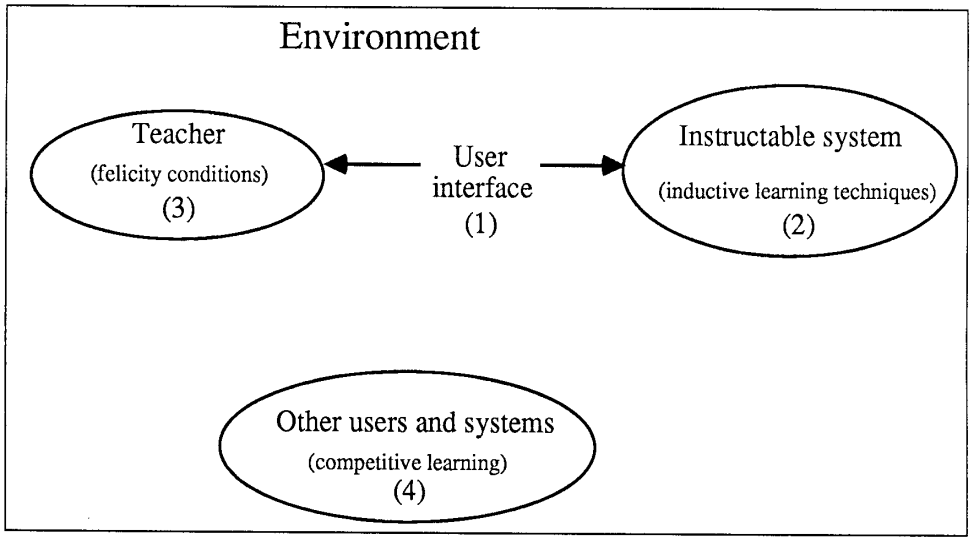

Figure 2: Four requirements for autonomous learning.

1. Learning in the user interface
2. Inductive learning techniques
3. Teaching constraints (felicity conditions)
4. Ability to cope with a competitive learning environment

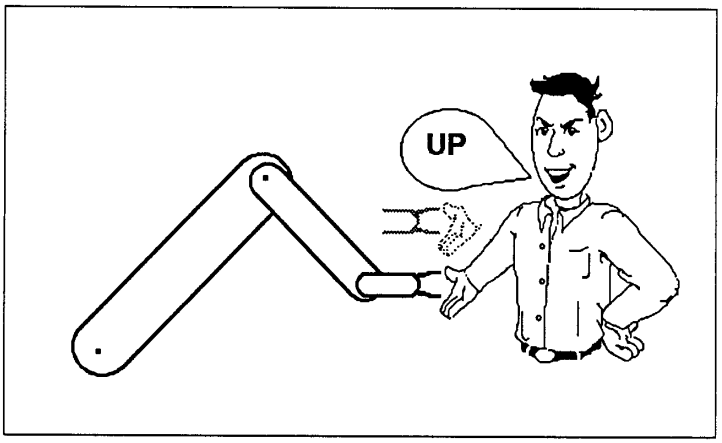

Figure 3: Teaching a verbal correction

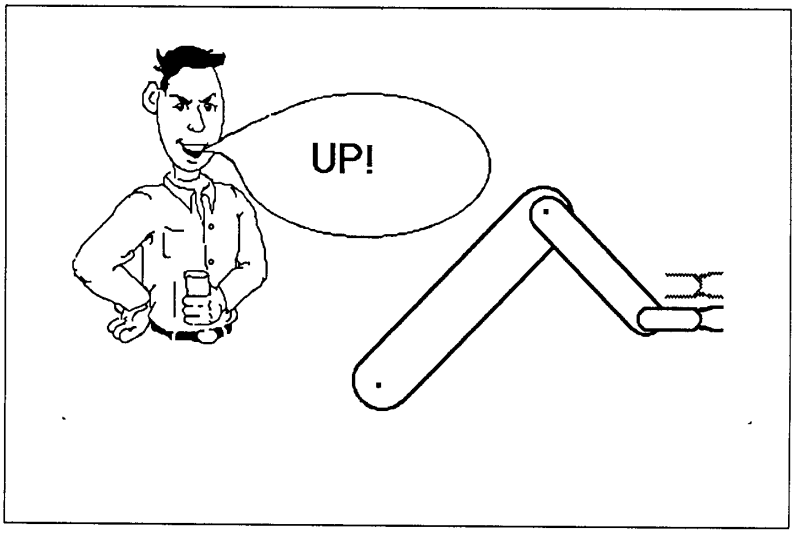

Figure 4: Invoking a verbal correction

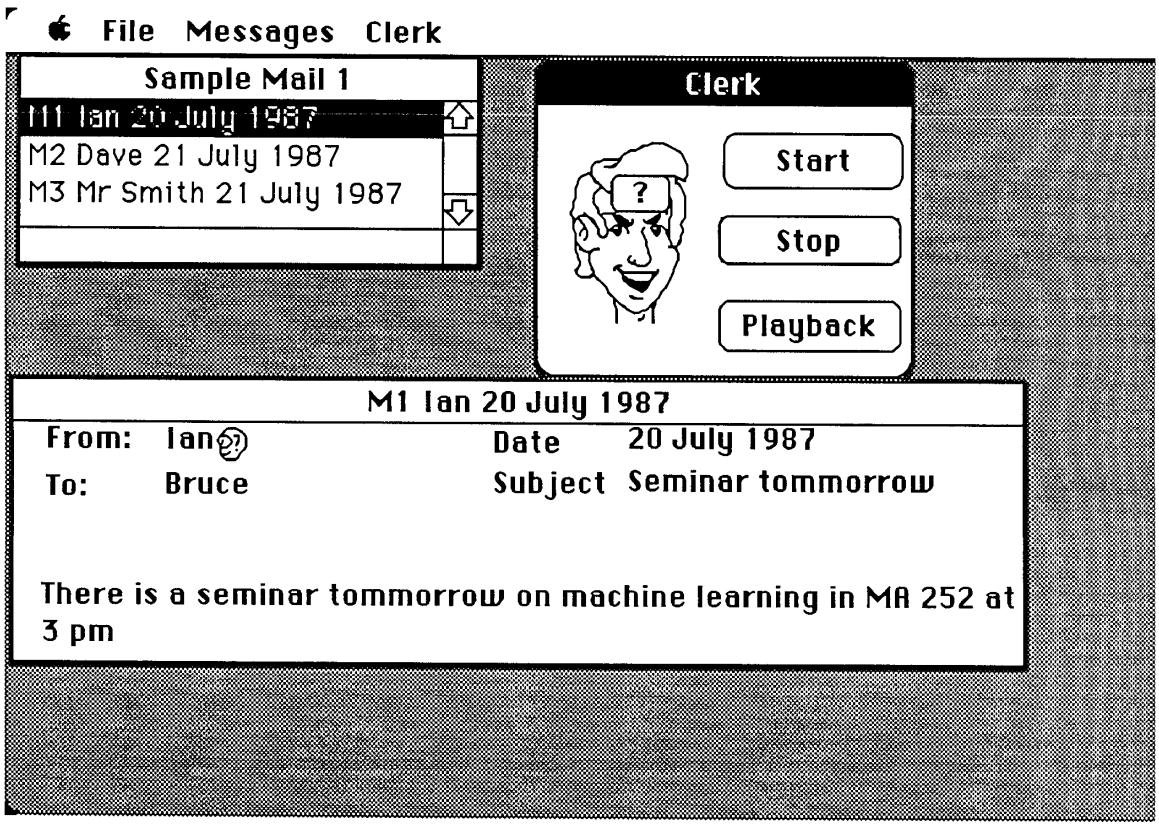

Figure 5: The automated clerk

construct	form	example
conditionals	if ... then ... else ...	if Name is "John Doe" then alert manager else (no action)
iterations or loops	for, while, repeat	for all files in folder "New Reports" print the file
nested control structure	structures within structures	for all folders for all files in folder print file
data structures	structured information	*Name* John Doe *Phone* (403) 234-3467 *Address* 4432 69 St NW *Postal* E4A-1S4
variables	set of objects	let file be "letter1" let printer be "laser" (then later) print file on printer

Example Procedure
open "Mail" for each new mail item (*New-Mail-Item*) open *New-Mail-Item* let *Sender* be sender part of data structure *New-Mail-Item* if there is no folder *Sender* then create a new folder *Sender* copy *New-Mail-Item* into *Sender* folder end for loop

Table 1: Constructs that must be acquired by an instructable system

Quantifiers (permanent)	*Correction*
Tiny	2
Smaller	0.25 previous
Normal	120
Larger	1.5 previous
Huge	240

Qualifiers (temporary)	*Correction*
Zoom	double size
Creep	halve size

Motions		*Axis*
Up	*or* Down	vertical (z)
Left	*or* Right	horizontal (y)
In	*or* Out	horizontal (x)
Extend	*or* Flex	z–rotation
Curl	*or* Uncurl	y–rotation
Twist	*or* Untwist	x-rotation

Control	*effect*
Clear	Reset voice control, correction bases, robot alter system
(Three user defined panic words)	Robot Abort

Table 2: Verbal corrections are made in unit amounts. Quantifiers set the base amount, qualifiers modify the amount for the current correction, and motions determine the dimension of correction. The maximum correction size is 960, and the minimum 1

LIST OF FIGURES

1 Types of autonomy. A practical instructable system would include all but motivational autonomy. . 21
2 Four requirements for autonomous learning. 22
3 Teaching a verbal correction 23
4 Invoking a verbal correction 23
5 The automated clerk 24

LIST OF TABLES

1 Constructs that must be acquired by an instructable system 25
2 Verbal repertoire for online verbal editing (from [38]) 25