Editing by Example

ROBERT P. NIX
Yale University

An editing by example system is an automatic program synthesis facility embedded in a text editor that can be used to solve repetitive text editing problems. The user provides the editor with a few examples of a text transformation. The system analyzes the examples and generalizes them into a program that can perform the transformation to the rest of the user's text. This paper presents an overview of the design, analysis, and implementation of a practical editing by example system. It studies the problem of synthesizing a text processing program that generalizes the transformation implicitly described by a small number of input/output examples. A class of text processing programs called *gap programs* is defined and the problems associated with synthesizing them from examples are examined, leading to an efficient heuristic that provably synthesizes a gap program from examples of its input/output behavior. The editing by example system derived from this analysis has been embedded in a production text editor. By developing an editing by example system that solves a useful class of text processing problems, this work demonstrates that program synthesis is feasible in the domain of text editing.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Program Complexity**]: Nonnumerical Algorithms and Problems—*pattern matching*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages; H.2.3 [**Database Management**]: Languages—*data manipulation languages (DML)*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation*; H.4.1 [**Information Systems Applications**]: Office Automation—*word processing*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*; I.5.4 [**Pattern Recognition**]: Applications—*text processing*; I.5.5 [**Pattern Recognition**]: Implementation—*interactive systems*; I.7.1 [**Text Processing**]: Text Editing—*languages*

General Terms: Languages, Theory

Additional Key Words and Phrases: Automatic programming, gap patterns, grammatical inference, inductive inference, text editing

1. INTRODUCTION

An editing by example (EBE) system is an automatic program synthesis facility embedded in a text editor that can be used to solve repetitive text editing problems. The user provides the editor with a few examples of a text transformation. The EBE system analyzes the examples and generalizes them into a program that can perform the transformation to the rest of the user's text.

This research was supported by NSF grant MCS8002447. A preliminary version of this paper appeared in the *Conference Record of the 11th Annual Symposium on Principles of Programming Languages*, (Salt Lake City, Utah, January 1984). This paper summarizes the results of the author's Ph.D. dissertation, submitted to the Department of Computer Science, Yale University, New Haven, CT.
Author's current address: Multiflow Computer, Inc., 175 North Main St., Branford, CT 06405.

```
Recent research is said to have traced the earliest known use
of @i[O.K.] to the @i[Boston Morning Post] of 23 March 1839. It
was not until nearly a hundred years later that, greatly helped
by radio and television, it won its present popularity in England.
It is made to serve as an adjective (@i[That's O.K.]) and
occasionally attributive (@i[Advertising is in these days a
socially O.K. profession]); it supersedes the old formulas of assent
@i[Very well], @i[All right], and @i[Right oh], ... It has bred
a jocular variant @i[Okidokey].
```

Fig. 1. A document formatted using Scribe.

```
Recent research is said to have traced the earliest known use
of {\sl O.K.} to the {\sl Boston Morning Post} of 23 March 1839. It
was not until nearly a hundred years later that, greatly helped
by radio and television, it won its present popularity in England.
It is made to serve as an adjective ({\sl That's O.K.}) and
occasionally attributive ({\sl Advertising is in these days a
socially O.K. profession}); it supersedes the old formulas of assent
{\sl Very well}, {\sl All right}, and {\sl Right oh}, ... It has bred
a jocular variant {\sl Okidokey}.
```

Fig. 2. The document when changed to TEX.

This paper reports on a theory, a design, and an implementation of an EBE system that has been implemented within *U* [22], a production text editor. To demonstrate the system, suppose that a user of *U* wants to change a paper which has been formatted using the Scribe document formatting language [25], a fragment of which is shown in Figure 1, to use the TEX document formatting language [15], as shown in Figure 2. This text transformation has many steps, but for the sake of an example we will concentrate on the problem of changing all uses of Scribe's italic notation, `@i[O.K.]`, to TEX's slanted font notation, `{\sl O.K.}`. To use *U*'s editing by example facility to make this change, the user enters *U* and begins the EBE session by selecting, or marking, the first example:

```
@i[O.K.]
```

The user then issues a command to the editor that tells it that the selected text is an example of the sort of thing that should be changed; that is, this text is the sort of *input* that the transformation should affect. The user then manually transforms the text to TEX format, using the editor commands normally used to make the change on a single instance of the text:

```
{\sl O.K.}
```

Once the line has been changed, the user selects it and issues another command that informs the editor that the selected text is the *output*. That is, this line of text is the sort of thing the user would like the editing by example system to produce when it finds some text resembling the input. At this point, the system synthesizes a program that generalizes the transformation expressed by the example. The system's generalization of a single example is a trivial program that transforms all instances of the literal input text of the example to the literal

output text. Since the other occurrences that the user wants to change are not simply repetitions of @i[O.K.] another example is given by selecting and transforming @i[Boston Morning Post] in a similar way. The editing commands that are used when concocting this second example are not required to be the same as, or even similar to, the commands given when concocting the first example.

After analyzing the second example, the editor shows the user the synthesized program in a specialized notation for string search and transformation:

@i[-1-] ⇒ {\slu -1- }

This notation defines a simple program called a *gap program*. Gap programs have two parts: the part preceding the ⇒, which is called the *gap pattern*, and the part following the ⇒, which is called the *gap replacement*. The gap pattern is a string matching pattern composed of constants and variables that describes the format of the fragments of text that the user wants to change. The constants in the expression above are the characters in the typewriter font, like @i [and {\slu, which match their literal text (we use "␣" as a visible space character). The variables in the input pattern are signified by the numbers between hyphens. A variable (also called a *gap*) matches any sequence of characters up to the constant string that follows the variable in the pattern. In this pattern, the variable *-1-* matches the characters between an occurrence of @i [and the first] that follows.

Each of the elements of the output replacement is a constant string or a variable from the input pattern. Programs in this language execute by searching for some part of the user's text that matches the input pattern. When matching text is found, it is replaced with a concatenation of the constants of the output pattern together with those parts of the text that are matched by variables contained in the output pattern. For example, when the input pattern @i[-1-] is matched against the text fragment ... as an adjective (@i[That's O.K.]) and occasionally ..., it matches the text @i[That's O.K.] and binds the variable *-1-* to the text That's O.K. The matched text is then deleted and the fragments {\slu, That's O.K., and } are inserted in its place. The searching process is continued after the point of replacement, and the program stops when no matching text is found. The gap program synthesized by the EBE system will finish off this step of the Scribe/TEX conversion process, and the user can perform the rest of the steps either by using the editor or by starting a new session with the EBE system.

2. A PRACTICAL EBE SYSTEM

This paper proposes a practical design for an editing by example system. A practical EBE system is one that a user would turn to out of choice when faced with a text processing problem whose solution demands either programming or drudgery. A practical EBE system must satisfy several criteria: It must be powerful enough to write programs that can solve the text processing problems that the user encounters. It must be easy and natural to use, something that will be determined both by the engineering details of the user interface and by the requirements for information imposed by the system. And it must be efficient, both in computational terms and in the amount of information that it requires of the user.

Perhaps the simplest systems that possess some of these properties are the program transcription facilities, or keystroke macros, that are present in many text editors such as EMACS and Z [18, 29, 31]. A keystroke macro is a record of the sequence of editor commands given when solving a single instance of an editing problem; the stored sequence of commands may be reexecuted when the user is faced with the next instance of the problem. Keystroke macros are an effective tool, and there have been several experiments with applying the keystroke macro style of "programming by example" to other domains: Smith's Pygmalion system [28], Curry's PAD system for programming by abstract demonstration [5], Lieberman and Hewitt's Tinker system [16], and Halbert's work on a programming by example facility for the user interface of the Xerox Star [10]. However, we did not take the program transcription approach to building an EBE system. While we felt that a generalized program transcription system would be a useful text processing facility, we were interested in building an EBE system that tried to perform more ambitious generalizations of the behavior shown it.

Given this desire to perform ambitious generalizations, there are still an enormous number of different ways to design and build an EBE system. Our first stab at reducing the scope of the problem was to form a simple and not too restrictive model of the process carried out by an editing by example system. In our model, the goal of the EBE system is to find a *target program* that will solve the user's text processing problems. Toward this end, the EBE system collects *sample data* that describes the desired behavior of the target program and uses a *synthesis procedure* to map from the sample data to a runnable program. If the user is completely satisfied with the synthesized program, it can be run over and over again until through with the editing task. On the other hand, if the program is not satisfactory, the user can cause the system to create a better program by supplying more data to the synthesis procedure and beginning the process anew.

This view of editing by example raises several questions: What sort of programs does the EBE system synthesize? What sort of information does the user provide the EBE system? How does the system synthesize the programs from the information? What sort of interface does the user see? The answers to these four questions are closely interrelated, but we will attempt to treat them one at a time. We begin by discussing the kind of program that we will synthesize, and we then decide on the information upon which we base the synthesis. These two decisions greatly determine the structure of the system, and within that framework we then describe the development of an algorithm for text program synthesis. We close with a brief sketch of the user interface and a conclusion.

The decisions that we made in the design of the EBE system were motivated by a complexity-theoretic analysis of the difficulty of the problems encountered; however, length constraints force us to state these theoretical results without proof. The interested reader may find the proofs in the author's doctoral dissertation [22].

3. WHAT SORT OF PROGRAMS?

The goal of an EBE system is to synthesize programs that help a user transform the text in some regular manner. Text is an ubiquitous data structure that can be used in a natural way to represent almost anything, so it is possible that these

programs could be called on to perform arbitrary computation. However, in order to build an effective and practical EBE system, we restrict our attention to solving some of the typical problems encountered while editing text.

Many problems come to mind. The user might be performing a pattern directed scan and edit as was shown in the opening example. The user might be performing some knowledge-based function on his text, such as renumbering a list or changing digits like `9` to names of months like `September`. The user might be performing a specialized procedure on his text: sorting some lines, adding up columns of numbers, filling and justifying paragraphs, or performing the join of a database relation. Or the user might be manipulating the text as if it represented a more complicated data structure such as a program parse tree.

Although many problems come to mind, we wanted to make progress on a practical EBE system, and so we had to choose to concentrate on one class of them. We chose to concentrate on synthesizing programs that scan and edit text, as in the introductory example.

While scanning and editing problems can be solved using general purpose programs, research into automatic programming has not yet yielded a practical method for reliably, robustly, and efficiently synthesizing general purpose programs from example information. Thus the approach of adapting a general purpose program synthesis strategy to synthesize programs that just happen to be scanning text would probably not yield a practical system. The approach that we took instead was to consider synthesizing programs of limited power that are specialized to string scanning.

String scanning programs are often specified using a grammatical pattern matching notation. The study of grammatical inference is concerned with the problem of synthesizing patterns from examples; some surveys of grammatical inference include those of Biermann and Feldman [4], Fu and Booth [8], and Angluin and Smith [3]. Perhaps the two best known formalisms for describing the syntactic structure of text are regular expressions and context free grammars. Unfortunately, although many algorithms have been developed for synthesizing regular expressions, context free grammars, and their subclasses from examples (see Angluin and Smith's survey [3]), none of these algorithms performs well enough in terms of running time, amount of data required, and quality of hypotheses proposed to be used as the pattern synthesis component of a useful EBE system.

However, pattern matching formalisms less powerful than regular expressions have been shown to be quite useful in text processing applications (cf. POPLAR [19]). Guided by these systems, and by our own experience with building and using text processing tools [6, 7], we decided to build the EBE system around a text processing language of limited power whose programs could be effectively synthesized from examples. The programs in the limited language are called *gap programs*.

3.1 Gap Programs

Gap programs are the class of *pattern* ⇒ *replacement* programs introduced in the opening example; the *pattern* is a *gap pattern*, and the *replacement* is a *gap replacement*. Gap patterns bear a resemblance to Angluin's regular pattern

languages [2] and Shinohara's extended regular pattern languages [26, 27]. A *gap pattern* G over an alphabet Σ is a sequence of alternating *strings* and *gaps*, $s_0 g_1 s_1 g_2 s_2 \ldots g_n s_n$. The strings s_i are drawn from Σ^+ (although s_0 may be the null string, except when $n = 0$), the gaps g_i are distinct symbols drawn from a gap alphabet Σ_G that is disjoint from Σ, and the number of gaps n is greater than or equal to 0. The *constant subsequence* of a gap pattern, denoted $c(G)$, is the string $s_0 s_1 s_2 \ldots s_n$. Similarly, the *gap subsequence* of a gap pattern, denoted $g(G)$, is the string $g_1 g_2 \ldots g_n$.

Our first sample gap pattern is made up from a single constant string:

@i [O.K.]

This gap pattern matches the constant string @i [O.K.]. The second sample gap pattern contains a gap:

@i [*-1-*]

The characters *-1-* together make up a single gap symbol. This pattern will match strings that begin with @i [and end in], with the gap spanning the characters in between. The next gap pattern, which does not have a leading constant string:

-1-]

will match strings ending in]. Another example,

Dear␣*-1-*, *eol*
Congratulations␣ *-2-* !␣You␣have␣been␣selected

denotes a gap pattern that uses two gaps to match the first few lines of a form letter. The special constant *eol* matches the end of the line. Each gap symbol must be distinct; for example, the symbol *-1-* must occur only once in the gap pattern. This gap pattern matches a phone number:

(*-1-*)␣ *-2-* – *-3-*.

This text,

(*-1-*)␣ *-2-* – *-3-*

is not a gap pattern because gap patterns must be end in a constant string. This is also not a gap pattern,

(*-1-*)␣ *-2-* *-3-*.

because the gap symbols *-2-* and *-3-* must be separated in the pattern by some constant string.

When a gap pattern is matched against a piece of text, each of the gap symbols in the pattern is bound to the substring of the text that is matched by the gap. This substring is defined so that the matching process is deterministic; a gap symbol g_i that is followed in the gap pattern by a constant string s_i will match text only so long as it does not include s_i. Formally, define the set of *legal substitutes* for a gap g_i that is followed by the string s_i to be the set of all strings $\alpha \in \Sigma^*$ in which the leftmost occurrence of s_i as a substring in the string αs_i is at the end of αs_i. Then the language $L(G)$ defined by a gap pattern $G = s_0 g_1 s_1 g_2 s_2 \ldots g_n s_n$ is the set of all strings of the form $s_0 \alpha_1 s_1 \alpha_2 s_2 \ldots \alpha_n s_n$ where

each α_i is a legal substitute for g_i. A string s is *matched* by a gap pattern G if $s \in L(G)$; a set of strings S is matched by G if $S \subset L(G)$.

For example, when the gap pattern *-1-* `abc` is matched against `xyzabc`, the gap *-1-* matches `xyz` and the pattern matches the entire string. When the pattern matches the string `abc`, it binds the gap *-1-* to the null string. When it matches `ababc`, it binds *-1-* to `ab`. However, when the pattern is matched against the string `abcxabc`, it matches only the prefix `abc` and fails to match the entire string because the gap *-1-* is defined so as not to match a string that includes `abc`. Formally, define a *parse* of string $s \in L(G)$ relative to a gap pattern $G = s_0 g_1 s_1 g_2 s_2 \ldots g_n s_n$ to be a sequence of n strings $p_1, p_2, \ldots, p_n$ such that $s = s_0 p_1 s_1 p_2 s_2 \ldots p_n s_n$ and each p_i is a legal substitute for the corresponding g_i.

The gap pattern of the gap program matches text and parses it into those pieces that match the constants and those pieces that match the gaps. Then the *gap replacement expression* is used to compute the new string that will replace the string matched. A gap replacement expression R for a gap pattern G over an alphabet Σ with gap symbols $g_1, g_2, \ldots, g_n$ is a string from $(\Sigma \cup \{g_1, g_2, \ldots, g_n\})^*$. Gap replacement expressions are not interesting objects in isolation; they are only of interest when they have been combined with a gap pattern G into a gap program. A *gap program* P is a pair consisting of a gap pattern G and a replacement expression R for G; the program is denoted $G \Rightarrow R$. Here is an example of a gap program that will change the phone number `(203) 436-0715.` to `203-436-0715.`:

`(203) 436-0715.` ⇒ `203-436-0715.`

This gap program generalizes the transformation to apply to all phone numbers of that form:

-1-`(`␣*-2-*`-`*-3-*`.` ⇒ *-1-*`-`*-2-*`-`*-3-*`.`

This gap program replaces an area code with the word `Call`:

-1-`(`␣*-2-*`-`*-3-*`.` ⇒ `Call`␣*-2-*`-`*-3-*`.`

This gap program will delete a phone number entirely:

-1-`(`␣*-2-*`-`*-3-*`.` ⇒

This one will interchange the area code with the first three digits:

-1-`(`␣*-2-*`-`*-3-*`.` ⇒ *-2-*`(`␣*-1-*`-`*-3-*`.`

And this last program performs another nonsensical transformation, duplicating the digits of the number in a pleasant pattern:

-1-`(`␣*-2-*`-`*-3-*`.` ⇒ *-1-* *-2-* *-3-* *-3-* *-2-* *-1-*

The intuitive descriptions of the effects of these gap programs may be formalized as follows. If x and y are strings in Σ^* and $P = G \Rightarrow R$ is a gap program, then $P(x) = y$ if and only if G matches x yielding the parse $p_1, p_2, \ldots, p_n$ and if y is equal to R with p_i substituted for each occurrence of g_i.

Gap programs are a fairly weak text transformation language; some elementary properties of gap patterns and programs include:

(1) Gap patterns parse strings uniquely into constants and gaps.
(2) Gap patterns may be matched against text in linear time.
(3) Gap patterns define languages that are a proper subclass of the regular languages.
(4) The language generated by a gap pattern is either a singleton or infinite.
(5) The set of languages defined by gap patterns is not closed under union, intersection, or complement.
(6) Gap programs are not closed under composition; there are transformations computable by the composition of two gap programs that are not computable by a single gap program.
(7) (Due to Dana Angluin.) Given an arbitrary finitely specified function, that is, a finite collection of arbitrary input/output pairs, there exist two gap programs that can be composed together to transform each of the given inputs to the corresponding output.

4. SYNTHESIZING GAP PROGRAMS FROM I/O EXAMPLES

The user must have some way to tell the EBE system what he or she wants the target program to do. There are two easily garnerable sources of information about the target program: one is the sequence of commands that the user employed while editing an example, and the other is the appearance of the example text before and after the edit. The sequence of commands is called a *trace* of the target program, and the change in appearance is called the program's *input/output* behavior.

The information contained in the command trace can be made to include everything contained in the input/output samples, and more besides; so at first glance it seems obvious that an editing by example system should use traces as its principal source of information. However, there are some problems with using traces. The major problem is that traces are an unreliable source of information about the user's intent. For example, string search and cursor movement commands can often play the same role in moving on to the next example, and the string search command will communicate much more about the user's intent, but it is probably more likely that a cursor movement command will be used if the next example is visible on the screen. Another problem is that a system that uses traces might require the user to use similar commands in a similar order when given two examples, and such a requirement would run counter to the free-form nature of interactions in a well-designed editor. A final problem is that a system that generalizes from traces would have to be given some knowledge of the semantics of every editor command; requiring that this knowledge be embedded in the EBE system would inhibit the extensibility of the editor and would also probably reduce the portability of the EBE system design.

The problems associated with using traces as the principal source of information in the EBE system led us to concentrate our efforts on algorithms that work from input/output examples. Although input/output examples contain less information than traces, the information that is present in the input/output examples is not as susceptible to error.

Given that we are going to use input/output examples, how many is it reasonable to require that the user give in order to specify a program? While the amount of example data that a user can be imposed upon to provide is subject to many factors, such as the smoothness of the user interface, his knowledge of programming, and his mood, we suspect that people's tolerances are small. We suspect that five is too many examples to have to provide, that four is too many, that three is probably too many, and that two may well be too many. A single input/output example would be ideal. Unfortunately, a single example cannot impart the pattern that describes the text that the user would like to transform, and there is also not much of a basis for deciding on a nontrivial transformation that maps the input string of the single example to the single output. It is unreasonable to expect synthesis from a single example to yield useful programs.

The gap program synthesis algorithm that we have developed can usually converge to the target gap program after the user provides two or three examples of the target function's input, and we describe a heuristic in Section 6 that results in a single output example being all that is usually required.

An algorithm for gap program synthesis takes as input a certain number of input/output examples that describe the behavior of the text transformation that the EBE system user would like to perform. It analyzes these examples and proposes a gap program that can transform each of the inputs to the corresponding output.

The algorithm succeeds if it proposes a gap program that performs the desired transformation to the rest of the user's text. If the algorithm can identify an arbitrary gap program after being given adequate data, then we say that the algorithm *identifies the class of gap programs in the limit.* If the algorithm uses examples of what the target gap program does, then we say that it is working from *positive data,* whereas if it also uses examples of what it should not do, then we say that it is basing its synthesis on *positive* and *negative* data. These concepts were first introduced and studied by Gold [9], and they have since been explored by other researchers in inductive inference; Angluin and Smith give a good survey of work in inductive inference [3].

An algorithm that synthesizes gap programs in the limit from input/output examples could work by examining the set of samples given by the user and returning the "best" gap program that can perform the transformation shown. The best gap program can be defined so that as more and more examples are given to the system, the best program would be guaranteed eventually to be the program that the user had in mind. In an effort to understand the problems involved with building such a system, we investigated the complexity of various subproblems related to gap program synthesis. One of our findings was:

THEOREM 1. *The problem of deciding whether a set of input/output samples describes a transformation that can be effected by a gap program is an NP-complete problem. This problem remains NP-complete even when there are only three input/output pairs in the set.*

Thus an algorithm whose goal is to synthesize a gap program that can perform the transformation described by a set of input/output samples must solve an NP-hard problem. Moreover, this problem remains NP-hard even when there are

only three input/output pairs in the example set. This negative result coincided with our intuition that tackling the gap program synthesis problem monolithically was too hard.

However, we still wanted to build an EBE system that could synthesize gap programs. Toward this end, we finessed the problem by dividing and conquering, and decomposed the problem of synthesizing a gap program into two steps. The first step is to find the "best" gap pattern that matches all of the input strings in the sample set. This gap pattern yields a parse of the input samples, and the second step of the algorithm is to attempt to find a replacement expression that can rearrange the parsed inputs to yield the output samples.

This decomposed process differs from the process of finding a gap program as a whole in two respects. The first is that the decomposed process can be done more efficiently. We have found that although each of the two steps of the decomposed process is also NP-hard, they can be solved in time polynomial in the size of the input when the number of samples is bounded. We have developed efficient heuristics that the EBE system actually uses to perform each of the phases, and we have been able to prove that the heuristics still identify gap programs in the limit from positive data.

The second way that the decomposed process differs is that the first step, that of finding a gap pattern, is performed independently of the second step of finding a gap replacement. It may be that the gap pattern found in the first step parses the input strings in such a way that it is impossible for any replacement expression to rearrange the parsed fields to form the output. In this case, the algorithm terminates with the answer "more data required". We have been able to prove that the addition of new relevant data can make the decomposed process work, by making it find a gap pattern that parses the input so that a replacement expression can be found. The decomposed process gains efficiency over monolithic gap program synthesis by sometimes requiring the user to supply more data than a monolithic process would require; in practice, this penalty is rarely paid.

The gap program synthesis algorithm that we describe below finds the best gap pattern that matches the input samples, and then finds a replacement expression that can map the inputs to the outputs.

4.1 Descriptive Gap Pattern Synthesis

Our definition of a "best" gap pattern that matches a set of strings S is that the best gap pattern is one that finds the greatest number of common distinctive features in the set. We call such a pattern *descriptive*; a gap pattern G is a *descriptive gap pattern* for a set of strings S if:

(1) G matches all of the strings in S.
(2) G has the greatest number of constant symbols of any gap pattern that matches S.
(3) Of the patterns that satisfy the previous two constraints, G has the fewest number of gaps.

The first two criteria form the core of the definition; we want gap patterns that find the largest number of common constants in the sample data. The third criterion is intended to remove from consideration those gap patterns that find

all of the common constants, but contain extraneous gaps. The name "descriptive" is justified by a result that shows that a descriptive gap pattern for a set of strings defines a minimal gap pattern language that contains the strings. The following results help to characterize the difficulty of the descriptive gap pattern synthesis problem:

THEOREM 2. *Finding a descriptive gap pattern for a given set of strings is NP-hard.*

THEOREM 3. *There is an algorithm that can find a descriptive gap pattern for n strings of length at most l in time* $O(l^{3n+1} \log l)$.

The algorithm of Theorem 3 is of theoretical interest only, since its running time is not practical. We actually synthesize descriptive gap patterns using a combination of two heuristics: one that approximates the constant substring of the pattern, and one that tries to insert gaps into those constants to form a gap pattern.

(1) *Finding the Constants.* The constants are approximated using an algorithm that approximates the longest common subsequence (LCS) of a set of strings. The LCS of a pair of strings can be found in polynomial time using a variety of dynamic programming algorithms [11, 13, 30], although Maier showed that the problem is NP-hard when the number of strings in the set is not bounded [17]. We approximate the LCS of a set of strings $s_1, s_2 \ldots, s_n$ by first ordering the set from shortest string s_1 to longest string s_n, and then computing the iterative pairwise approximation $\mathrm{LCS}(s_n, \mathrm{LCS}(s_{n-1}, \ldots \mathrm{LCS}(s_2, s_1)))$. This heuristic seems to produce good results in practice. Our current implementation of the heuristic makes use of a pairwise LCS algorithm due to Hirschberg [11], and runs in $O(nl^2)$ time and linear space. As an example of its output, when the system is set to analyzing the strings:

```
@i[Boston Morning Post]
@i[Well all right]
@i[Right oh]
```

it finds the common constants:

```
@ i [ ␣ ]
```

In this case, these common constants are indeed a longest common subsequence of the samples. However, in this case there happen to be other common subsequences of equal length, and in general the heuristic is not guaranteed to be an exact LCS when the set contains more than two samples.

Our selection of the common constant "␣" may seem a little too fortuitous, since the samples all contained common constants `i`, `g`, `t`, and `o` that could have been chosen just as easily. We describe a tokenization heuristic in Section 6 that relies less on making such fortuitous selections.

(2) *Inserting the Gaps.* The next step of the descriptive gap pattern synthesis heuristic is to insert gaps into the constant string to make it into a gap pattern. We have not been able to classify the complexity of the gap insertion problem, although we suspect that the problem is NP-hard in general. However, we have

been able to find an algorithm that runs in polynomial time for a fixed number of strings:

THEOREM 4. *Given a constant string c and a set of n strings S, each of length bounded by l, there is an algorithm with running time* $O(l^{3n+4} \log l)$ *that determines the minimal number of gaps that have to be inserted into c to make it match S.*

This algorithm is also solely of theoretical interest. In practice, we perform gap insertion using a simple heuristic that does a leftmost match of the constants against the sample strings and inserts gaps where they are required. For example, the first three characters of the common subsequence, @i [, are matched against the first three characters of each sample. The samples do not all contain a ␣ as the fourth character, so a gap is required before the ␣ in the pattern, and a gap is also required before the]. This heuristic runs in time $O(nl^2)$ and results in the pattern:

@i [*-1-* ␣ *-2-*]

The constant synthesis and gap insertion heuristics can be combined into a heuristic algorithm that solves the gap pattern synthesis problem.

THEOREM 5. *The descriptive gap pattern synthesis heuristics identify the gap pattern of a target program in the limit from the data given in the input examples.*

4.2 Replacement Expression Synthesis Algorithm

Once a descriptive gap pattern is found that can describe the structure of the input strings, we must then find a way to produce the corresponding output strings using that structure. For example, if these three lines were our three input samples:

```
@i[Boston Morning Post]
@i[Well all right]
@i[Right oh]
```

then the following descriptive gap pattern would be found:

@i [*-1-* ␣ *-2-*]

The first gap in the pattern matches the string `Boston` in the first sample, `Well` in the second, and `Right` in the third, and so on:

```
 -1-          -2-
Boston    Morning␣Post
Well      all␣right
Right     oh
```

This collection of input fragments is called the input sample parse. The replacement expression synthesis problem is that of taking the fragments of text matched by gaps from the input, and a collection of output samples, say:

```
{\sl Boston Morning Post}
{\sl Well all right}
{\sl Right oh}
```

and finding a replacement expression that will produce each of the outputs from the corresponding input parse.

This problem is equivalent to that of deciding whether the output samples can be tiled with the columns from the input parse and with new columns of constants chosen from the alphabet. While this example does not make the problem appear difficult, synthesizing a replacement expression from the given fragments is in general a difficult task:

THEOREM 6. *The problem of synthesizing a replacement expression that maps the parse of a given input sample set to a given set of outputs is NP-hard.*

However, our data does not seem to exercise the features that make the problem intractable, and in practice we solve this problem exactly, without recourse to heuristics or approximations.

The algorithm for finding a replacement expression has two phases. The first phase constructs a finite automaton for each input/output example that describes all of the different replacement expressions that can rearrange the particular input example to yield the particular output example. The next phase of the algorithm finds a single replacement expression that can simultaneously produce all of the output strings by intersecting the automata derived in the first step. The classical finite state machine intersection algorithms can be used [12], and a replacement expression that simultaneously produces each of the outputs from the corresponding input can be recovered by finding a path from the start state of the intersected automaton to the accepting state.

For example, if the two sample input/output pairs were

```
abxbay    ⇒ ababa
cddxddcy  ⇒ addcddc
```

then the descriptive gap pattern matching the two inputs would be *-1-* x *-2-* y. When this pattern is matched against the first input, it parses that input into two gaps, ab and ba. The algorithm then constructs the machine in Figure 3 that represents all possible ways of writing the first output using those inputs and the constants a and b. The next step is to construct a similar machine to describe the second output, shown in Figure 4.

In the second stage, the algorithm intersects these two machines, which results in a machine, shown in Figure 5, that encodes all possible ways that the constraints of both input/output pairs can be simultaneously satisfied. This particular machine generates only one string, the string a *-2-* *-2-*, which corresponds to the single replacement expression that can perform the transformation. The gap program that transforms the inputs to the outputs is then *-1-* x *-2-* y ⇒ a *-2-* *-2-*.

The finite state machines of the first phase can be built in time proportional to the lengths of the output strings multiplied by the number of gaps in the input pattern. If l is a bound on the length of the output strings, then a particular gap from the input can occur at no more than l different points in the output. Thus the machines constructed in the first phase of the algorithm have no more than l states and $O(|g(G)|l)$ transitions. The worst case running time of the second phase can be proportional to the product of the sizes of the machines constructed in the first phase, and so we have shown

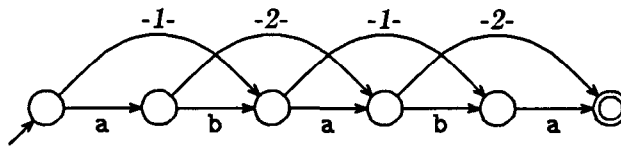

Fig. 3. Finite state machine for the output sample ababa.

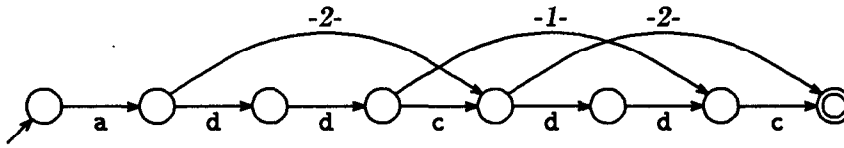

Fig. 4. Finite state machine for the output sample addcddc.

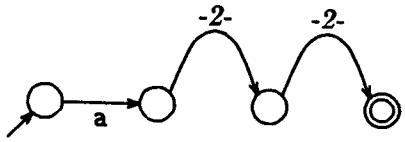

Fig. 5. The intersection of the machines in Figures 3 and 4.

THEOREM 7. *A replacement expression that produces n output strings by mapping* $|g(G)|$ *gap fragments taken from n input strings of length at most l can be constructed in time* $O(|g(G)|^n l^n)$.

This problem is NP-hard, which implies that we probably should not expect to improve on this worst case performance by very much; however, the finite state machine intersection algorithm seems to perform quite well in practice. In practice, the machines constructed in the first phase of the algorithm are long and skinny, because the string contained in a particular gap usually does not occur in the output in very many places. The intersection of two of these skinny machines M_j and M_k can be implemented to run in time roughly proportional to the size of the resulting machine $M_j \cap M_k$, and the intersection is a machine that is usually skinnier than either M_j or M_k. So this algorithm performs well in practice, running in time closer to $O(nl)$ than $O(|g(G)|^n l^n)$, and in fact this is exactly the replacement expression synthesis algorithm that is used by the EBE system. The following can be shown to be true:

THEOREM 8. *The replacement synthesis algorithm converges to the target replacement expression once the pattern synthesis heuristic has found the target pattern.*

THEOREM 9. *The descriptive gap pattern synthesis heuristic and the replacement expression algorithm together make up a gap program synthesis heuristic that can identify gap programs in the limit from positive data.*

5. DESCRIPTIVE GAP PROGRAM SYNTHESIS PERFORMANCE

The gap program synthesis algorithm that we have sketched above can be shown to identify gap programs in the limit from positive data; however, it would be nice to have some assurance that it will do so within the limit of the user's patience. Unfortunately, this assurance is impossible to come by, because the speed with which the system converges to a target gap program depends upon the quality of the sample data provided by the user.

The identification of the gap pattern is the part of the EBE process that is most susceptible to variations in the quality of sample data, since the replacement synthesis algorithm computes the gap replacement exactly. In an attempt to gain a better understanding of how well the pattern synthesis algorithm performs on a small amount of data, say two or three examples, we studied the algorithm's performance on several different sets of randomly generated test data. The trends found in the study may be simply summarized: fewer gaps, longer constants, and shorter gap substitutes make target gap patterns easier to identify; more gaps, shorter constants, and longer gap substitutes makes them harder.

These studies were helpful in identifying a common aspect of sample data that slows the gap pattern synthesis procedure's convergence to the target pattern. As an example, these two input samples might be given by the user in an effort to make the system synthesize the target pattern @i [*-1-*]:

```
@i[Boston Morning Post]
@i[Well all right]
```

The descriptive gap pattern for this sample set is @i [*-1-* ␣ *-2-* ␣ *-3-*], and the system will not converge to the target program until the user supplies an example that does not contain a ␣. However, the system will still be able to synthesize a useful gap program, albeit a noisy one, because the pattern fragment *-1-* ␣ *-2-* ␣ *-3-* matched exactly the same text in this particular sample set as the gap *-1-* does in the target pattern. We say that patterns like @i [*-1-* ␣ *-2-* ␣ *-3-*] are *strongly compatible* with patterns like @i [*-1-*] on a given set of samples. Patterns that are strongly compatible with the target are tolerable hypotheses, because the system will still be able to find a replacement expression that can produce the outputs, and it will thus be able to synthesize a useful gap program.

When the system fails to find the target gap program, it usually manages to find one with a strongly compatible pattern; however, the system does occasionally fail in more serious ways. The most common of these is when it is unable to find any gap pattern at all to match the sample data; from experience, such failures usually occur because the user is asking the system to perform a task that cannot be handled by gap programs. Another, less common, manner of failure is for the system to find a gap pattern that is not strongly compatible with the target. Such patterns usually do not parse the input samples in a way that lets them be transformed to the output samples; so while the system has been able to generate a gap pattern, it will not be able to generate a replacement expression. Adding another input example will often allow the system to converge to the target gap pattern, or to one that is strongly compatible with the target.

6. DESCRIPTIVE GAP PROGRAM SYNTHESIS HEURISTICS

The EBE system uses four heuristics to augment the basic gap program synthesis algorithm. The first two heuristics help to make strongly compatible gap patterns more like the target pattern, the third makes the synthesized gap patterns more "reasonable," and the last reduces the number of output examples that are normally required to specify a replacement expression.

Tokenization. Tokenization is a heuristic whereby the system performs an a priori grouping of the characters of the samples so that it will not, for example, notice that two samples such as `Boston Morning Post` and `Right oh`, share the common characters `i`, `g`, and `o`. The tokenization heuristic that is currently in use forms tokens out of runs of alphabetic characters and runs of numeric characters and leaves other characters to form single character tokens; for example, the 19 characters in `Boston␣Morning␣Post` are viewed as consisting of the five tokens: `Boston`, `␣`, `Morning`, `␣`, and `Post`. This particular tokenization scheme is a completely arbitrary choice. If the system cannot find a transformation that works with respect to the tokenization, it reanalyzes the samples using a character-at-a-time tokenization.

Pattern Reduction. Pattern reduction is a heuristic for making descriptive gap patterns less descriptive by trying to convert a strongly compatible pattern to the target pattern. The pattern reduction heuristic examines a gap program for blocks of constants and gaps that are copied en masse into the replacement expression and coalesces such blocks into a single gap as long as the resulting program will still transform the examples. For example, the gap program @i [*-1-* ␣ *-2-* ␣ *-3-*] ⇒ { \sl␣ *-1-* ␣ *-2-* ␣ *-3-* } would be pattern reduced to @i [*-1-*] ⇒ { \sl␣ *-1-* }, so long as the new gap program can still parse and correctly transform the examples. The heuristic treats runs of gaps that do not occur in the replacement expression, that is, those whose text is being deleted, as though they belonged to contiguous blocks.

Gap Bounding. Gap bounding limits the number of end-of-line boundaries that a gap is allowed to cross while matching the text of a file, in order to keep the program from running amok and matching a thousand lines when it matched only two lines in the examples. If a particular gap spans at most l end-of-line boundaries in any input sample, then the system restricts it to spanning no more than $\lfloor 1.5l \rfloor$ end-of-line boundaries when searching the text.

Unmatched Input Samples. This heuristic reduces the number of output samples that are needed by removing the requirement that every input sample be matched with a corresponding output sample. Using this heuristic, the system analyzes all of the input samples and produces a descriptive gap pattern. It uses this pattern to parse those inputs that have a corresponding output, and then generates the class of replacement expressions that can produce the outputs from the parsed inputs. If there is more than one replacement expression in the set, then it chooses the shortest, which is usually the one with the largest number of gap symbols. This heuristic is important because the input examples are probably already present in the user's text and can be provided to the EBE system simply

by selecting them and giving a single command, while the output examples are not already present and each one usually has to be created either from scratch or by modifying an input example. This heuristic thus significantly reduces the amount of effort involved in providing examples to the EBE system.

7. DESCRIPTIVE GAP PROGRAM SYNTHESIS

These four heuristics can be combined with the gap pattern and replacement synthesis procedures to form a practical gap program synthesis procedure. This procedure is invoked whenever the EBE system's current program hypothesis fails to perform the function specified by a given input or output sample. The algorithm (Figure 6) can usually synthesize a target gap program based on the information contained in two or three input examples and one output example.

We demonstrate the algorithm with an example; many more examples may be found in the full paper [20]. In this example, the user would like to take a program filled with LISP function definitions:

```
(define (factorial n)
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
(define (halts f)
  . . . )
```

and insert a "comment template" before each function:

```
;*** (factorial n)
;*** ----------------------------------
;*** (perspicuous description here)
;***
(define (factorial n)
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
;*** (halts f)
;*** ----------------------------------
;*** (perspicuous description here)
;***
(define (halts f)
  . . . )
```

To specify this transformation to the EBE system, the user chooses to give the two `(define` lines as input examples and the comment template for `factorial` as an output example. (The user could have chosen to give the full text of the function bodies as input examples; this would have worked, but the synthesized programs would have been noisier, e.g., because the function bodies both contain many right parentheses.) The system tokenizes the two input examples:

bol (define ␣ (factorial ␣ n) *eol*
bol (define ␣ (halts ␣ f) *eol*

finds the constants that they have in common:

bol (define ␣ (␣) *eol*

and then inserts gaps to form a descriptive gap pattern G:

bol (define␣(*-1-* ␣ *-2-*) *eol*

It tokenizes the single output example in the same way, parses the inputs using

Inputs:
  a set of input/output pairs $S = \{\langle i_1, o_1\rangle, \langle i_2, o_2\rangle, \ldots, \langle i_n, o_n\rangle\}$;
  a set of unpaired inputs $I = \{i_{n+1}, i_{n+2}, \ldots, i_{n+m}\}$;
  and a tokenization function T.
Output:
  a gap program P or an indication of failure (not shown).

Tokenize the samples in S and I using T;
Approximate a descriptive gap pattern G common
  to $\{i_1, i_2, \ldots, i_n, i_{n+1}, i_{n+2}, \ldots, i_{n+m}\}$;
Use G to parse $\{i_1, i_2, \ldots, i_n\}$;
Synthesize the shortest replacement expression R that
  maps i_k to o_k for $k = 1, 2, \ldots, n$;
Perform pattern reduction on G and R;
Bound the gaps of G;
Return $P = G \Rightarrow R$;

Fig. 6. The gap program synthesis algorithm.

the gap pattern, and builds a finite automaton that describes all possible replacement expressions that can produce the single output from the first input. It chooses the shortest such replacement expression R (preferring to use gaps in lieu of constants when there is a conflict:

```
;***␣( -1-␣-2- ) eol
;***␣---------- eol
;***␣(perspicuous␣description␣here) eol
;*** eol
(define␣( -1-␣-2- ) eol
```

The pattern reduction heuristic merges the fragment *-1-* ␣ *-2-* into a single gap *-1-* in both G and R, and the gap bounding heuristic limits this gap to matching characters within a single line, yielding the gap program:

```
bol (define␣( -1- ) eol
        ⇒
;***␣( -1- ) eol
;***␣---------- eol
;***␣(perspicuous␣description␣here) eol
;*** eol
(define␣( -1- ) eol
```

This gap program will serve to insert comment templates in the rest of the user's text.

8. IMPLEMENTATION

The gap program synthesis algorithm that we have developed is embedded within the EBE subsystem of a screen editor called *U* [22]. *U* is a full function screen editor implemented within the T programming system [23, 24], a LISP-like programming environment. The principal implementation of *U* is on the Apollo Domain MC68000-based workstation, although implementations also exist for VAX/Unix and VAX/VMS. (Domain, MC68000, Unix, VAX, and VMS are trademarks of Apollo Computer, Motorola, AT&T Bell Laboratories, DEC, and DEC, respectively.) The user interface of the *U* editor is similar to that of Wood's Z [31], which in turn was inspired by the work of Irons [14]. Perhaps the most

important feature of *U* is its general purpose undo command, which enables fearless use of the EBE subsystem.

The *U* user interacts with the EBE subsystem using five *U* commands:

(1) a command that initializes an EBE session, which creates an EBE session window or clears the existing one of any previously given samples;
(2) a command that specifies that a selected piece of text is an input sample;
(3) a similar command that specifies that some text is an output sample, which will usually be paired with the last input sample supplied;
(4) a command that runs the current gap program hypothesis, either by single-stepping it or by applying its transformation to a selected context;
(5) and a command that allows the user to modify the state of the EBE subsystem: fixing or deleting examples, specifying gap programs by hand, etc.

The EBE subsystem commands are implemented in the style of the rest of the editor commands. As one example, the standard editor selection mechanism is used to specify examples. Another example is that the user interface of the command for running the gap program closely resembles that of the editor's querying global-replace command.

The EBE subsystem invokes the gap program synthesis procedure every time the state of the example set changes, that is, every time an input or output example is given. The synthesized program is immediately displayed in a window, so the user gets immediate feedback about whether adequate samples have been given. This arrangement is feasible because the gap program synthesis procedure is quite efficient: it usually takes a second or two of elapsed time to produce a program (in an untuned implementation). If the gap program synthesis operation were more costly, it would probably have been better to have the user invoke it explicitly with a separate command.

9. CONCLUSION

The general form of the EBE system sketched in this paper was determined by a sequence of design decisions; these decisions, in order of importance, were

- to try to develop a useful and practical system for automating repetitive text processing tasks;
- to automate the tasks through a program synthesis system, rather than through a novel user interface to a program transcription system;
- to take a formal approach to solving this problem, rather than, for instance, a knowledge-based approach;
- to concentrate on automating the solution to problems solvable by simple text scanning and replacement programs;
- to develop a system that would base its hypotheses on positive data, rather than taking advantage of negative data as well;
- to base the system's analyses on the input/output behavior of the target function, rather than on traces or other sources of information;
- to require more than one example of the target function's behavior in order to form interesting generalizations, rather than trying to intuit interesting generalizations from a single example;

- to use formal language style patterns in the text scanning programs, rather than using control-structure oriented pattern matching as in SNOBOL;
- to use gap patterns to describe the structure of the text to transform;
- to use gap replacement expressions to describe how to perform the transformation;
- and to use a heuristic gap program synthesis procedure, rather than one that always guarantees to find a gap program concomitant with the demonstrated behavior.

The primary contribution of this work lies in demonstrating the feasibility of program synthesis in the domain of text editing. We developed, analyzed, and implemented an editing by example system and embedded it in a production text editor. The system seems to be an effective aid in automating the solution of a useful class of text processing problems.

Most of the credit for this success should go to the choice of the domain. Text editing is an interactive activity that is oriented around the incremental and (usually) unstructured manipulation of a large collection of data. Small-scale text processing problems constantly crop up during the course of these manipulations, and many of these problems can be solved by simple, syntactically oriented text processing programs. A few examples suffice to specify a good fraction of these programs, and the text editing environment makes these examples easy to produce and provide. The text editing domain is ideal for programming by example research.

Another contribution of this work is in providing an application for the techniques of inductive inference, an area of reseach that has seen a great deal of theoretical development but heretofore has had very few applications. This application, and the others that hopefully will follow, may help to focus inductive inference research on addressing problems of practical importance.

This work indicates a direction for program synthesis research: to find and develop applications for the programs that lie within the range of the program synthesis techniques that have been developed. If such research proves fruitful, it may spur further development in this area, which may help the field to evolve toward the eventual goal of automating the programming process.

The greatest weakness of this work is that the EBE system has not been used by a large community because the *U* editor did not become generally usable until this project was nearly complete (it is quite usable now). The design and evaluation of the system is based on the author's personal experience with building, using, and supporting text processing tools; while this experience is not inconsequential, it still represents only one man's view. It would have been better to have had more feedback on the system, both from knowledgeable programmers and from naive word-processors.

Another weakness is that gap programs are not powerful enough to express the solution of many text processing problems. The author's doctoral dissertation considers some extensions to gap programs [20], but while these extensions increase the capabilities of the system, it is clear that much more sophisticated programs cannot be derived from a few input/output examples. Different approaches must be taken. The future of this research lies in studying other ways in which the power of programming can be brought smoothly out into the user interface.

ACKNOWLEDGMENTS

This work benefitted from many discussions with Dana Angluin, my thesis advisor at Yale. I would also like to thank the other two members of my reading committee, Michael Fischer and Alan Perlis. The *U* editor was written by the author, but Nat Mishkin is largely responsible for the current status of its implementation on the Apollo. I gratefully acknowledge the editorial advice of Judy Martel and Mary-Claire van Leunen.

REFERENCES

1. AHO, A. V., KERNIGHAN, B. W., AND WEINBERGER, P. J. AWK—A pattern matching and scanning language. *Softw. Pract. Exper. 9*, 4 (April 1979), 267–280.
2. ANGLUIN, D. Finding patterns common to a set of strings. *J. Comput. Syst. Sci. 21* (1980), 46–62.
3. ANGLUIN, D., AND SMITH, C. A survey of inductive inference: theory and methods. Res. Rep. 250, Dep. Computer Science, Yale University, Oct. 1982.
4. BIERMANN, A. W., AND FELDMAN, J. A. A survey of results in grammatical inference. In *Frontiers of Pattern Recognition*, Academic Press, New York, 1972.
5. CURRY, G. Programming by abstract demonstration. Ph.D. dissertation, University of Washington, 1977; also appeared as Computer Science Department Tech. Rep. 77-08-02.
6. ELLIS, J. R., MISHKIN, N. W., NIX, R. P., AND WOOD, S. R. A BLISS programming environment. Res. Rep. 231, Dep. Computer Science, Yale University, June 1982.
7. ELLIS, J. R., MISHKIN, N. W., VAN LEUNEN, M.-C., AND WOOD, S. R. Tools: An environment for timeshared computing and programming. *Softw. Pract. Exper.* (Oct. 1983).
8. FU, K. S., AND BOOTH, T. L. Grammatical inference: Introduction and survey, parts 1 and 2. *IEEE Trans. Syst. Man, Cybern. SMC-5* (1975), 95–111 and 409–423.
9. GOLD, E. M. Language identification in the limit. *Inf. Control 10* (1967), 447–474.
10. HALBERT, D. C. An example of programming by example. M.S. thesis, Dep. Electrical Engineering and Computer Sciences, University of California at Berkeley, 1981; also an internal report of Xerox Office Products Division, Palo Alto, Calif., 1981.
11. HIRSCHBERG, D. A linear space algorithm for computing maximal common subsequences. *Commun. ACM 18*, 6 (June 1975), 341–343.
12. HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.
13. HUNT, J. W., AND SZYMANSKI, T. G. A fast algorithm for computing longest common subsequences. *Commun. ACM 20*, 5 (May 1977), 350–353.
14. IRONS, E. T., AND DJORUP, F. M. A CRT editing system. *Commun. ACM 15*, 1 (Jan. 1972), 16–20.
15. KNUTH, D. E. *The TeXbook.* Addison-Wesley, Reading, Mass., 1984.
16. LIEBERMAN, H., AND HEWITT, C. A session with Tinker: Interleaving program testing with program design. In *Conf. Rec. 1980 LISP Conf.* (Stanford University, 1980), 90–99.
17. MAIER, D. The complexity of some problems on subsequences and supersequences. *J. ACM 25* (1978), 322–336.
18. MEYROWITZ, N., AND VAN DAM, A. Interactive editing systems: Part II. *ACM Comput. Surv. 14*, 3 (1982), 353–415.
19. MORRIS, J. H., SCHMIDT, E., AND WALDER, P. Experience with an applicative string processing language. In *Proc. 7th ACM Conf. Principles of Programming Languages* (Jan. 1980), 32–46.
20. NIX, R. P. Editing by example. Ph.D. dissertation, Dep. Computer Science, Yale University, 1983; also appeared as Yale University Dep. Computer Science Res. Rep. 280.
21. NIX, R. P. Editing by example. In *Proc. 11th ACM Symp. Principles of Programming Languages* (Salt Lake City, Jan. 1984), 186–195.
22. NIX, R. P., AND MISHKIN, N. W. *U* Editor user's and programmer's manual. Internal memo., Dep. Computer Science, Yale University, 1983.
23. REES, J. A., AND ADAMS, N. I. T: A dialect of Lisp or, lambda: the ultimate software tool. In *Proc. 1982 ACM Symp. Lisp and Functional Programming* (Aug. 1982).

24. REES, J. A., AND ADAMS, N. I. T user's manual. Internal memo., Dep. Computer Science, Yale University, 1982.
25. REID, B. K. A high-level approach to computer document formatting. In *7th ACM Symp. Principles of Programming Languages* (Jan. 1980), 24–31.
26. SHINOHARA, T. Polynomial time inference of extended regular pattern languages. In *Proc. Software Science and Engineering of Computer Science* (Kyoto, Japan, 1982).
27. SHINOHARA, T. Polynomial time inference of pattern languages and its application. In *Proc. 7th IBM Symp. Mathematical Foundations of Computer Science*, 1982.
28. SMITH, D. C. Pygmalion: A computer program to model and stimulate creative thought. Ph.D. dissertation, Computer Science Dep., Stanford University, 1975; also appeared as AIM-260, June 1975, and as a book from Birkhauser Verlag, 1977.
29. STALLMAN, R. M. EMACS, the extensible, customizable, self-documenting display editor. In *Proc. ACM SIGPLAN/SIGOA Symp. Text Manipulation* (Portland, Ore., June 1981), 147–160. The conference proceedings appeared as *SIGPLAN Notices*, vol. 16, no. 6, June 1981.
30. WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *J. ACM 21* (1974), 168–173.
31. WOOD, S. R. Z: The 95% program editor. In *Proc. ACM SIGPLAN/SIGOA Symp. Text Manipulation* (Portland, Ore., June 1981), 1–7. The conference proceedings appeared as *SIGPLAN Notices*, vol. 16, no. 6, June 1981.

Received April 1984; revised May 1985; accepted May 1985