1. Introduction

People who employ interactive text editors are frequently confronted with simple, but repetitive, editing tasks. These can be frustrating to perform manually, too variable and individually not significant enough to automate by a custom-built program, and difficult to address with structured editors because the text lacks formal structure.

Three such tasks are illustrated in the Appendix. They all involve reformatting a textual database: in the first case an address list, in the second a database of match scores, and in the third a list of references. Each can be accomplished using an editor with a macro definition capability (e.g. EMACS; Stallman, 1981) or a stand-alone macro facility (e.g. TEMPO; Pence & Wakefield, 1988), but this requires careful advance planning and is hard to get right first time. Moreover, any programming solution suffers from excessive rigidity—there is no avoiding the fact that the tasks are defined incrementally and may be extended as more examples are encountered. Editing macros in particular tend to be difficult to extend unless they are expressed in some abstract form, and this is likely to put off the non-programming user. In practice, people are likely to accomplish such tasks by manually editing each entry, perhaps using global search-and-replace to perform sub-tasks where feasible.

Programming by example seems a natural way to approach these problems. Rather than having users program a task, they need only specify examples of how to perform it, from which a program is synthesized that performs the task itself. Programming-by-example schemes have been proposed for office information systems (e.g. Halbert, 1984; MacDonald & Witten, 1987), operating system interaction (e.g. Waterman *et al.*, 1986), robot programming (e.g. Andreae, 1984, 1985; Heise, 1989), and graphical editing (e.g. Maulsby *et al.*, 1989).

Another possible approach is to use techniques of predictive text generation—which exploit the statistical redundancy of language to accelerate and amplify user inputs—in conjunction with adaptively-formed language models. For example, we have built a device called the REACTIVE KEYBOARD that accelerates typewritten communication with a computer by predicting what the user is going to type next (Darragh *et al.*, in press). While this is suitable for enhancing the ease and rate of initial text entry, especially for physically limited people, it is not particularly well-matched to the task of editing structured or semi-structured data.

Nix (1983, 1984) has investigated the application of "by example" techniques to text editing. His approach is to seek a transformation that maps blocks of input text into corresponding blocks of output text. While this works well for rigidly-structured editing tasks like that of Task 2 in the Appendix (which was in fact taken from Nix's thesis), it breaks down when faced with less artificial transformations that involve rules and exceptions, like those of Tasks 1 and 3. In contrast, the scheme developed in the present paper adopts a procedural stance: editing actions are recorded and generalized into a program that transforms input into output. This seems to offer better potential

for complex, ill-structured, editing problems where many different cases may arise.

2. Principles

We begin by stating some simple principles on which the work is based. Some of these were suggested by an informal preliminary experiment in which we presented 12 subjects with a number of different editing tasks, all involving reformatting blocks of semi-structured information, and observed them while they went about the task using a simple interactive editor (the Macintosh MINIEDIT; Chernicoff, 1984).

Use an abstract editing model that provides a small set of simple operations. As Nix (1983, 1984) pointed out, users can accomplish an editing task in different ways, and this confounds procedural models of their performance. For example, moving down to the end of the second line of the next paragraph can be done with innumerable different combinations of basic *move* commands. Consequently our editing model provides a single *select* operation that subsumes all sequences of moves. Many editing actions—such as search-and-replace—are in effect canned procedures and need not be provided in an effective programming-by-example environment.

Record the local context of each editing action in a rich, redundant, manner. To recover the information lost by ignoring the details of individual *move* commands it is necessary to infer what information determines the target of a *select* action. Because the user might have in mind any of a variety of different selection procedures (e.g. skipping a constant number of words or lines, moving to the beginning of a word or paragraph, finding a certain left or right context), it is advisable to record as much relevant information as possible.

Base the program on the sequence of commands performed when editing a single block of information. We assume that the user wishes to edit semi-structured databases that comprise clearly-defined blocks of text, as illustrated in the Appendix. Programming by example is made much easier if the user demonstrates the procedure on the first block and informs the system when it is complete. The alternative on the one hand of predicting the next editing action whenever enough evidence has been accumulated to do so with confidence (as the METAMOUSE system for graphical editing does; see Maulsby *et al.*, 1989) inevitably exacts a price either in conservatism or over-eagerness to predict. The alternative on the other of allowing the user to demonstrate the procedure on several blocks invites free variation in task execution, which exacerbates the difficulty of learning.

Extend the program dynamically if errors occur in subsequent blocks. In actual repetitive editing tasks, subsequent blocks of text often diverge from the format of the first—the databases do not have a completely rigid, repetitive, structure. To accommodate this the user must be able to extend the procedure on the fly to account for new circumstances. This is a kind of on-line debugging of programs taught by example.

Allow the user a practice period in which to discover a suitable procedure. We observed that users perform repetitive editing tasks by trial and error in the beginning, and settle on a suitable editing procedure only after some experimentation. It is valuable to provide a practice mode to allow a suitable procedure to be discovered. This could simply take the form of a command that resets the first block of text to its original state, to be invoked after each practice attempt.

3. Design

Here we discuss the design of a procedural programming-by-example editing system; specific implementation decisions are covered in Section 4. It is assumed that the editor supports the notion of current text position and the following primitive actions (summarized in Table 1):

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

There are, of course, alternative ontologies of editing actions. For example, *delete* could be considered a special case of *insert* using the empty string; *locate* could be treated as *selecting* the empty string. From our point of view, these are not important issues.

These actions constitute an extremely primitive editor, whose usefulness would be greatly enhanced by additional operations. Many common ones can be accommodated within the programming-by-example scheme but do not affect its design. Examples are

- *search* finds a specified text pattern and *selects* it;
- *cut* deletes the currently-selected text and saves it in a buffer;
- *paste* inserts the contents of the buffer at the current position.

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

3.1 RECORDING TRACES

Editing traces are recorded as sequences of actions and attributes. The four primitives above—*insert, locate, select, delete*—correspond to four types of

action. In practice these operations may not be primitive as far as the user is concerned—for example, locating a position may require many *next-line* and *next-character* commands; selecting text may require *mark* and *copy-region* commands—however, they are mapped into actions by the recording process. *Insert* and *select* have a parameter that records the text string involved.

Locate and *select* require positional information. With these actions are recorded a number of attributes that characterize the place at which the action occurred. The attributes rely on a lexical decomposition of the text into units such as characters, words, lines, paragraphs, and the file itself. Of course, many difficult issues are involved here (e.g. see Witten & Bell, 1990, for a discussion of the question of word identification alone), and different applications will involve different lexical categories. However, we are not promoting our particular choice of attributes but rather the methods of generalization that are based on them.

Three basic kinds of attribute have been identified, and are summarized in Table 1. The first characterizes the local context of the current position by specifying a stretch of preceding text and of succeeding text. The second gives positional information relative to enclosing units of text, such as beginning, middle, or end of a file, paragraph, or line: we call these "lexical" attributes. They provide less local, but much less detailed, information than the local context. The third measures relative distance from the previous position, and can be expressed in various units such as characters, words, lines, and paragraphs.

We deliberately leave open the details of these attributes. Particular choices are discussed in Section 4, but the idea of editing through procedural programming-by-example transcends them. Note that it is not necessary to include as attributes local lexical features such as capitalization and punctuation, for these can be determined by generalizing the local context (and the parameter in the case of a *select* action).

3.2 CREATING PROGRAMS

Creating programs involves generalizing a trace—which is in effect a straight-line program—into a procedure that includes both variables and control structures such as loops and branches. The idea is to identify steps that are to be merged into a single procedure step, and generalize their attributes.

This is a process of induction, and it is inevitable that any proposed solution will have shortcomings in particular situations. Fortunately, however, the interactive nature of the editing environment allows the user to correct for deficiencies in induction. Here we identify desirable properties of any generalization algorithm: designing specific algorithms that fulfil these properties is a topic of current research. A simple method is described in Section 4 and evaluated, through examples of its performance, in Section 5.

Identifying steps to merge. The aim of merging different steps of the trace is to increase the predictability of actions. In general terms, one seeks the smallest procedure that accounts adequately for the trace. However, there is an inevitable trade-off between the accuracy with which the procedure

"accounts for" the trace and the size of the procedure. This can be formulated in terms of the entropy of the predictions. Unfortunately, algorithms to find the best procedure that represents a particular trade-off are infeasible because they involve enumeration of all possible procedures (Gaines, 1976). Consequently practical systems form "context" models that merge steps when they are sufficiently similar and when they occur in sufficiently similar contexts (Witten, 1987). Translated into the editing domain, this results in these rules of thumb:

- only merge steps that involve the same primitive actions;
- when deciding whether to merge two steps, take account of their neighbors.

The first rule results from the fact that the procedure formed must be executable in order to provide any advantage to the user—underspecified actions such as *insert-or-select* will not help. However, actions with different parameters can be merged. For example, there may be circumstances under which two *select* actions with different character strings should be merged by finding a regular expression that covers them both. Similarly, *insert* actions that specify different text may be merged: although this will not produce an executable program step it may reflect unpredictability that is inherent in the task.

Just because the steps involve the same actions does not necessarily mean that they play the same role in the procedure. Merging steps indiscriminately, even though they match, can reduce predictability by losing the correct context within the procedure. The second rule highlights the importance of context, but does not provide a rationale for deciding how much context to use.

Another way of restricting merging is to take into consideration the structure of the program being created. For example, the discipline of structured programming suggests the rule

- do not merge nodes if this creates a branch into an already-formed loop.

This eliminates overlapping loops but permits nested ones, and is used in the implementation described in Section 4.

Generalizing attributes and parameters. With *locate* and *select* actions are associated attributes that must be combined when steps are merged, as must the parameters of *select* actions. A number of different methods of generalizing items having various data types have been identified in the machine learning literature (e.g. Michalski, 1983). We need ways of generalizing character strings (*select* parameters and local contexts), lexical attributes (beginning, middle, or end of textual unit), and numeric attributes (relative distances).

The last two are simple. In virtually all realistic procedures, lexical attributes affect the procedure only if they remain constant at every occurrence of a step. This will allow a step to be restricted to the beginning of a line or paragraph, for example—a common requirement when editing structured

data. Thus lexical attributes are discarded whenever they differ between steps.

The same holds for numeric attributes—executing a step on every third word, for example, causes the "word" measure to remain constant. These attributes also serve the purpose of restricting the region or "focus of attention" in which the system must search for a certain context. For example, if the number of words and lines skipped since the last editing action varies but the number of paragraphs remains constant at zero, attention is restricted to the current paragraph when seeking a new context in which to execute the current step. Moreover, the sign of these attributes dictates the direction of search when locating a new position.

Generalizing character strings. Character strings, such as the parameters of *select* actions, can be generalized by identifying a string expression (for example, a regular expression) that covers the different examples. Here we identify some of the issues in designing a generalization method for strings; a practical, heuristic, method will be sketched in Section 4.2.

It is trivial to find a regular expression that satisfies a given set of examples, for it need be no more than a list of the examples. To further constrain the problem one might seek the *smallest* regular expression that satisfies the examples. This can be found by enumerating all expressions from the smallest up, but unfortunately this is computationally intractable and, in fact, efficient procedures do not exist (Angluin, 1978).

A much simpler approach is to seek the longest common contiguous substring occurring in the examples. Then one could define a generalization to be any string that contains this substring. This effectively treats strings as a fixed part flanked by variable parts—not a very comprehensive basis for detecting lexical structure in strings.

Nix (1983) generalized this approach by defining a "gap pattern" to be a sequence of alternating strings and gaps, strings being constant and gaps matching any text. The general problem of finding a gap pattern that fits a set of sample data is intractable when the strings are long. However, he developed a heuristic procedure that attempts to find the "best" gap pattern that matches a set of strings, namely the one that

- maximizes the number of constant symbols;
- minimizes the number of gaps (subject to the previous constraint).

The procedure first finds the longest common (non-contiguous) subsequence of the set of strings. The maximal-length ordered sequence of characters common to two strings can be found efficiently using dynamic programming (Hirschberg, 1975), and the method can be adapted to deal with more than two strings. Nix's procedure then places sufficient gaps in the longest common subsequence to make it unify with all strings in the set.

Many editing tasks are sensitive to the pattern of character *classes* in the strings being manipulated, where classes are assigned according to a hierarchy such as that of Figure 1. An improvement on Nix's formulation is to consider *typed* gaps that specify the class of omitted character strings.

Although identifying mergeable steps only provides positive examples of character strings that are associated with the step, negative examples will come from executing the procedure on different blocks of text. Indeed, having formed a procedure from a trace, negative examples may be found by re-applying that procedure to the same trace and finding places where incorrect predictions are made because of over-generalization. None of the above methods cater for negative examples, except the enumeration technique for regular expressions.

The problem of generalizing the context attribute is slightly different from that of generalizing *select* parameters, for their extent is not well-defined—it is not clear how far the preceding context should extend to the left, nor the succeeding context to the right. The simplest solution is to select a unit that constitutes the context, say a word. However, this eliminates the possibility of finding rules that depend on larger contexts. It would be better to carry along contexts at several different lexical levels: word, line, sentence. Since different examples at each level will be amalgamated by the character-string generalization method, this would allow the discovery of any regularities that occur, regardless of their level.

3.3 EXECUTING AND EXTENDING PROGRAMS

Once a program has been created from the trace of actions on the first block of text, it is executed on subsequent blocks. The process of "executing" a program step depends on the particular action involved. For actions that need to find a position in the text it is essentially a process of pattern recognition, while for other actions it is more straightforward.

Consider the simple cases first. A step that specifies an *insert* action may or may not specify the text that is to be inserted. If not, the user is simply invited to type it. If so, the suggested insertion is presented for approval. If it is not approved, the system has to decide whether to generalize the program step into an *insert* action with unspecified text, or create an alternative step with the new text parameter and link it in to the program. This is the same decision as whether to merge two trace steps together when constructing the procedure from the trace.

If the user indicates that an *insert* action is inappropriate, the system has no option but to ask for the correct action and amend the procedure accordingly using the same method as for initial procedure construction.

Other actions, like *delete*, *cut* and *paste*, are executed in the same way, by informing the user of the proposed action, awaiting confirmation, and soliciting the correct action in the case of an error.

Now for the more complex cases, when the action involves finding a position in the text. The text must be scanned from the current position until the attributes are matched. The direction of scan is indicated by the sign of the *relative-distance* attributes. Its extent is given by the size of the smallest unit (word, line, paragraph, ...) whose *relative-distance* has remained zero in previous executions of this step. The match pattern is given by the context specifications associated with the program step, and the parameter in the case

of a *select* action. The lexical attributes of the position (beginning, middle, or end of a unit of text such as file, paragraph, or line, etc.) which have remained constant in previous executions should be respected in the scan.

If a position is found that satisfies these constraints, the user is informed of the proposed position and action. If the action is correct but the position is incorrect, the correct location is requested and the attributes associated with the program text are updated accordingly. If the action is incorrect, the correct one is solicited and the program amended accordingly.

When updating a program step to accommodate a new location in which it should be executed, there are two cases. If the correct position was overlooked by the scanning procedure, the program step is too specialized and must be generalized so that it would find the correct position next time. This involves generalizing the context attributes (and the parameter, in the case of a *select* action) to subsume the new positive example. However, if the scan finds a position that precedes the one indicated by the user, the program step is too general and must be altered to avoid finding the incorrect position in future. This involves specializing the context attributes (and the parameter, in the case of a *select* action) to inhibit matching the new position, effectively treating the incorrect position as a negative example. It may also be necessary to generalize these attributes to ensure that the correct position is indeed matched, if it is not already.

Suppose now that the scan does not succeed in finding a position that satisfies the constraints represented by the attribute values. Rather than reporting to the user that the program step cannot be executed, a match can be sought with a generalized version of the attributes. This can be done in three ways. The extent of the scan can be increased, the lexical attributes can be over-ruled, or the context patterns can be generalized. It is not clear how best to balance these three possibilities. In any case, if weakening the attribute constraints does allow the scan to find a position for the program step, the system suggests the action as before, and if it is accepted, generalizes the attributes to subsume the new position just as it would if the user had suggested it.

In summary, whenever executing a program step involves finding a position, the attributes associated with that step are used to constrain the position. Two kinds of bugs are anticipated. Overgeneralization bugs cause the procedure to match a step with a place which should not have been matched. Overspecialization bugs cause it to miss places that should have been matched. Both cases are handled by generalizing or specializing the attributes of the program step accordingly.

A third kind of bug, incorrect program structure, occurs when a predicted action is actually incorrect (in contrast to the correct action being predicted at an incorrect position). Procedure execution is suspended and the user demonstrates the correct action, or sequence of actions, manually. The system will then merge the new trace fragment in with the existing program.

4. Implementation

So far the issues have been identified but some details of how to deal with them remain unresolved. A basic version of these ideas has been implemented which operates within the Macintosh MINIEDIT, a simple interactive point-and-click editor (Chernicoff, 1984). This has a text window, a text cursor, and two pull-down menus, *file* and *edit*. Within the window the user can

- type, in which case text is inserted at the text cursor position;
- click, which moves the text cursor to the position of the mouse cursor;
- press and sweep, which selects (and highlights) a stretch of text;
- double click, which selects the word at the mouse cursor.

The *file* menu gives access to conventional file operations such as open, save, and quit; these are not recognized within the learning system. The *edit* menu allows

- cut, which deletes the currently-selected text and saves it in a buffer;
- paste, which pastes the contents of the buffer at the cursor position;
- undo, which revokes the last editing action.

The following subsections parallel those of Section 3 and give more details of the system's operation, illustrated on Task 1 of Appendix A, which involves reformatting a list of addresses.

4.1 RECORDING TRACES

Figure 2 shows a sequence of attributes and actions that constitute an editing trace. Actions include typing a string of characters, clicking to reposition the cursor, selecting a stretch of text (or a single word by double-clicking), cutting (either by menu selection or the delete key), and pasting. File operations are not considered to be actions. The system ignores extraneous actions such as consecutive mouse-clicks when recording a trace. The typing and selecting actions have a parameter that records the text string involved.

4.2 CREATING PROGRAMS

The program-creation mechanism generalizes the trace of Figure 2 into the procedure of Figure 3.

Nodes are considered to be mergeable whenever their actions (and parameters, where appropriate) are identical. For example, T1, T3, and T5 are mergeable, as are T2, T4 and T6. Whenever nodes are merged, loops are formed. However, mergeable nodes are only actually merged provided this does not create a branch into the body of an already-formed loop. For example, step T6 of the trace in Figure 2 is not merged with state S2 of Figure 3, since this would involve branching into a loop.

For select actions, the attributes are discarded. This assumes that the text selected is the only clue to the location of the action—an oversimplification for some tasks, but one that sidesteps the difficult problem of balancing constraints on the context against constraints on the selection parameter when executing such actions. Thus only for click actions do attributes actually need to be generalized, and this proceeds as follows.

Relative-position attributes. These are generalized by simply dropping the attribute if different values are encountered.

Absolute-position attributes. Within each unit (file, paragraph, line, word), the "beginning" and "end" values can be generalized to "extreme", while the "middle" value can only be generalized to "any". For example, `beginning-of-line` and `end-of-line` are generalized to `extreme-of-line`; if `middle-of-line` is present too then the position-within-line constraint is abandoned. Different absolute-position attributes are treated according to the word, line, paragraph, file hierarchy. For example, since `beginning-of-line` implies `beginning-of-word`, when both features are present the second is dropped.

Textual attributes. Character strings are generalized according to a heuristic that finds common subsequences. Individual characters are related by the hierarchy of Figure 1, and this relationship is extended in the obvious way to strings of consecutive characters from the same class. Then common subsequences are identified. For example, when trace elements 1 and 3 of Figure 2 are coalesced, the "before" strings "`Bix,¤`" and "`N.W.,¤`" must be combined.[1] First their common subsequence "`,¤`" is identified. Then the pattern "`[letter],¤`" is constructed from the common subsequence and the first string, where the square bracket notation indicates a sequence of characters from that class and other characters (comma and space) are interpreted literally. However, this does not match the second string, and so the pattern is generalized to "`[character],¤`" (abbreviated to "`[c],¤`" in Figure 3).

The heuristic method that combines textual attributes is biased towards discovering maximally specific patterns that characterize common subsequences of the character classes present in the strings. For example, the longest common subsequence of "`299-2299`" and "`222-8888`" is "`2...22...`", but the heuristic finds the pattern "`2...-....`" instead.

Example of generalization. An example of generalization is shown in Figure 4. The sequence generalizer determines that trace elements T1, T3, and T5 should be merged, and their attributes are combined into those for state S1 of the program in Figure 3.

---

[1]The symbol "¤" is used to make the space character visible.

4.3 EXECUTING AND EXTENDING PROGRAMS

Once a program has been created from the trace of actions on the first block of text, it is executed on subsequent blocks.[2] This is easiest to explain in terms of an example.

Consider applying the program of Figure 3 to the second block (in this case, line) of Task 1 in the Appendix. State S1 finds the first comma (after the name) and S2 suggests a RETURN character, which is accepted by the user. A position satisfying S1 (after "`Suite␣1,␣`") is found before one satisfying S3, and so S2 suggests another RETURN , also accepted. Now S1 finds the comma after "`Banff␣Blvd.`" and suggests a RETURN here. This is not what the user intended. He rejects the prediction and is invited to position the cursor correctly.

This is an example of an overgeneralization bug. These are handled by storing with each node "negative examples" (or patterns generalized from several negative examples) that indicate a mismatch with the node. For example, in the current circumstance the prior context "`Blvd.,␣`" and the posterior context "`N.W.,␣`" are recorded as negative examples with state S1 of the program. These will prevent S1 from matching a position with either of these contexts in future. Also, the context of the correct position indicated by the user is merged with that specified in the state to ensure that the correct position will indeed be chosen in the future. In this case the position's prior and posterior contexts are "`N.W.,␣`" and "`Calgary,␣`" which already match the patterns "`[character],␣`" and "`[character]`" respectively that are stored with the state.[3]

Continuing with the example, the program continues by predicting correctly the RETURN character following "`Calgary,␣`". This leaves it in state S2. Now it fails to identify a position for either of S2's successors, states S1 and S3. Because the pattern in S3, "`284-4983`", is specific, it is generalized up one level of the hierarchy of Figure 1, to "`[digit][op][digit]`", and a match is sought again. This does indeed find the "`229-4567`" at the end of the second line of the task, and so the program predicts that this item will be selected. (If a match had not been found, the pattern "`[digit][op][digit]`" would have been generalized further, to "`[alphanumeric][non-alphanumeric][alphanumeric]`", and so on.) When the user accepts this prediction, the select action of state S3 is generalized to "`2[digit]-4[digit]`" to accommodate the two patterns it has actually been applied to.

---

[2] In fact, it might have been better to undo the modifications and re-execute the procedure on the first block of text as well, both to provide an initial round of debugging and to show the user what has—and what has not—been learned. In the present case the procedure would fail on the first block in exactly the same way as it does on the second.

[3] The current system is blind to other negative examples that may occur prior to the target position specified by the user. Again, it may be better to undo and re-execute the program step to identify bugs early.

On the third and subsequent blocks of text up to the fifth, the system predicts all actions correctly. However, on the "`Helen¤Binnie`" line, where the telephone number "`(405)220-6578`" has a different format, it selects just "`220-6578`" and predicts a cut action. The teacher rejects this action and selects the whole string "`(405)220-6578`". Then the string generalization algorithm is re-invoked to generalize the pattern, which was previously "`2[digit]-[digit]`", to "`[character]2[digit]-[digit]`".

On the sixth block a RETURN is predicted immediately prior to "`S.E.,¤`", because only "`N.W.,¤`" is stored as a negative example. When the user rejects this prediction and indicates the correct position for the click action, the system generalizes the negative example pattern to match "`S.E.,¤`" too.

5. Evaluation

The program has been tested on the three tasks given in the Appendix. Despite the fact that many components have been implemented in a simplistic manner, it performed well on all tasks.[4] Table 2 summarizes the results.

On the address-list task (Task 1), manual execution of the procedure on the first block resulted in a trace of 8 steps (that of Figure 2), which was used to form a 6-node procedure (Figure 3). Execution of this procedure on the second block resulted in 10 editing actions, 9 of which corresponded to predictions accepted by the user. However, since a position selection and a corresponding action are both presented together to the user, these 9 predictions actually corresponded to only 5 dialog boxes being generated, and accepted. There were a total of three uses of the debugger (described in the previous section).

The second task was included since it has been studied by Nix (1983) as an illustration of his editing-by-example system. The trace was immediately generalized into the correct program.

The reference-list task (Task 3) was more complex. The second block differs from the first in that three authors are included instead of one, a volume number is present, and there is no month. The third block has two authors but no volume or page numbers. The final two blocks do not introduce any new cases. Nevertheless, despite this degree of variation, the great majority of predictions made by the system turned out to be correct.

The effort in performing the three tasks, measured in both keystrokes and mouse-clicks, was compared with and without the programming-by-example system. In Task 1 only one-third as many mouse-clicks, and one-third as many keystrokes, were required using the system. Task 2 produced even greater savings because without the learning system the fixed characters "`GameScore[¤winner¤']`" and "`;¤loser¤'`" were typed over and over again

---

[4]However, the tasks were used when developing and debugging the program, so this should not be construed as a proper test of the program's competence.

(although real users would likely be more resourceful). Task 3 required slightly over half as many mouse-clicks and keystrokes because the debugger had to be used to instruct the system to form branches and supply inputs. Of course, these figures depend heavily on the number of blocks that are edited, but they give some idea of the savings for tasks of this size.

6. Conclusions

Programming by example is a promising way of helping users with repetitive editing tasks involving semi-structured text. Although particularly suitable for non-programmers who have no alternative but to perform editing manually, it is probably useful for programmers too—even knowledgeable users in a UNIX environment with a rich set of software tools at their disposal often end up reformatting files manually.

The procedural approach adopted here, in which a trace of user actions is generalized into a program, offers significant advantages over a pattern-match-and-replace scheme such as Nix's (1983, 1984) that operates on input and output only. The disadvantage that traces are confusing because users have many ways of doing the same thing is alleviated by (a) using a simple, abstract model of editing instead of raw low-level commands, and (b) forming a program as soon as possible and using it to suggest actions to the user, thereby curtailing free variation in performance of the task. Executing the program on successive examples and "debugging" it where necessary provides a natural way to extend it incrementally, thus avoiding the need to think in advance about problem definition; this fits the procedural model well.

Our pilot implementation has numerous shortcomings. Embedded in a "toy" editor, it has no pretensions to being a usable software tool. A decision was taken to simplify each component as much as possible in order to get a working prototype. For instance, in *select* actions the attributes are discarded and notice is taken of the selected text alone; this means that one cannot select by context. More generally, no serious attempt has been made to resolve the general question of conflict between constraints imposed by different attributes; this deserves further study. There are important deficiencies in the program-construction method, for despite its attempt to forbid ill-structured programs, the node-merging policy is rather simplistic and merges nodes far too readily, constructing spaghetti-like programs that produce anomalous behavior in all but the simplest situations. For example, if Task 3 were extended with more examples of references having different formats (books with publisher and place of publication, papers in edited collections, etc.) the program construction method would effectively break down and useful predictions would cease to occur.

Some parts of the system rely on a lexical characterization of text that is essentially *ad hoc*. For example, attributes such as distance measures and position indicators are based on a division of the text into characters, words, lines and paragraphs. The string generalization method relies on a particular hierarchy of character classes. The definition of each of these units should really be sensitive to the type of text being edited.

Despite its shortcomings, the pilot implementation has demonstrated the viability of a procedural programming-by-example approach to repetitive text editing. It provides a substantial amount of assistance in the three example tasks, as shown in Table 2, by learning the essence of the procedure on the first block of text and executing it on the remaining blocks, with only minor debugging being necessary. However, it is not robust enough to support proper human factors tests of the efficacy of the procedural approach to semi-structured text editing with actual users.[5]

We are beginning a re-implementation in EMACS which is intended to overcome some of the problems noted above, and provide a useable editing tool. EMACS already includes comprehensive editing facilities and can be extended by writing LISP code. User interface aspects, which have consumed considerable effort in the current implementation, are already taken care of. Existing code to make lexical decisions (about what constitute words, paragraphs, etc.) is accessible via procedure calls. The editor can be placed in different "modes," and this makes lexical decisions sensitive to the type of text being edited. The fact that so much is already provided means that we will be able to concentrate more on programming-by-example aspects such as the question of identifying nodes to merge, alternatives for which are already being studied.

Acknowledgements

We gratefully acknowledge the key roles Dave Maulsby and Bruce MacDonald have played in helping us to develop and articulate these ideas, and the stimulating research environment provided by the Knowledge Science Lab at the University of Calgary. This research is supported by the Natural Sciences and Engineering Research Council of Canada.

References

Andreae, P.M. (1984) "Constraint limited generalization: acquiring procedures from examples," *Proc. American Association of Artificial Intelligence National Conference*, Austin, Texas; August.

Andreae, P.M. (1985) "Justified generalization: acquiring procedures from examples." Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT, Boston, Massachusetts.

Angluin, D. (1978) "On the complexity of minimum inference of regular sets," *Information and Control 39*: 337–350.

Chernicoff, S. (1984) *Macintosh revealed: Volume 2—Programming with the toolbox.*

---

[5]Testing systems which involve any kind of adaptation or learning is a major undertaking, as we have found from our experience of testing the METAMOUSE system for graphical programming by example (Maulsby *et al.*, 1989) and the PREDICT system for keyboard acceleration (Darragh & Witten, in preparation).

Darragh, J.J., Witten, I.H. and James, M.L. (in press) "The Reactive Keyboard: a predictive typing aid," *IEEE Computer.*

Darragh, J.J. and Witten, I.H. (in preparation) *The Reactive Keyboard.* To be published by Cambridge University Press, Cambridge, England.

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

Halbert, D. (1984) "Programming by example." Research Report OSD–T8402, Xerox PARC, Palo Alto, California.

Heise, R. (1989) "Demonstration instead of programming." M.Sc. Thesis, Department of Computer Science, University of Calgary, Canada.

Hirschberg, D. (1975) "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM 18*(6): 341–343.

MacDonald, B. A. & Witten, I. H. (1987) "Programming computer controlled systems by non-experts," *Proceedings of the IEEE SMC Annual Conference,* 432–437. Alexandria, Virginia.

Maulsby, D.L., Kittlitz, K.A., & Witten, I.H. (1989) "Metamouse: specifying graphical procedures by example," *Proceedings of ACM SIGGRAPH,* 127–136. Boston, Massachusetts.

Maulsby, D.L., James, G.A. and Witten, I.H. (1989) "Evaluating interaction in knowledge acquisition: a case study," *Proceedings of the European Knowledge Acquisition Workshop*, 406–419. Paris, France.

Michalski, R.S. (1983) "A theory and methodology of inductive inference," In *Machine Learning: an artificial intelligence approach*, edited by R.S. Michalski, J. Carbonell and T. Mitchell, pp. 83–134. Tioga, Palo Alto, California.

Nix, R. (1983) "Editing by example." Ph.D. Dissertation, Computer Science Department, Yale University.

Nix, R. (1984) "Editing by example," *Proc. ACM Symposium on Principles of Programming Languages*: 186–195. Salt Lake City, Utah; January.

Pence, J. & Wakefield, C. (1988). *Tempo II.* Affinity MicroSystems, Boulder, Colorado.

Stallman, R.M. (1981) "EMACS—the extensible, customizable, self-documenting display editor," *SIGOA Newsletter 2*(1/2):147–156; Spring/ Summer.

Waterman, D., Faught, W., Klahr, P., Rosenschein, S. and Wesson, R. (1986) "Exemplary programming: applications and design considerations." In *Expert systems: techniques, tools and applications*, edited by P. Klahr and D. Waterman, pp. 273–309. Addison-Wesley.

Witten, I.H. (1987) "Modeling behaviour sequences: principles, practice, prospects," *Future Computing Systems 2*(1): 55–81.

Witten, I.H. and Bell, T.C. (1990) "Source models for natural language text," *Int J Man-Machine Studies 32*(5): 545–579.

Appendix: Example tasks

TASK 1: ADDRESS LIST

Input

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

Output

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

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

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

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

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

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

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

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

TASK 2: SIMPLE DATABASE

Input

```
Cardinals 5, Pirates 2.
Tigers 3, Red Sox 1.
Red Sox 12, Orioles 4.
Yankees 7, Mets 3.
Dodgers 6, Tigers 4.
Brewers 9, Braves 3.
Phillies 2, Reds 1.
```

Output

```
GameScore[ winner 'Cardinals'; loser 'Pirates'; scores [5, 2]].
GameScore[ winner 'Tigers'; loser 'Red Sox'; scores [3, 1]].
GameScore[ winner 'Red Sox'; loser 'Orioles'; scores [12, 4]].
GameScore[ winner 'Yankees'; loser 'Mets'; scores [7, 3]].
GameScore[ winner 'Dodgers'; loser 'Tigers'; scores [7, 3]].
GameScore[ winner 'Brewers'; loser 'Braves'; scores [9, 3]].
GameScore[ winner 'Phillies'; loser 'Reds'; scores [2, 1]].
```

TASK 3: REFERENCES

Input

```
%A Abi-Ezzi, S.S.
%D 1986
%T An implementer's view of PHIGS
%J Computer Graphics and Applications
%P 12-23
%O February
%K *

%A Ackley, D.H.
%A Hinton, G.E.
%A Sejnowski, T.J.
%D 1985
%T A learning algorithm for Boltzmann machines
%J Cognitive Science
%V 9
%P 147-169
%K *

%A Addis, T.R.
%A Hinton, G.E.
%D 1987
%T A framework for knowledge elicitation
%J Proc First European Conference on Knowledge Acquisition
%O September
%K *

%A Allen, J.F.
%A Koomen, J.A.
%D 1986
%T Planning using a temporal world model
%J Artificial Intelligence
%O March
%K *

%A Allen, J.F.
%D 1983
%T Maintaining knowledge about temporal instances
%J Comm ACM
%V 26
%P 832-843
%O May
%K *
```

Output

Abi-Ezzi, S.S. (1986), "An implementer's view of PHIGS", Computer Graphics and Applications, pp. 12-23, February.

Ackley, D.H., Hinton, G.E. and Sejnowski, T.J., (1985), "A learning algorithm for Boltzmann machines", Cognitive Science, Vol. 9, pp. 147-169.

Addis, T.R. and Hinton, G.E. (1987), "A framework for knowledge elicitation", Proc First European Conference on Knowledge Acquisition, September.

Allen, J.F. and Koomen, J.A. (1986), "Planning using a temporal world model", Artificial Intelligence, March.

Allen, J.F. (1983), "Maintaining knowledge about temporal instances", Comm ACM, Vol. 26, pp. 832-843, May.

Captions for Tables and Figures

Primitive actions		**parameter**	**positional information**
	insert	<text>	
	locate		<attributes>
	select	<text>	<attributes>
	delete		

Attributes		**details**	
	context	before, after	
	lexical	beginning	file
		middle of	paragraph
		end	line
	relative distance		
		characters	
		words	
		lines	
		paragraphs	

Table 1 Summary of actions and attributes

	Block number	Actions performed	Predictions accepted	Uses of debugger	Nodes in procedure
Task 1	1 (trace)	8			6
	2	10	9	1	6
	3	8	8	0	6
	4	8	8	0	6
	5	8	7	1	6
	6	8	7	1	6
	the rest	8	8	0	6
Task 2	1 (trace)	11			13
	the rest	11	11	0	13
Task 3	1 (trace)	20			20
	2	26	18	3	28
	3	22	20	2	28
	the rest	20	20	0	28

Table 2 Performance of the system on three example tasks

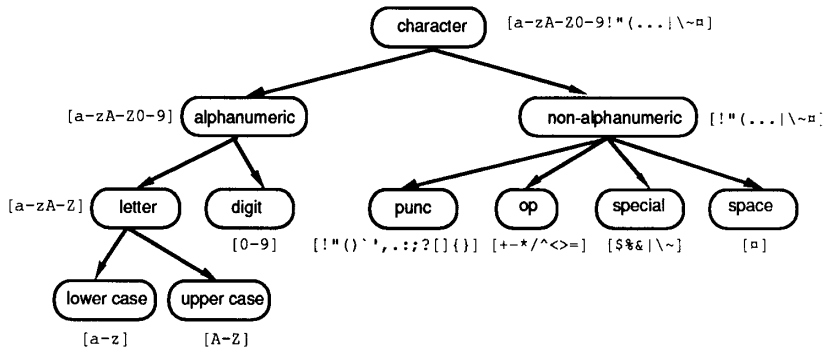

Figure 1 Hierarchy of character classes

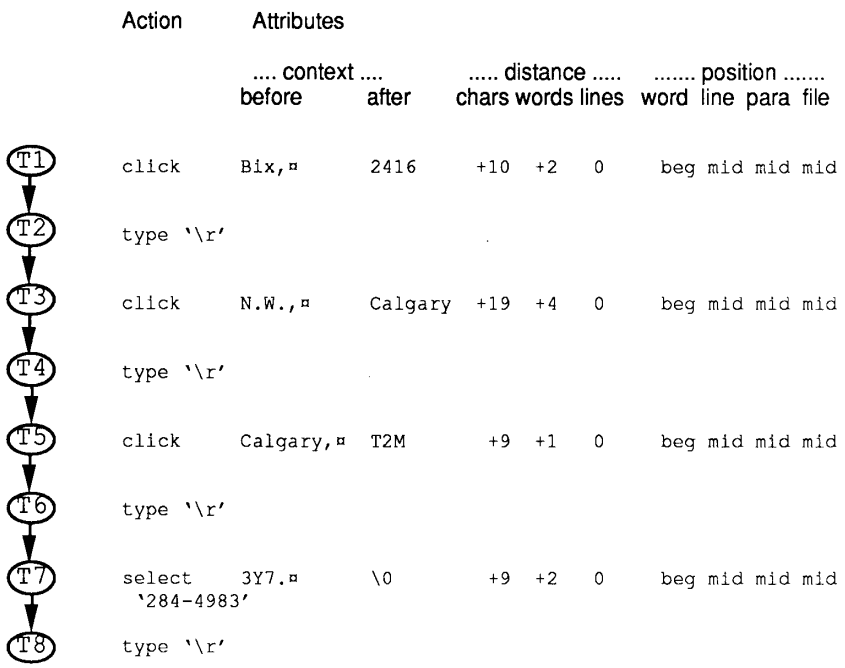

Figure 2 Trace for first block of address list task (Task 1)

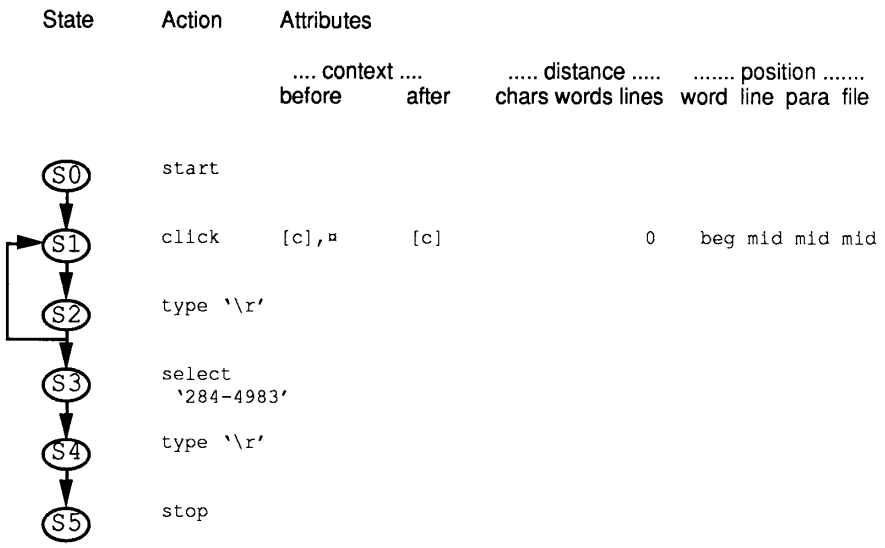

Figure 3 Program created from trace of Figure 2

		.... context		..... distance			..... position			
		before	after	chars	words	lines	word	line	para	file
T1	click	Bix,¤	2416	+10	+2	0	beg	mid	mid	mid
T3	click	N.W.¤	Calgary	+19	+4	0	beg	mid	mid	mid
T5	click	Calgary,¤	T2M	+9	+1	0	beg	mid	mid	mid
			⇓							
S1	click	[c],¤	[c]			0	beg	mid	mid	mid

Figure 4 Generalizing trace elements T1, T3, and T5 into program step S1