This research was supported by the National Science Foundation, grant No. MCS8002447.

Editing by Example

August, 1983

Research Report 280

Robert Peter Nix

Author's current address: Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

This report reproduces a dissertation presented to the Faculty of the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy, December, 1983.

ABSTRACT

Editing by Example

Robert Peter Nix

Yale University, 1983

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 dissertation presents the design, analysis, and implementation of a practical editing by example system. In particular, we study the problem of synthesizing a text processing program that generalizes the transformation that is implicitly described by a small number of input/output examples. We define a class of text processing programs called *gap programs*, characterize their computational power, study the problems associated with synthesizing them from examples, and derive an efficient heuristic that provably synthesizes a gap program from examples of its input/output behavior.

We evaluate how well the gap program synthesis heuristic performs on the text encountered in practice. This evaluation inspires the development of several modifications to the gap program synthesis heuristic that act both to improve the quality of the hypotheses proposed by the system and to reduce the number of examples required to converge to a target program. The result is a gap program synthesis heuristic that can usually synthesize a target gap program from two or three input examples and a single output example.

The editing by example system derived from this analysis has been embedded in a production text editor. The system is presented as a group of editor commands that use the standard interfaces of the editor to collect examples, show synthesized programs, and run them. By developing an editing by example system that solves a useful class of text processing problems, we demonstrate that program synthesis is feasible in the domain of text editing.

Acknowledgements

I owe a great debt to my advisor, Dana Angluin. I thank her for her guidance, her keen eye for correctness, her willing ear, and her friendship.

Alan Perlis has been a constant source of ideas during my years here at Yale; I hope someday to have explored a noticeable fraction of them.

I thank Michael Fischer, the third member of my committee, for encouraging some of the theoretical developments.

This work was supported by National Science Foundation grant MCS8002447. Computing facilities were provided under the good auspices of John O'Donnell, director of the Yale Computer Science Department Computing Facility.

I received most of my education in Computer Science from my peers, the Tools group: John Ellis, Nat Mishkin, Steve Wood, and later Mary-Claire van Leunen. It is difficult to express the joy of working with such people; I can only hope for an opportunity to refresh that memory.

U, the screen editor in which the editing by example system has been implemented, was originally written by the author; however, Nat Mishkin is largely responsible for the current state of its implementation on the Apollo. Jim Philbin also made important contributions, porting *U* to Unix and VMS. Jonathan Rees and Norman Adams developed T, my favorite programming language and the environment in which *U* came to fruition.

Thanks to Udi Shapiro, for shaming me into finishing.

Judy Martel carefully read several drafts of this thesis, corrected many of my errors of style, and pointed out many flaws in the presentation; I thank her for contributing to my development as a writer. Mary-Claire van Leunen skimmed the high spots, skipped the dull bits, and offered good editorial advice.

I spent a very pleasant five years in New Haven: for this, I happily thank Norm Adams, Mark Burstein, Gail Craddock, John Ellis, Josh Fisher, Paul Hudak, John Levine, Geoff Lowney, Judy Martel, Nat Mishkin, Judy Mishkin, John O'Donnell, Susan Temple

O'Donnell, Jim Philbin, Jonathan Rees, John Ruttenberg, Margie Ruttenberg, Udi Shapiro, Lydia Spitzer, Cathy Van Dyke, Mary-Claire van Leunen, Maggie Wolf, and Steve Wood.

I am grateful for the constant love and support of Sue Nelson Nix, and I dedicate this thesis to her.

Table of Contents

Chapter 1

THE PROBLEM

Text editors are interactive programs that help people compose and edit text on a computer. They make it easy for people to incrementally modify their text in a free-form and natural style. Most of the time spent in an interactive computing environment is spent using a text editor, and interactive editing has come to be an ubiquitous paradigm for good user interface design.

The bulk of the editor user's time is spent on fresh, unique, and non-repetitive tasks, such as dashing off a letter or writing this dissertation; but there are often occasions when the user finds himself doing something repetitious and dull.

As an extreme example, consider the plight of Professor I. K. Jones, who over the course of years has accumulated a large collection of information into a personal database of considerable magnitude. Professor Jones has painstakingly accreted disks and disks filled with lists of addresses of his colleagues, scores of American League baseball games, cross-referenced indices of his many publications, grain futures quotations, and a whole lot more. His collection of data has grown so vast and chaotic that he cannot find a thing in it. He knows of only one solution to his crisis: he must reorganize his varied bits of data so that they can be uniformly manipulated by a database system. This reorganization process is problematical because each of his personal database files must be converted from its current idiosyncratic format to a standard format for eventual incorporation into the database.

Professor Jones has the choice either of writing programs to transform each of his file formats to the standard one, or of transforming his files manually with his editor. In his case, the mass of data is so large that he would probably choose to write programs to perform the conversion. On the other hand, if he were interested only in converting a file containing the scores of the twenty-odd baseball games played on one weekend in May, he would probably choose to do it by hand, in his editor. And if Jones were not a facile programmer, he might even convert an entire year of baseball statistics manually.

Writing a program involves a fair amount of overhead, work that is not directly connected with solving the particular problem. By far the largest portion of this overhead lies in learning how to write computer programs in the first place. Even assuming that the user knows how to program (Jones is an Egyptologist), he still has a lot of work to do. In order to write a program, he has first to formulate an abstract model of the concrete problem at hand and then follow through with the mechanics of actually implementing his abstraction: writing the program, debugging it, running it, checking to see whether it did the right thing, and then debugging it some more. Of course, the advantage of programming is that once this development process is completed, the program can convert his files in seconds.

On the other hand, converting the files by hand within an editor involves almost no overhead. Professor Jones created and updated his files with his editor, and he uses the editor every day in the normal course of his work, so he does not have to think at all about how he would use it to solve this particular problem. But meticulously transforming large files like these involves a considerable amount of tedious work. This work is productive, in the sense that every step makes tangible progress towards the solution of the problem, but it is tedious and repetitious all the same.

1.1 Editing by example

In what follows, we report on a theory, a design, and an implementation of a new and interesting way of resolving dilemmas like this one. This mechanism has the advantages of both of Professor Jones's options, and it ameliorates some of their disadvantages. This mechanism is ***editing by example***, or EBE. An EBE system is a facility embedded in a text editor that can be used to solve repetitive text editing problems. An EBE system takes a few examples of a text transformation and produces a program that generalizes a rule governing the examples. Once produced, this program can be used to perform the generalized transformation to the rest of the user's text.

To demonstrate editing by example, suppose that Jones wants to transform a long list of baseball scores like these:

```
Yankees 3, Orioles 1.
Brewers 12, Cardinals 5.
Dodgers 5, Braves 4.
Braves 3, Dodgers 0.
Reds 4, Mets 2.
Pirates 2, Phillies 1.
```

to a database input format:

```
GameScore[ winner 'Yankees'; loser 'Orioles'; scores [ 3, 1 ] ];
GameScore[ winner 'Brewers'; loser 'Cardinals'; scores [ 12, 5 ] ];
GameScore[ winner 'Dodgers'; loser 'Braves'; scores [ 5, 4 ] ];
GameScore[ winner 'Braves'; loser 'Dodgers'; scores [ 3, 0 ] ];
GameScore[ winner 'Reds'; loser 'Mets';  scores [ 4, 2 ] ];
GameScore[ winner 'Pirates'; loser 'Phillies'; scores [ 2, 1 ] ];
```

He could write a program to make this transformation, but he decides instead to use the editing by example facility that has just been introduced into his editor. He enters the text editor and begins his EBE session by selecting, or marking, his first example:

```
Yankees 3; Orioles 1.
```

He then issues a command to the editor that tells it that the selected text is an example of the sort of thing that he wants to change. That is, this line of text is the sort of *input* that he wants his transformation to affect. He then manually transforms that text to the database format, using the editor commands he would normally use to make the change on a single instance of the text:

```
GameScore[ winner 'Yankees'; loser 'Orioles'; scores [ 3, 1 ] ];
```

Once he has finished changing the line, he 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 he would like the editing by example system to produce. At this point, he could give a command telling the editor to synthesize a program that generalizes the transformation expressed by his example. However, he knows that 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, and since the rest of the scores are not simple repetitions of `Yankees 3, Orioles 1.`, he gives another example by selecting and transforming the `Brewers` score in the same way.

After providing the second example, Jones feels that the system should be able to make a decent generalization, so he asks to see it. The editor shows him the synthesized program in a specialized notation for string search and transformation:

```
    bol -1- ␣ -2- ,␣ -3- ␣ -4- . eol
                                        ⇒
    GameScore[␣winner␣' -1- '␣loser␣' -3- ';␣scores␣[␣ -2- ,␣ -4- ␣]␣]; eol
```

This notation defines a simple program with two parts: the part preceding the "⇒", which is the input pattern, and the part following the "⇒", which is the output replacement. The input 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 "`,␣`" and "`GameScore[␣winner␣`", which match their literal text (␣ is a visible space character), along with the special constants *bol* and *eol* that match the unprintable text fragments beginning-of-line and end-of-line. The variables in the input pattern are signified by the numbers between dashes. A variable matches any sequence of characters up to the constant string that follows it in the pattern. In this pattern, the variable *-1-* matches the characters between the beginning of the line and the first space.

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 used in the output pattern. For example, when the input pattern fragment "`,␣` *-3-* `␣`" is matched against the text "`, Orioles `", the variable *-3-* is bound to the text "`Orioles`", and every occurrence of a *-3-* in the output replacement expression is replaced with that text. The searching process is continued after the point of replacement, and the program stops when no matching text is found.

Jones decides that the synthesized program looks like it will work, so he gives a command that runs it in single-stepping mode. In this mode, the editor asks the user for confirmation before transforming the text that matches the pattern. The `Dodgers` score is selected as the next part of the file that looks something like the two examples given so far. Jones confirms that this is a good choice, so the system replaces the "`Dodgers...`" line with the appropriate "`GameScore[...`" line and asks him whether the transformation was correct. It was, and Jones thinks that the program will work for the rest of scores, so he finishes the job by telling the system to continue and transform the rest of the file.

The next problem on Jones's agenda is slightly different. He has a large paper on unionization among pyramid construction workers that he originally formatted using the TEX document formatting system [53], and he would like to make the file acceptable to Scribe [73, 74], another document formatter. This conversion process is rife with small text transformation problems that can be solved using an EBE system. For example, he has to change the TEX notation for describing figures:

```
\figure{
Foreman         10
Brickwright      3
Slave Driver     0
Slave            0
}{Pyramid Construction Wage Scales}
```

to a different form for Scribe:

```
@Begin(Figure)
@Begin(Format)
Foreman          10
Brickwright      3
Slave Driver     0
Slave            0
@End(Format)
@Caption(Pyramid Construction Wage Scales)
@End(Figure)
```

As with the baseball scores, Jones gives this example by selecting the input text, giving it as an input example, changing it to the desired output, and then giving that text as an output example. After Jones gives a second example of this transformation on a similar figure entitled "`Pyramid Worker's  Life Expectancy (by Occupation)`", the editing by example system has the following incorrect hypothesis:

```
bol \figure{ eol
Foreman␣␣␣␣␣␣␣␣ -1- eol
Brickwright␣␣␣␣ -2- eol
Slave␣Driver␣␣␣ -3- eol
Slave␣␣␣␣␣␣␣␣␣␣ -4- eol
}{Pyramid␣ -5- } eol
                                     ⇒
@Begin(Figure) eol
@Begin(Format) eol
Foreman␣␣␣␣␣␣␣␣ -1- eol
Brickwright␣␣␣␣ -2- eol
Slave␣Driver␣␣␣ -3- eol
Slave␣␣␣␣␣␣␣␣␣␣ -4- eol
@End(Format) eol
@Caption(Pyramid␣ -5- ) eol
@End(Figure) eol
```

The system found that the text begins on a line with "`\figure{`" and ends on a line that contains the caption notation, and it also discovered a transformation that could map the input examples to the outputs. However, a great deal of accidental commonality in the two examples was found as well: the two figures both listed occupations connected with pyramid construction, so they shared the names of the occupations; and both captions happened to begin with the word "`Pyramid `". While this program will serve to find and transform the next figure that shares these traits, it will ignore figures whose caption, for instance, starts with any word other than "`Pyramid `". So Jones provides a third example that does not share these unintended common features by transforming

```
\figure{
\include{overrun.graph}
}{Construction Time Overruns}
```

in the same way. After this figure is given as an example, the editing by example system generates a more streamlined program which serves to transform the rest of the figures in the paper:

```
bol \figure{ eol
-1- eol
}{ -2- } eol
                    ⇒
@Begin(Figure) eol
@Begin(Format) eol
-1- eol
@End(Format) eol
@Caption( -2- ) eol
@End(Figure) eol
```

Most of the other transformations involved in converting Jones's paper from TEX to Scribe can be done in a similar fashion, by giving examples for each particular transformation and converting all of the other instances of the transformation with one run of the synthesized program. When the process of conversion from TEX to Scribe is viewed incrementally, each of the steps is simple, and the exceptions and special cases that are bound to occur can be handled individually with little effort.

These two scenarios involved massive text processing tasks, the sort of tasks that can consume hours, and the sort of tasks that are not done every day. An editing by example system can also be used to solve smaller problems; indeed most of the applications of an EBE system fall into this category.

For example, suppose that a programmer is involved in debugging some network software, and has written a program called "`chaos_server_creator`" which is used to create network servers. The server creator has a bug which causes it to erroneously create a large number of "`name_server`" programs. When the programmer hears about this bug, he lists out in an editable transcript the processes that are running on the network server's machine and discovers that there are indeed a large number of runaway name servers:

```
$
$ ld -u //gamma/sys/node_data/proc_dir

Directory "//gamma/sys/node_data/proc_dir":

       uid           name

16FCFE96.E000009C  chaos_ncp
16FCFE97.7000009C  chaos_server_creator
16FCFE95.2000009C  display_manager
16FCFE96.A000009C  mbx_helper
1704B389.3000009C  name_server.102
1704B3EC.6000009C  name_server.103
1704B3F1.9000009C  name_server.104
1704B3F8.C000009C  name_server.105
1704B3FE.F000009C  name_server.106
1704B404.2000009C  name_server.107
1704B410.5000009C  name_server.108
1704B41A.8000009C  name_server.109
1704B423.B000009C  name_server.110
16FCFE97.4000009C  net_mail_daemon
1704E64D.3000009C  process_28

15 entries.
$
```

The runaway name servers are sopping up all of the resources on the network server computer, so the first step in fixing the bug is to kill them off. To kill processes on a remote machine, the programmer has to run a special version of the "sigp" program that he keeps in his "toolbox" directory, supplying it with the unique identifier's (uid's) of the processes that he wants to kill (the pair of hexadecimal numbers in the left column). The programmer can kill off these processes by issuing the following set of commands to the command interpreter.

```
~toolbox/sigp -u 1704B389 3000009C
~toolbox/sigp -u 1704B3EC 6000009C
~toolbox/sigp -u 1704B3F1 9000009C
~toolbox/sigp -u 1704B3F8 C000009C
~toolbox/sigp -u 1704B3FE F000009C
~toolbox/sigp -u 1704B404 2000009C
~toolbox/sigp -u 1704B410 5000009C
~toolbox/sigp -u 1704B41A 8000009C
~toolbox/sigp -u 1704B423 B000009C
```

To produce this sequence of commands, he selects the line containing

```
    1704B389.3000009C  name_server.102
```

as an input example, changes the line to read

```
    ~toolbox/sigp -u 1704B389 3000009C
```

and selects it as an output example. He does the same thing for the line named "name_server.103", and asks the system to generalize the two examples, yielding the following program:

```
    bol 1704B -1- . -2- C␣␣name_server. -3- eol
                                                   ⇒
    ~toolbox/sigp␣-u␣1704B -1- ␣ -2- C eol
```

He then applies the program to the rest of the list, which transforms it to the process termination commands. He can then give the commands as input to the command interpreter, which will run the process-killing program for each of the runaway processes. This entire procedure takes about fifteen seconds, and the programmer can go on about fixing the bug that caused the problem.

In each of these scenarios a text editor user is faced with the problem of changing the format of some parts of a larger body of text. The EBE system provides a way for the user to write programs that can perform such transformations. The programs work by hunting out the text to be transformed, parsing it into its constituent fragments, and rearranging the fragments as desired.

1.2 Goal and approach of the thesis

This dissertation proposes a practical design for an EBE system. A practical editing by example system is one that a person like Professor Jones would turn to out of choice when faced with a text processing problem whose solution demands either programming or drudgery. The practicality of the system arises out of a combination of several factors: First, it must be powerful enough to synthesize programs that can solve the text processing problems that the user encounters. Second, it must be easy and natural to use, which is determined both by the engineering details of the user interface and the requirements for information imposed by the system. Third, it must be efficient, both in computational terms and in the amount of information that it requires of the user. Fourth and foremost, a practical EBE system must *exist* and *work*, otherwise practicality is not an interesting issue.

The simultaneous satisfaction of these requirements requires compromise. The desire for a powerful system is the most difficult to satisfy; our ability to efficiently synthesize useful programs from examples is nowhere near as advanced as our ability to come up with text processing problems that *look* like they can be solved by example. What we present in this thesis are schemes for synthesizing text processing programs that solve *some* of the text processing problems a user might face. We would like to think that we have captured quite a few of them, but we know for certain that there are some that we've missed.

Since we are not able to solve all of the text processing problems that we would like to, it is extremely important that we have a clear notion of which problems lie within the system's range. It is unlikely that a user would be willing to go to the effort of giving examples to a system that has failed him in the past without explanation. The only way we know of getting a clear notion of the system's capabilities, and of having an explanation ready, is to formulate an analytical model of those capabilities. Thus a large portion of this dissertation is devoted to a formal analysis of the power of one scheme for text program synthesis.

In the next chapter, we begin the development of an EBE system by surveying related research in the areas of programming by example, inductive inference, program synthesis from examples, and grammatical pattern inference. We then examine the sequence of decisions that led to the development of the system described in Section 1.1 by considering both the feasibility of various system design alternatives and the impact that pursuing them would have on the user of the system. As a result of these considerations, we decide to develop an editing by example system that works by synthesizing a class of text processing programs called *gap programs* from examples of their input/output behavior.

We go on to define gap programs and to examine some of their basic properties. We then analyze the computational complexity of the problems associated with identifying them from input/output examples. The results of this analysis inspire the development of a heuristic algorithm for gap program synthesis. We prove that, given adequate data, the heuristic can be guaranteed to synthesize the user's intended gap program.

An experimental study of the gap program synthesis heuristic's behavior leads to the formulation of several modifications that improve its performance on small amounts of data. The gap program synthesis algorithm that results can usually synthesize a target gap program from two or three examples of its input and a single example of its output. This algorithm has been implemented within the EBE subsystem of the *U* editor [66]; the details of *U*'s EBE user interface are presented.

We go on to examine some proposals for extending the EBE system to create more sophisticated programs, and we close with a brief conclusion.

Chapter 2

RELATED WORK

Ours is the first work on program synthesis from examples in a text editor; however, a large body of other research has bearing on the issues that we address in this thesis. The first step of our survey of this work is to review the approaches that have been taken to providing facilities for programming by example. In general, programming by example research has concentrated on developing novel facilities for transcribing programs; the facilities developed do not generalize their sample data in an interesting way. We want more ambitious generalizations, so we go on to survey the theory of inductive inference, which provides a formal framework for viewing the process of generalization from examples. We then present a short overview of the techniques used by automatic program synthesis systems. Finally, we survey research on grammatical inference, which is an approach to the problem of discovering patterns common to a set of strings.

2.1 Programming and program transcription systems

Text processing problems like those posed in Chapter 1 are typically handled through programming. On the Unix system [50], for example, the user has many ways available to him for programming a solution to the problem, each with its own difficulties and demands for expertise: he can write a general purpose tool, which would probably be a C program that solves this general class of problems; he can write a special purpose program in a systems programming language, like C, that performs exactly the transformations needed; he can write the same sort of special purpose program, but in a string processing language such as SNOBOL, Icon, or awk; or he can piece together a program from several general purpose file transformation filters using the shell programming language. If the user's editor is programmable, he can write a program within the editor's embedded programming language that composes editor primitives into a special purpose command for this task. If his editor is not programmable, but the user is familiar with its internal structure and implementation, then he might add in some special-purpose editor commands to help him get the job done.

Some of these options require less effort than others, perhaps because the resulting programs are shorter, or perhaps because they run in an environment that makes writing and debugging them easier. But all of the options require some knowledge of programming and the desire to use that knowledge to solve a particular text processing task. If the user does not know how to program, or does not feel like programming, or regards the problem facing him as too trivial to program, then he is likely to solve the problem manually using the basic command set of his editor.

While doing this, the user will probably use a similar set of commands to solve each instance of the problem. If he uses exactly the same commands to solve each instance, then the simplest way to turn these commands into a program is to record the commands used to solve one instance of the problem and to "play them back", or re-execute them, to solve the next instance of the problem. Such program transcription facilities are widespread and are usually called the macro language in most text editors, such as EMACS [85] or Z [98], although this facility is called "programming by example" in the *bb* editor described by Meyrowitz and van Dam [62].

A similar sort of programming by example facility, but in a different domain, has been implemented by Halbert within SmallStar, a Smalltalk prototype of Xerox's Star office system [38]. In SmallStar, the user specifies a program in CUSP, Xerox's customer programming language for the Star, by graphically tracing its execution. For example, to write a program that prints all "documents" in a "file folder," the user enters "start recording" mode and goes through the actions he would normally take to pull a document out of a folder and print it. The system uses a simple heuristic to decide which of the objects being manipulated stand for variables in the program, but the user must specify loops and other control structures explicitly.

Several other systems have used the "by example" paradigm to make the process of programming more natural: Smith's Pygmalion system [83], was an earlier experiment with developing a system like SmallStar that also included a program transcription component. Curry [20] implemented a system for programming by abstract demonstration that is similar in spirit to Smith's. Zloof's Query-By-Example system [99, 100] provides a template-style interface to a relational database. In QBE, the user formulates a relational query by filling in the slots of a table portraying this query. QBE and other related systems [101, 102, 103] are unambiguous programming languages that take advantage of the tabular nature of their domains to give the act of programming a "by example" flavor. Lieberman and Hewitt's Tinker system experiments with a program development method that interleaves testing and design [57]. In Tinker, a program is developed by supplying examples of the use of sub-functions and expanding the definitions of these functions when the computation on the example data demands it. Hatfield developed a design for an interactive document formatting language in which formatting styles are specified by demonstrating them on a template [40]. Industrial robots receive some of their programming by being "led" through the mechanical motions they are to perform [90]. This same style of programming by example has been used in Perlman's Tortis system for teaching programming concepts to young children (mentioned by Smith [83]).

These systems all represent interesting research in user interfaces for programming, but they do not perform interesting generalizations of the behavior shown them.

2.2 Inductive inference

Inductive inference is the theoretical foundation for the process of program synthesis from examples. Researchers in inductive inference have focused on finding and characterizing solutions to the problem:

> Given a set of examples that conform to some unknown rule, how do we choose the "best" explanation for the behavior shown?

Inductive inference is a familiar task for any student who has taken a standardized test. These tests often present students with the start of a sequence, such as:

> 1, 3, 7, 15, 31,...

and ask them to select "the" next element of the sequence from the choices provided. To find this element, the student must divine the rule governing the generation of the sequence (or be a talented guesser).

Such examinations test the student's ability to function as an inductive inference algorithm with three inputs, two of which are implicit in this case. The explicit input is a set of examples which are implicitly described in some language. In this case, the example set is a single numeric sequence and its presentation language is the language of arithmetic. The other implicit input to the algorithm defines the range of explanations for the sequences that should be considered by the student in forming his guess. This set is difficult to define in this case: the examiners probably mean it to encompass all functions that can be expressed as reasonably simple combinations of the arithmetic operators familiar to a high school student. The student's job is to find a function f in the allowable range of explanations with $f(1)$ equal to 1, $f(2)$ equal to 3, ..., and $f(6)$ equal to one of the choices provided. Of course there are an infinite number of functions that have this initial behavior; these tests implicitly ask for the simplest one.

Gold, motivated by the problem of characterizing the process of language learning, introduced an abstract setting for studying inductive inference [32]. The inductive inference problem for languages is the problem of deciding which language L of a set of languages U is characterized by a set of examples E. Each example in E shows a string and indicates whether or not that string is in the target language. For instance, suppose that U is the set of regular languages over the symbols $\{0,1\}$, then

$$E=\{<0,in>,\ <1,out>,\ <0011,in>,\ <10101,out>,\\ <101011,in>,\ <1010111,out>,\ <1010110,in>,\ ...\}$$

might characterize the regular language L consisting of the binary strings of even parity. In Gold's work the process of inductive inference is assumed to take place on an ever expanding, potentially infinite list of examples in which every possible string occurs at least once in the list along with an indication of whether or not it is in the target language. Gold defined an algorithm I to *identify a language in the limit* if, as more and more examples are provided, I eventually settles on one explanation that fits the facts and does not subsequently change this explanation. A class of languages U can be identified in the limit if every $L \in U$ can be identified in the limit.

Identification by enumeration is a general inference technique introduced by Gold in which I tries each of the elements of U in some fixed order L_1, L_2, L_3, ... and returns the first L_i consistent with the examples seen so far. When this algorithm is presented with a new fact that is not covered by its current explanation, it switches to the next language in the enumeration that works. In general, an algorithm cannot decide that it has successfully identified a function in the limit because new data may conflict with its current conjecture.

Identifying functions can be related to identifying languages by viewing a function f as defining the language consisting of all pairs $<x,f(x)>$ for every x in the domain of f. Using an enumerative algorithm, Gold showed that a set of functions F can be identified in the limit from input/output examples if it is recursively enumerable with a decidable halting problem. For languages, as opposed to functions, this identification requires examples both of strings that are in the language and strings that are not. To demonstrate this, Gold proved that if U contains all finite sets and at least one infinite set, then it is possible for an adversary to keep an inference algorithm supplied only with positive examples from converging to a correct answer. This implies, for example, that an arbitrary element of the regular sets cannot be identified in the limit from positive data.

Gold distinguished two ways of presenting examples to an inductive inference algorithm: *text* and *informant*. Presentation by text is passive: the ordering of the infinite list of examples is fixed in advance. An inductive inference algorithm working from a text presentation is limited to reading the next example from the list and returning its current conjecture. On the other hand, example presentation via an informant is active: an inference algorithm can ask the informant if a particular string is an element of the target language and use the informant's answer to formulate another query. In an abstract sense, these two presentation techniques are equivalent (although there are settings where they are not [45]), because an inference algorithm working from text can act as if it is working with an informant by reading and remembering examples until the answer to its question appears in the example list. Of course, there is a considerable practical difference between the two techniques.

A great deal of interesting research in inductive inference has been done since Gold's initial papers. Blum and Blum [15] extended Gold's work by defining identification in the limit relative to a complexity bound, thus allowing identification in the limit for classes of functions that do not have decidable halting problems. Many researchers have designed algorithms and heuristics that inductively infer various classes of functions and languages; some of these will be discussed in the following sections. Further discussion of inductive inference can be found in Angluin and Smith's survey [5].

The theory of inductive inference provides an elegant formal setting for examining the problem of generalization from examples, but it does not offer practical techniques for program synthesis.

2.3 Program synthesis systems

2.3.1 Program synthesis from I/O behavior

Perhaps the earliest program synthesis systems were those of Fredkin, Pivar, Finkelstein, Persson, and others [29, 68, 69, 70], who did research in the early 60's on the problem of extrapolating numerical sequences. Their programs worked by testing the example sequence for special-case properties, such as a constant difference between every adjacent pair of numbers. These systems were successful at identifying and extrapolating the function generating an example sequence because the domain of numerical sequences was restricted enough so that a small number of special case rules could be composed together to make a powerful and robust system. This outcome was heartening, but extrapolating from sequences to programs has proven to be much more difficult.

Artificial Intelligence researchers working on Automatic Programming have approached the problem of program synthesis by developing knowledge-based systems for reasoning about programs [8, 59, 60, 61, 89]. Perhaps the most ambitious project in this area is PSI, carried out by Green and his colleagues at Stanford [34, 35, 36, 39, 80]. This project's principal focus was on understanding, codifying, and implementing the complex heuristics that are used by real programmers as they go about their work. Along the way though, a few systems that synthesized simple list manipulation functions from I/O behavior were implemented [34]. These systems were based on schemas — they check the input/output examples to see if they fit a particular stylized type of program and synthesize a variation of the stylized program if they do.

Summers presented a more formal approach to this style of Lisp program synthesis [87, 88] that has since been refined by many other researchers [47, 48, 49, 54, 55]. Summers' system, THESYS, takes several examples of the input/output behavior of a list processing function and tries to construct a simple recursive function involving the Lisp primitives *car*, *cdr*, *cons*, *atom*, and *cond* that can effect the transformation described in the examples. THESYS constructs the program by transforming the input/output examples to a canonical form that ignores the actual atoms being manipulated, and then searching for a recurrence that describes the relations among the canonicalized examples.

Shapiro approached the problem of program synthesis from the point of view of program debugging [76, 77, 78, 79]. Shapiro's Model Inference System fixes a "buggy" Prolog program fragment when it encounters an example of the input/output behavior of the program that is incompatible with the current hypothesis. Shapiro's system works by accepting facts, generating axioms to explain the known facts, debugging the axioms when a contradictory or unexplained fact is found, and generating queries to differentiate between candidate axioms. The Model Inference System depends on an oracle, namely the user, to provide examples of the target function's input/output behavior and to answer questions of the form "*is* $f(x) = y$?"; good examples, that cause the system to revamp its buggy hypotheses, can speed the program synthesis process.

2.3.2 Program synthesis from traces

Analysis based solely on the input/output behavior of a demonstrated function ignores a great deal of information: the actual operations that were used to produce the function's behavior. The record of these operations is called a *trace*.

Biermann has developed several systems that have explored techniques for program synthesis from traces. He began by synthesizing formal machines from traces of their state transitions. He developed algorithms for synthesizing both finite state machines [10] and Turing Machines [11]. Subsequently, he experimented with the synthesis of programs written in an Algol-like language [13] and a restricted form of Lisp [14]. The Algol-like program synthesis system typifies his approach: the user begins an "autoprogramming" session by declaring the name of the function that he wishes to synthesize, the names of its parameters, and the names of any intermediate values he requires. Next he simulates each step of the target program's execution by going through actions of the form "I := I+1", and "A[J] := A[I]". The system analyzes this information and tries to infer the existence of loops and conditionals. The inference process is carried out by looking for blocks of statements that are equivalent modulo updates to loop induction variables. Biermann reports the synthesis of a small compiler for an Algol-like language using this technique [13].

Waterman, Faught, and others [25, 26, 94] coined the name "exemplary programming" to describe the function of EP, a programming by example system that was designed to address the trivial day-to-day programming needs of a user interacting with a computer. EP examines the text of a user's interactions with systems programs (such as a network file transfer program), with the goal of constructing "agents" (programs) that will subsequently handle the details of this sort of interaction automatically. For example, EP would synthesize a file transfer agent by analyzing one example of the dialogue between a user and a network file transfer program. The synthesized agent is named by the user and stored in a library with other agents, to be subsequently invoked when the user wants to perform the task again. EP's agent synthesis process involves almost no generalization on the part of the system.

Other systems have done some form of program synthesis from traces. Automatic programming efforts such as Green's PSI have made use of a program trace [39, 80, 34, 35, 36]. Bauer [9] implemented a system that extended Biermann's by allowing all program actions to be annotated with a justification. For example, a trace in Biermann's system might contain the statement "I := I+1", while in Bauer's the user could communicate more intent by saying "Since J<5 then I := I+1". Witten implemented a scheme for communicating simple programs to a pocket calculator by giving examples [97].

There has been a great deal of research in the area of program synthesis, and it will not be surveyed further here. The interested reader should consult Balzer's early paper [8], Green, et.al.'s report [34], or the survey of Smith [84]. There has also been considerable research done in the area of machine learning, which is not closely related to our work in editing by example. Dietterich, London, Clarkson, and Dromey's survey [21] examines work on machine learning, and a compendium of current research may be found in the book of Michalski, Carbonell, and Mitchell [63].

Most of the work in program synthesis has concentrated on synthesizing general purpose programs from examples. Such programs can vary widely in their structure and behavior, and it is thus not surprising that a small amount of example data does not serve to describe very many interesting programs. All of this research is experimental; none of these systems have been put to production use as a method of writing programs.

2.4 Finding patterns common to a set of strings

The programs described in Chapter 1 consist of two parts: a pattern expression that describes the format of the text that is to be transformed, and a replacement expression that computes the transformation. In this section we survey some of the work relevant to the problem of synthesizing the pattern expressions from examples; a broader overview may be gathered from the surveys of Biermann and Feldman [12], Fu and Booth [30], or Angluin and Smith [5].

Pattern expressions are used to describe the syntactic structure of the text that is to be transformed; perhaps the two best known formalisms for describing the syntactic structure of text are regular expressions and context free grammars.

A well known result of Gold shows that if a class of languages includes all finite languages and at least one infinite language, then no algorithm can identify arbitrary elements of this class in the limit from positive data [32]. Regular expressions define a set of languages that satisfy these criteria, so no algorithm can converge to an arbitrary regular expression without some added information. However, several different forms of additional information make identification possible.

One form of additional information is negative data; an inference algorithm that is provided with negative data as well as positive data can identify regular expressions from examples. For example, an algorithm could work by successively drawing hypotheses from a complete list of all regular expressions and testing its current hypothesis drawn from the list against the positive and negative samples provided. If the hypothesis either does not match some positive sample or successfully matches some negative sample, then the algorithm can change its guess to the next expression in the enumeration. This algorithm is guaranteed to converge to a correct expression because such an expression occurs at some earliest location in the enumeration, and all of the preceding expressions will eventually demonstrate that they match some negative sample or fail to match a positive one.

Regular expressions can be identified solely from positive data if an inference algorithm is provided with some extra piece of information about the samples that restricts the number of times that an adversary can force the algorithm to change its current hypothesis. For example, if it is known that the target regular expression is, say, less than 1,000 characters long, then the algorithm could work by guessing the regular expression that generates the minimal enclosing language containing the current set of samples, and changing its guess when it is provided with a sample that is not in the language generated by its current hypothesis. This algorithm converges for two reasons: it guesses an expression generating a minimal language, which either is equal to the target language or fails to contain strings that are in the target language; and there are a finite number of candidate expressions that are less than 1,000 characters in length, so the algorithm can change its hypothesis only a finite number of times before converging. The bound on the size of the target expression does not have to be given explicitly. For example, the algorithm could assume that some initial set of samples restricts the search space; e.g. an initial sample set of {abad,ababad,abababad} might be assumed to imply that the target expression is no longer than (abad|ababad|abababad).

Although it is possible to identify regular expressions from examples, efficient algorithms for this task have not been developed. Angluin [3] showed that the problem of finding a minimal length regular expression that matches a given set of positive and negative samples is NP-hard. The problem remains NP-hard even when the class of regular expressions considered does not use union, or when it disallows the use of closure.

There has been somewhat more success in synthesizing finite automata from examples, although Gold showed that the corresponding problem of finding minimal finite automata from positive and negative examples is NP-hard [33]. Feldman [27] formulated a general strategy for enumerating automata that structures the search space so that unworkable solutions can be quickly pruned. Pao and Carr [67] presented an enumerative algorithm based on a variation of this strategy that identifies finite automata from a combination of input/output behavior and queries about the acceptability of strings. Biermann and Feldman [10] applied a similar idea to the identification of sequential machines from input/output information.

Trakhtenbrot and Barzdin [92] showed that if the sample set includes positive and negative data that classifies all strings not exceeding a given length, then there is a polynomial time algorithm which finds a deterministic finite automaton with the minimal number of states compatible with the sample. Their algorithm runs in polynomial time because its input has been padded; if the size of the alphabet is 2, and samples up to length 20 are required to express the intricacies of the target automaton, then the algorithm needs more than 2,000,000 samples. Angluin [3] presents results that characterize how much this padding can be reduced and retain the polynomial running time.

Angluin found a polynomial time algorithm for inferring a deterministic finite state machine from positive data and a polynomially bounded number of queries [6], which is related to the problem for which Pao and Carr developed an enumerative algorithm [67]. This algorithm starts with an initial set of strings that are accepted by some target finite automaton. The sample set is assumed to exercise the automaton completely in the sense that the machine passes through every one of its states while accepting the strings in the set. The algorithm then makes a polynomial number of queries concerning the acceptability of other strings derived from the initial set, and it uses the answers to these queries to synthesize the target machine in polynomial time.

Finite automata are a cumbersome notation for describing the structure of text, and thus regular expressions would be a more desirable target for a text processing system. A regular expression synthesis algorithm could work by using a finite automata identification procedure to synthesize an automaton that recognizes the samples and then transforming the automaton to a regular expression. But the process of converting deterministic finite automata to regular expressions can create regular expressions that are exponentially larger than the input automata [22], and regular expression minimization is an intractable problem [86]. Even if the finite automaton is the minimum state automaton that accepts the samples, the resulting regular expression may not be the minimal expression that generates a language that contains the samples.

The synthesis of context-free grammars has also been studied by many researchers, but it appears to be more difficult than regular expression synthesis. Feldman [27] and Feldman, Gips, Horning, and Reder [28] considered enumerative schemes for identifying context-free grammars from examples that make use of Feldman's scheme for trimming the search space. Wharton [95] also presents an enumerative algorithm for identifying context-free grammars from examples.

Crespi-Reghizzi, Melkanoff, and Lichten [18], and Crespi-Reghizzi, Guida, and Mandrioli [19] have developed efficient techniques for synthesizing subclasses of context-free grammars. In these systems the user is required to specify the unlabeled parse tree of each sample by bracketing the parts of the samples that are produced by each subtree. The task of the system is to find a good way of labeling the interior nodes of the parse tree with the names of the productions that they represent.

Angluin developed polynomial time algorithms for two special classes of languages: the reversible languages [7] and the pattern languages [4]. The reversible languages are a subclass of the regular languages [7]; a regular language L is k-reversible for a nonnegative integer k if whenever u_1vw and u_2vw are in L and $|v| = k$, then for every z, u_1vz is in L if and only if u_2vz is in L. For example, the language describing the set of binary strings of even parity is a zero-reversible language. Angluin presents an algorithm for finding the smallest k-reversible language containing a positive sample that runs in polynomial time for each fixed k. Reversible languages do not appear to be applicable to describing the patterns encountered in text.

The pattern languages are generated from *patterns*, which are a concatenation of constants and variables, for example, $12xx$ or $32xy5zxy$. The language generated by a pattern is the set of strings obtained by substituting non-null constant strings for the variables of the pattern. The language of $45xx2$ includes the strings 45112 and 451221222, but not the strings 452, 45132, or 666. Angluin presents an algorithm for finding a smallest pattern language containing a given set of positive samples and shows that this algorithm runs in polynomial time if the pattern contains only one distinct variable [4].

Shinohara's extended regular pattern languages [81, 82] are related to Angluin's pattern languages, except that the variables in each pattern can occur only once, and the variables are permitted to match the null string. For example, $35x2$ and $1x3y4z2$ are extended regular patterns, but $4x5x2$ is not. Shinohara presents an algorithm for identifying extended regular pattern languages in the limit from positive data, and he has applied the algorithm in a domain that bears an interesting resemblance to editing by example. Shinohara's idea is to use the algorithm to provide "a data entry facility with a learning function". If the user were entering, for example, a bibliographic database, then he would begin his session by typing in the first few entries:

```
$
Author:   Angluin, D.
Title:    Inductive Inference of Formal Languages from Positive Data
Journal:  Inform. Contr. 45
Year:     1980
$
Author:   Maier, D.
Title:    The Complexity of Some Problems on Subsequences and
          Supersequences
Journal:  JACM 25
Year:     1978
$
...
```

where "$" is a special symbol that delimits the records. After a few records have been entered, the system would infer the structure of the records in the form:

`Author:` w`Title:` x`Journal:` y`Year:` z

It would then emit the constant parts of the form, such as "`Author:   `", wait for the user to enter the field w, and go on to the next field when the user is done entering the author.

Shinohara's work was performed independently of ours, but extended regular patterns bear a close resemblance to the pattern matching notation introduced in Chapter 1 and developed in Chapter 4, which we call gap patterns (we call the variables "gaps"). There are two important differences between the notations: variables may not occur at the end of a gap pattern; and the text matched by a variable is not allowed to contain the string that follows the variable in the gap pattern. These restrictions arise from the application: gap patterns are used to search through large pieces of text for fragments that resemble the examples given, while extended regular patterns are used in a more controlled environment to synthesize a data entry form.

With the exception of Shinohara's work, the techniques for synthesizing patterns from examples have not seen practical application. This dissertation presents both an algorithm that performs an interesting kind of pattern synthesis from positive data and the complete design of a practical system that makes use of the algorithm to provide a facility for editing by example.

Chapter 3

DESIGN ISSUES IN EBE

There are an enormous number of different ways to design and build an EBE system. Our first stab at reducing the scope of the problem is to form a simple and not too restrictive model of the process carried out by an editing by example system:

$$\begin{tikzcd} \textit{Sample Data} \arrow[r] & \textit{Synthesis Procedure} \arrow[r] & \textit{Runnable Program} \arrow[ll, bend right] \end{tikzcd}$$

The goal of the EBE system is to find a ***target program*** that will solve the user's text processing problems. Towards this end, the EBE system collects *sample data* that describes the desired behavior of the target program and uses its *synthesis procedure* to map from the sample data to a *runnable program*. If the user is completely satisfied with the synthesized program, he can run it over and over again until he is through with his editing task. On the other hand, if the program does not satisfy him, he 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. The remainder of the chapter will discuss the first two questions, the answers to which will determine the structure of the system to a great degree. The third question will be dealt with in

Chapter 4, which contains the detailed development of a particular algorithm for text program synthesis. The user interface will be presented along with the rest of the implementation in Chapter 5.

3.1 What sort of programs does the system synthesize?

The goal of an EBE system is to synthesize programs that help a user transform his text in some regular manner. Text is a 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. But we do not know how to effectively synthesize arbitrary programs, so instead we will restrict our attention to some of the typical applications for text processing programs inside a text editor.

Many applications come to mind. The user might be performing a pattern directed scan and edit as was shown in the examples in Chapter 1. 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.

Many applications come to mind, but if we are to make progress on a practical editing by example system, then we must concentrate on one of them. We have chosen to concentrate on synthesizing programs that scan and edit text.

A typical sort of task handled by a text scanning and editing program is that of transforming a large file containing many Lisp function definitions:

```
(defun factorial (n)
    (cond ((<= n 1) 1)
          (t (* n (factorial (- n 1))))))

(defun sort (l)
    ...)

(defun halts (f)
    ...)
```

to be in a different form:

```
(define (factorial n)
    (cond ((<= n 1) 1)
          (t (* n (factorial (- n 1))))))

(define (sort l)
    ...)

(define (halts f)
    ...)
```

The process of changing one of these definitions can be viewed as consisting of two distinct phases: looking for the next function definition that should be changed, and changing that definition once it is found.

3.1.1 Pattern matching

The first phase can be handled by a fragment of the synthesized program that searches for a distinct configuration of text. There are a variety of well-known styles in which the target of the search could be specified: formal language notations such as regular expressions or context free grammars [42], procedural string matching code as in SNOBOL [37], *ad hoc* string scanning code such as that implemented in the runtime libraries of many programming languages such as C [51] or Bliss [23], and recursive symbol scanning code as is commonly written in Lisp. There are two criteria on which we could decide to prefer one notation to another: the first is how well the notation expresses the patterns that appear in text; the second is whether a program synthesis system can effectively synthesize the text scanning patterns that the notation expresses.

The conclusion to be drawn from the survey of program synthesis systems in Chapter 2 is that research into program synthesis has not yet yielded a practical method for reliably, robustly, and efficiently synthesizing general purpose programs from example information. The approach of adapting a general purpose program synthesis strategy to the synthesis of general purpose programs that just happen to be scanning text would probably not yield a practical system. The approach that we will take instead is to consider synthesizing programs of limited power that are specialized to string scanning.

Context free grammars are often used to describe the syntactic structure of programming languages, and there have been a number of systems that use variants of context free grammars to specify programming language parsers, the best known of which is probably Johnson's yacc [46]. Context free grammars can represent all of the structures expressible by regular expressions, and so one might expect them to be preferred to regular expressions for text processing problems. However, this extra power comes at considerable cost; context free grammars are a verbose and clumsy notation for expressing the structures that commonly occur in text and have not been widely used in this application. They are also difficult to synthesize from examples: only enumerative algorithms have been proposed.

Regular expressions represent text structures more concisely, and they and some of their subclasses have been widely applied to text scanning and editing [2, 91]. The text editor ed [50] uses regular expressions to specify the target text of an edit. The well-known Unix programs grep, egrep, and fgrep [75, 50] search a collection of files for lines that match a pattern that is specified by a regular expression. The Unix file processing tool awk [1] uses a subclass of regular expressions to filter out the lines to be processed; the programming language POPLAR [64, 65] has a similar text scanning language. Lesk's lexical analyzer generator lex [56] uses regular expressions to specify the token-level syntax of programming languages. Many systems implement a wildcard syntax for specifying file names that is a subclass of regular expressions, e.g. in the Unix shell "`*.c`" names all files ending in "`.c`".

Regular expressions seem to be powerful enough for many text processing applications, but it is difficult to synthesize them from examples. As we saw in the survey in Section 2.4, finite automata can be identified from positive data if it is assumed, for example, that a sample set exercises all of the states of the automaton. Finding a minimal finite automaton from positive and negative data is NP-hard [33, 3], and perhaps more importantly can require quite a few samples [3]. If the algorithm is allowed to ask the user if some particular string would be acceptable, then it can converge to an answer after making a number of queries that is polynomial in the number of states in the target machine [6], but $O(n^2)$ queries are too many to realistically impose on a user.

These are algorithms for finding finite automata, not regular expressions. The finite automaton can be converted to a regular expression, but the resulting regular expression can be very large, and is unlikely to be the one that the user had in mind. The finite automaton itself could be used as the hypothesis, but finite automata are large and cumbersome objects that are not suitable for showing to a text editor user (in Section

5.14, we will discuss why we want to show the program to the user).

Another problem with regular expressions is that they are a little too expressive. A given set of strings can be matched by an infinite number of regular expressions, and even if the strings are assumed to exercise all of the features of the target regular expression, the number of regular expressions that still could be *the* pattern common to the strings is very large. Regular expressions are not tailor-made for the text processing application; they are more general than is normally required, and are not ideally suited to expressing the standard structures of text. The subclasses of regular expressions that express commonly encountered text structures are difficult to characterize, and because of this, it is hard to come up with preference criteria that can enable a synthesis algorithm to choose the most "natural" explanation from among several candidate regular expressions.

Regular expressions are not fully supported by most of the systems mentioned above — of those mentioned, only egrep and lex support the full regular expression notation, the others support restricted subclasses. A common restriction allows only those expressions with single-character alternation and closure; this class contains patterns like `(a|b|c)` and `a*b(a|b|c)*d` but not patterns like `(ab|cd)`, `(ab)*`, and `((a|b)c)*`. Another common restriction allows only constant strings and wildcards. A wildcard is a symbol, often "*", that is used to match any sequence of characters. For example, the pattern `a*b` matches strings starting with `a` and ending with `b`, and the pattern `*command*.t` matches strings that contain the substring `command` and end in "`.t`".

These restricted subclasses of regular expressions seem to be adequate for their applications, and the right subclass of regular expressions can go a long way towards expressing the surface-level syntactic structures commonly encountered in text. While text is a ubiquitous medium that can be used to express arbitrarily complicated data structures, the structures that are commonly encountered are not very complicated. When text does have some structure, then that structure is usually "flat" and does not require a complicated pattern notation to be described.

SNOBOL presents a different point of view on text scanning constructs. SNOBOL's string matching facilities come in two parts: an automatic backtracking control structure that turns SNOBOL patterns into a universal computation engine, and a large set of pattern matching primitives that are used to conveniently match text. The pattern matching primitives work relative to a pattern matching cursor, and include facilities like:

a constant `'string'` matches if that particular string is at the current cursor location, `LEN(n)` matches any `n` characters, `BREAK('string')` matches all characters up to one of those in the string, and `SPAN('string')` matches runs of characters contained in the string. Programs can often be written using the primitive pattern matching language without recourse to backtracking.

We cannot expect to have much success with synthesizing programs that can employ all of the variations of SNOBOL's backtracking pattern matching control structure, because that mechanism is as powerful as a general-purpose programming language, and the world has not been able to build systems that effectively synthesize general purpose programs from example information. But it may be feasible to fix on one particular control structure, and to leave the program synthesis algorithm with the problem of deciding how to conjoin primitives similar to SNOBOL's within that fixed control structure.

We have adopted this strategy. The control structure that we have fixed on is one that performs a sequential scan through a file, matching a pattern against the text as it goes, and transforming the text that matches the pattern. Within that fixed control structure we have experimented with the synthesis of patterns formed out of specific primitives. The class of patterns that we have been most successful in synthesizing are called *gap patterns*, which are the patterns presented in the scenarios in Section 1.1. Gap patterns will be defined formally in Chapter 4, but as an example of a gap pattern, the first line of the Lisp function definition

```
(defun factorial (n)
```

could be matched by the gap pattern

bol `"(defun "` --- `" ("` --- `")"` *eol*

This pattern defines a class of strings that contains the constants *bol*, "`(defun `", "` (`", "`)`", and *eol*, interspersed with two *gaps*, which can be filled with any sequence of characters that does not contain the constant string that follows the gap. When this gap pattern is interpreted within the framework of the fixed control structure that we have adopted, it defines a string scanning procedure that looks for the string "`(defun `" at the beginning of a line, followed by the string "` (`" next on that line, and ending in a "`)`" at the end of the line.

3.1.2 Output generation

The gap pattern will help us locate the text; we must then transform the text to the form specified by the examples. As we saw from the scenarios presented in Section 1.1, this transformation can require the synthesized program to copy and replicate substrings of the input, and to introduce new constant strings into the output. The only difficult part of this process is that of parsing the input into named substrings, or fields. This parsing problem is similar to the pattern matching problem; indeed, the pattern shown above can be augmented with a field labeling mechanism to yield a pattern that names the strings that match the gaps:

```
    bol "(defun " -1- " (" -2- ")" eol
```

The pattern matches the text in the same way, but the pattern matching routine that interprets it has the added task of recording the bounds of the text that matches the gaps labeled *-1-* and *-2-*. Once some matching text is found and parsed, the function definition form can be edited to conform to its new syntax by specifying the text that should replace the text matched by the pattern. If the operations of copying gaps and inserting new constants are the only ones involved, then the replacement text can be specified using a notation similar to that used to specify gap patterns:

```
    bol "(define (" -1- " " -2- ")" eol
```

This notation defines a string that is equal to the replacement pattern with each gap symbol replaced by the text that matched the corresponding gap symbol in the gap pattern. The text processing program that consists of the input scanning pattern and the output replacement expression is called a *gap program*. We write gap programs as pattern/replacement pairs separated by a ⇒.

```
    bol "(defun " -1- " (" -2- ")" eol     ⇒
    bol "(define (" -1- " " -2- ")" eol
```

When this gap program is applied to the line of text:

```
    (defun factorial (n)
```

it transforms it to the line:

```
    (define (factorial n)
```

Chapter 4 presents an in-depth discussion of gap program properties and the problems associated with efficiently synthesizing them from example information. Chapter

6 discusses some extensions to the gap pattern and gap replacement notations.

3.2 What sort of information does the user provide the EBE system?

A fundamental aspect of the design of an editing by example system is the sort of information that the system requires from the user. Many types of information could be required: input/output behavior, traces, declarations, answers to questions, negative data. Traces and input/output behavior can be collected most naturally, so we will first consider which of these would be most appropriate for our applications, and postpone consideration of the others until Chapter 6.

3.2.1 Traces or I/O behavior?

There are two types of information about the target program that are easy for an EBE system user to provide: 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. The program transcription systems discussed in Section 2.1 (e.g. keystroke macros) are examples of trace-oriented systems that do not perform generalization. A trace-oriented EBE system that generalizes the command traces could be implemented to keep track of the commands used during several sample edits with the goal of generalizing the structure of these command sequences into a program that can perform the change. However, we would like to point out several problems with using traces as the source of information, and make the claim that the extra evidence contained in a trace is often useless.

One problem with traces is that the user's intent is often masked by the editor commands that he employs. Suppose that the user wanted to carry out the transformation described in Section 3.1 and change the form of every function definition in a file containing many Lisp function definitions. An essential step in this process is to locate the next function definition form to be changed. One way of doing this, which communicates some of the user's intent, is to search for the next occurrence of the word defun, but most editors provide a large variety of movement commands that do not correspond so simply with the user's intent. For example, the user could advance to the next function definition by giving several "move down to the next line" commands, or by "moving forward over words" until he gets to the right place, or by pointing at the definition with a pointing device, and so on. In most editors there are a large number of ways of moving around in the text, and very few of them impart any measure of the reason why the user chose to focus on one piece of text while ignoring others.

The structure of the text to be changed is not the only thing that can be masked by a trace. Once the user gets to the function definition that he wants to modify, he has a large variety of ways of changing it at his disposal. The Yale editor Z [98], for example, implements nearly a dozen simple commands for deleting a piece of text, and the extensible editor EMACS [85] provides even more deletion commands. If several commands are required, there is often nothing that constrains the order in which they can be given, and the user may well give them in a different order in each of his examples. An EBE system could deal with this difficulty by requiring the user to use a similar set of commands in a similar order when editing each example given to the system, but such a requirement goes against the grain of good user interface design. In a well designed editor, the user's interactions are free-form; he can jump willy-nilly around the text, making changes at the bottom line first and changing the top line somewhere in the middle. This freedom of action is a fundamental part of the user interface, and any EBE system must take it into account. An EBE system based on traces would restrict this freedom.

Another problem with traces is that an EBE system that uses them can make an editor more difficult to extend. Many editors are extensible, in that they have internal interfaces that allow programmers to add new facilities to the editor. Allowing for editor extensibility in a trace-oriented system is a problem, because the EBE system must have some knowledge of the meaning of every new command. Setting aside the difficult problem of representing the meaning of a new command, the sheer overhead involved in communicating knowledge of every new command to the EBE system could make it a bottleneck for editor development; such bottlenecks are often their own demise.

Requiring the EBE system to know the semantics of every editor command could also be a barrier to portability. Different editors have different ideas about the detailed semantics of their command sets, and embedding the semantics in the EBE system could make it difficult to transfer the system to a different editor.

One way around some of these problems is to design a trace-oriented system that works with a fixed set of primitive editing operations. For example, all of the current and future editor commands that delete various numbers of characters, lines, and words can be implemented in terms of a single, low-level, deletion primitive. The haphazardness of the user's interactions can be reduced by pre-processing them into a canonical order before analysis. The operations can be sorted so that they appear to happen from top to bottom, and redundant operations can be removed. However, once this is done, it is not clear that the remaining problem greatly differs from synthesis on the basis of input/output behavior — a great deal of the added information that separates traces from input/output behavior has been lost.

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, they have none of the problems that traces do.

3.2.2 How much data is needed?

Given that we are going to use input/output examples, how many should be required of the user to specify a function? The obvious answer to this question is, "As few as possible." In fact, we would have an ideal system if we could follow the lead of Green, Shaw, and Swartout's EXAMPLE system [80] and base our program synthesis on the evidence of a single input/output example.

Unfortunately, a single example cannot impart the pattern that describes the text that the user would like to transform, and so it is unreasonable to expect synthesis from a single example to yield useful programs. An essential function of text processing programs is to locate the next subpart of the file that is to be transformed and to skip over the parts that are not of interest. There is no basis for guessing the general rule governing the appearance of the interesting text from the evidence of one example. The only reasonable pattern would seem to be the pattern that matches the literal input text of the single example, and more examples must be provided in order to specify more interesting patterns.

There is also not much of a basis for deciding on a non-trivial transformation that maps the input string of the single example to the output. One might choose to find the shortest or simplest program that can effect the transformation, perhaps by making use of algorithms that find the least-cost edit distance of two strings [93], but there is no good reason to prefer this program to the trivial program that replaces all of the occurences of the literal input with the literal output.

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 (but there's no such thing as a free lunch). The program synthesis algorithms presented in Chapter 4 can often converge to the target gap program after the user provides two or three examples of the target function's input, and we introduce a heuristic in Section 5.9 that results in a single output example being all that is usually required.

Now that we have decided on the sort of programs that we are going to synthesize, and the data that we will use to synthesize them, it is time to embark on a detailed study of the techniques for doing the synthesis.

Chapter 4

GAP PROGRAMS

Gap programs are the class of *pattern* $\Rightarrow$ *replacement* programs introduced in Chapter 1. Here we formalize the definition of gap programs and examine some of their basic properties. We then study the problems associated with identifying gap programs from input/output examples by defining what we mean by the best gap program that fits a set of data, characterizing how much and what sort of data is needed to make the user's target program be the best one, and classifying the computational complexity of various aspects of synthesizing such a gap program. The complexity analysis motivates the development of an efficient algorithm that computes an approximation to the best gap program for the given data, and we prove that this algorithm will converge in the limit to the target program.

4.1 Gap patterns

A *gap pattern* G over an alphabet Σ is a sequence of alternating *strings* and *gaps*, $s_0g_1s_1g_2s_2...g_ns_n$. 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 $s_0s_1s_2...s_n$. Similarly, the *gap subsequence* of a gap pattern, denoted $g(G)$, is the string $g_1g_2...g_n$.

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

`this␣is␣a␣gap␣pattern`

This gap pattern matches the constant string "`this is a gap pattern`". Our notation for constant strings will be those strings appearing in a font like "this␣", where "␣" is meant to be a visible representation of the space character. The second sample gap pattern contains a gap:

    this␣ --- ␣gap␣pattern

The characters "---" together make up a single gap symbol. This pattern will match strings that begin with "this␣" and end in "␣gap␣pattern", with the gap spanning over those characters in between. The next gap pattern does not have a leading constant string:

    --- ␣pattern

This pattern will match strings ending in "␣pattern". The following gap pattern matches quoted strings:

    " *-1-* "

The symbol "*-1-*" is also a gap symbol. Our convention is to use gap symbols like *-1-* and *-2-* when there is some reason to assign a specific name to a gap, and to use the anonymous gap symbol --- for those gaps that we have no reason to name. 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. Each gap symbol must be distinct. For example, the symbol *-1-* can occur once in the gap pattern. An exception to this rule is made for the anonymous gap symbol ---; the following gap pattern matches the same strings as the previous one:

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

Although the anonymous gap symbol --- appears twice, this appearance is not meant to constrain both of its occurrences to match the same strings. Our final example gap pattern,

    (*-1-*)␣ *-2-* - *-3-* .

matches telephone numbers.

This is not a gap pattern

    this␣is␣not␣a␣gap␣pattern ---

because all gap symbols must be followed by a constant string. This is also not a gap pattern

```
this␣ -1- -2- not␣a␣gap␣pattern
```

for the same reason.

We now formally define the set of strings matched by a particular gap pattern. The definition closely parallels the intuitive descriptions given above, but it has the added restriction that the text that is matched by a gap must not contain the string that follows the gap in the pattern. 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* or *spanned* by a gap pattern G if $s \in L(G)$. Similarly, G *matches* or *spans* a set of strings S if $S \subseteq L(G)$. We say that two gap patterns G and H are equal, written $G = H$, if $L(G) = L(H)$. We define G to be equivalent to H, written $G \equiv H$, if the text of the gap patterns are equal when all of the gap symbols have been replaced by "---". The following lemma demonstrates that equality and equivalence are the same.

Lemma 1: If G and H are gap patterns over the same alphabet Σ, where $|\Sigma| \geq 3$, then $G \equiv H$ if and only if $G = H$.

Proof: If $G \equiv H$, then $L(G) = L(H)$ because gap substitutes are defined independently of the gap symbol involved; the converse is the interesting question.

Suppose that $L(G) = L(H)$ but that $G \not\equiv H$. The unique shortest element of $L(G)$ is $c(G)$, and the unique shortest element of $L(H)$ is $c(H)$, so $L(G) = L(H)$ implies that $c(G) = c(H)$. If $G \not\equiv H$, then G and H must have a different number of gaps, or have the same number of gaps but in different places. Suppose, without loss of generality, that $|g(G)| \geq |g(H)|$ and let i be the index of the leftmost gap in G that is in a different place in H:

$$\begin{aligned} G &= s_0 g_1 s_1 \ldots g_{i-1} s_{i-1} g_i s_i \ldots g_n s_n \\ H &= s_0 g_1 s_1 \ldots g_{i-1} s'_{i-1} g'_i s'_i \ldots g'_m s'_m \end{aligned}$$

Case $i \leq m$: The i'th gaps of G and H are in different places, so one of s_{i-1} or s'_{i-1} must be shorter; suppose that s_{i-1} is shorter than s'_{i-1}. For concreteness, suppose that the last character of s_{i-1} is an a and the first character of s_i is a b. This means that the

$|s_{i-1}|$'th and $|s_{i-1}|+1$'st characters of s'_{i-1} are also an a and a b. If c is some character in Σ different from a and b, then when gap g_i is filled with a c:

$$s_0s_1...s_{i-1}\mathtt{c}s_i...s_n$$

The resulting string cannot be matched by H because H has a b at that position, H has no gap at that position, and if H matched some other character with a previous gap it would leave the a matching the c at that position. But this string is a member of $L(G)$, which is equal to $L(H)$, which is a contradiction.

Case $i > m$: In this case, there is at least one extra gap in G past the last gap in H, and actually i would equal m. The same argument given in the previous case can be applied with s_{i-1} playing the role of the short string and s'_m playing the role of the longer one.

Thus if $L(G) = L(H)$, then $G \equiv H$. □

In all that follows we will assume that $|\Sigma| \geq 3$, and we will write $G = H$ instead of $G \equiv H$.

Gap patterns break the strings of their language up into fields in a natural way. Define a *parse* of a string $s \in L(G)$ relative to a gap pattern $G = s_0g_1s_1g_2s_2...g_ns_n$, denoted $parse(G,s)$, to be a sequence of n strings $p_1,p_2,...,p_n$ such that $s = s_0p_1s_1p_2s_2...p_ns_n$ and p_i is a legal substitute for g_i. For example, if Σ = {1, 2, (,), ␣} and G = "(*-1-*)␣ *-2-* - *-3-* .", then $L(G)$ includes the following strings (white space has been inserted in the strings to separate the portions generated by the different components of the gap pattern):

(212)␣ 222 - 1212 .

(21212)␣ 2121212 - 12212 .

( )␣ - .

((()))))))␣ (()()()()()((- ----12121 .

The first two strings are obviously in the language; the third, in which all of the gaps match the null string, is also in $L(G)$; and although the fourth string is not beautiful, it too is in the language. Some examples of strings that are not included in $L(G)$:

(212)222-1212.

(122)␣121-1222

(212)␣212-1212.␣(212)␣121-1211.

The first string fails to be in $L(G)$ because it is missing the "␣" after the ")", the second is missing the closing ".", and the third string includes extraneous text after the first ".". Note that the third string could be in the language if, for example, the definition of match allowed the pattern to match the second ")␣" rather than the first.

Why do we impose the restriction that the text that a gap can match must not include the following string? The reason is that we want the gap program's actions to be determinate. We use gap patterns as the parsing component of text processing programs that run without human intervention, and if these programs are not deterministic, then we will have less confidence in the correctness of their unmonitored behavior. The following result shows that the gap pattern matching process is deterministic:

Property 2: If $s \in L(G)$, then $parse(G,s)$ is unique.

Proof: This property is a simple consequence of the definition of legal substitute. Suppose that there were two ways of parsing a string s using G. If there are two different parses, then they must differ in some leftmost position, say position i. So the two parses are $p_1,p_2,...p_{i-1},p_i,p_{i+1},...,p_n$ and $p_1,p_2,...p_{i-1},p_i',p_{i+1}',...,p_n'$. If p_i' is the shorter of p_i and p_i', then $p_i's_i$ is a strict prefix of p_is_i, which implies that s_i occurs in p_is_i before the end, which in turn implies that p_i is not a legal substitute for g_i. And so, by contradiction, $parse(G,s)$ is unique. □

Gap patterns may be efficiently matched against their text.

Property 3: A gap pattern G may be matched against a string F in time $O(|G|+|F|)$.

Proof: Gap pattern parsing is deterministic, so a matching algorithm has only to find the locations of G's sequence of constant strings $s_0,s_1,...s_n$ in F. There are several well known algorithms suitable for finding the next s_i in F in linear time: Wiener's algorithm [96], Boyer and Moore's algorithm [16], or Knuth, Morris, and Pratt's algorithm [52]. A scanning procedure that uses one of these algorithms will run in linear time. □

The gap pattern notation is weak; it cannot illuminate the structural complexities expressible by context free grammars, or even by regular expressions. The following three results categorize this weakness:

Property 4: Gap patterns define sub-regular languages.

Proof: The regular expression denoting the language generated by the gap pattern fragment g_is_i over an alphabet Σ is:

$$\Sigma^*s_i - \Sigma^*s_i\Sigma^+$$

The fragment Σ^* denotes all strings; Σ^*s_i denotes all strings ending in s_i; Σ^+ denotes all non-null strings; $\Sigma^*s_i\Sigma^+$ denotes all strings where s_i occurs before the end; and the difference $\Sigma^*s_i - \Sigma^*s_i\Sigma^+$ generates all strings with s_i at the end but without an occurrence of s_i before the end. Note that the representation of the regular expression in terms of the standard primitives (closure, alternation, and concatenation) may be exponentially larger than the text written above. The regular expression denoting the language generated by an entire gap pattern is the concatenation of the languages generated by each of the fragments:

$$s_0(\Sigma^*s_1 - \Sigma^*s_1\Sigma^+)(\Sigma^*s_2 - \Sigma^*s_2\Sigma^+)...(\Sigma^*s_n - \Sigma^*s_n\Sigma^+)$$

and so gap patterns define sub-regular languages. □

Gap patterns are strictly sub-regular; their patterns cannot define all of the languages definable with regular expressions. For example, the following property shows that no gap pattern can generate the regular language {x,y}.

Property 5: If G is a gap pattern then $L(G)$ either is composed of one string or is infinite.

Proof: If there are no gaps in G, or if the alphabet Σ consists of a single character, then $L(G)$ consists of the single string composed of the constants of G. If there is a gap in G, say g_i, and $|\Sigma| \geq 2$, then let a be an element of Σ that is not equal to the first character of the string s_i that follows g_i in G. Then $a^* = \{a,aa,aaa,...\}$ forms an infinite set of valid substitutes for g_i, and thus $L(G)$ is infinite. □

Property 6: Gap patterns are not closed under union, intersection, or complement.

Proof: Let Σ be {a,b,c} and let G be the gap pattern --- b --- a and H be --- c --- a. We will demonstrate each of these properties using these two gap patterns as examples.

Union: $L(G) \cup L(H)$ includes the two strings "ba" and "ca". The only gap pattern that can match both of these is --- a, but this matches "a" as well, which is matched neither by G nor by H. (Note that the fact that all gap languages are either singletons or infinite also shows that they are not closed under union.)

Intersection: $L(G) \cap L(H)$ includes the two strings "bca" and "cba". There are only three gap patterns that match both of these strings: G, H, and --- a. None of these gap patterns capture the requirement that all strings in the intersection must contain both a "b" and a "c" before the final "a", e.g. $L($--- a$)$ contains "ba", which is not contained in $L(H)$.

Complement: $L(G)'$ includes the two strings "b" and "c". No gap pattern can match both of these strings simultaneously. □

This ends our discussion of the elementary properties of gap patterns. Now we will discuss gap programs, by first introducing their output-producing component, the gap replacement expressions.

4.2 Gap replacements and programs

A ***gap replacement R for a gap pattern G*** over an alphabet Σ with gap symbols $\{g_1, g_2, \ldots, g_n\}$ is a string from $(\Sigma \cup \{g_1, g_2, \ldots, g_n\})^*$. By convention, the anonymous gap symbol "---" never appears in a replacement expression. Gap replacements 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 gap replacement R for G; the program is denoted $G \Rightarrow R$. Gap programs bear an interesting resemblance to the string processing language based on Labeled Markov Algorithms proposed by Galler and Perlis [31].

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-* .

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

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

This gap program matches phone numbers in the "(203)" area code, and deletes the area code:

(203)␣ *-1-* - *-2-* . ⇒ *-1-* - *-2-* .

And this last changes numbers from area code "(211)" to "(203)":

(211)␣ *-1-* - *-2-* . ⇒ (203)␣ *-1-* - *-2-* .

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. If S is a set of pairs of strings $\{<i_1,o_1>, <i_2,o_2>, \ldots, <i_m,o_m>\}$, then we say that a gap program P is *concomitant* with S if $P(i_j) = o_j$ for every pair $<i_j,o_j>$ in S.

We do not use gap programs on isolated strings; rather, we use them to search for and modify subparts of a larger file. This process of repetitively applying the text transformation defined by a gap program to the text of a file is captured by a definition of the *search and replacement* operation. When a gap program P being interpreted under the search and replacement operation is applied to a string $F = axb$ then the result is the string ayb if and only if x is the leftmost substring of axb that is matched by G and $P(x) = y$. Repetitive applications of this program are made to the suffix b.

For example, if the gap program

bol *-1-* ␣ *-2-* ,␣ *-3-* ␣ *-4-* *eol* ⇒
bol *-3-* ␣lost␣to␣the␣ *-1-* ,␣ *-2-* ␣to␣ *-4-* . *eol*

were applied to the input data

```
Yankees 10, Baltimore 3
    0 0 2 0 1 0 0 7 0 10
    1 1 0 0 0 0 1 0 0 3
Mets 6, Chicago 5
    3 0 0 0 0 0 0 0 3 6
    0 0 4 0 1 0 0 0 0 5
Angels 2, Detroit 0
    0 0 0 0 1 0 0 1 0 2
    0 0 0 0 0 0 0 0 0 0
```

then it would modify the text to be:

```
Baltimore lost to the Yankees, 10 to 3.
    0 0 2 0 1 0 0 7 0 10
    1 1 0 0 0 0 1 0 0 3
Chicago lost to the Mets, 6 to 5.
    3 0 0 0 0 0 0 0 3 6
    0 0 4 0 1 0 0 0 0 5
Detroit lost to the Angels, 2 to 0.
    0 0 0 0 1 0 0 1 0 2
    0 0 0 0 0 0 0 0 0 0
```

The power of gap programs is limited by the expressive power of gap patterns. One consequence of this is the following result, which shows that more can be computed by taking another pass over the text.

Property 7: 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.

Proof: There is no single gap program that can transform these strings as shown:

```
cab$    ⇒  c
dba$    ⇒  d
```

The proof is by an exhaustive analysis of the possible gap programs. First observe that the only gap patterns that can match both input strings are the patterns "--- $", "--- a --- $", and "--- b --- $". This is the case because all gap patterns matching the strings must end in $; the only other constants that can be in a matching pattern are a and b; and they cannot both be in a matching pattern because they occur in the order ab in the first sample and ba in second. The output strings are composed of different characters, so the replacement expression must not contain any constants, but only gaps. None of the programs leaves the output characters isolated in a gap: "--- $" leaves both the c and the d sharing the gap with the string ab; the first gap in "--- a --- $" matches db; and the first gap in "--- b --- $" matches ca. So no gap program exists that can make this transformation in a single pass.

The transformation can be effected by the composition of these two gap programs:

-1- a *-2-* $ ⇒ *-1-* *-2-* $
-1- b$ ⇒ *-1-*

The first program deletes the a from each sample, and then the second program completes the transformation. □

Angluin pointed out that two gap programs can be composed together to compute any finitely specified function on sets of strings.

Property 8: (Angluin) Given a set of pairs of n strings $S=\{<i_1,o_1>,<i_2,o_2>,...,<i_n,o_n>\}$ such that $i_j \neq i_k$ for $j \neq k$, and such that there is some gap pattern G that matches all of the i_j, there exist two gap programs P_1 and P_2 such that $P_2(P_1(i_j))=o_j$ for all pairs in S.

Proof: Let {x,y,#} be three symbols not in Σ, then the first program P_1 is:

$$\begin{array}{ll} G \Rightarrow & \mathtt{x}G\mathtt{yx}\sigma_1\mathtt{yx}i_1\mathtt{yxyx\#x}G\mathtt{yx}\sigma_1\mathtt{yx}i_2\mathtt{yxyx\#x}G\mathtt{yx}\sigma_1\mathtt{yx}i_3\mathtt{yxyx\#}\ldots\mathtt{x}G\mathtt{yx}\sigma_1\mathtt{yx}i_n\mathtt{yxyx\#} \\ & \mathtt{x}G\mathtt{yx}\sigma_2\mathtt{yx}i_1\mathtt{yxyx\#x}G\mathtt{yx}\sigma_2\mathtt{yx}i_2\mathtt{yxyx\#x}G\mathtt{yx}\sigma_2\mathtt{yx}i_3\mathtt{yxyx\#}\ldots\mathtt{x}G\mathtt{yx}\sigma_2\mathtt{yx}i_n\mathtt{yxyx\#} \\ & \mathtt{x}G\mathtt{yx}\sigma_3\mathtt{yx}i_1\mathtt{yxyx\#x}G\mathtt{yx}\sigma_3\mathtt{yx}i_2\mathtt{yxyx\#x}G\mathtt{yx}\sigma_3\mathtt{yx}i_3\mathtt{yxyx\#}\ldots\mathtt{x}G\mathtt{yx}\sigma_3\mathtt{yx}i_n\mathtt{yxyx\#} \\ & \ldots \\ & \mathtt{x}G\mathtt{yx}\sigma_t\mathtt{yx}i_1\mathtt{yxyx\#x}G\mathtt{yx}\sigma_t\mathtt{yx}i_2\mathtt{yxyx\#x}G\mathtt{yx}\sigma_t\mathtt{yx}i_3\mathtt{yxyx\#}\ldots\mathtt{x}G\mathtt{yx}\sigma_t\mathtt{yx}i_n\mathtt{yxyx\#} \end{array}$$

The gap pattern of P_1 is any gap pattern G that matches all of the input samples in S. The replacement expression is a t by n matrix of fragments of symbols where fragment $[k,l]$ is:

$$\mathtt{x}G\mathtt{yx}\sigma_k\mathtt{yx}i_l\mathtt{yxyx\#}$$

The G in the replacement expression is a shorthand for the direct substitution of the symbols of G, which has the effect of copying the input string over into the output. The i_l in the replacement expression fragment stands for a substitution of the text of the l'th input string.

The idea behind the construction of P_2 is to parse the output of P_1 in such a way that there are tn different gaps, each labeled -$[k,l]$- for $1 \leq k \leq t$ and $1 \leq l \leq n$, that match the symbol σ_k when applied to sample $P_1(i_l)$, but match the empty string everywhere else. For each k and l, the $[k,l]$ gap fragment in the gap pattern of P_2 is:

--- $\mathtt{x}i_l\mathtt{yx}$ -$[k,l]$- $\mathtt{yx}$ --- #

When this fragment is matched against the corresponding fragment in the string $P_1(i_l)$:

$\mathtt{x}i_l\mathtt{yx}\sigma_k\mathtt{yx}i_l\mathtt{yxyx\#}$

the gap -$[k,l]$- matches the symbol σ_k. When the fragment is matched against the corresponding fragment in the strings $P_1(i_m)$, for $m \neq l$:

$\mathtt{x}i_m\mathtt{yx}\sigma_k\mathtt{yx}i_l\mathtt{yxyx\#}$

the gap -$[k,l]$- matches the null string. An arbitrary output can be constructed given gaps with these properties. If $o_j = \sigma_{k_1}\sigma_{k_2}\sigma_{k_3}...\sigma_{k_r}$, then the gap fragments -$[k_1,j]$- -$[k_2,j]$- -$[k_3,j]$- ... -$[k_r,j]$- can be concatenated together to yield output o_j without producing any of the characters of the other outputs.

The composition $P_2(P_1(i_j)) = o_j$ for $1 \leq j \leq n$. □

This finishes our development of the basic properties of gap programs; the rest of this chapter concentrates on the problem of identifying gap programs from examples.

4.3 The longest common subsequence problem

This section is an aside that discusses the Longest Common Subsequence problem, a problem that will be of value in our study of the identification of gap programs from examples. The Longest Common Subsequence (LCS) problem is that of finding a common subsequence of maximal length that occurs in every string of a set of n strings $Q = \{q_1, q_2, ..., q_n\}$. For example, the LCS of these two strings:

```
The⊔Yankees⊔beat⊔Baltimore,⊔10⊔to⊔3.
The⊔Mets⊔beat⊔Chicago,⊔6⊔to⊔5.
```

is this sequence:

```
T h e ⊔ e s ⊔ b e a t ⊔ i o , ⊔ ⊔ t o ⊔ .
```

The LCS is not always unique. For example, when we add another string to the pair above:

```
The⊔Yankees⊔beat⊔Baltimore,⊔10⊔to⊔3.
The⊔Mets⊔beat⊔Chicago,⊔6⊔to⊔5.
The⊔Angels⊔beat⊔Detroit,⊔2⊔to⊔0.
```

then an LCS is either this sequence:

```
T h e ⊔ e s ⊔ b e a t ⊔ o , ⊔ ⊔ t o ⊔ .
```

or this one:

T h e ␣ e s ␣ b e a t ␣ i , ␣ ␣ t o ␣ .

Researchers have viewed the problem of finding a LCS for two strings as an abstraction of the file comparison problem, and thus have given it a fair amount of attention. The problem was mentioned as being solvable in polynomial time by Chvatal, Klarner, and Knuth in 1972 [17]. In 1974, Wagner and Fischer [93] presented the first published algorithm that found the LCS of two strings; their algorithm is a special case of a more general dynamic programming algorithm for solving the least-cost edit distance problem. The LCS algorithm has a running time of $O(mn)$ when applied to two strings of size m and n; unfortunately, it also requires $O(mn)$ space. More efficient solutions to this problem have since been found: Hirschberg showed how to implement the algorithm to run in $O(m+n)$ space [41]; and Hunt and Szymanski [43] developed an algorithm that usually beats the $O(n^2)$ time bound. The latter algorithm runs in time $O((r+n)\ log\ n)$, where r is the number of ordered pairs of positions in which the two strings have a character in common.

These algorithms can be adapted to find the Longest Common Subsequence of more than two strings. This adaptation leads to an algorithm that solves the problem in time and space proportional to the product of the lengths of the strings. Such an algorithm is polynomial for a fixed number of strings, but has an exponential running time when the number of strings grows without bound. This running time is to be expected, because Maier [58] has shown that the general problem of finding the Longest Common Subsequence of an unbounded number of strings is NP-hard.

PROBLEM: *Longest Common Subsequence (LCS)*

INSTANCE: An alphabet Γ, a set of strings $Q = \{q_1, q_2, \ldots q_n\}$ drawn from Γ^*, and a positive integer k.

QUESTION: Does there exist a subsequence common to the q_i in Q of size k?

Theorem 9: (Maier) The Longest Common Subsequence problem is NP-complete [58].

Theorem 10: (Maier) The Longest Common Subsequence problem remains NP-complete even when the alphabet Γ has only two symbols [58].

In what follows, we will use these two results of Maier's to show the computational intractability of many aspects of the identification of gap programs from examples, and we will also use a heuristic for finding the LCS of a set of strings as part of a practical gap program synthesis procedure.

4.4 Finding gap programs from examples

A procedure for identifying gap programs from examples might look something like Algorithm 1 below. The algorithm starts with one sample input/output pair and has as its initial hypothesis the gap program that replaces all occurrences of that particular input string by the output string. It then enters a loop in which it accepts new input/output pairs that further describe the behavior of the target gap program. The current program hypothesis is retained as long as it agrees with the behavior described by the new input/output example; when the hypothesis fails to agree with the evidence, then the procedure tries to find a "good" replacement program that does agree. The algorithm resembles an enumerative algorithm; indeed, setting aside for the moment the stated goal of choosing a good new *P*, an enumerative search for any *P* consistent with the evidence may be used to find a new gap program when the current one stops working. The algorithm differs from an enumerative one in its insistence on replacing a failed hypothesis with a good one — not just any one will do; it wants one that will serve as good intermediate behavior for the system.

```
$S \leftarrow \{<\textit{first input, first output}>\}$;
$P \leftarrow \textit{first input} \Rightarrow \textit{first output}$;
loop
    use P as the current guess;
    $s \leftarrow <\textit{new input, new output}>$;
    $S \leftarrow S \cup \{s\}$;
    if $P(s_{input}) \neq s_{output}$ then
        $P \leftarrow$ a "good" new P concomitant with S;
forever;
```

Algorithm 1: Iterative synthesis of gap programs

What are some reasonable definitions of "good"? A good program could be the shortest, simplest, most concise gap program that works. Or we might actually want the most complex gap program, the one that finds as tricky a transformation as possible. Another plausible definition is the gap program that corrects the flaws of the previous hypothesis, but with as few changes to the previous hypothesis as possible. Each of these criteria for "goodness" has something to recommend it, as do many others that could be proposed.

But leaving aside our desire for a good gap program, how hard is it to find *any* gap program at all that will work? In the following theorem, we show that the general problem of finding a gap program that fits a set of sample data is intractable. Not only is it NP-hard, but it is still NP-hard even when the set of sample data contains only three input/output pairs. The Gap Program Existence decision problem captures the essence of the problem of finding a gap program that fits a set of sample data:

PROBLEM: *Gap Program Existence*

INSTANCE: An alphabet Σ, and a set of pairs of strings $S = \{<i_1,o_1>, <i_2,o_2>, \ldots, <i_n,o_n>\}$ from Σ^* describing the input/output behavior of a text transformation.

QUESTION: Does there exist a gap program P such that $P(i_j) = o_j$ for all $<i_j,o_j>$ in S?

This problem is NP-complete, even when S contains a bounded number of samples.

Theorem 11: The Gap Program Existence problem is NP-complete when n, the number of input/output samples, is greater than or equal to three.

Proof: A nondeterministic procedure to determine Gap Program Existence would simply guess P and verify that $P(i_j) = o_j$ for all of the input/output pairs in S. The verification procedure runs in polynomial time, so this problem is in NP.

We prove that this problem is NP-complete by reduction from the Longest Common Subsequence problem over an alphabet of size two, a problem that was proven NP-complete by Maier [58] (see Section 4.3). Suppose that we have an instance of the Longest Common Subsequence problem over an alphabet $\Gamma = \{x,y\}$ that consists of a set of n strings $Q = \{q_1,q_2,\ldots,q_n\}$ from Γ^* and a positive integer k. We outline the construction of an instance of the Gap Program Existence problem over a set of three input/output samples S such that there is a common subsequence of size k in Q if and only if there a gap program P computing the transformation described by S. The alphabet Σ consists of the symbols $\{x,y\} \cup \{a_1,a_2,\ldots a_k\} \cup \{\$_1,\$_2,\ldots\$_k\} \cup \{\#,\%\}$. The constructed sample set S consists of three input/output pairs. The three input strings are:

$$x\#y\#a_1xa_1a_1ya_1\#a_2xa_2a_2ya_2\#...a_kxa_ka_kya_k\#$$
$$\#\#a_1\$_1a_1a_1\$_1a_1\#a_2\$_2a_2a_2\$_2a_2\#...a_k\$_ka_ka_k\$_ka_k\#$$
$$\#\#a_1a_1a_1\#a_2a_2a_2\#...a_ka_ka_k\#$$

After the initial input string fragments, <x#y#, ##, ##>, the input consists of k repetitions of the fragment $<a_ixa_ia_iya_i\#,\ a_i\$_ia_ia_i\$_ia_i\#,\ a_ia_ia_i\#>$, where i ranges from 1 to k. The output strings corresponding to these inputs in S are:

$$q_1\%q_2\%...q_n\%$$
$$\$_1\$_2...\$_k\%\$_1\$_2...\$_k\%...\$_1\$_2...\$_k\%$$
$$\%\%...\%$$

These output strings consist of n repetitions of the fragment $<q_i\%,\ \$_1\$_2...\$_k\%,\ \%>$, for i ranging between 1 and n.

We show that if there is a common subsequence of size k in Q then there is a gap program that will transform the inputs to the corresponding outputs. A common subsequence of size k in Q is a sequence of k symbols $z_1z_2...z_k$ where each z_j is either an x or a y. This common subsequence can be used to construct the gap program to transform the inputs of S to the corresponding outputs. The pattern used in the gap program is of the form:

$$\text{-}x\text{-}\ \#\ \text{-}y\text{-}\ \#\ g_1\ g_2\ ...\ g_k$$

where the g_j are pattern fragments that we will specify below. The gap -x- matches the string x in the first sample, and the null string in the second and third. Similarly, the gap -y- matches the string y in the first sample and the null string in the other two. The gap fragments substituted for g_j differ depending upon whether symbol j in the common subsequence is an x or a y. If z_j is an x, then the gap pattern fragment a_j -x_j- a_j --- # is used. In the other case, if z_j is a y then g_j is the fragment --- a_ja_j -y_j- a_j#. In the case of the former, the gap named -x_j- matches the symbol x in the first sample, the symbol $\$_j$ in the second sample, and a null string in the last sample. Similarly, in the latter case -y_j- matches a y, a $\$_j$, and the null string, respectively.

The replacement expression is constructed in pieces, one piece to account for each of the output fragments $<q_j\%,\ \$_1\$_2...\$_k\%,\ \%>$. All three strings in this fragment end in a % symbol, and so the replacement expression yielding the fragment also ends in the constant %. The primary constraint on this construction is finding a way to account for the $\$_i$

symbols in the second fragment — the $\$_i$ symbols do not occur in the other two fragments, so they must be the product of a copy of a gap matched in the input. When one or the other of the gaps -x_i- and -y_i- match the symbol $\$_i$ in the second sample, it also matches either an x or a y, respectively, in the first sample. When this gap is used to copy the $\$_i$ to the second output, it also copies the x or y to the first one. This parallel copying is valid because the symbols matched by each of the gaps -x_i- or -y_i- in the first input form a common subsequence $z_1z_2...z_k$ of each q_j in the first output. The rest of the replacement expression may be formed from the gaps -x- and -y-. These two gaps match either an x or a y, respectively, in the first sample and the null string in the other two samples, and so they may be used to fill in those parts of q_j that are not part of the common subsequence. The constructed gap program performs the transformation from each input string to the corresponding output string.

To illustrate this construction, let $Q = \{$xxyx, xyyx, xyxyx$\}$ and k be 3. Then the set of input samples constructed is:

$$\text{x\#y\#}a_1\text{x}a_1a_1\text{y}a_1\#a_2\text{x}a_2a_2\text{y}a_2\#a_3\text{x}a_3a_3\text{y}a_3\#$$

$$\#\#a_1\$_1a_1a_1\$_1a_1\#a_2\$_2a_2a_2\$_2a_2\#a_3\$_3a_3a_3\$_3a_3\#$$

$$\#\#a_1a_1a_1\#a_2a_2a_2\#a_3a_3a_3\#$$

and the corresponding set of output samples is:

$$\text{xxyx\%yxyxy\%xyxyx\%}$$

$$\$_1\$_2\$_3\%\$_1\$_2\$_3\%\$_1\$_2\$_3\%$$

$$\%\%\%$$

then the program constructed from the common subsequence xyx would be:

$$\text{-}x\text{-}\ \#\ \text{-}y\text{-}\ \#\ a_1\ \text{-}x_1\text{-}\ a_1\ \text{---}\ \#\ \text{---}\ a_2a_2\ \text{-}y_2\text{-}\ a_2\#\ a_3\ \text{-}x_3\text{-}\ a_3\ \text{---}\ \#\ \Rightarrow$$

$$\text{-}x_1\text{-}\ \text{-}x\text{-}\ \text{-}y_2\text{-}\ \text{-}x_3\text{-}\ \%\ \text{-}y\text{-}\ \text{-}x_1\text{-}\ \text{-}y_2\text{-}\ \text{-}x_3\text{-}\ \text{-}y\text{-}\ \%\ \text{-}x_1\text{-}\ \text{-}y_2\text{-}\ \text{-}x_3\text{-}\ \text{-}y\text{-}\ \text{-}x\text{-}\ \%$$

One can verify that this program maps the inputs to the outputs as indicated.

Conversely, if there is a program $P = G \Rightarrow R$ that maps the inputs of S to the outputs, then we will show that there must be a common subsequence of size k in Q. This will be derived from the following collection of facts:

1. The n % symbols in each of the output strings do not occur in the input, so the replacement expression R must contain the % symbols as constants. So R may be broken up at each of these constants into pieces that account for

each of the output fragments $<q_j\%,\ \$_1\$_2...\$_k\%,\ \%>$ separately.

2. Each of the $\$_i$ symbols in the second output arises from a gap copied from the input, because the $\$_i$ do not occur in the other two output strings.
3. Adjacent $\$_i$ and $\$_{i+1}$ symbols must come from different gaps because they do not occur next to each other in the input. So each particular $\$_i$ must be *isolated* as the sole symbol matched by some gap in the second input; call this gap $\text{-}g_i\text{-}$.
4. In order to isolate a $\$_i$ in the input, the gap pattern G must contain as constants each of the symbols to either side of the $\$_i$ matched. That is, G must contain the fragment $a_i\ \text{-}g_i\text{-}\ a_i$.
5. There are two $\$_i$ symbols in the second input string, and each of the two $\$_i$ symbols is surrounded by two distinct a_i, so any gap pattern that isolates both of the $\$_i$ symbols must contain four a_i symbols as constants.
6. The third input string contains only three a_i symbols, so a pattern with four a_i could not match this input, and thus any gap pattern that matches the input can isolate at most one of the $\$_i$.

The choice of which $\$_i$ is isolated amounts to choosing whether x or y is the i'th member of the common subsequence spanning Q, because any gap pattern that isolates a $\$_i$ in a gap also isolates either an x or a y in the same gap in the first input. When the n instances of the symbols $\$_1\$_2...\$_k$ are inserted in the second output, a corresponding sequence of k x or y symbols will be inserted in each of the n strings $q_1,q_2,...q_n$. And thus there is a common subsequence of size k in Q if there is a gap program performing the mapping specified by S.

The sample set S may be constructed in polynomial time, so the Gap Program Existence problem is NP-complete even when the sample set is of size three. □

4.5 Complexity of finding a gap pattern

This section considers the problem of finding a gap pattern to match a set of strings. It does not consider the problem of finding a "good" gap pattern — any gap pattern at all will do. This is not a very important problem for gap program synthesis from positive data, because when working from positive data, any gap pattern at all will *not* do, a good one is required. Nevertheless, the problem is interesting because it seems easy at first glance, but turns out to be intractable. It is surprising, but true, that in general it is hard to find a gap pattern to match a set of strings.

PROBLEM: *Gap Pattern Existence*

INSTANCE: An alphabet Σ, a set of strings $S = \{s_1, s_2, ... s_n\}$ drawn from Σ^*.

QUESTION: Does there exist a gap pattern G such that $S \subseteq L(G)$?

Theorem 12: Gap Pattern Existence is NP-complete.

Now that the result has been stated, it would be best to skip on to the next section, and forego the proof. In Section 4.8 we will point out that this problem can be solved in polynomial time when the number of strings in S is bounded.

Proof: This problem is in NP, as one can verify in linear time that a nondeterministically chosen gap pattern matches a set of strings. We show that this problem is NP-complete via a reduction from the Longest Common Subsequence problem. An instance of LCS is a set of strings $Q = \{q_1, q_2, ..., q_n\}$ over an alphabet Γ, and a positive integer k, along with the question of whether the strings in Q have a common subsequence of length k. We construct an instance of the Gap Pattern Existence problem with an alphabet $\Sigma = \Gamma \cup \{\$, \#, \&\}$. The set of strings S consists of $n+1$ strings, with the first n being based on the corresponding strings in Q. If $q_i = q_{i1} q_{i2} ... q_{il_i}$, then

$$s_i = q_{i1}\#q_{i2}\#...\#q_{il_i}\#\$$$

String s_{n+1} in S will be called the *lock string*. If $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_t\}$, and m is a positive integer equal to one greater than the length of the longest string in Q, then the lock string is formed from k repetitions of the string ###...###$ (containing m # symbols) concatenated with the string $\gamma_1\gamma_2...\gamma_t$. In addition, every character in the lock string is preceded by the & character. This is a representation of the lock string that uses an exponentiation notation in which $(x)^n$ signifies a string consisting of n consecutive copies of x:

$$s_{n+1} = ((\&\#)^m \&\$\&\gamma_1\&\gamma_2...\&\gamma_t)^k \&\$$$

This completes the construction of S.

If $\gamma_{l_1}\gamma_{l_2}...\gamma_{l_k}$ forms a subsequence common to Q of size k, then the following gap pattern will match all elements of S:

$$G = \text{---}\ \gamma_{l_1}\ \text{---}\ \#\ \text{---}\ \gamma_{l_2}\ \text{---}\ \#\ ...\ \text{---}\ \gamma_{l_k}\ \text{---}\ \#\ \text{---}\ \$$$

Conversely, if there is a gap pattern G that matches the strings in S, then we will show that there must be a subsequence of size k common to the strings in Q. First, some properties that must hold true of any gap pattern G that matches S.

1. G must end in \$, because that is the last symbol of each string in S.
2. The constants of G can contain no \$ before the end, because the first n samples of S contain only the one \$.
3. No constant string in G is longer than one symbol, because the & symbol separates every pair of symbols in the lock string, but does not occur in any other string in S.
4. G is of the form "--- a_1 --- a_2 ... --- a_p --- \$" for $a_i \in \Gamma \cup \{\#\}$.
5. Fewer than m of the a_i symbols are a #, because the first n strings in S contain fewer than m # symbols.

We claim that at least k of the a_i in G must be symbols from Γ. If G contains fewer than k symbols from Γ, then G can be broken at each occurrence of an a_i from Γ into no more than $k-1$ pieces:

$$\begin{aligned} G = {} & \text{---}\ \#\ \text{---}\ \#\ \ldots\ \text{---}\ \gamma_{l_1} \\ & \text{---}\ \#\ \text{---}\ \#\ \ldots\ \text{---}\ \gamma_{l_2} \\ & \quad \cdot\ \cdot\ \cdot \\ & \text{---}\ \#\ \text{---}\ \#\ \ldots\ \text{---}\ \gamma_{l_{k-1}} \\ & \text{---}\ \#\ \text{---}\ \#\ \ldots\ \text{---}\ \$ \end{aligned}$$

Each of the fragments terminating in a symbol from Γ can span no more than one occurrence of the substring $(\&\#)^m\&\$\&\gamma_1\&\gamma_2\ldots\&\gamma_t$ in the lock string. The lock string contains k occurrences of this substring, and thus the final fragment from G, --- # --- # ... --- \$, can be absorbed by the initial $(\&\#)^m\&\$$ fragment in the k'th occurrence of the substring, and G cannot match the final \$. And so G cannot match all of S without containing at least k constants from Γ. These constants form a common subsequence of Q, and thus if there is a G matching S then there must be a common subsequence of size k in Q.

This transformation may be computed in polynomial time, and so the Gap Pattern Existence problem is NP-complete. □

4.6 Decomposing the problem

We have shown that it is difficult to find a gap program that performs the actions specified by a set of input/output samples. The problem is NP-hard, even when the number of input/output samples is bounded. But we would like to be able to construct a system that can solve this problem efficiently enough to be a viable component of an interactive text editor.

Towards this end, we finesse the problem by dividing and conquering, and decompose the problem of synthesizing a gap program into two steps. The first step finds the "best" gap pattern G that matches all of the input strings in the sample set. This gap pattern yields a parse of the the input samples of S, and the second step of the algorithm attempts to find a replacement expression R that rearranges 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. The problem as a whole is NP-hard even when there are only three samples; however we will show that although each of the two steps of the decomposed process are also NP-hard, they can be solved in time polynomial in the size of the input when the number of samples is bounded.

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 will show 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 by sometimes requiring the user to supply more data than a monolithic gap program synthesis process would require; in practice, this penalty is rarely paid.

In the following sections, we define what we mean by a "best" gap pattern, present a heuristic algorithm for synthesizing that pattern from a set of strings, and give an algorithm for synthesizing a replacement expression from the parse yielded by the gap pattern. We argue that this procedure identifies a gap program in the limit from positive data; and perhaps more importantly, that it has good intermediate behavior as well.

4.7 Descriptive gap patterns

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 spans 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. As an example, if S is the following string:

```
The Yankees beat Baltimore, 10 to 3.
```

then the following gap pattern is the unique descriptive gap pattern for the set:

```
The␣Yankees␣beat␣Baltimore,␣10␣to␣3.
```

When we add a second string to S:

```
The Yankees beat Baltimore, 10 to 3.
The Mets beat Chicago, 6 to 5.
```

there is still just one descriptive gap pattern, namely:

```
The␣ -1- e -2- s␣beat␣ -3- i -4- o -5- ,␣ -6- ␣to␣ -7- .
```

When a third sample is included:

```
The Yankees beat Baltimore, 10 to 3.
The Mets beat Chicago, 6 to 5.
The Angels beat Detroit, 2 to 0.
```

then there are two descriptive gap patterns:

```
The␣ -1- e -2- s␣beat␣ -3- o -4- ,␣ -5- ␣to␣ -6- .

The␣ -1- e -2- s␣beat␣ -3- i -4- ,␣ -5- ␣to␣ -6- .
```

One might wonder why a gap pattern like "--- ." does not serve to describe the structure of the input strings just as well as one of the descriptive gap patterns. The reason that we prefer the descriptive gap patterns is that we want be able to identify a particular gap pattern quickly, with a small number of positive examples. To keep the number of examples required low, we prefer to have a criterion that jumps to conclusions based on slim evidence to one that chooses to ignore evidence that is there. Such behavior also seems preferable if we are to do synthesis from positive data: an algorithm that chooses to ignore what could be a significant feature common to the samples would probably continue to ignore the feature when more positive data is given.

We would like to find an algorithm that will always identify the gap program that will solve the user's problems. But, the output of any program synthesis system is determined by its input, so instead we will characterize the kind and number of inputs that will be required by our algorithms to identify a particular gap program. The characterization given here results from a worst-case analysis; something closer to an average case analysis may be found in Chapter 5.

We characterize the convergence properties of algorithms that identify descriptive gap patterns by making use of a set of strings called the *expanded once* strings of G, or $eo(G)$. The set of strings $eo(G)$ generated from a gap pattern G over an alphabet Σ are exactly those strings from $L(G)$ that are of length $|c(G)|$ and $|c(G)|+1$. For example, if $\Sigma = \{a,b,c\}$, then eo(a --- bc --- c) is the set:

```
{abcc,
 aabcc, abbcc, acbcc,
 abcac, abcbc}
```

Lemma 13: If G and H are gap patterns over the same alphabet Σ, where $|\Sigma| \geq 3$, then $G = H$ if and only if $eo(G) = eo(H)$.

Proof: Lemma 1 on page 41 states that $G = H$ if and only if $L(G) = L(H)$, but all of the strings constructed in the proof are of length $|c(G)|$ or $|c(G)|+1$, and are thus members of $eo(G)$. □

In practice, the input strings are tokenized (see Section 5.11), and the alphabet size can be regarded as unbounded. When the alphabet is of unbounded size the gap pattern $G = s_0 g_1 s_1 g_2 s_2 \ldots g_n s_n$ can be characterized by two samples with the aid of two symbols \$ and # that do not occur in the constants of G. That is, G is the unique descriptive gap pattern that matches this set of strings:

$$s_0 \$ s_1 \$ s_2 \$ \ldots \$ s_n$$
$$s_0 \# s_1 \# s_2 \# \ldots \# s_n$$

These two samples make the underlying gap pattern easy to identify, not because they each contain many repetitions of the same character, but because their gaps are filled with symbols in such a way that they cannot be mistaken. The second sample does not contain a $ symbol, and the first does not contain a #, so neither of these characters can be part of any constant string in a gap pattern that matches the two strings, but all of the other characters in the sample can. So the two strings pin down the underlying descriptive gap pattern unambiguously. For similar reasons, these two strings also identify the pattern:

$$s_0 s_1 s_2 \ldots s_n$$
$$s_0 \# s_1 \# s_2 \# \ldots \# s_n$$

The user can uniquely identify a gap pattern by giving two similar samples that do not contain inessential features in common. In practice, the system cannot count on the user to give examples that are quite as noise-free as these. And a formal treatment that assumes a finite alphabet demands that we restrict our ability to conjure up symbols like $ and #. In this case, a more technical result applies:

Lemma 14: If $|\Sigma| \geq 3$, then any gap pattern G is the unique descriptive gap pattern of a certain sample set of size $|g(G)|+1$ each of whose strings has length $\leq |c(G)|+1$.

Proof: If G is $s_0 g_1 s_1 g_2 s_2 \ldots g_n s_n$, and a_i are symbols from Σ differing from the first symbol of s_i and the last symbol of s_{i-1}, $1 \leq i \leq n$, then G is the unique descriptive gap pattern for the following subset S of $eo(G)$:

$$s_0 s_1 s_2 \ldots s_n$$
$$s_0 a_1 s_1 s_2 \ldots s_n$$
$$s_0 s_1 a_2 s_2 \ldots s_n$$
$$\cdot \quad \cdot \quad \cdot$$
$$s_0 s_1 s_2 \ldots a_n s_n$$

One of the samples in S is $c(G)$, so $c(G)$ is of maximum length for any gap pattern matching S, and is the unique such sequence of constants. The argument given in Lemma 1 suffices to show that a gap pattern H that matches S with $c(H) = c(G)$ must contain a gap at every point that G contains a gap, and so G is the unique pattern with the maximal number of constants and the minimal number of gaps. □

The following theorem provides something of a formal justification for the use of the term "descriptive". It shows that a descriptive gap pattern defines a language that tightly encloses the sample strings that were used in its derivation.

Theorem 15: Let G be a descriptive gap pattern G for a sample set S. Then $L(G)$ is a minimal gap pattern language containing S. That is, if H is a gap pattern, and $S \subseteq L(H) \subseteq L(G)$, then $H = G$.

Proof: Suppose that there is another gap pattern H with $S \subseteq L(H) \subseteq L(G)$; we show that H must equal G. First, we know that $|c(H)| \leq |c(G)|$, because $|c(G)|$ is maximal for all gap patterns that span S. Also, the shortest string in $L(G)$ is $c(G)$, and there is no other string of that length in $L(G)$, so $c(H)$ must be equal to $c(G)$, and thus G and H could differ only in the number and placement of their gaps. H cannot have fewer gaps, because G is descriptive. If H has more gaps, or gaps that occur in different places, then the construction of Lemma 1 shows that there must be a member of $eo(H)$ that is not in $eo(G)$, which contradicts $L(H) \subseteq L(G)$. □

4.8 Synthesizing descriptive gap patterns from examples

This section discusses ways of synthesizing a descriptive gap pattern that matches a set of samples. It turns out that this task is computationally difficult, so we present several progressively more efficient algorithms that are successively more heuristic and approximate.

Our first algorithm, Algorithm 2, is an existential algorithm that starts by considering the set of all gap patterns that match a set of strings and then winnows that set down until only the descriptive patterns are left. This algorithm will obviously find a descriptive gap pattern if the set can be matched by any gap pattern at all, and no proof of this fact will be supplied.

```
Input: A set of strings S.
Output: A descriptive gap pattern for S;

Consider the set of all gap patterns G matching S;
Discard all but the patterns with the largest number of constants;
Discard all but the patterns with the fewest number of gaps;
return one of the remaining gap patterns;
```

Algorithm 2: Finding a descriptive gap pattern.

An existential algorithm like Algorithm 2 will obviously not do; it will be too slow. We need something constructive, like Algorithm 3. This algorithm does not require knowledge of all the gap patterns matching a set. It works by first finding the constants of the pattern, and then inserting a minimal number of gaps into this string to make it into a gap pattern that matches the set. Setting questions of implementation aside for the moment, it is easy to see that this algorithm works, and that it will correctly identify a descriptive gap pattern for a set of strings.

```
Input: A set of strings $S$.
Output: A descriptive gap pattern for $S$;

Find $c(G)$, the constants of a descriptive gap pattern matching $S$;
Insert the minimal number of gaps into $c(G)$ to transform
    it to a gap pattern $G$ that matches $S$;
return $G$;
```

Algorithm 3: Finding a descriptive gap pattern.

4.8.1 Complexity of descriptive gap pattern synthesis

The following theorem characterizes the inherent computational difficulty of the approach taken by these two algorithms. The theorem and its proof may seem to be redundant, in that Theorem 12 shows that finding any sort of gap pattern at all to match a set is NP-hard, but we reproduce it here because the reduction in the proof is more natural than that in Theorem 12. The problem of finding *any* gap pattern that matches a set of strings is usually easy in practice, because the set of strings is usually simple in structure; however, the simplicity of the reduction contained in the following proof hints that descriptive gap patterns are more difficult to find in practice. While we could usually solve the former problem using an exact algorithm, we should consider heuristic algorithms for finding descriptive gap patterns.

PROBLEM: *Finding a Descriptive Gap Pattern*

INSTANCE: An alphabet Σ, a set of strings $S=\{s_1,s_2,\dots s_n\}$ drawn from Σ^*, and two non-negative integers c and g.

QUESTION: Is there a gap pattern G matching S with $|c(G)|\geq c$ and $|g(G)|\leq g$?

Theorem 16: The problem of Finding a Descriptive Gap Pattern is NP-complete.

Proof: By reduction from the Longest Common Subsequence problem. An instance of LCS is a set of strings $Q=\{q_1,q_2,\dots,q_n\}$ over an alphabet Γ and a positive integer k. The Descriptive Gap Pattern problem constructed from this has an alphabet

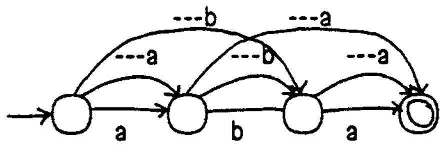
$\Sigma = \Gamma \cup \{a_1, a_2, \ldots, a_n\} \cup \{\$\}$, $c = k+1$, $g = k+1$, and the set of strings S of size n. Each s_i in S is constructed from the corresponding $q_i = q_{i1} q_{i2} \ldots q_{il_i}$ in Q as follows:

$$s_i = q_{i1} a_i q_{i2} a_i \ldots q_{il_i} a_i \$$$

There is a common subsequence of size k in Q if and only if there is a gap pattern matching S with at least $k+1$ constants and no more than $k+1$ gaps. If there is a common subsequence of size k, say $\gamma_1 \gamma_2 \ldots \gamma_k$, then the gap pattern --- γ_1 --- γ_2 ... --- γ_k --- \$ matches all of the samples and satisfies the bounds. Conversely, if there is a gap pattern containing $k+1$ constants matching S, then we know that none of the constants in the pattern are a_i, and thus at least k of these constants are symbols from Γ. These k symbols form a common subsequence of size k in Q. □

A descriptive gap pattern can be found in polynomial time when there are a bounded number of strings in the sample set. For example, if S has n members, each of length bounded by l, then there is an algorithm running in time $O(l^{3n+1} \log l)$ that will find a descriptive gap pattern matching S.

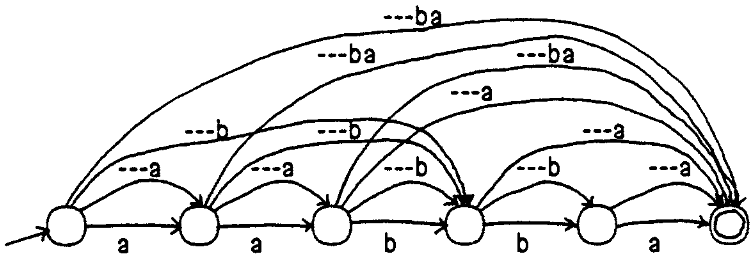
This algorithm works by constructing a concise representation of all possible gap patterns matching the sample set, and then winnowing out a descriptive gap pattern from that representation. The first step in constructing the representation of all possible gap patterns matching the set is to construct a concise representation of all possible gap patterns that match a particular sample. This representation is a finite automaton [42] that recognizes, or generates, all possible gap patterns that match the sample. The finite automaton M_i constructed for sample $s_i \in S$ has $|s_i|+1$ major states, named with the integers from 0 to $|s_i|$, where state 0 is the start state, and state $|s_i|$ is the sole accepting state. M recognizes strings formed from the alphabet $\Sigma \cup \{---\}$, where --- is the anonymous gap symbol. One class of state transitions in the automaton are those that go from state $j-1$ to state j on the symbol s_{ij}, the jth character of s_i; these transitions are used to generate some of the constant portions of the gap pattern. There is also a transition through a sequence of l auxiliary states from state $j-1$ to state k in M_i on the string --- $\sigma_1 \sigma_2 \ldots \sigma_l$ if the substring $s_{ij} s_{ij+1} \ldots s_{ik}$ can be matched by the gap pattern --- $\sigma_1 \sigma_2 \ldots \sigma_l$ but the substring $s_{ij} s_{ij+1} \ldots s_{ik-1}$ cannot be matched by the gap pattern --- $\sigma_1 \sigma_2 \ldots \sigma_{l-1}$. These two kinds of state transitions are the only non-failure transitions in M_i. For example, if s_1 were aba, then M_1 would be the automaton:



This automaton accepts the following set of twelve strings which, not coincidentally, are all the gap patterns that match the string aba.

aba	--- aba	--- ba
ab --- a	--- ab --- a	--- b --- a
a --- ba	--- a --- ba	a --- a
a --- b --- a	--- a --- b --- a	--- a --- a

If the second sample s_2 were the string aabba then M_2 would be an automaton recognizing a language consisting of 73 distinct strings:



The algorithm computes a finite automaton of this kind for each of the samples in the set, yielding n finite automata $M_1, M_2, \ldots M_n$. Each of these finite automata encodes the set of all of gap patterns that match a particular string; the intersection of these sets is the set of gap patterns that simultaneously match all of the sample strings. The intersection can be computed efficiently by forming the intersection of the machines using the classical finite state machine intersection algorithm that constructs an intersection machine containing a state for each pair of states in the two machines being intersected [42]. Intersection is associative, so this process can be done pairwise by first forming $M_1 \cap M_2$ yielding M' and then forming $M' \cap M_3$, etc.. The resulting machine $M = (M_n \cap (M_{n-1} \cap (\ldots \cap (M_2 \cap M_1))))$ recognizes those gap patterns that match all strings in the sample set.

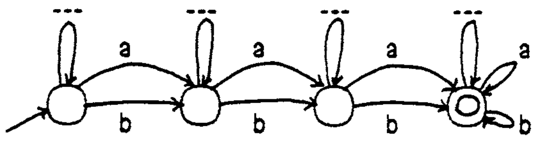
The running time of the algorithm that constructs M is polynomial if the number of samples in S is assumed to be bounded. If s_i is of length l, then there are no more than $l-j+1$ transitions leaving each major state j, and each of these transitions is labeled with a string of length at most $l-j+1$, so there are no more than $O(l^3)$ states in M_i, including the auxiliaries. Each of the M_i can be constructed in time $O(l^3)$, and so the total running time of the machine construction phase is $O(nl^3)$. The intersection of two finite automata M_i and M_j can be implemented to run in time $O(|M_i|\ |M_j|)$, and it yields a finite automaton of size $O(|M_i|\ |M_j|)$. Thus, the entire intersection phase runs in time $O(l^{3n})$ and yields a machine M with $O(l^{3n})$ states.

The machine M can be used to solve the Gap Pattern Existence problem. To solve the problem, simply minimize the number of states in M, which may be done in $O(l^{3n})$ time using an algorithm that takes advantage of the acyclic nature of the machines. If M has any states left at all after the minimization process is complete, then a gap pattern exists that matches all of the strings of S. A particular gap pattern that matches the samples may be generated by finding some path from M's start state to M's accepting state and returning the strings encountered along the traversal. Thus we have shown:

Theorem 17: A gap pattern to match a set of n strings of length bounded by l may be synthesized in time $O(l^{3n})$. Thus, the Gap Pattern Existence problem may be solved in polynomial time for a bounded number of strings.

We still have not found a descriptive gap pattern; M encodes all gap patterns that match the set, and we wish to find the one that does so with the greatest number of constants and the fewest number of gaps. We would like to find such a pattern in polynomial time; however, there may be an exponentially growing number of accepting paths through M, so a brute-force search through all of the strings accepted by M is out of the question.

To find a descriptive gap pattern using M, we have to find the strings accepted by M that contain the greatest number of constants, and among those, isolate the ones with the fewest number of gaps. One can test to see if the language accepted by M contains any gap patterns with at least k constants by intersecting M with an automaton that generates all gap patterns that contain at least k constants. Here is an example of a finite state automaton that generates all gap patterns containing at least three constants over the alphabet {a,b} with gap symbol ---:



This automaton has $O(k)$ states, and its intersection with M can be constructed in time $O(kl^{3n})$. Thus one can discover in $O(kl^{3n})$ time whether S can be matched by a gap pattern with at least k constants. The number of constants in any gap pattern that can match S is bounded by the length of the shortest string in S, which is bounded by l. And so the gap patterns with the maximum number of constants in them that are recognized by M can be found in $log\ l$ applications of the intersection algorithm using a binary searching process. The whole process takes $O(l^{3n+1}\ log\ l)$ time. A similar technique can then be used to further reduce the gap patterns to those that contain the minimal number of gaps.

Theorem 18: A descriptive gap pattern for a sample set S consisting of n samples of length at most l can be synthesized in $O(l^{3n+1}\ log\ l)$ time. The Descriptive Gap Pattern Synthesis problem can be solved in polynomial time for bounded n.

This algorithm for descriptive gap program synthesis is not practical, and while its performance can probably be improved somewhat, we have chosen to develop a heuristic to solve this problem.

4.8.2 A heuristic approach

The algorithms that we have described for finding a descriptive gap pattern are not efficient enough to be practical; our first serious crack at solving this problem is encoded in Algorithm 4. This algorithm is heuristic; it starts by computing an approximation to $c(G)$ by finding the Longest Common Subsequence of the sample strings. The next step is to attempt to convert this common subsequence to a gap pattern matching the set by inserting the minimal number of gaps required to make the gap pattern match S. If this step succeeds, then the gap pattern synthesized is returned; if it does not, then the algorithm reports that it failed to find a pattern.

The first step of the algorithm is to find the constants that are to go into the gap pattern; the second step is to find the places in that constant string that require gaps. We will consider how to perform each of these two steps in turn.

```
Input: A set of strings $S$.
Output: A pseudo–descriptive gap pattern for $S$;

Find an approximation to $c(G)$ by finding the LCS of $S$;
Try to insert the minimal number of gaps into the LCS to
    transform it into a gap pattern $G$ that matches $S$;
return $G$ or signal failure;
```

Algorithm 4: Approximating a descriptive gap pattern.

4.8.3 Finding the constants

Our first approximation for finding the constants of the gap pattern is to find the Longest Common Subsequence of the strings. This way of computing the constants of a gap pattern is not completely reliable, because the longest common subsequence of a set of strings is not always related to a descriptive gap pattern matching the set. For example, if $S=\{$ccaba,bcca$\}$, the longest common subsequence of S is cca, but the only gap pattern that matches S is --- b --- a. So an algorithm that tries to convert the LCS of the strings into a gap pattern matching the strings cannot always succeed. However, we will show that the algorithm will eventually converge to a correct descriptive gap pattern as more examples are added to the set. For example, if the string aba is added to S, then the LCS becomes ba and a gap pattern based on this string can be found.

The LCS of a set of strings is also hard to compute, so a further layer of heuristic approximation is required. Algorithm 5 gives our heuristic for the LCS. The algorithm considers the samples from shortest to longest and takes its initial hypothesis for a common subsequence to be the shortest sample. It then refines its hypothesis by computing the exact longest common subsequence of its current guess with each of the strings s_2, s_3, ... s_n in turn.

```
Sort the inputs by increasing length, yielding $I_1$, $I_2$, ..., $I_n$.
$c \leftarrow I_1$;
for $j \leftarrow 2$ to $n$ do
    $c \leftarrow LCS(c, I_j)$;
return $c$;
```

Algorithm 5: Approximating the LCS of a set of strings

The longest common subsequence operation is not associative, so this algorithm will not necessarily find the exact longest common subsequence of a set of strings. But it will do pretty well in practice, and indeed will converge to $c(G)$ in the limit.

Theorem 19: Algorithm 5 computes $c(G)$ in the limit.

Proof: One property of the *LCS* is that if c is a subsequence of a string I, then $LCS(c,I)$ is equal to c. If G is the gap pattern generating the set of samples, then in the limit all strings in $L(G)$ will be processed by the algorithm. The unique shortest string in $L(G)$ is $c(G)$, and the strings are considered in order of increasing length, so eventually I_1 will be equal to $c(G)$. Once this happens, c will be initially set to I_1, which is $c(G)$, which is a subsequence of all strings in $L(G)$, and the successive refinements of setting c to $LCS(c,I_j)$ will leave c equal to $c(G)$. □

If the input strings are all of length l, then this algorithm can be implemented to run in time $O(nl^2)$ using Hirschberg's algorithm for computing the LCS of two strings [41]. Alternatively, if r is a bound on the number of ordered pairs of positions in the input strings that contain equal characters, then this could be implemented to run in time $O(n(l+r)\ log\ l)$ using Hunt and Szymanski's algorithm [43].

4.8.4 Inserting the gaps

The next step of the algorithm is to take the string of constants returned by the first step and to find a way to insert the minimal number of gaps into that constant string to get a gap pattern that matches the samples.

We have attempted, without success, to classify the complexity of the problem of transforming a longest common subsequence of a set of strings into a gap pattern that can match the set. We suspect that the problem is NP-hard, but we have not been able to prove it.

On the other hand, we do have an algorithm that solves the problem in time that is polynomial in the total length of the strings when the number of strings in the sample set is bounded. The algorithm uses the descriptive gap pattern synthesis algorithm of Theorem 18 as a subroutine. It adds the common subsequence c to the sample set, and then performs the descriptive gap pattern synthesis algorithm on this new set. If pattern synthesis yields a descriptive gap pattern G with $c(G)=c$, then it mimics G's way of inserting the gaps into c. If the algorithm returns a descriptive gap pattern that has fewer constants than c, or if it cannot find a gap pattern to match S at all, then there is no way to insert gaps into c to transform it into a gap pattern matching S, for if there were then the gap pattern based on c would have been the descriptive gap pattern matching S.

If l is a bound on the length of each string in S, the descriptive gap pattern synthesis algorithm runs in time $O(l^{3n+1} \log l)$, so this algorithm for gap insertion runs in time $O(l^{3n+4} \log l)$, since it increases the size of S by one string.

Theorem 20: The Gap Insertion problem can be solved in $O(l^{3n+4} \log l)$ time, which is polynomial for a bounded number of samples.

In practice, the problem of finding a way of inserting gaps into a constant string to get a gap pattern that matches the samples can be adequately solved using simple heuristics. The heuristic shown in Algorithm 6 treats the string returned by the constant-finding step as a gap pattern with zero gaps and tries that pattern against each of the strings in the sample set. If the pattern does not match some string, then the algorithm tries to make some local modifications to the gap pattern to make it match the string. If the modifications are successful, then it continues processing the rest of the strings; if the modifications fail, then it gives up.

```
Input: $I_1, I_2, \ldots I_n$, a set of strings;
        $c$, a subsequence common to the $I_j$;
Output: A gap pattern $G$ with $c(G) = c$ that matches the $I_j$, or
        an indication that no pattern could be found;

The initial gap pattern is a common subsequence without gaps, and
   the inputs are considered sorted from shortest to longest.
$G \leftarrow c$;
loop for $j \leftarrow 1$ to $n$ do
   if $G$ does not match $I_j$ then
      make local modifications to $G$ to make it match $I_j$;
      if $G$ cannot be modified to match $I_j$ then fail;
endloop;
delete extraneous gaps from $G$;
return $G$ or a signal of failure;
```

Algorithm 6: Heuristic gap insertion algorithm

The heuristics used to modify G to make it match a sample are the core of the algorithm, and have been broken out as Algorithm 7. We have two heuristics: one that inserts gaps, and one that deletes or moves them. The gap pattern starts out without any gaps, so we prefer to insert gaps as long as that leads to a gap pattern that matches the string being considered. If the gap insertion heuristic cannot insert gaps into G to make it match a particular I_j, then it will delete gaps from G and try inserting gaps again. If the second gap insertion fails, then the algorithm will give up and return an indication that G could not be modified to make it match the samples.

```
insert gaps in $G$ to make it match $I_j$;
if gap insertion fails to make $G$ match $I_j$ then
    move and delete the gaps in $G$, such that $G$
        still matches $I_1, I_2, \ldots, I_{j-1}$;
    insert gaps in $G$ to make it match $I_j$;
    if gap insertion fails to make $G$ match $I_j$ then
        signal failure;
```

Algorithm 7: Performing local modifications to a gap pattern.

The gap deletion heuristic is also run at the end of the gap insertion process in an effort to reduce the number of gaps in the gap pattern.

Our heuristic approach to inserting gaps into the gap pattern is called *leftmost-match gap insertion* and is implemented by Algorithm 8. The algorithm takes as input a sample string I and a gap pattern G encoded as a common subsequence c together with a boolean vector g that indicates where the gaps in c lie. Its output is a new gap pattern G' that subsumes G; G' has the same constants and gaps as G, except that it may have some additional gaps as well. We call this algorithm leftmost-match gap insertion because it tends to match the constants of the subsequence as far to the left as it can, which is to say that it tends to insert the gaps as far to the right in the constant subsequence as it can.

The leftmost-match gap insertion algorithm works by scanning G and I in parallel from left to right, considering suffixes of the gap pattern and input string as it goes. It terminates either when the gap pattern suffix under consideration matches the current input string suffix, or if it runs out of input string to match before running out of gap pattern. The former kind of termination yields a gap pattern that matches the string, while the latter is a signal that this particular heuristic has failed and that another must be tried. Two kinds of action are taken to make one of these eventualities come to pass.

```
Inputs: I, a sample input string of length l;
        G, a gap pattern encoded as:
            c, a common subsequence of length n;
            g, a boolean vector of length n in which g[j] is
              true iff there is a gap before c[j] in G.

j ← 1;
k ← 1;
loop while k ≤ |I|;
    until G[j..n] matches I[k..|I|];
    if c[j] = I[k] then
        j ← j + 1;  k ← k + 1;
    else
        g[j] ← true; k ← k + 1;
endloop;
if k>|I| then the gap insertion process failed;
```

Algorithm 8: Leftmost-match gap insertion

In the first case, if the leading constant character of the pattern suffix is equal to the first character of the string suffix, then the algorithm assumes that these two characters would match in the target gap pattern, and so it goes on to consider the trailing suffixes of both the pattern and the string. In the second case, if the two leading characters are different, then the algorithm modifies the gap pattern to contain a gap before the leading character of the pattern suffix, and immediately uses the new gap to skip over the leading character of the string suffix.

As an example of this algorithm's behavior, if the input string I is the string:

```
The Mets beat Chicago, 6 to 5.
```

And G is the constant gap pattern "`The␣es␣beat␣io,␣␣to␣.`", then this algorithm will insert six gaps in order to make the pattern match the string:

The␣ *-1-* e *-2-* s␣beat␣ *-3-* i *-4-* o,␣ *-5-* ␣to␣ *-6-* .

When a second input string is considered:

```
The Yankees beat Baltimore, 10 to 3.
```

Then the gap pattern must have one more gap inserted between the "o" and the "," to match the string:

`The␣ -1- e -2- s␣beat␣ -3- i -4- o -5- ,␣ -6- ␣to␣ -7- .`

We say that the gaps of a gap pattern G are a *superset* of the gaps of a gap pattern H if $c(G)=c(H)$ and at every point in $c(H)$ that H has a gap, G also has a gap. The following lemma shows that, with the proper input, the leftmost-match gap insertion heuristic will find a gap pattern whose gaps are a superset of the target pattern's gaps.

Lemma 21: If H is a gap pattern, and leftmost-match gap insertion is applied to the string $c(H)$ with a sample set equal to $eo(H)$, then the algorithm will successfully insert gaps into the string to transform it to a gap pattern G that matches the samples, and the gaps of G will be a superset of the gaps of H.

Proof: Any gap pattern G with $c(G)=c(H)$ will match the string $c(H)$, so the only strings in $eo(H)$ that might require gap insertion are those of length $|c(H)|+1$. All of the strings of length $|c(H)|+1$ have one gap expanded and are either of the form $s_0s_1...s_{i-1}\mathtt{x}s_i...s_n$, where $\mathtt{x}$ is some symbol that is not equal to the first character in s_i, or are of the form $s_0s_1...s_{i-1}\mathtt{y}s_i...s_n$, where $\mathtt{y}$ is the same as the first symbol in s_i. Let $eo_{i\mathtt{x}}(H)$ be that subset of $eo(H)$ that has the i'th gap expanded with a symbol that is not equal to the first symbol of s_i, and let $eo_{i\mathtt{y}}(H)$ be that subset of $eo(H)$ that has the i'th gap expanded with the same symbol as the first symbol of s_i. Note that $eo_{i\mathtt{y}}(H)$ is either empty or contains one string.

If the i'th gap has already been inserted into G, then one can verify that Algorithm 8 will not be invoked on any member of $eo_{i\mathtt{x}}(H)$ or $eo_{i\mathtt{y}}(H)$, because G will already match that string. If the i'th gap has not yet been inserted into G and the current input sample is a member of $eo_{i\mathtt{x}}(H)$ then Algorithm 8 will be invoked. The algorithm will find that the first character of s_i does not equal $\mathtt{x}$, and it will then insert a gap into G preceding s_i, which will make G match the input and all other members of $eo_{i\mathtt{x}}(H)$ or $eo_{i\mathtt{y}}(H)$. If the current input sample is a member of $eo_{i\mathtt{y}}(H)$, then the $\mathtt{y}$ will be equal to the first symbol in s_i, but will differ from some other symbol in s_i; if $\mathtt{y}$ were equal to all members of s_i, then it would not be a valid substitute for the gap i. In these circumstances, Algorithm 8 will insert a gap before the first member of s_i that is not equal to $\mathtt{y}$; this gap will be useful only in matching that single element of $eo(H)$.

Thus, on input $eo(H)$, the gap insertion algorithm will transform $c(H)$ into a gap pattern G whose gaps are a superset of the gaps of H. □

Note that this lemma also shows that the gap insertion heuristic will insert no more than $2|g(H)|$ gaps into $c(H)$ on the input data $eo(H)$.

The extraneous gaps inserted by the leftmost-match gap insertion heuristic may cause problems. For example, if the algorithm is given as input a gap pattern $G = $ abc and a first sample string $I_1 = $ aabc, then it will modify G to be "a --- bc". If the second input I_2 is babc, then G will be modified again to be "--- a --- bc". If the third input I_3 is abbcabc, then there is no way for gaps to be *inserted* into G to allow it to match I_3; the only gap pattern with constants abc that matches abbcabc is "--- abc", and a gap must be deleted from G to transform it thus.

Some of these extraneous gaps can be removed by the gap deletion heuristic given in Algorithm 9. This algorithm scans the gaps in G starting from the left. It removes a gap from G, yielding a new gap pattern G', and then tests if G' still matches all of the samples that it used to. If G' matches the samples, then the change is made permanent in G, otherwise the modifications are undone.

```
Inputs: S, a set of strings;
        G, a gap pattern that matches S;
Output: G, perhaps with some gaps deleted;

Scan the gaps -j- of G from the left:
   $G' \leftarrow G$ with $s_{j-1}$ -j- $s_j$ changed to $s_{j-1}s_j$;
   if $G'$ matches S then
      $G \leftarrow G'$;
return G;
```

Algorithm 9: Gap deletion heuristic

This heuristic is invoked whenever the leftmost-match gap insertion algorithm fails to be able to modify G to match a particular string. If the heuristic is able to remove some gaps from G, then the leftmost-match gap insertion heuristic is attempted again. For example, the first few steps of the scenario given above proceed in the same way: if the gap insertion algorithm is given as input a gap pattern $G = $ abc, and a first sample string $I_1 = $ aabc, then leftmost-match gap insertion will successfully modify G to be "a --- bc". If the second input I_2 is babc, then a gap will be inserted again to make G be "--- a --- bc". However, if the third input I_3 is abbcabc, then there is no way for gaps to be *inserted* into G to allow it to match I_3. In this case, though, the second gap can be deleted to yield the pattern "--- abc" which matches the previous two inputs, and also happens to match I_3.

Lemma 22: If the set of strings S is equal to $eo(H)$, and G is a gap pattern that matches S whose gaps are a superset of those of H, then after running Algorithm 9 on G and S, G will be equal to H.

Proof: Each of the gaps in G that are not in H can be removed, one at a time, and G will still match all strings in $eo(H)$. On the other hand, suppose that the i'th gap that G shares with H is removed. In this case, if x is a character different from the first character of s_i, (assuming $|\Sigma| \geq 2$), then G will not be able to match $s_0 s_1 \ldots s_{i-1} \mathtt{x} s_i \ldots s_n$, which is a member of $eo(H)$. □

The running time of the gap deletion heuristic on a gap pattern G and a string of length l is $O(l|\mathcal{g}(G)|)$, which is dominated by $O(l^2)$. Leftmost-match gap insertion also runs in $O(l^2)$. On n inputs, the gap insertion heuristic might invoke leftmost-match gap insertion twice per input, and the deletion heuristic once per input, so the running time is bounded by $O(nl^2)$. In practice, the actual running time is considerably better.

The gap insertion algorithm is not guaranteed to find a way to insert gaps into a constant sequence to make it match a set of strings. Even if it can find some sort of gap insertion, it is not guaranteed to be a good one. However, it seems to perform pretty well in practice, and we show in the next section that in the limit it correctly inserts the gaps into the constants returned by the constant-finding heuristics.

4.8.5 A heuristic descriptive gap pattern synthesis algorithm

Algorithm 6 can be combined with Algorithm 5 to yield an algorithm that is a worthwhile heuristic for approximating a descriptive gap pattern.

```
Input: A set of strings $S = \{I_1, I_2, \ldots, I_n\}$
    sorted in order of increasing length.
Output: G, a pseudo-descriptive gap pattern for S.

$G \leftarrow$ result of Algorithm 5 on S;
$G \leftarrow$ result of Algorithm 6 on G and S;
return G;
```

Algorithm 10: Heuristic for approximating a descriptive gap pattern

One can show that this algorithm identifies a descriptive gap pattern in the limit:

Theorem 23: Algorithm 10 identifies a descriptive gap pattern in the limit.

Proof: We have already shown in Theorem 19 that Algorithm 5 will identify the constants of the descriptive gap pattern in the limit. So assuming that the algorithm has correctly identified $c(G)$, we need only show that the repeated applications of Algorithm 8 from within Algorithm 6 will place the minimal number of gaps into their required positions. The input strings I_1, I_2, ..., I_n are presented in order of increasing length, so after some point the initial part of the sequence is $eo(G)$. Lemma 21 shows that on this input Algorithm 8 will insert gaps into the right locations in the constants, although it may insert extra gaps as well. Lemma 22 shows that Algorithm 9 will remove the extraneous gaps as soon as it is invoked, and it will be invoked at least once every time Algorithm 6 is called. □

4.9 Synthesizing gap replacements from examples

Once a descriptive gap pattern is found that describes 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 input samples:

```
The Yankees beat Baltimore, 10 to 3.
The Mets beat Chicago, 6 to 5.
The Angels beat Detroit, 2 to 0.
```

Algorithm 10 would find the following descriptive gap pattern to describe the structure of the input set:

The␣ *-1-* e *-2-* s␣beat␣ *-3-* o *-4-* ,␣ *-5-* ␣to␣ *-6-* .

The first gap in the pattern matches the string "Yank" in the first sample, "M" in the second, and "Ang" in the third. The second matches "e", "t", and "l", and so on:

-1-	*-2-*	*-3-*	*-4-*	*-5-*	*-6-*
Yank	e	Baltim	re	10	3
M	t	Chicag		6	5
Ang	l	Detr	it	2	0

The problem addressed in this section is that of taking the fragments of text matched by gaps from the input, and a collection of output samples:

```
Yankees 10, Baltimore 3.
Mets 6, Chicago 5.
Angels 2, Detroit 0.
```

and finding some way of explaining how the outputs could be produced by copying gaps from the input strings and inserting new constant strings as required. That is, how do we synthesize a replacement expression that can construct the set of output strings from the sample inputs? What sort of processing is required to be able to generate this replacement expression?

-1- e *-2-* s␣ *-5-* ,␣ *-3-* o *-4-* ␣ *-6-* .

That this problem is at all challenging is not evident from this example. If we examine the output strings and the gaps matched by the input, we can easily see how to construct a replacement expression. It takes only a glance to see which parts of the output are produced from gaps *-1-*, *-3-*, *-4-*, *-5-* and *-6-*. The first gap, for example, matches "Yank", "M", and "Ang" in the input, and it is the only available source for these strings in the output. The second gap presents only a slightly more difficult problem. It matches an "e" in the first input string, and one might have to pause for moment to decide which, if any, of the three "e"'s in the first output result from copies of that gap. But only for a moment, for the second gap simultaneously matches a "t" in the second input sample, and there is only one possible destination for that "t" in the second output. So the problem does not appear to be very difficult.

In the following, we show that the problem of creating a replacement expression is theoretically intractable. We then present an algorithm that solves the problem in polynomial time for a bounded number of sample strings, and we point out that the running time of this algorithm is reasonable on the sort of data encountered in practice.

4.9.1 Complexity of gap replacement synthesis

The essence of the problem of constructing a gap replacement is to solve the problem of finding a way of building several large pieces of text out of a given set of smaller ones. The Gap Replacement Synthesis decision problem formalizes this question:

PROBLEM: *Gap Replacement Synthesis*

INSTANCE: An alphabet Σ, a set of vectors of strings S called the input parse, and a single vector of strings X called the output. S is the set $\{s_1, s_2, \ldots s_n\}$, where each of the s_i is a vector of strings $<s_{i1}, s_{i2}, \ldots, s_{im}>$, and each s_{ij} is a string from Σ^*. X is a single vector of strings $<x_1, x_2, \ldots x_n>$ where each x_i is a string from Σ^*.

QUESTION: Can X be expressed as a componentwise concatenation of the vectors of strings in S? That is, does there exist a sequence of indices $j_1,j_2,...j_l$, $l\geq 0$, such that each x_i in X is equal to the concatenation of the strings $s_{j_1i},s_{j_2i},...s_{j_li}$?

While this statement of the problem does not mention the possibility of inserting new constant characters, that added operation can be viewed as simply having an input parse that includes all strings of the form $<$a,a,...,a$>$ for each a$\in\Sigma$.

Theorem 24: Gap Replacement Synthesis is NP-complete.

Proof: This problem can be solved in nondeterministic polynomial time by an algorithm that guesses the sequence of indices $j_1,j_2,...j_l$ and then confirms that the concatenation of $s_{j_1i},s_{j_2i},...s_{j_li}$ is equal to x_i for each of the x_i in X.

We will show that this problem is NP-complete by reduction from the Longest Common Subsequence problem (LCS). Given an instance of LCS $= <\Gamma,Q,k>$, we construct an instance of Gap Replacement Synthesis$=<\Sigma,S,X>$ as follows: The alphabet Σ will be $\Gamma\cup\{\$\}$, where \$ is some character not in Γ. If Γ contains t symbols $\{\gamma_1,\gamma_2,...,\gamma_t\}$, then the set of vectors S will consist of t vectors of length $m+1$:

$$<\gamma_1,\gamma_1,...,\gamma_1,\$> <\gamma_2,\gamma_2,...,\gamma_2,\$> \cdots \qquad <\gamma_t,\gamma_t,...,\gamma_t,\$>$$

along with tm other $m+1$-vectors:

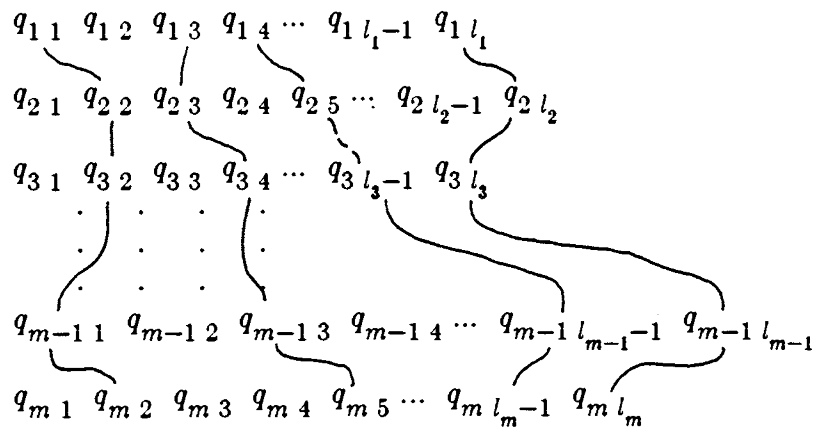
$$\begin{array}{cccc} <\gamma_1,\Lambda,...,\Lambda,\Lambda> & <\gamma_2,\Lambda,...,\Lambda,\Lambda> & \cdots & <\gamma_t,\Lambda,...,\Lambda,\Lambda> \\ <\Lambda,\gamma_1,...,\Lambda,\Lambda> & <\Lambda,\gamma_2,...,\Lambda,\Lambda> & \cdots & <\Lambda,\gamma_t,...,\Lambda,\Lambda> \\ \cdot\ \cdot\ \cdot & \cdot\ \cdot\ \cdot & & \cdot\ \cdot\ \cdot \\ <\Lambda,\Lambda,...,\gamma_1,\Lambda> & <\Lambda,\Lambda,...,\gamma_2,\Lambda> & \cdots & <\Lambda,\Lambda,...,\gamma_t,\Lambda> \end{array}$$

And the output vector X will be the $m+1$-vector:

$$<q_1,q_2,...,q_k,\$\$\$...\$\$\$>$$

The first m components of X are simply copies of the strings in the LCS sample set, and the last component is a string composed of k consecutive \$ symbols. Now we will show that there is a common subsequence of size k in our instance of LCS if and only if there is a replacement expression in the instance of Gap Replacement Synthesis that we have constructed.

Suppose that there is a common subsequence of size k in Q. This means that there is a scheme for threading the strings of Q so that there are k strands of non-overlapping thread passing through identical symbols in each of the q_i:


$q_{1\,1}$ $q_{1\,2}$ $q_{1\,3}$ $q_{1\,4}$ ⋯ $q_{1\,l_1-1}$ $q_{1\,l_1}$
$q_{2\,1}$ $q_{2\,2}$ $q_{2\,3}$ $q_{2\,4}$ $q_{2\,5}$ ⋯ $q_{2\,l_2-1}$ $q_{2\,l_2}$
$q_{3\,1}$ $q_{3\,2}$ $q_{3\,3}$ $q_{3\,4}$ ⋯ $q_{3\,l_3-1}$ $q_{3\,l_3}$
$q_{m-1\,1}$ $q_{m-1\,2}$ $q_{m-1\,3}$ $q_{m-1\,4}$ ⋯ $q_{m-1\,l_{m-1}-1}$ $q_{m-1\,l_{m-1}}$
$q_{m\,1}$ $q_{m\,2}$ $q_{m\,3}$ $q_{m\,4}$ $q_{m\,5}$ ⋯ $q_{m\,l_m-1}$ $q_{m\,l_m}$

The components x_1 through x_m of X are exactly the strings in Q, so this diagram reveals a simple way of constructing each of the x_i in X from the parts in S. The pieces of these strings that do not lie on threads can be built character by character out of the members of S that have one non-empty slot containing a single character. The particular order that this is done in does not matter — the unthreaded portions of x_2 can be added before those in x_1, or vice-versa. Once all of the characters to the left of a thread have been assembled into X, that entire thread can be laid in at once by adding in the member of S that contains the character in the thread in positions 1 through m. This member of S also contains a \$ in component $m+1$. It is correct to add it in because component $m+1$ of X contains a string consisting of k \$'s, each of \$'s is added exactly when a thread is encountered, and there are exactly k threads.

Suppose that there is a sequence of indices $j_1, j_2, \ldots j_l$, $l \geq 0$, such that each x_i in X is equal to the concatenation of the strings $s_{j_1\,i} s_{j_2\,i} \ldots s_{j_l\,i}$. In particular x_{m+1}, which consists of k \$ symbols, must be constructed from members of S corresponding to k of these indices. All of the members of S that contain the \$ symbol in position $m+1$ are of the form $<\gamma, \gamma, \ldots \gamma, \$>$ for some $\gamma \in \Gamma$. The k symbols of Γ occurring in each of these s_i form a subsequence common to the members of Q of size k.

Thus, there is a common subsequence of size k in Q if and only if there is a replacement expression X drawn from the parts in S. This construction can be done in polynomial time. □

4.9.2 A gap replacement synthesis procedure

In practice, finding a gap replacement given a gap program is not quite so hard as this NP-completeness result would imply. Indeed, in practice we can solve this problem exactly because our data does not seem to exercise the features that make it intractable.

The gap pattern found by the algorithms in Section 4.8 yields a particular parse of the input strings into constants and gaps. For example, suppose that we have the following two input/output pairs as examples:

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

The descriptive gap pattern matching the inputs is *-1-* x *-2-* y. In the first example, the first gap matches the string ab and the second matches ba. These two gaps and some auxiliary constants can be arranged to form the output ababa using any one of eight replacement expressions:

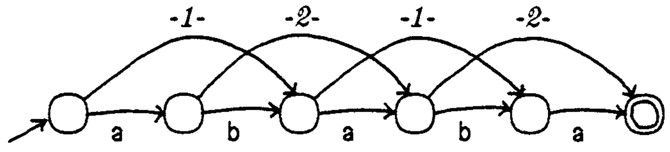
```
ababa          a -2- -2-
aba -2-        -1- aba
ab -1- a       -1- a -2-
a -2- b a      -1- -1- a
```

Similarly, in the second example, the first gap matches the string cdd and the second gap matches ddc, and there are five ways of writing the output addcddc:

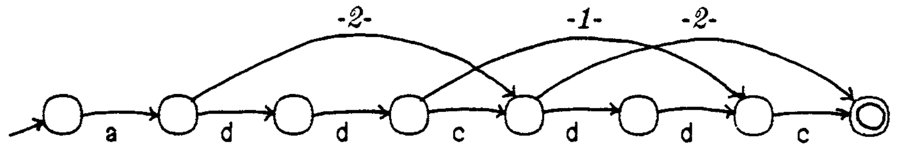
```
addcddc        a -2- ddc
addc -2-       a -2- -2-
add -1- c
```

The task of the algorithm described in this section is to find a single replacement expression that will simultaneously transform both of the input examples parsed by a gap pattern to the corresponding output examples.

The algorithm we use 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 yield the output example. For input/output pair $<i_j,o_j>$ the corresponding finite state machine M_j has $|o_j|+1$ states numbered 1 through $|o_j|+1$. State 1 is the start state, and state $|o_j|+1$ is the sole accepting state. There are two classes of transitions in M_j: those arising from the constants in any matching replacement expression and those arising from the gaps. There is a transition from state k to state $k+1$ on the symbol $o_j[k]$; and there is a transition from state k to state $k+l$ on gap symbol g_m if the substring $o_j[k..k+l-1]$ is equal to the text matched by gap g_m in i_j. The following is a picture of the finite state machine that captures all of the ways that the output ababa can be written in terms of constants and gaps when gap -*1*- matches ab and gap -*2*- matches ba:

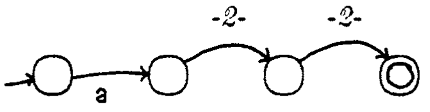


And the following is the finite state machine for the second sample in which the output string is addcddc, the first gap matches cdd, and the second gap matches ddc:



Notice that each of the machines recognize the languages given above.

The next step of the algorithm is to find some replacement expression that produces all of the output strings by intersecting the machines derived in the first step. The classical finite state machine intersection algorithms can be used [42], and in this case the intersection would be the following machine:



In this case the replacement expression is unique. The only string recognized by this automaton corresponds to the replacement expression a -*2*- -*2*-, and so the resulting program is:

-*1*- x -*2*- y $\Rightarrow$ a -*2*- -*2*-

The replacement synthesis algorithm is given in Algorithm 11.

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 $O(|\mathcal{g}(G)|l)$ transitions. The worst case running time of the second phase can be proportional to the product of

```
Inputs: A set of strings $O = <o_1, o_2, ..., o_n>$ and
        a parse of the inputs $S$
Output: A replacement expression $R$ that produces $O$ from
        the components in $S$ and any necessary constants, or an
        indication that no $R$ could be found;

if all $o_j$ in $O$ are the empty string then
   return the empty replacement expression;

loop for each $o_i$ in $O$ do
   $M_i \leftarrow$ finite automaton representing all replacement
        expressions yielding $o_j$ from $S$;
endloop;

$M \leftarrow M_1$
loop step $i$ from 2 to $n$ do
   $M \leftarrow M \cap M_i$;
   if $M$ is the empty automaton then
      return failure;
endloop;

return a replacement expression consisting of the symbols
       encountered along one accepting path through $M$;
```

Algorithm 11: Replacement synthesis algorithm

the sizes of the machines constructed in the first phase, and so we have shown:

Theorem 25: Given a set of n input/output samples $S = \{<i_1, o_1>, <i_2, o_2>, ..., <i_n, o_n>\}$, and a gap pattern G that matches the input samples, Algorithm 11 will find a replacement expression R if one exists in time $O(|g(G)|^n l^n)$.

The NP-completeness result in Theorem 24 implies that we should not expect to improve on this worst case performance by very much, but the algorithm seems to perform well in practice. In practice, the machines constructed in the first phase are long and thin — 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 is exactly the replacement expression computation algorithm that is implemented in the running system.

The algorithm can be shown to converge to the target replacement expression once the pattern synthesis algorithm has found the target:

Theorem 26: Assuming that the gap pattern G used to parse the inputs is the correct one, Algorithm 11 will identify the replacement expression component of a gap program in the limit from positive data.

Proof: The intersection machine M encodes all possible replacement expressions that can construct a set of output strings from a parsed set of inputs, so if G is the target gap pattern then the target replacement expression R must be one of the replacement expressions represented in M. By definition, all of the possible sample input/output pairs $<i_j, o_j>$ can be transformed by the program $P = G \Rightarrow R$; we will show that every other replacement expression R' results in a gap program $P' = G \Rightarrow R'$ that will fail to transform some input/output pair in the same way that P does. Adding these input/output pairs to the sample set will cause R' to be eliminated from M; eventually, only R will be represented in M.

The target expression R is of the form $x_1 x_2 ... x_k$ where each of the x_i is either a gap symbol from G or a symbol from the alphabet. Let R' be some other replacement expression that can transform the input/output pairs presented so far. R' differs from R in some leftmost position i; that is, R' is of the form $x_1 x_2 ... x_{i-1} x_i' ... x_l'$ with $x_i \neq x_i'$. If x_i and x_i' are both symbols from the alphabet, then $P'(c(G))$ will not equal $P(c(G))$, because both of these output strings are made up solely of the constants of the replacement expressions. If x_i and x_i' are different gap symbols, then an input that binds the two gaps to strings starting with different symbols will result in a corresponding difference in the outputs. Similarly, if x_i is a constant and x_i' is a gap, or vice-versa, then an input that binds the leading character of the gap to something other than the constant will cause a corresponding difference in the outputs. If R is a strict prefix of R', or vice-versa, then an input that binds the gaps that occur in the suffix of R' will produce a different, longer output. □

4.10 Summary of results

This chapter presented a study of the problem of synthesizing gap programs from examples of their input/output behavior. In an effort to understand how to formulate an efficient solution to this problem, we investigated the complexity of various subproblems related to gap program synthesis. We considered the complexity of both the general problem, where the number and length of the input/output pairs is not bounded, and a more restricted problem in which the number of input/output pairs is bounded, although the length is not. The following table summarizes our results; the entries are either NPC, if the problem is NP-complete, or an upper bound for solving the particular problem for n strings each no longer than l characters.

Problem	*General*		*Fixed n*	
Gap program synthesis	NPC	(Thm. 11)	NPC	(Thm. 11)
Gap pattern synthesis	NPC	(Thm. 12)	$O(l^{3n})$	(Thm. 17)
Descriptive gap pattern synthesis	NPC	(Thm. 16)	$O(l^{3n+1} \log l)$	(Thm. 18)
Longest Common Subsequence	NPC	(Thm. 9)	$O(l^n)$	
Gap insertion	?		$O(l^{3n+4} \log l)$	(Thm. 20)
Replacement synthesis	NPC	(Thm. 24)	$O(l^n \lvert g(G) \rvert^n)$	(Thm. 25)

An algorithm that is guaranteed to find a gap program exhibiting some sample behavior must solve an NP-complete problem, and this problem remains NP-complete even when the number of input/output pairs is bounded. This complexity result led us to decompose the problem into more tractable pieces and to find the gap pattern and the replacement expression independently of each other. The independence of the two computations implies that a gap pattern is no longer constrained to parse the input samples in such a way that a replacement expression exists that maps the parsed inputs to the output samples.

In order to get a gap pattern that parses the inputs in a potentially useful way, the "best" gap pattern matching a set of input samples is defined to be a descriptive gap pattern matching the set, that is, a gap pattern matching the set with the largest number of constants and the fewest number of gaps. We showed that descriptive gap pattern synthesis is NP-hard, and although algorithms for finding a descriptive gap pattern for a fixed number of samples run in polynomial time, they do not seem to be practical, and so we developed heuristics for approximating descriptive gap patterns.

The heuristic approximation is computed by Algorithm 10, which first computes a plausible candidate for the constants of the descriptive gap pattern, and then inserts gaps into that string to make it match all of the samples. The constants are approximated by a heuristic for the longest common subsequence of the set of input strings, Algorithm 5, which runs in time $O(nl^2)$. Theorem 19 shows that this heuristic approximation converges in the limit to the constants of the descriptive gap pattern for the sample set. The algorithm then uses the gap insertion heuristic implemented by Algorithm 6 to insert gaps into this constant sequence in $O(nl^2)$ time to make it into a gap pattern matching the input samples. Theorem 23 shows that this entire process converges in the limit to the descriptive gap pattern matching the input samples.

The descriptive gap pattern yields a parse of the input strings, and the task of the replacement expression is to rearrange that parse and introduce new constants to form the output strings. Finding a replacement expression given a parse is NP-hard, although the problem can be solved by Algorithm 11 in $O(l^n|g(G)|^n)$ time for a bounded number of output samples. However, in contrast to gap pattern synthesis, it does not appear to be hard in practice to find an exact replacement expression. In practice, the $O(l^n|g(G)|^n)$ algorithm seems to run in time closer to $O(nl)$, and so we actually use this algorithm to find a replacement expression in the running system.

```
Input:  a set of input/output pairs $S=\{<i_1,o_1>,<i_2,o_2>,...,<i_n,o_n>\}$;
Output: a gap program $P$ or an indication of failure.

Approximate a descriptive gap pattern $G$ common to $\{i_1,i_2,...,i_n\}$
   using Algorithm 10;
Use $G$ to parse $\{i_1,i_2,...,i_n\}$;
Compute a replacement expression $R$ that maps the parsed $i_k$ to $o_k$
   using Algorithm 11;

Return $P = G \Rightarrow R$;
```

Algorithm 12: A gap program synthesis algorithm

The descriptive gap pattern heuristics and the replacement expression algorithm make up a gap program synthesis procedure that can identify gap programs in the limit from positive data.

Theorem 27: Algorithm 12 identifies gap programs in the limit from positive data.

Proof: Theorem 23 proves that the approximate descriptive gap pattern synthesis algorithm identifies gap patterns in the limit from positive data. Theorem 26 shows that once the target gap pattern has been identified, the identification of the replacement expression will surely follow. Thus Algorithm 12 converges to the target program. □

We have presented an algorithm that will identify gap programs in the limit from positive data. In the next chapter, we analyze its short-term performance, present heuristics that help it to function effectively on text, and describe its implementation and user interface within a text editor.

Chapter 5

IMPLEMENTATION

In Chapter 4 we developed a gap program synthesis algorithm and showed that it could identify gap programs in the limit from positive data. While identification in the limit is certainly a property to be desired, one would like the further assurance that the algorithm will perform within the limit of the user's patience. In an attempt to gain an understanding of how well the algorithm performs on a small amount of data, say two or three examples, this chapter starts by presenting a study of the algorithm's performance on several different sets of randomly generated test data. This study brings out some of the aspects of both the input data and the target program that affect the gap program synthesis process.

The chapter also presents four heuristics that help to make the gap program synthesis algorithm into a useful tool for editing by example. Three of these heuristics, ***pattern reduction***, ***tokenization***, and ***gap bounding***, are used to transform the synthesized descriptive gap pattern to a more natural gap pattern. The fourth heuristic reduces the number of output examples that are normally required.

The complete gap program synthesis algorithm has been integrated into the editing by example subsystem of the *U* editor [66]. The last portion of this chapter discusses some of the issues pertaining to user interfaces to EBE systems, and presents the details of the user interface to *U*'s EBE subsystem.

5.1 Good data; the simple answer

An EBE system user wants to apply a transformation to a piece of text that can, hopefully, be performed by a gap program $P = G \Rightarrow R$. The identification of G is the part of the EBE process that is most susceptible to variations in the quality of sample data, since Algorithm 11 computes the gap replacement R exactly, so we will begin our study of the performance of the gap program synthesis algorithm by concentrating exclusively on evaluating the sort of data that Algorithm 10 needs to identify a gap pattern.

The simple answer for what constitutes good data to identify a gap pattern $G = s_0 g_1 s_1 g_2 s_2 \ldots g_n s_n$ is a set of input strings that unambiguously identifies G. For example, if # is a symbol that is not contained in any of the constant strings of G, then these two strings will pin G down:

$$s_0 s_1 s_2 \ldots s_n$$
$$s_0 \# s_1 \# s_2 \# \ldots \# s_n$$

Or if # and % are two symbols that are not in $c(G)$, then these strings might be convenient:

$$s_0 \# s_1 \# s_2 \# \ldots \# s_n$$
$$s_0 \% s_1 \% s_2 \% \ldots \% s_n$$

It is not necessary that the gap-filling strings be the same character, or that they be a single character; if the gaps are filled with strings that are easy to distinguish from each other and from the intended constants of the gap pattern, then it is easy identify the gap pattern.

However, the EBE system cannot count on the user to provide data that is this unambiguous; if it did, the user of our system would undoubtedly have been better off if we instead had concentrated on enhancing the facilities for writing gap programs manually. The question that concerns us is characterizing how well the system works on ambiguous data. We have already shown that the system will identify gap programs in the limit; we would like to study its performance when the number of samples is small.

5.2 Gap pattern synthesis experiment design

Experiments were run in order to gain a qualitative understanding of the average-case performance of Algorithm 10, the gap pattern synthesis algorithm. The goal of the experiments was to determine the impact that properties of the target gap pattern and the input data have on the algorithm's ability to identify gap patterns from examples. In each experimental trial a random gap pattern G was generated and the algorithm was set to identifying G from randomly generated elements of $L(G)$. The number of elements of $L(G)$ needed to identify G was recorded, as was a quantity, that we will define in Section 5.4, that measures the number of intermediate hypotheses that were similar to G. One should keep in mind that the quantitative results based on random data that are reported here should not be taken to be a basis for predicting the system's performance on text; the purpose of these experiments is to gain a qualitative understanding of the problems encountered in gap pattern synthesis.

There are many properties of the target gap pattern and of the sample data that can affect the algorithm's performance; in this study, we examined five of these properties:

- the distribution of elements of the constant strings of G;
- the lengths of the constant strings of G;
- the number of gaps in G;
- the distribution of the gap substitutes in members of $L(G)$; and
- the lengths of the gap substitutes in members of $L(G)$.

Three sets of experiments were performed to measure the effects of various combinations of these parameters. In the first set, the results of which are presented in Section 5.3, the symbols that filled both the constant strings of G and the gaps of the members of $L(G)$ were chosen from an alphabet of 100 equiprobable symbols. The second set explored the system's performance when the size of the alphabet was changed from 100 to 25, and the third set used a distribution that attempted to approximate that of text. The results of these tests are presented in Sections 5.5 and 5.6.

All three sets of experiments had the same structure. The number of gaps in the target gap pattern G was chosen, and then the length of the constant strings in the gap pattern was decided upon. For a particular number of gaps n and a particular size of constant strings l, G was created by randomly choosing $l(n+1)$ symbols from the alphabet to generate $c(G)$ and then inserting the n gaps at uniform intervals in the constant string.

For example, if the alphabet consisted of the 26 symbols a through z, l was 4, and n was 2, then any of these gap patterns could be one of those generated:

```
wbeb --- etvs --- ijso
asji --- sadj --- skfa
spbm --- qldy --- gpav
```

For each random G, random members of $L(G)$ were produced by choosing a particular number to be the length of the gap substitute strings and then randomly generating legal substitutes of that length for each gap in G. For example, if G was the pattern `wbeb --- etvs --- ijso` and the chosen gap substitute length was 8, then the algorithm might be asked to try to identify G from the following set of strings of length 28:

```
wbebdwohbgrletvsmfxnljvqijso
wbebpfjjqvgmetvscgdhtpjyijso
wbebmpoozvgqetvsftxmqcvbijso
wbebqxzkktcsetvsdqoybvkhijso
wbebrsxpfvhzetvsyvljhqvqijso
wbebiyobntlgetvsvswreyphijso
wbebuwdyzpsuetvszslxhoetijso
wbebwmdysqjxetvskfrmptlbijso
wbebvbsztpfketvsdmkksbvvijso
wbebtctmupiketvsxfomraxzijso
```

Once the random gap pattern G and random members of $L(G)$ have been produced, the experimental trial proceeds by computing an approximation to the descriptive gap pattern for the first two strings in the set using Algorithm 10. If the synthesized pattern is equal to G, then the experiment is terminated. If the pattern is not equal to the target, then the algorithm is re-run with the first three samples, and then the first four, and so on until it manages to synthesize G. In this case, an analysis of the first two strings

```
wbebdwohbgrletvsmfxnljvqijso
wbebpfjjqvgmetvscgdhtpjyijso
```

yields the descriptive gap pattern

```
wbeb --- g --- etvs --- j --- ijso
```

When the synthesis procedure is re-run with the first three samples

```
wbebdwohbgrletvsmfxnljvqijso
wbebpfjjqvgmetvscgdhtpjyijso
wbebmpoozvgqetvsftxmqcvbijso
```

it synthesizes the pattern

```
wbeb --- g --- etvs --- ijso
```

When the fourth example is added, the system successfully synthesizes the target gap pattern. This test run would have supplied one piece of data concerning random gap patterns with 2 gaps and with constants of length 4 generated uniformly from the 26 symbols a through z being identified by samples generated by filling the gaps with strings of length 8 generated uniformly from the 26 symbols a through z.

5.3 Experiment on [1,100]

The results of the experiments in which the gap and constant strings were both drawn from an equiprobable alphabet of size 100 are summarized in Table 1. The table is organized in four sections, one each for results pertaining to gap patterns with 1, 2, 4, and 8 gaps; each section has four rows, one each for data pertaining to the constant lengths 1, 2, 4, and 8; each row has 7 entries, one each for data pertaining to the gap substitute lengths 1, 2, 4, 8, 16, 32, and 64. For example, the entry in the 8 gap section in the row for constant length 8 and the column for gap substitute length 64 gives the average number of strings of length 584 that were needed to identify a random gap pattern with 72 constants and 8 gaps. Four runs were taken to compute each average, and the entry ** appears if *any* of the four runs could not identify G within 15 samples.

The trends in the table may be simply summarized: fewer gaps, longer constants, and shorter gap substitutes make gap patterns easier to identify; more gaps, shorter constants, and longer gap substitutes make them harder. To help provide a more detailed explanation of the algorithm's performance, we introduce the notion of *compatible* gap patterns.

5.4 Compatibility

Suppose that the current hypothesis of the gap pattern synthesis algorithm is a gap pattern G', and for the sake of concreteness, suppose that this hypothesis is based on a sample set consisting of two strings $\{\alpha,\beta\}$. We distinguish two ways in which G' can differ from the target gap pattern G: G' can be *compatible* with G on this sample set by parsing α and β in such a way that the algorithm can still find a replacement expression

N	CL \ GL	1	2	4	8	16	32	64
1	1	2.00	2.25	2.25	2.75	2.75	3.75	6.75
	2	2.00	2.00	2.25	2.50	3.50	4.25	4.75
	4	2.00	2.00	2.00	3.25	2.75	3.50	5.50
	8	2.00	2.00	2.00	2.00	3.00	3.75	5.00
2	1	2.25	2.00	3.25	3.25	4.00	7.50	**
	2	2.00	2.00	2.00	2.75	3.25	5.75	**
	4	2.00	2.25	2.00	2.75	3.75	4.00	6.00
	8	2.00	2.00	2.00	2.50	3.25	4.25	6.50
4	1	2.00	2.50	2.75	**	**	**	**
	2	2.25	2.25	2.50	3.00	5.00	**	**
	4	2.00	2.25	2.25	3.50	3.75	4.75	7.00
	8	2.00	2.25	2.50	3.00	4.25	5.00	6.25
8	1	2.00	2.00	6.25	**	**	**	**
	2	2.00	2.00	3.25	4.00	5.00	**	**
	4	2.00	2.00	3.25	3.50	3.75	7.00	**
	8	2.00	2.25	3.00	4.00	4.25	5.00	7.25

Table 1: Average number of strings from [1,100] needed to synthesize G.

N	CL \ GL	1	2	4	8	16	32	64
1	1	100	100	100	100	100	79	87
	2	100	100	100	100	83	91	91
	4	100	100	100	100	75	91	83
	8	100	100	100	100	87	100	93
2	1	87	100	66	67	38	32	18
	2	100	100	100	79	58	55	45
	4	100	100	100	87	91	83	72
	8	100	100	100	100	100	81	86
4	1	100	75	83	35	24	2	0
	2	100	100	100	62	53	26	3
	4	100	87	100	62	83	52	62
	8	100	100	87	62	83	79	60
8	1	100	100	32	11	0	0	0
	2	100	100	70	41	26	2	0
	4	100	100	45	66	62	57	28
	8	100	100	87	64	66	75	54

Table 2: Average percentage of strongly compatible hypotheses; [1,100].

R' that can transform the inputs to the outputs; or G' can be *incompatible* in that it parses the inputs in such a way that is different enough from G's parse so that no such replacement expression can be found.

A compatible G' is only a minor nuisance to the user, because the system will be able to synthesize a complete gap program using a compatible G'. Given a complete gap program, it is easy for the user to provide examples that can erase the differences between the synthesized program and the target, and in Section 5.8 we present a heuristic that often allows the system to do this without requiring the user to give more examples. On the other hand, an incompatible G' will cause the system to be unable to synthesize a complete gap program, and will leave the user wondering whether he should give up or provide more data.

The definition of compatibility is not constrained enough for our purposes, in that a compatible G' for some sample set S may bear little resemblance to the target gap pattern. In practice, we are interested only in those compatible patterns that are similar to the target. To define such patterns, suppose that the gap pattern $G' = c_0 h_1 c_1 h_2 c_2 ... h_m c_m$ matches a string α and parses it into the fields $q_1, q_2, ..., q_m$. That is, α is equal to the concatenation of the strings $c_0 q_1 c_1 q_2 c_2 ... q_m c_m$, and each of the q_i is a legal substitute for gap h_i in the pattern fragment $h_i c_i$. If the target pattern G parses α into fields $p_1, p_2, ..., p_n$ then we say that G' is *strongly compatible* with G on a string α if the bounds of the text matched by each gap q_i in $parse(G',\alpha)$ are contained within the bounds of the text matched by some p_j in $parse(G,\alpha)$. That is, G' is strongly compatible with G on α if that part of α matched by the fragment $g_i s_i$ in G is matched by $h_j c_j h_{j+1} c_{j+1} ... h_{j+k} c_{j+k} s_i$ in G', where $k \geq 0$, and h_j and c_{j+k} may be missing. We say that G' is *not strongly compatible* with G on α if G' is not strongly compatible with G on α. To extend the definition to sets of strings, we say that G' is *strongly compatible* with G on a set of strings S if it is strongly compatible with G on *every* element of S, and that G' is *not strongly compatible* with G on S if G' is not strongly compatible with G on *any* element of S.

If the target gap pattern G is equal to $s_0 g_1 s_1 g_2 s_2 ... g_n s_n$, then, by definition, all of the input samples can be parsed by this pattern. In general, if the inputs are the strings α and β, the target G parses them as

$$\begin{array}{cccccccccc} & p_{\alpha 1} & & p_{\alpha 2} & & p_{\alpha 3} & & & p_{\alpha n} & \\ s_0 & & s_1 & & s_2 & & s_3 & \cdots & & s_n \\ & p_{\beta 1} & & p_{\beta 2} & & p_{\beta 3} & & & p_{\beta n} & \end{array}$$

A gap pattern G' that is strongly compatible with G on $\{\alpha,\beta\}$ may find additional common constants inside each of the fields parsed by a gap of G, but does not find common constants in fields that are parsed by different gaps of G. That is, a strongly compatible G' respects the gap/constant columnar partitioning imposed by G. For example, a strongly compatible G' can notice that the strings $p_{\alpha 2}$ and $p_{\beta 2}$ both contain an `x`, and can thus break that single field in G up into two fields separated by an `x` in G'.

As an example of a not strongly compatible gap pattern, if the target G is "`a --- b --- c`", and these two samples are analyzed

```
accbaac
aaabbbc
```

Then "`a --- aa --- c`", the descriptive gap pattern for the set, is not strongly compatible with G.

Strong compatibility implies compatibility. If G is the gap pattern of the target gap program, and G' is strongly compatible with G, then the gap program synthesis system will be able to construct a gap program to effect the mapping described by the samples.

Theorem 28: If the program $P = G \Rightarrow R$ computes the transformation described by a sample set $S = \{<i_1,o_1>,<i_2,o_2>, \ldots,<i_n,o_n>\}$ and G' is strongly compatible with G on all i_k in S, then there is a replacement expression R' such that the program $P = G' \Rightarrow R'$ computes the transformation in S.

Proof: R' can be constructed from R be replacing each of the gap symbols in R by the sequence of constants and gaps in G' that match the same text as the gap symbol in G. For example, if the text matched by the fragment g_is_i in G is matched by $h_jc_jh_{j+1}c_{j+1}\ldots h_{j+k}c_{j+k}s_i$ in G', then every occurence of g_i in R can be replaced by $h_jc_jh_{j+1}c_{j+1}\ldots h_{j+k}c_{j+k}$ to yield the same text. $\square$

We have stated that compatible patterns are desirable because they lead to workable programs, and so strongly compatible patterns must also be desirable, but what are the chances that the descriptive gap pattern synthesis algorithm will find gap patterns that are strongly compatible with the target gap pattern? The experimentally determined chances are summarized in Table 2. Each entry in the table gives the average of the percentage of strongly compatible hypotheses generated during each 4 trials. For example, if one trial took 6 input samples, and made 5 proposals of which 3 were strongly compatible, then it would contribute a 60% compatibility percentage to the average. If the other 3 runs had compatiblity percentages of 75%, 83%, and 100%, then the figure reported for that class of gap patterns would be 79%. Note that the final, correct, proposal was counted as a strongly compatible hypothesis, and that sample sets of size 1 were neither analyzed nor counted.

This table has the same trends as Table 1: fewer gaps, longer constants, and shorter gap substitutes mean that more of the intermediate hypotheses are strongly compatible; more gaps, shorter constants, and longer gap substitutes make a larger percentage not strongly compatible. In Section 5.10 we show that the system will often converge to the target gap program as soon as it finds one strongly compatible gap pattern; thus a strong compatibility percentage above 25% probably means that the system performs tolerably well on such input.

The gap pattern synthesis process performs poorly when there are unintended features common to the text of the gap substitutes. Strong compatibility is a way of distinguishing between two classes of unintended common features found by the descriptive gap pattern synthesis algorithm: intra-gap features that occur within the same gap of the target gap pattern, and inter-gap features that occur between different gaps of the target. If all unintended features are intra-gap, then the gap pattern is strongly compatible with the target on the samples; if any of the unintended features are inter-gap, then the gap pattern is not strongly compatible with the target on the samples.

5.4.1 Intra-gap features

Intra-gap features occur in those parts of the strings α and β which are parsed by G as

$$s_0 \quad \begin{array}{c} p_{\alpha 1} \\ p_{\beta 1} \end{array} \quad s_1$$

that are instead parsed by G' as

$$s_0 \quad \begin{array}{c} p'_{\alpha 1} \\ p'_{\beta 1} \end{array} \quad s'_1 \quad \begin{array}{c} p''_{\alpha 1} \\ p''_{\beta 1} \end{array} \quad s_1$$

G' found that the gap substitutes $p_{\alpha 1}$ and $p_{\beta 1}$ had the additional constant s_1' in common.

For the purposes of simplifying the discussion contained in this section, we will assume that the algorithm has correctly found the constants of the target gap pattern, and that intra-gap features are found only among the strings that are serving as gap substitutes. In this setting, how many intra-gap features should one expect there to be?

The descriptive gap pattern synthesis algorithm finds the common symbols by computing the longest common subsequence (LCS) of the strings. The average length of the LCS of a pair of randomly chosen strings is an indication of the number of intra-gap features that will be normally found.

For example, if both strings are of length 32 over an equiprobable alphabet of size 100, then the empirically determined average LCS is 0.75 symbols long. If the strings are of length 64, then the empirically determined average length is 1.2. The average LCS does not tend to be very long until the two input strings get to be longer than the number of characters in the alphabet: the average LCS of two strings of length 64 over an alphabet of size 26 has length 5.3, and when the alphabet is of size 5, the average length rises to 21.

The gap pattern synthesis algorithm finds the features common to more than two samples by iteratively computing the LCS of the samples. For example, for three strings X, Y and Z, it computes $LCS(LCS(X,Y),Z)$. The average length of the LCS of two strings of size 64 over an alphabet of size 100 is 1.25 symbols. The experimentally determined average for three strings of this length is around 0.6 symbols, which is only a small decrease from 1.25. When a fourth sample is added, the decrease is even less dramatic, with an experimentally determined average of 0.5 symbols. The reason for small decreases in average size is that once a string gets to be a single character long, the chances that its single character occurs somewhere within a much longer string are high.

The performance of the gap pattern synthesis algorithm when trying to identify a gap pattern with $|g(G)|$ different gaps can be modeled by treating each of the gaps independently. The total average number of intra-gap features found is equal to the sum of the average lengths of the LCS of each gap. As an example, if the system is processing four samples that describe a target gap pattern with four gaps in which each gap has been filled with a string of size 64, the experimentally determined average of the number of intra-gap features found is around 2.0 symbols.

This model of the likelihood of intra-gap features indicates that the probability of having no intra-gap features goes to zero fairly slowly. This property is apparent to the user — with the algorithm as it stands, the user will either have to give a large number of input examples or explicitly supply input examples that have the unintended common features removed. Section 5.8 proposes a general-purpose heuristic that can help to solve this problem by removing portions of the gap pattern that are not needed to perform the transformation.

5.4.2 Inter-gap features

Inter-gap common features result in incompatible gap patterns, and are thus more serious. What are the chances of having inter-gap common features that are large enough to make a descriptive gap pattern not be strongly compatible? To get an idea, consider the neighborhood surrounding the text matched by the constant string s_1 of the target gap pattern G when matching two strings α and β,

$$\begin{array}{ccc} p_{\alpha 1} & & p_{\alpha 2} \\ & s_1 & \\ p_{\beta 1} & & p_{\beta 2} \end{array}$$

If there are inter-gap features common to $p_{\alpha 1}$ and $p_{\beta 2}$ in the gap pattern G', then its parse of the same text might look something like

$$\begin{array}{ccc} p'_{\alpha 1} & & p''_{\alpha 1}s_1p_{\alpha 2} \\ & s'_1 & \\ p_{\beta 1}s_1p'_{\beta 2} & & p''_{\beta 2} \end{array}$$

Of course, the situation could be considerably more complicated. For example, G' could have found inter-gap common features between the first and third gaps, even though they are not adjacent. G' could also have found features common to $p'_{\alpha 1}$ and $p_{\beta 1}s_1p'_{\beta 2}$ or common to $p''_{\alpha 1}s_1p_{\alpha 2}$ and $p''_{\beta 2}$. To simplify matters, we will ignore these complications, although the second has a significant impact on the problem.

By choosing to match a portion of $p_{\alpha 1}$ with a portion of $p_{\beta 2}$, G' has decided not to use the common feature s_1. If G' is descriptive, then $c(G')$ has the largest number of constants of any gap pattern matching α and β, and thus this choice means that s'_1 must be longer than s_1. The probability that there is such an s'_1 long enough to override s_1 is roughly related to the probability that the LCS of $p_{\alpha 1}$ and $p_{\beta 2}$ is longer than s_1. The chances of this event decrease as the length of s_1 increases; this can be seen in Table 2, where the percentage of strongly compatible hypotheses increases as the length of the constant strings in the target gap pattern increases.

There is also an informal argument for why there should not be very many inter-gap features. It is reasonable to suppose that the same field taken from two different examples might contain features that are unintentionally in common, so intra-gap features are to be expected. On the other hand, the chances that two different fields taken from two different examples have features in common is smaller. And this, taken with the fact that the unintentional common features have to be large enough to override consideration of the common constants that lie between the two fields, implies that a descriptive gap pattern is rarely not strongly compatible with the target gap pattern.

5.5 Experiment on [1,25]

The same experiment was repeated with the distribution of constant strings and gap substitute strings changed from random strings over an equiprobable alphabet of size 100 to random strings over an equiprobable alphabet of size 25. The results are summarized in Tables 3 and 4. Each of the entries in the tables are an average of four trials; the symbol ** occupies slots in which *any* of the four trials could not identify the target pattern within 15 samples.

These tables exhibit the same trends as Tables 1 and 2: fewer gaps, longer constants, and shorter gap substitutes make gap patterns easier to identify; more gaps, shorter constants, and longer gap substitutes make them harder. Reducing the size of the alphabet from 100 to 25 increased the likelihood of both inter-gap and intra-gap features (notice how the numbers jump when the size of the gap substitutes is greater than the size of the alphabet). This resulted in an increase in the number of samples required to identify a random gap pattern, and a decrease in the percentage of intermediate hypotheses that were strongly compatible with the target pattern.

5.6 Experiment on pseudo-text

The experimental studies just presented use sample data made up from equiprobable symbols drawn from alphabets of size 100 and size 25. Such data makes the study of the performance of the algorithm easier, but such data bears little resemblance to the kind of data that is normally processed by the EBE system. The EBE system processes text.

N	CL \ GL	1	2	4	8	16	32	64
1	1	2.00	2.25	2.75	3.25	3.75	8.00	**
	2	2.00	2.00	2.00	3.50	5.50	7.25	**
	4	2.00	2.00	2.50	3.75	5.50	7.00	**
	8	2.00	2.00	2.00	3.50	6.75	8.00	**
2	1	2.25	2.75	3.50	4.00	7.75	**	**
	2	2.00	2.25	3.00	4.00	7.75	**	**
	4	2.00	2.75	3.00	3.25	6.75	9.25	**
	8	2.50	2.00	3.00	4.25	5.00	7.00	**
4	1	2.25	3.00	7.00	**	**	**	**
	2	2.25	2.25	4.50	6.75	**	**	**
	4	2.50	2.25	3.25	4.25	5.75	**	**
	8	2.25	2.00	3.25	4.00	5.75	10.75	**
8	1	2.00	4.25	**	**	**	**	**
	2	2.00	2.75	4.75	8.00	**	**	**
	4	2.25	3.25	3.75	5.25	9.00	**	**
	8	2.50	3.00	3.50	5.25	7.25	12.25	**

Table 3: Average number of [1,25] strings needed to identify a random G.

N	CL \ GL	1	2	4	8	16	32	64
1	1	100	100	87	79	83	91	84
	2	100	100	100	70	80	87	90
	4	100	100	83	68	77	92	81
	8	100	100	100	72	83	95	92
2	1	100	75	70	50	25	6	3
	2	100	100	79	58	39	18	1
	4	100	100	87	70	65	65	46
	8	100	100	75	75	68	67	78
4	1	87	66	18	2	1	0	0
	2	100	87	57	39	10	0	0
	4	100	100	56	50	53	32	5
	8	100	100	79	35	63	67	71
8	1	100	36	9	0	0	0	0
	2	100	62	27	16	0	0	0
	4	100	45	75	52	29	3	0
	8	100	75	66	60	59	63	35

Table 4: Average percentage of strongly compatible hypotheses; [1.25].

While nothing hard and fast can be said about the structure of text, it is safe to say that an equiprobable set of 100 symbols does not capture its properties.

We ran a second set of experiments using random data that was generated using a distribution that bears a closer resemblance to text. The properties of text that it captures are:

- A small number of tokens dominate the constants of the gap pattern. These tokens include the punctation characters, such as "." and ",", and delimiting characters like *eol*.
- Over a third of the tokens in the gaps are space characters, and space characters do not occur very often in the constants. The majority of the rest of the tokens in the gaps are distinct tokens representing individual words, although there is a significant sprinkling of the punctuation characters.

Most of the short examples that have been given seem to contradict the second property; space characters are used as delimiters and play an important role in parsing the text. But while this is true for short examples, the system's performance on short examples is not at issue. In longer examples, space characters do not play a significant role in the constants of the target gap pattern simply because there are usually many spaces in every example, and there are probably not a like number of gaps in the target pattern. The actual pseudo-text distribution used was the following:

class	*constant filler*	*gap filler*
space character	5%	35%
4 punctuation characters	20% each	5% each
"words" from [1,500]	15%	45%

Random gap patterns and sample strings were generated in the same way as in the previous test, although 40 samples were used in these tests rather than 15. Table 5 gives the average number of samples spent trying to identify a pattern; a ** occupies those slots where more than 40 samples were analyzed without finding a G.

Note that the performance is horrible; the problem is simply that in order to converge to the target G the samples must contain no intra-gap features, and space characters are just too likely for this to ever happen when computing the LCS of the longer gap substitutions. For example, the probability that there are no spaces in a gap substitute of length 8 is 0.65^8, which is around 0.03, so there is less than a one in thirty chance that any one sample will contain no spaces, and only about a 0.72 chance that one

N	CL \ GL	1	2	4	8	16	32	64
1	1	2.00	2.00	9.50	14.25	**	**	**
	2	2.25	2.00	4.50	**	**	**	**
	4	3.25	3.25	7.75	**	**	**	**
	8	2.00	2.00	2.00	**	**	**	**
2	1	2.75	6.25	10.25	**	**	**	**
	2	2.50	3.50	8.50	**	**	**	**
	4	3.50	3.25	10.00	**	**	**	**
	8	2.25	3.75	7.25	**	**	**	**
4	1	3.00	5.50	**	**	**	**	**
	2	2.75	4.50	8.00	**	**	**	**
	4	3.25	4.75	9.50	**	**	**	**
	8	3.25	4.50	17.25	**	**	**	**
8	1	5.50	**	**	**	**	**	**
	2	2.50	7.00	15.00	**	**	**	**
	4	3.00	4.50	15.25	**	**	**	**
	8	3.00	9.25	10.00	**	**	**	**

Table 5: Average number of pseudo-text strings needed to identify a random G.

N	CL \ GL	1	2	4	8	16	32	64
1	1	100	100	66	76	87	62	25
	2	100	100	75	48	67	75	47
	4	100	91	61	48	52	47	77
	8	100	100	100	91	47	35	52
2	1	93	39	44	20	15	5	0
	2	91	60	57	18	37	12	25
	4	100	62	56	54	47	62	25
	8	100	78	57	34	20	27	12
4	1	85	42	4	0	0	0	0
	2	100	80	38	30	5	2	0
	4	100	47	76	33	10	5	5
	8	100	35	23	22	27	30	5
8	1	60	10	0	0	0	0	0
	2	100	39	31	0	0	0	0
	4	100	65	48	3	0	5	2
	8	100	38	16	22	22	5	0

Table 6: Average percentage of strongly compatible hypotheses; pseudo-text.

of 40 samples will be without spaces. The compatibility figures are a little more encouraging; many of them are over 25%, and in the next section we will introduce a heuristic that will take advantage of this fact.

The gap pattern synthesis system seems to perform better in practice than this last experiment indicates. Gap programs are typically used to manipulate text that is in some sort of tabular form, and this statistical model does not capture such text's properties of regularity and orderliness. These experiments also do not capture the fact that the user of an EBE system wants to help the system to converge to a working program, and wants to provide data that makes this possible. On the other hand, these tests do indicate that the system cannot deal well with long and noisy input samples. We have found that when a short input is added to such sample sets, the spurious intra-gap and inter-gap features vanish, and the target gap pattern is quickly synthesized.

5.7 Summary of gap pattern synthesis experiments

These tests and the resulting analysis may be easily summarized. Long gap constants, a small number of gaps, and short gap substitutes make gap patterns easier to identify; short gap constants, a large number of gaps, and long gap substitutes make gap patterns harder to identify.

There are two ways in which the gap pattern synthesis algorithm fails to find a hypothesis. The algorithm may find unintended intra-gap features in the samples, or it may find unintended inter-gap features. If all of the extraneous common features that it finds are intra-gap, then it will synthesize a gap pattern that is strongly compatible with the target pattern, and it will be able to produce a program, albeit a noisy one, that can map the input samples to the outputs. If any of the extraneous features are inter-gap, then the synthesized pattern will not be strongly compatible with the target, and the system will probably not be able to create a gap program that can effect the transformation.

The tests indicate that intra-gap features are prevalent in random samples, and that inter-gap features occur less often; our experience with the system indicates that these observations both hold true in practice. The next section proposes a heuristic called *pattern reduction* that attempts to remove intra-gap features.

5.8 Pattern reduction

The first heuristic extension to the gap program synthesis algorithm that we shall consider is one that tends to transform a strongly compatible gap pattern to the target pattern.

The virtue of the descriptive gap pattern synthesis algorithm is that it finds the largest number of constants common to the input samples; this is also one of its greatest flaws. For example, if the user is trying to convert Scribe italic notation:

```
A long and grungy passage, some of @i[which is in italic], and
@i[some of which is not] in italic, and some of which just happens
to @i[end up in] italic.
```

to the corresponding TEX notation:

```
A long and grungy passage, some of {\sl which is in italic}, and
{\sl some of which is not} in italic, and some of which just happens
to {\sl end up in} italic.
```

and he selects the first two italicized phrases as examples, then a descriptive gap pattern will net him the following gap program:

@i[*-1-* ␣ *-2-* ␣ *-3-* ␣ *-4-*]   ⇒   {\sl␣ *-1-* ␣ *-2-* ␣ *-3-* ␣ *-4-* }

The three space characters are probably not an intended feature; noticing them leads to a gap pattern that is strongly compatible with the target, but it would be better not to notice them.

One way of handling this problem is to have a heuristic that knows that space characters are not often significant, and that adjusts the gap pattern so that all of the "--- ␣ ---" sequences are reduced to a single gap "---". This approach will work for all those cases in which the spaces happen to be an insignificant feature that just happened to be common to the input samples. However one of the functions of the gaps and constants that are in the gap pattern is to carve the matched text up into fields that can be manipulated separately. If the space that is removed happens to be a significant separator between two gaps that have widely separated destinations in the replacement expression, then this heuristic will result in a gap pattern that parses the inputs in such a way that they cannot be mapped to the outputs. We do not use this heuristic.

Pattern reduction is a more general heuristic for reducing the number of extraneous constants found in a gap pattern. The idea is to examine the descriptive gap pattern G and the replacement expression R for those blocks of G that are copied *en masse* into R.

In the example above, all of the constituents, including the constants, of the contiguous block

```
-1- ⊔ -2- ⊔ -3- ⊔ -4-
```

of the gap pattern occur within the same contiguous block in the replacement expression

```
{\s|⊔ -1- ⊔ -2- ⊔ -3- ⊔ -4- }
```

If this block of gaps and constants were reduced to a single gap, then the resulting gap program

```
@i[ -1- ]  ⇒  {\s|⊔ -1- }
```

would still transform the examples in the same way, and would be more likely to successfully transform the next occurrence of the Scribe italic notation in the way that was intended. The heuristic treats runs of gaps that do not occur in the replacement expression, i.e. those whose text is being deleted from the input, as belonging to contiguous blocks.

The pattern reduction heuristic is applied only insofar as it leaves the general sense of the pattern the same. Pattern reduction is not performed if the reduced pattern does not match the inputs (which is possible), and it is not performed if it matches them in such a way that they cannot be transformed to the outputs. And in order to not drastically change the general sense of the pattern, the heuristic will not reduce away the leading and trailing constants of the pattern.

The pattern reduction heuristic can be applied only if the gap program synthesis algorithm has been able to find a replacement expression that can map the text parsed by the unreduced G' to the output. In other words, about the only time that this heuristic can be applied is when the pattern synthesis procedure has been able to find a gap pattern G' that is strongly compatible with the target pattern.

There is a problem with applying the gap reduction heuristic. One of the roles played by the gap pattern in a gap program is to distinguish the text that is supposed to be transformed from the text that is supposed to be skipped. The pattern reduction heuristic works from positive data, and has no knowledge of the text that the program's gap pattern is ***not*** supposed to match. For example, if the user wants to insert the text "`phone: `" in front of every phone number in an address list, he might have the following gap program in mind *bol* (*-1-*)␣ *-2-* - *-3-* *eol* ⇒ *bol* `phone:␣(` *-1-*)␣ *-2-* - *-3-* *eol*

After being given a few examples of what the program should do, the EBE system will probably come up with exactly the target program, and will then proceed to perform pattern reduction and get

bol *-1-* *eol* ⇒ `phone:␣` *-1-* *eol*

This gap program will insert "`phone: `" at the beginning of every line of the list, even those that have nothing to do with phone numbers.

The problem, in a nutshell, is that the system using the pattern reduction heuristic no longer identifies gap programs in the limit from positive data. We are not sure how to deal with this. We have had enough experience with situations in which pattern reduction does the right thing to know that we want something like it; on the other hand, we have *not* had enough experience with situations in which it does the wrong thing to have had the need to come up with user interface facilities for getting around the problem.

5.9 One output example will do

The gap program synthesis algorithm developed in Chapter 4 can be modified so that it often requires a small number of input examples and only *one* output example.

The program synthesis algorithm has two phases: a gap pattern synthesis phase that finds an approximation to the descriptive gap pattern common to the inputs, and a gap replacement synthesis phase that finds a way of constructing a gap replacement that can yield the outputs. The pattern synthesis phase works from input examples; the replacement synthesis phase works from output examples and the parse of the input examples.

The program synthesis algorithm can be modified so as to not require matched pairs of inputs and outputs. The modified algorithm has two classes of data: input/output example pairs as before, and unpaired input examples without a corresponding output. The pattern synthesis algorithm is run on both the paired and unpaired inputs, and approximates a descriptive gap pattern that can match all of the strings in both sets. The replacement synthesis algorithm is modified to consider only the paired examples: the paired inputs are parsed using the gap pattern synthesized from both the paired and unpaired inputs, and then the replacement synthesis algorithm is applied to the paired outputs to find a replacement expression that can construct them from the corresponding inputs.

For example, suppose that the user wants to compute a transformation that will map the input "`@i[italicized]`" to the output "`{\sl italicized}`", and also perform the analogous transformation to the input "`@i[slanted]`". The descriptive gap pattern common to the two inputs "`@i[italicized]`" and "`@i[slanted]`" is "`@i[` *-1-* `]`", and this pattern matches the first example binding the string "`italicized`" to *-1-*. The replacement synthesis algorithm then constructs a representation of all replacement expressions that yield the output "`{\sl italicized}`" using the gap *-1-* along with any necessary constants. In this case, it comes up with two replacement expressions: "`{\sl␣italicized}`" and "`{\sl␣` *-1-* `}`". The replacement synthesis algorithm is modified to prefer using the shortest expression, and where there are two of the same length it prefers the one that uses the largest number of gap symbols. In this case, the algorithm decides to use the replacement expression "`{\sl␣` *-1-* `}`".

Using this algorithm instead of one that requires paired input/output examples greatly reduces the effort required of the EBE system user. Supplying input examples usually requires little effort because they are almost always already present in the user's text; on the other hand, supplying output examples requires considerably more effort on the part of the user because they are usually not already present in the text and must be created.

To make this concrete, suppose that it takes 3 seconds of the user's time to locate and give an input example, 15 seconds to manually transform the input text to the desired output text, and an additional 2 seconds to give the output text as an example. If the user has to give 2 input examples and make up and give the 2 corresponding output examples, then it takes a total of 40 seconds. On the other hand, if the user gives two input examples and only one output example, then the process takes only 23 seconds, and the user is spared the task of making a redundant edit to the second example. If a third example is required, then the system that needs pairs takes 60 seconds (and *two* redundant edits), while the system that needs only one output takes 26 seconds.

The modified algorithm performs well; statistics to support this contention are presented in Section 5.10. In practice, the greatest source of ambiguity lies in finding the gap pattern, and once a workable gap pattern is found the shortest replacement expression that produces the outputs is almost always the desired one.

5.10 Gap program synthesis performance

This section presents test results that describe the performance of the gap program synthesis algorithm, Algorithm 12, with the addition of the pattern reduction heuristic and the heuristic for reducing the number of output examples required. The first step in each experiment was to generate a random gap pattern G using the method developed in Section 5.2. The next step was to produce a random replacement expression R to go along with G. An exact algorithm is used in the gap program synthesis process to construct the replacement expression, and we observed that the system's performance did not seem to be particularly sensitive to the actual replacement expression used, so the replacement expressions in the tests reported here were all sequences of length ten in which half of the symbols were randomly chosen from $g(G)$ and the other half were randomly chosen characters from the alphabet. For example, a random gap program with 2 gaps and constants of length 4 might be

```
wbeb -1- etvs -2- ijso  ⇒ -2- -2- av -1- bwe -1- -2-
```

Once a random gap program $P = G \Rightarrow R$ is produced, the next step is to synthesize a collection of sample data from which the synthesis algorithm will try to synthesize a program. In these tests the generated data consisted of 50 input/output pairs in which each input was a random element of $L(G)$ generated in the same way as in Section 5.2, and each output was the result of applying P to that input.

The goal of each experimental trial was to synthesize a gap program that could transform each of the 50 input samples to their corresponding output. Each trial was terminated as soon as the system had created such a program, even if the synthesized program was not equal to the program P that generated all of the data.

Tables 7 and 8 summarize the results of the experiment run on text made from a random selection of 100 equiprobable characters. Each entry in the tables records the average number of input samples and output samples used as evidence in the course of developing a gap program that could compute the transformation described by the 50 input/output pairs. As in Section 5.1, the entries are grouped by the characteristics of the target gap pattern: four major sections for the number of gaps in the target pattern,

N	CL \ GL	1	2	4	8	16	32	64
1	1	2.00	2.00	2.25	2.12	2.37	2.12	3.00
	2	2.00	2.00	2.25	2.00	2.00	2.25	2.25
	4	2.00	2.00	2.25	2.50	2.50	3.00	3.00
	8	2.50	2.00	2.25	2.50	2.00	2.50	2.75
2	1	2.50	2.00	3.00	2.50	3.25	4.00	10.66
	2	2.00	2.00	2.00	2.50	3.00	3.50	5.50
	4	2.00	2.00	2.00	2.25	2.50	3.75	3.00
	8	2.00	2.25	2.00	2.50	3.00	2.75	3.00
4	1	2.50	2.50	4.00	3.75	7.50	**	**
	2	2.00	2.00	2.50	2.50	3.00	5.00	4.00
	4	2.25	2.25	2.25	2.25	2.25	3.00	3.25
	8	2.00	2.25	2.50	2.25	2.50	3.25	3.00
8	1	3.00	3.25	4.75	7.50	**	**	**
	2	2.50	2.25	2.75	2.75	6.25	10.50	**
	4	2.25	2.25	2.25	2.75	2.75	3.50	4.25
	8	2.25	2.50	2.50	2.75	2.50	3.00	3.00

Table 7: Average number of inputs needed to identify P; [1,100].

N	CL \ GL	1	2	4	8	16	32	64
1	1	1.00	1.00	1.00	1.00	1.00	1.00	1.20
	2	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	4	1.00	1.00	1.00	1.00	1.00	1.00	1.25
	8	1.50	1.00	1.00	1.00	1.00	1.00	1.25
2	1	1.50	1.00	1.00	1.00	1.25	1.75	1.00
	2	1.00	1.00	1.00	1.00	1.00	1.25	1.00
	4	1.00	1.00	1.00	1.00	1.00	1.50	1.25
	8	1.00	1.00	1.00	1.00	1.25	1.50	1.25
4	1	1.25	1.00	1.00	1.50	1.00	**	**
	2	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	4	1.25	1.00	1.00	1.00	1.00	1.00	1.50
	8	1.00	1.00	1.25	1.00	1.25	1.00	1.25
8	1	1.75	1.75	2.50	1.50	**	**	**
	2	1.50	1.25	1.25	1.00	1.00	1.00	**
	4	1.25	1.00	1.00	1.00	1.00	1.00	1.00
	8	1.25	1.00	1.00	1.00	1.00	1.50	1.50

Table 8: Average number of outputs needed to identify P; [1,100].

N	CL \ GL	1	2	4	8	16	32	64
1	1	2.00	2.25	2.25	2.25	2.75	3.25	3.25
	2	2.00	2.25	2.50	2.50	3.25	3.00	3.00
	4	2.25	2.00	2.25	3.00	3.00	3.00	4.00
	8	2.25	2.25	2.25	2.25	2.75	3.00	3.00
2	1	2.25	2.50	5.50	3.25	6.00	5.00	4.50
	2	2.75	2.50	2.75	2.75	3.75	3.50	6.50
	4	2.25	2.50	2.00	2.50	3.25	3.50	5.00
	8	2.25	2.00	2.50	2.75	3.00	3.00	3.25
4	1	3.00	3.75	5.00	7.00	**	**	**
	2	3.00	2.25	3.50	4.00	6.33	**	**
	4	2.75	2.75	2.75	3.00	3.25	4.25	5.66
	8	2.75	2.25	3.00	3.00	3.25	3.50	4.00
8	1	5.50	9.25	10.00	**	**	**	**
	2	3.12	3.12	3.62	9.33	**	**	**
	4	3.00	2.75	3.00	2.75	3.75	4.75	**
	8	3.00	3.00	2.75	3.00	3.25	3.50	**

Table 9: Average number of inputs needed to identify P; [1,25].

N	CL \ GL	1	2	4	8	16	32	64
1	1	1.00	1.00	1.00	1.00	1.00	1.50	2.00
	2	1.00	1.00	1.00	1.00	1.25	1.25	1.75
	4	1.25	1.00	1.00	1.00	1.00	1.50	1.75
	8	1.25	1.00	1.00	1.00	1.25	1.50	1.75
2	1	1.25	1.00	1.00	1.00	1.25	1.33	2.00
	2	1.75	1.00	1.00	1.25	1.00	1.50	1.75
	4	1.25	1.00	1.00	1.00	1.75	1.50	1.50
	8	1.25	1.00	1.00	1.00	1.50	2.00	2.00
4	1	1.75	1.75	2.00	1.33	**	**	**
	2	2.00	1.00	1.00	1.25	1.33	**	**
	4	1.50	1.00	1.00	1.50	1.50	1.25	2.00
	8	1.75	1.00	1.50	1.50	2.00	2.00	2.50
8	1	2.75	2.75	1.50	**	**	**	**
	2	2.12	1.75	1.50	3.33	**	**	**
	4	2.00	1.25	1.00	1.00	1.50	2.25	**
	8	1.75	1.00	1.25	1.75	1.75	2.00	**

Table 10: Average number of outputs needed to identify P; [1,25].

N	CL \ GL	1	2	4	8	16	32	64
1	1	2.43	2.87	3.27	3.62	2.50	3.25	**
	2	2.25	2.37	2.14	3.75	6.25	4.00	**
	4	2.50	2.75	3.25	3.25	3.50	4.00	4.75
	8	2.00	2.25	2.75	4.25	2.50	4.25	4.50
2	1	3.75	2.25	3.00	4.00	7.33	5.66	**
	2	3.00	2.25	4.75	3.75	3.50	6.00	**
	4	3.00	3.00	3.50	3.00	3.50	4.00	5.75
	8	2.25	3.25	3.75	3.50	4.50	5.50	5.00
4	1	3.75	5.00	2.50	3.00	**	**	**
	2	4.00	2.75	3.25	3.00	8.00	8.50	**
	4	3.25	4.25	2.75	4.00	4.75	6.00	9.66
	8	3.75	3.50	4.00	4.00	5.00	3.75	5.33
8	1	5.50	7.00	**	**	**	**	**
	2	4.25	4.00	10.00	7.33	11.00	**	**
	4	4.75	3.50	3.75	4.50	4.66	6.75	**
	8	4.00	3.25	3.50	3.75	5.00	4.50	**

Table 11: Average number of inputs needed to identify P, pseudo-text.

N	CL \ GL	1	2	4	8	16	32	64
1	1	1.18	1.00	1.36	1.75	1.25	1.75	**
	2	1.12	1.12	1.14	2.00	3.00	2.25	**
	4	1.25	1.00	2.25	1.25	2.25	3.00	3.50
	8	1.00	1.00	1.75	2.25	1.50	2.75	3.50
2	1	2.25	1.00	1.50	2.25	2.66	3.00	**
	2	2.00	1.25	1.75	2.75	2.50	4.00	**
	4	2.00	1.00	2.00	2.00	2.50	2.66	3.25
	8	1.25	1.50	2.50	2.00	3.50	3.25	3.50
4	1	2.25	1.25	1.50	2.00	**	**	**
	2	3.00	1.50	1.50	2.00	2.00	3.50	**
	4	2.25	1.75	1.75	3.00	3.00	3.75	4.33
	8	2.75	1.75	2.25	2.00	3.25	2.75	4.00
8	1	2.62	1.50	**	**	**	**	**
	2	2.50	2.50	2.33	2.00	4.00	**	**
	4	3.75	1.75	2.00	3.50	2.66	3.75	**
	8	3.00	1.50	1.75	2.50	3.25	3.50	**

Table 12: Average number of outputs needed to identify P, pseudo-text.

4 rows for the length of the constant strings in the target, and 7 columns for the length of the strings used as gap substitutes in the random input samples. An entry of ** appears if the synthesis procedure could not converge within 15 input and output samples.

The algorithm performs very well. When given fairly short data to analyze, it usually needed two or three input samples and a single output sample to find a transformation that could map all 50 inputs to their outputs. Even with longer samples, the algorithm usually required between two to four input samples and one output sample.

The algorithm's good performance can be explained by examining the compatiblity percentage figures shown in Table 2 on page 92. That table shows the average percentage of the synthesized gap patterns that are strongly compatible with the target gap pattern. The pattern reduction heuristic has the tendency of mapping a strongly compatible gap pattern to the target gap pattern, or at least to a gap pattern that is as good as the target on the samples analyzed. The algorithm is likely to converge to a workable gap program as soon as it is able to synthesize a single strongly compatible gap pattern. A class of data that has a non-negligible compatibility percentage, say one that is over 25%, is thus likely to be easy to identify.

A second set of tests was performed for the distribution of the equiprobable alphabet of size 25; these tests are summarized in Tables 9 and 10. The algorithm performed well, but not quite as well as for an alphabet size of 100 because the probability of synthesizing gap patterns that are not strongly compatible with the target is higher when the alphabet is smaller.

The third set of tests performed used the pseudo-text distribution presented in Section 5.6; these tests are summarized in Tables 11 and 12. Those classes of gap patterns that had a strong compatility percentage over 25% were fairly easily identified, but the input samples were much too noisy in the other cases.

5.11 Tokenization and gap bounding heuristics

This section presents two heuristics, *tokenization* and *gap bounding*, that while unimportant to the theoretical treatment of the problem of identifying gap programs from examples, are of essential importance to the practical application of the formal techniques.

The problem addressed by *tokenization* is very simple. The "natural" gap program to transform this set of strings:

```
The Yankees beat Baltimore, 10 to 3.
The Mets beat Chicago, 6 to 5.
The Angels beat Detroit, 2 to 0.
```

to this set:

```
Yankees 10, Baltimore 3.
Mets 6, Chicago 5.
Angels 2, Detroit 0.
```

is this one:

```
The⊔ -1- ⊔beat⊔ -2- ,⊔ -3- ⊔to⊔ -4- .    ⇒
-1- ⊔ -3- ,⊔ -2- ⊔ -4- .
```

Yet the system that we have described thus far would produce this program:

```
The⊔ -1- e -2- s⊔beat⊔ -3- o -4- ,⊔ -5- ⊔to⊔ -6- .    ⇒
-1- e -2- s⊔ -5- ,⊔ -3- o -4- ⊔ -6- .
```

This program seems wrong. Why does this program seem wrong? It is noisy, and while it can correctly transform each of the input samples to each of the output samples, it will be unable to transform the first baseball score in which the name of the winning team does not contain the letter "e". It seems wrong because one would not normally notice the fact that all of the baseball teams had the letter "e" in them, and one would not appreciate having this pointed out. On the other hand, if the system had found a shared ";" in this spot, instead of an "e", then we would have been quite pleased, because we would have noticed the same thing ourselves.

In order to wean the system from this objectionable performance, we will give the system "knowledge" of the standard lexical conventions for reading text, so that it can avoid violating them. The implementation of a straightforward version of this embedded knowledge is simple. The system reads tokenized text and performs its analysis in terms of the tokens, rather than doing it character by character. For example, the 98 characters in the three input samples above are read and analyzed as the following set of 45 tokens.

```
The ⊔ Yankees ⊔ beat ⊔ Baltimore , ⊔ 10 ⊔ to ⊔ 3 .
The ⊔ Mets ⊔ beat ⊔ Chicago , ⊔ 6 ⊔ to ⊔ 5 .
The ⊔ Angels ⊔ beat ⊔ Detroit , ⊔ 2 ⊔ to ⊔ 0 .
```

The descriptive gap pattern found for the tokenized set is:

```
The⊔ -1- ⊔beat⊔ -2- ,⊔ -3- ⊔to⊔ -4- .
```

which is exactly the sort of gap pattern that we want. The standard analysis is then done to create a gap replacement.

-1- ⊔ *-3-* ,⊔ *-2-* ⊔ *-4-* .

The tokenization heuristic can be viewed formally as an expansion of the alphabet. Before we had a small fixed alphabet, such as the ASCII character set, and now we have one of unbounded size that has as its members all of the lexemes that our tokenization heuristics allow. One of the ramifications of the decision to use tokenization is that the formal results proven relative to a bounded-size alphabet become less relevant. In a sense, this is all for the better, because most of the time the existence of a bound on the size of the alphabet was working against us — making our results clumsier to state and harder to prove.

The second heuristic is *gap bounding*, which is a simple bit of post-processing done on the synthesized gap programs. In our definition of the gap pattern matching process the only constraint imposed on a gap is that it cannot match any text that contains the constant string following the gap in the pattern. We do not constrain the amount of text that can be matched by a gap, and so a single gap can merrily match 10,000 lines of text, even if it matched less than two lines in the examples in all of the examples given.

Gap bounding is a way of dealing with the problem of gap programs running berserk in this manner. After the gap pattern is synthesized, we impose an *a priori* bound on the number of characters or lines that can be matched by the gap. For example, if a gap matched at most 3 lines in the examples given, we restrict it to matching no more than 4 lines of text when it is searching the text. The exact gap bounding function that we employ is to restrict a gap that matches text that crosses at most l end-of-line boundaries in any example to crossing no more than $\lfloor 1.5l \rfloor$ end-of-line boundaries when scanning the file. Although this changes the expressive capabilities of the gap programs, it does so in an uninteresting way, because the gap bound can be arbitrarily extended by adding more samples involving longer gaps.

5.12 A practical algorithm for gap program synthesis

Algorithm 13 encodes a practical gap program synthesis process that is used to create a new gap program when the EBE system's current hypothesis does not work.

```
Inputs: a set of input/output pairs $S=\{<i_1,o_1>,<i_2,o_2>,...,<i_n,o_n>\}$;
        a set of unpaired inputs $I=\{i_{n+1},i_{n+2},...i_{n+m}\}$;
        and a tokenization function $T$.
Output: a gap program $P$ or an indication of failure.

Tokenize the samples in $S$ and $I$ using $T$;
Approximate a descriptive gap pattern $G$ common to
   $\{i_1,i_2,...,i_n,i_{n+1},...i_{n+m}\}$ using Algorithm 10;
Use $G$ to parse $\{i_1,i_2,...,i_n\}$;
Compute the shortest replacement expression $R$ that maps $i_k$ to $o_k$
   for $k=1,2,...,n$ using Algorithm 11;
Perform pattern reduction on $G$ and $R$;
Bound the gaps of $G$;
return $P = G \Rightarrow R$;
```

Algorithm 13: A practical gap program synthesis algorithm

To demonstrate each of the phases of the algorithm, we will follow the transformation of the address list shown in Figure 1 to a series of form letters like the one shown in Figure 2. To demonstrate the transformation, the user selects the input example

```
Mr. John Doe
136 Lenape Lane
Sioux Falls, North Dakota
```

transforms it the form letter of Figure 2, and selects that text as an output example. He then gives the second address as another input example

```
Dr. Penelope Wise
7213 Central Park West
New York, New York
```

and the system begins its analysis. The first step of the analysis is to tokenize the two input samples

```
Mr. John Doe
136 Lenape Lane
Sioux Falls, North Dakota

Dr. Penelope Wise
7213 Central Park West
New York, New York

Ms. Candice Barr
5565 Cormorant Avenue
Ewa Beach, Hawaii

Mr. James Jones
711 Hamilton Road
Arlington, Virginia
```

Figure 1: An address list.

```
Mr. John Doe
136 Lenape Lane
Sioux Falls, North Dakota

Dear Mr. Doe,

Over twenty years ago, our genealogical research organization
began a far-reaching investigation of the history of the
distinguished families of North Dakota.  We found the Doe Family
to play a prominent role in this history, beginning with Ichabod
Doe, one of the founding fathers of the village that went on to
become Sioux Falls, and continuing up to present day times.

The complete collected history of the Doe Family, bound in
rich naugahyde and destined to become a family heirloom, can
be yours by sending only $117.98 along with postage and handling
to the North Dakota Heritage Foundation.

Sincerely yours,
Ezra K. Sterling, Ph.D.

PS: For a limited time only, additional copies, ideal for
library bequests, are available at the reduced rate of just
$47.98 each.
```

Figure 2: A sample output form letter.

bol Mr . ⊔ John ⊔ Doe *eol*
136 ⊔ Lenape ⊔ Lane *eol*
Sioux ⊔ Falls , ⊔ North ⊔ Dakota *eol*
eol

bol Dr . ⊔ Penelope ⊔ Wise *eol*
7213 ⊔ Central ⊔ Park ⊔ West *eol*
New ⊔ York , ⊔ New ⊔ York *eol*
eol

It then computes an approximation to the Longest Common Subsequence of the two tokenized samples

bol . ⊔ ⊔ *eol*
⊔ ⊔ *eol*
⊔ , ⊔ ⊔ *eol*
eol

The next step is to insert gaps into the common subsequence to transform it to a gap pattern that matches both input samples

bol -1- .⊔ *-2-* ⊔ *-3- eol*
-4- ⊔ *-5-* ⊔ *-6- eol*
-7- ⊔ *-8-* ,⊔ *-9-* ⊔ *-10- eol*
eol

This gap pattern yields a unique parse of the two input samples, although the system actually parses only the single input sample that happens to be paired with an output.

-1-	*-2-*	*-3-*	*-4-*	*-5-*	*-6-*	*-7-*	*-8-*	*-9-*	*-10-*
Mr	John	Doe	136	Lenape	Lane	Sioux	Falls	North	Dakota

The replacement expression synthesis algorithm finds the shortest replacement expression that can produce the output example using these text fragments and any needed constant strings. The requirement that the expression be short, and that it use gap fragments wherever possible, leads to a replacement expression other than the literal text of the single output sample.

```
-1- .⊔ -2- ⊔ -3- eol
-4- ⊔ -5- ⊔ -6- eol
-7- ⊔ -8- ,⊔ -9- ⊔ -10- eol
eol
Dear⊔ -1- .⊔ -3- , eol
eol
Over⊔twenty⊔years⊔ago,⊔our⊔genealogical⊔research⊔organization eol
began⊔a⊔far-reaching⊔investigation⊔of⊔the⊔history⊔of⊔the eol
distinguished⊔families⊔of⊔ -9- ⊔ -10- .⊔⊔We⊔found⊔the⊔ -3- ⊔Family eol
to⊔play⊔a⊔prominent⊔role⊔in⊔this⊔history,⊔beginning⊔with⊔Ichabod eol
-3- ,⊔one⊔of⊔the⊔founding⊔fathers⊔of⊔the⊔village⊔that⊔went⊔on⊔to eol
become⊔ -7- ⊔ -8- ,⊔and⊔continuing⊔up⊔to⊔present⊔day⊔times. eol
eol
The⊔complete⊔collected⊔history⊔of⊔the⊔ -3- ⊔Family,⊔bound⊔in eol
rich⊔naugahyde⊔and⊔destined⊔to⊔become⊔a⊔family⊔heirloom,⊔can eol
be⊔yours⊔by⊔sending⊔only⊔$117.98⊔along⊔with⊔postage⊔and⊔handling eol
to⊔the⊔ -9- ⊔ -10- ⊔Heritage⊔Foundation. eol
eol
Sincerely⊔yours, eol
Ezra⊔K.⊔Sterling,⊔Ph.D. eol
eol
PS:⊔For⊔a⊔limited⊔time⊔only,⊔additional⊔copies,⊔ideal⊔for eol
library⊔bequests,⊔are⊔available⊔at⊔the⊔reduced⊔rate⊔of⊔just eol
$47.98⊔each. eol
```

The system then performs pattern reduction on this pattern/replacement pair, which notices that the pattern fragments "*-4-* ⊔ *-5-* ⊔ *-6-*", "*-7-* ⊔ *-8-*", and "*-9-* ⊔ *-10-*" are treated as blocks in the replacement expression and reduces them to single gaps. The final phase of the gap program synthesis process is gap bounding, which in this case restricts all of the gaps of the program's pattern to matching text within a single line. The result of this process is a program that will serve to produce the rest of the form letters.

```
bol -1- .⊔ -2- ⊔ -3- eol
-4- eol
-5- ,⊔ -6- eol
eol
```

⇒

```
-1- .␣ -2- ␣ -3- eol
-4- eol
-5- ,␣ -6- eol
eol
Dear␣ -1- .␣ -3- , eol
eol
Over␣twenty␣years␣ago,␣our␣genealogical␣research␣organization eol
began␣a␣far-reaching␣investigation␣of␣the␣history␣of␣the eol
distinguished␣families␣of␣ -6- .␣␣We␣found␣the␣ -3- ␣Family eol
to␣play␣a␣prominent␣role␣in␣this␣history,␣beginning␣with␣Ichabod eol
-3- ,␣one␣of␣the␣founding␣fathers␣of␣the␣village␣that␣went␣on␣to eol
become␣ -5- ,␣and␣continuing␣up␣to␣present␣day␣times. eol
eol
The␣complete␣collected␣history␣of␣the␣ -3- ␣Family,␣bound␣in eol
rich␣naugahyde␣and␣destined␣to␣become␣a␣family␣heirloom,␣can eol
be␣yours␣by␣sending␣only␣$117.98␣along␣with␣postage␣and␣handling eol
to␣the␣ -6- ␣Heritage␣Foundation. eol
eol
Sincerely␣yours, eol
Ezra␣K.␣Sterling,␣Ph.D. eol
eol
PS:␣For␣a␣limited␣time␣only,␣additional␣copies,␣ideal␣for eol
library␣bequests,␣are␣available␣at␣the␣reduced␣rate␣of␣just eol
$47.98␣each. eol
```

5.13 Failure

We have talked at great length about how the gap program synthesis algorithm can succeed, now we discuss how it fails.

One way in which the program synthesis process can fail is to successfully synthesize a gap program that transforms the inputs to the outputs, but to not produce the program that the user had in mind. This failure can be remedied by having the user provide more examples of the desired function's behavior; the results of the experimental performance analysis indicate that not very many more examples need to be provided, and in practice these examples are usually easy to come by.

The other way that the process can fail is that it can be unable to find any gap program at all that can perform the transformation shown in the examples. There are several causes for this problem: more examples might be required to steer the algorithm's heuristics to the target gap program, the *ad hoc* tokenization performed by the algorithm might be faulty, the transformation might not be expressible by a gap program, or the sample data might contain typographic errors.

The program synthesis algorithm may require more examples either when synthesizing an approximation to the descriptive gap pattern or when trying to produce the replacement expression. The pattern synthesis process fails if the common subsequence found does not lead to a gap pattern that matches the inputs; Theorem 23 shows that the user can eventually provide examples to make the pattern synthesis succeed. The replacement synthesis process fails if the synthesized gap pattern parses the examples in such a way that the parsed pieces cannot be rearranged to form the output strings. This can happens when the gap pattern found by the pattern synthesis process is not *the* pattern of the target gap program. Theorem 23 shows that the pattern synthesis process does converge to *the* correct pattern, and thus the user can recover from this problem by supplying more examples.

Another cause of failure is the tokenization heuristic. The tokenization heuristic imposes an *ad hoc* grouping on the characters of the examples which may preclude any gap program from transforming them. For example, if the goal was to transform all words like "`spineless`" and "`numberless`" to "`without spine`" and "`without number`", then the tokenization heuristic will cause the synthesis process to fail because it treats "`spineless`" as an atomic token whose internal structure is not manipulable. Our approach to dealing with this problem is to start the program synthesis process over again after a failure, but this time with a character-at-a-time tokenization. If the character-at-a-time analysis fails, then it fails for one of the other reasons that we discuss here. If it succeeds, then the program might be noisier than we would like, but it is better than nothing, and the pattern reduction heuristic might whisk away some of the objectionable features that are found.

The third kind of failure arises from limitations of gap programs. Gap programs are not very powerful, and there are a number of situations in which the transformation that the user wants to perform cannot be expressed as one. If a transformation cannot be expressed as a gap program, then no algorithm for gap program synthesis will ever be able to find a program to effect it. The question is how to deal with this eventuality. It would be nice if the system were able to authoritatively tell the user that no gap program exists to perform the transformation that he desires; unfortunately, Theorem 11 shows that discovering this piece of information is computationally difficult.

A typographic error in the examples is difficult to distinguish from a set of data that cannot be transformed by a gap program. The system could try to automatically perform something akin to spelling correction on the examples given, but this seems to be inappropriate, not to mention ill defined and slow. It seems better to provide the user with facilities for examining and modifying the examples he has given.

The approach that we have taken to these last two problems is to educate the user about the capabilities of the system, and to provide him with as much information as possible about the current state of the system. Gap programs are simple, and we would like the user to have a good idea about whether the transformation he desires can be handled by a gap program.

5.14 Editing by example in *U*

The program synthesis algorithm that we have developed is embedded inside of the EBE subsystem of a screen editor called *U* [66]. *U* is a full function screen editor implemented within the T programming system [71, 72]. The user interface of the editor is similar to that of Wood's Z [98], which in turn was inspired by many screen editors implemented at Yale and elsewhere, all originating in the work of Irons [44]. We will describe the details of the user interface only as they pertain to the EBE subsystem.

The gap program synthesis algorithm accepts pairs of strings as inputs and returns either a gap program that can transform the input strings to the outputs or an indication that no such gap program could be found. The objects in such a system are input samples, output samples, gap programs, and indications of failure; the user interface is responsible for presenting these objects and supplying usable and coherent mechanisms for manipulating them. In this section, we enumerate the kinds of interactions that a user might want to have with an EBE system, and describe the specifics of the *U* editor's implementation of these interactions.

The *U* user interacts with the EBE system using five *U* commands: *start-ebe-session*, *add-input-example*, *add-output-example*, *run-gap-program*, and *modify-ebe-state*. The first is an initialization command, the second two perform the basic operations on input/output examples, the fourth performs the basic operation on programs, and the last is an entry into a facility that provides a wide range of other operations on the objects of the system.

We will describe these commands by showing how they can be used to to change a file that makes use of Scribe's notation for italicizing text:

```
A paragraph filled with @i[italicized] words that flow
trippingly over the page.  @i[This is an entire sentence
in italics that spans several lines, and so the gap program
must contain a gap that can cross line boundaries.] @i[All
sorts of italic phrases.] @i[Even more italic phrases.]
@i[The author goes for cheap effects.] @i[Remember, gap
patterns cannot match parenthesis @i[so forget about nested
expressions.]] Check out the previous sentence after we're
done.
```

to a file that uses TEX's italic notation:

```
A paragraph filled with {\sl italicized} words that flow
trippingly over the page.  {\sl This is an entire sentence
in italics that spans several lines, and so the gap program
must contain a gap that can cross line boundaries.} {\sl All
sorts of italic phrases.} {\sl Even more italic phrases.}
{\sl The author goes for cheap effects.} {\sl Remember, gap
patterns cannot match parenthesis {\sl so forget about nested
expressions.}} Check out the previous sentence after we're
done.
```

5.14.1 Interacting with the EBE subsystem

The first editor command associated with EBE is *start-ebe-session*, an initialization command that clears the system's database of examples and displays a window that contains a succinct representation of the state of the EBE system called the EBE-*state-window*.

```
Inputs:
Outputs:
  no program
```

The window has a line each for the input and output examples and some space for displaying the current program hypothesis. Since the *start-ebe-session* command has just been given, there are no examples, and thus no program.

From the synthesis algorithm's point of view, the principal operation on input/output samples is that of *collection*; the user's point of view is different. To the user, each string in the sample must first be *created*. Making the creation operation easy and convenient is the reason for embedding the EBE system within a text editor. If the EBE system were not embedded in a text editor, but rather existed as an independent text transformation system, it would probably not be of much use.

Once the sample strings exist, some way must be found to bring them to the attention of the EBE system. The exact mechanism used for this depends on the editor, but in most text editors example collection can be implemented in a natural way. Most editors support some sort of *selection* mechanism that allows the user to specify a piece of text to be a parameter to an editing command. Example specification could be implemented via an editing command that accepts a string as a parameter and presents it as the next input or output example to the EBE system. Alternatively, if the editor supports the graphical object-oriented style of interaction, then one might instead specify an example by using a general purpose copying command to move the selected text to a place beside the other examples in an "example bin". The more naturally this operation can be made to fit within a particular editor's editing paradigm, the better.

In *U*, an input example is specified by selecting the text that contains the example and invoking the *add-input-example* command. For instance, the first input example might be the string "@i[italicized]". Once the text has been given as an example, a concise representation of it is registered in the EBE-state window:

```
Inputs:   "@i[itali..."
Outputs:
```
  no program

There is no program because the system needs at least one piece of output data in order to produce a program.

Output samples can often be easily produced by editing the text that had been selected as an input example. In this case, our hypothetical user would probably delete the characters "@i[", insert the characters "\sl{ ", and then change the "]" to a "}". Of course, he is free to do something more perverse; this is one of the advantages of having an EBE system that works from input/output behavior. No matter how the user produces it, once the output sample text exists in some form, the user can select it and bring it to the EBE system's attention by invoking the *add-output-example* command. This command adds the text to the system's example database paired with the last input example given, and triggers the process that analyzes the data to produce a gap program.

The EBE-state window is updated to reflect the system's new example and new program hypothesis:

```
Inputs:   "@i[itali..."
Outputs:  "{\sl ita..."
  "@i[italicized]"  ⇒  "{\sl italicized}"
```

Notice that the ***program synthesis*** operation was invoked implicitly. In *U*, program synthesis is performed every time the user provides a new example or modifies the example set in any way. This style of interaction allows the system to propose a sample program as soon as it can possibly find one. Gap program synthesis is cheap enough so that this is a viable way of structuring the user's interactions with the program synthesis operation. If we were using a more expensive program synthesis procedure, it would probably have been best to give the user the option of invoking the program synthesis operation manually.

The user could *execute* this gap program now, but he happens to want a more general program, so he selects the text beginning "`@i[This is an entire sentence...`" and gives the *add-input-example* command again. The addition of a new sample triggers the process of synthesizing a gap program that can simultaneously match both inputs and transform the first input to the corresponding output. The synthesis process is successful, and the EBE system is able to create the target program from two input examples and one output example:

```
Inputs:   "@i[itali..."   "@i[This ..."
Outputs:  "{\sl ita..."
  "@i[" -1- "]"  ⇒  "{\sl " -1- "}"
```

The *run-gap-program* command instigates a search forward from the current cursor location for the next piece of the text that matches the gap pattern. If it cannot find any text matching the pattern, then it tells the user that no such text can be found. If it can find such a piece of text, then it will highlight it, and ask the user:

```
Transform, Proceed, Skip, or Quit (t/p/s/q)?
```

If the user answers "`t`" for "`Transform`", then he is electing to stay in the EBE system's single-stepping mode. In this mode, the system replaces the highlighted text with the result of the replacement expression, and asks the user:

```
Try again (y/n)?
```

An answer of "n" for "No" would cause the system to exit this searching mode, and would leave the user at the location of the text last transformed. If the user answers "y" for "Yes", then the system searches for the next occurrence of text matching the gap pattern and continues the single-stepping process from there.

If the user had answered "p" for "Proceed" to the first question, then the gap program would have been iteratively applied to the rest of the file without further user intervention. In the current scenario, the program would successfully transform to TEX all of the the Scribe italicize commands that follow the highlighted text in the file (except the nested one). If the user had answered "s" for "Skip", then the highlighted text would not have been transformed, and the system would continue on to find the next piece of text that matches the gap pattern. If the user had answered "q" for "Quit", or given the standard *U* editor *cancel* command, then the gap program execution process would have been halted, and the user would have been returned to editing the file at the point where he halted the program execution.

The EBE system can produce a program that computes the wrong transformation. If the user executes the program anyway, then there is a chance that it will run amok and destroy the user's text. If this destruction is not reversible, then you can be sure that that will be the last time this particular user trusts anything to the EBE system. The *U* editor provides an *undo* facility, that can restore the text to a previous state; the undo facility is essential for the acceptance of the EBE system.

5.14.2 Other operations

The four commands *start-ebe-session*, *add-input-example*, *add-output-example*, and *run-gap-program* provide the core facilities for interacting with the editing by example system. These commands provide the essentials, but there are many other operations that the user might like to perform on the objects of the system.

For example, if the user has made a mistake in entering an example he has given to the system, then the only way that he could recover from this mistake using the four basic commands is to start over from scratch. Rather than that, the user might want to *view* the text of the examples that he has given, and *retract* the one containing the error, or perhaps *edit* it and ask the system to redo its analysis.

The user should also be able to view the synthesized programs. One might argue that the user should not see the programs, because he should not be bothered with this inessential detail. But the system that we are considering will not work until the user has provided adequate examples, and will work only if the user's target program is a gap program. If the user is able to see and understand the hypothesized gap program, then it can give him confidence that he has given enough examples to capture the general rule governing the transformation. Even if the user knows nothing about programming, and does not fully understand the displayed code, it is worth showing it to him because he might *learn* something. The purpose of the system is not to provide a crutch that prolongs ignorance of programming, but to provide a tool for helping people to solve their text processing problems; teaching the user about programming is probably the best way of helping him.

For this same reason, the user should be able to modify the system's synthesized programs, and write his own programs using the same notation.

The language that the system uses to express its synthesized programs should be able to express more programs than the system can synthesize. This is not critical in our implementation, since the system is written in T [71, 72], a dialect of Lisp, and the user has access to the power and notational elegance of T at all times. If the user has something that requires real programming, then he can write a program in a real programming language. But if the user has something that fits nicely into the concise sublanguage implemented by the EBE system, then he can either write that program himself or have the EBE system write it for him. Chapter 6 presents several approaches to extending the programming power of the gap program synthesis system.

All of these facilities, and others, are provided by the *U* editor's *modify-ebe-state* command. This command takes advantage of the editor's facilities to naturally (in terms of editor's standard interfaces) and cheaply (in terms of implementation) provide a superset of these operations.

The *modify-ebe-state* command creates a textual representation of the state of the EBE system, places the text into a window, and allows the user to edit it at will. Figure 3 shows how the EBE system's state would be represented for the current scenario.

The text displayed in the window contains executable expressions written in T; lines beginning with a ";***" are non-executable comments. There are two T expressions in the text, one that specifies the examples and one that specifies the gap program

```
;*** Current sample set:
(define-EBE-examples

 ;*** Example 1.
 (
   "@i[italicized]"  =>
   "{\sl italicized}"
 )

 ;*** Example 2.
 (
   "@i[This is an entire sentence" eol
   "in italics that spans several lines, and so the gap program" eol
   "must contain a gap that can cross line boundaries.]"

   ;*** No output for example 2 was supplied.
 )

)

;*** Current gap program hypothesis:
(define-gap-program
    "@i[" (-1- 3) "]"   =>  "{\sl " -1- "}"
)
```

Figure 3: Textual representation of EBE system state.

hypothesis. The first, the parenthesized expression beginning "(define-EBE-examples", defines the one-and-a-half example pairs that have been given so far in the EBE session. The first full set of examples consists of a pair of input and output strings separated by a "=>" and contained within a pair of parentheses. The second half-example consists of three strings and two "eol" symbols; it does not contain a "=>", and thus it has no output example.

The second T expression in the file defines the synthesized gap program using a notation that is similar to that used in Chapter 4. The only difference is the unfamiliar "(-1- 3)" contained in the gap pattern. This notation is used to specify the gap bound, which is the number of end-of-line boundaries that can be crossed in matching the pattern. In this case, "-1-" is the name of the gap, and the "3" specifies that the gap can skip over at most three end-of-lines in its search for a closing "]". An unparenthesized "-1-" would have been a shorthand for "(-1- 0)", a gap that is restricted to staying within a single line.

These two expressions make up a representation of the system's state which can be used for many purposes: The user can examine and modify the examples, perhaps with the aim of discovering and fixing a typographic error that is keeping the system from successfully synthesizing a program. A buggy example can be retracted simply by deleting it. The user can also modify the gap program, for example to change "`(-1- 3)`" to "`(-1- 10)`" to allow the gap to span ten lines rather than three. New gap programs or examples can be added from scratch. Gap programs and examples can be filed away for later use (perhaps for a bug report) by copying them to another file using standard editor commands.

When the user is finished viewing and modifying the state of the system, he commits his changes by giving the standard *close-window* command. The act of closing the window brings the modified textual representation of the state to the attention of the EBE system. The system incorporates the modifications by first clearing its internal databases and then reading and evaluating the text of the file as a set of executable T expressions. The observant Lisp-er will note that the arguments to both "`define-EBE-session`" and "`define-gap-program`" do not obey the standard syntax for evaluable forms; these two functions are both macros that provide special syntax. The side-effect of evaluating a "`define-EBE-session`" form is to add the examples in the form to the system's example database. The evaluation of the "`define-gap-program`" form has the side-effect of overriding the system's current gap program hypothesis with the defined gap program.

Our current implementation of the *modify-ebe-state* command is a cheap and efficacious hack, and there is certainly room for improving the interface that it provides. It is not very robust; the text is read and evaluated as T forms, and since the user is free to destroy the syntactic and semantic correctness of the forms, it is possible for him to end up chatting about his mistakes with the debugger for the underlying T system. Even if such drastic flaws can be corrected, such a system is also not set up to give timely context-dependent help to the user. For example, if the user incorrectly modifies the textual representation of the state, he won't find out about his mistake until he saves the text.

Chapter 6

EXTENSIONS

This chapter enumerates proposals for extensions to the editing by example system. To view the proposals in the proper setting, it is worthwhile to review the sequence of decisions that we made along the way to creating the EBE system that we described in the last five chapters. 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.

Although the gap program synthesis procedure and the user interface presented in Chapters 4 and 5 are critical to the implementation and use of the EBE system that we have described, their general form was determined by the decisions just enumerated, and their development did not require making other decisions of comparable importance.

The rest of this chapter will discuss ways in which some of these decisions could be made differently. The discussion is not intended to be exhaustive, but simply to indicate some reasonable avenues for expanding the capabilities of the system. These proposals have not been implemented within a working EBE system, although some of them were explored within an earlier test system. All of the proposals made are tempered by the desire that they be practical and implementable.

Four classes of extensions are proposed: extensions to gap patterns that expand their ability to find and parse text; extensions to gap replacement expressions that allow them to compute a wider range of useful functions; extensions that allow the system to take advantage of data other than input/output behavior; and user interface extensions that allow more powerful control structures to be specified.

6.1 Extending gap patterns

Gap patterns are a deterministic, sub-regular string matching notation. They consist of constants and gaps; the constants match only equal text, and the gaps match any text that does not contain the constants that follow them in the pattern. In this section we propose some extensions to the gap pattern notation and sketch how a program synthesis heuristic could produce programs that take advantage of the extensions. The extensions proposed are all conservative, in that we expect that an EBE system that incorporates them would still have practical behavior and performance.

Gap patterns may be extended by making the gaps of the pattern more selective about the text that they will match. We have already seen one example of such a restriction in the gap bounding heuristic that limits the number of end-of-line characters that a particular gap is allowed to match. This heuristic may be generalized by introducing the notion of a ***gap substitute grammar***.

A gap substitute grammar is a grammar associated with each gap in the pattern that defines the class of strings that may serve as substitutes for that gap. For example, the end-of-line restriction could be implemented by having a gap substitute grammar that generates all strings except for those that contain more than the maximum number of end-of-line characters, and except for those that contain the constant string that follows the gap. This particular class of gap substitute grammars happens to be expressible using regular expressions, and so gap patterns that are augmented with such gap substitute grammars are also sub-regular. In general, gap substitute grammars can define arbitrary classes of strings; however, since we do not know how to synthesize arbitrary grammars from examples, we will concentrate instead on considering various special case gap substitute grammars that play a useful role in text processing programs.

In the rest of this section we present five useful classes of gap substitute grammars: the first three classes are all sub-regular, the fourth is context-free, and the fifth is a context-sensitive extension.

6.1.1 Character class patterns

The grammars that correspond to the *character class* notation that is provided by many pattern matchers [23, 24, 50] form a useful class of sub-regular gap substitute grammars. Character classes are regular expressions that use a restricted form of alternation to define a pattern that matches a single character drawn from a particular set. A common notation used to express character classes is to enclose a list of the characters in the class within a pair of square brackets, e.g. "`[0123456789]`" defines a pattern that will match any decimal digit. Character subranges are usually specified using "`-`"; for example, "`[a-z]`" will match any lowercase alphabetic character and "`[a-zA-Z0-9]`" will match any alphanumeric character.

Gap patterns may be extended to use character classes by allowing a *character class gap* to take the place of any gap in the pattern. A character class gap is a gap that is restricted to matching runs of the characters that occur in the character class, and the matching process is defined so that character class gaps must not overlap with the string that follows them. Gap patterns can be viewed as a subclass of character class patterns in which the character classes used match any character in the alphabet. The notation we adopt for character class gaps is a conglomeration of the gap notation and the character class notation: for example, we denote an anonymous character class gap that matches all alphabetic characters by -[a-zA-Z]-; the gap would be given a distinguishing name, say the name *1*, using the notation -*1*:[a-zA-Z]-.

Character class programs are formed from character class gap patterns and the standard replacement expressions. A character class program that can change telephone numbers of the form "203 228 6750 " to "203-228-6750 " is

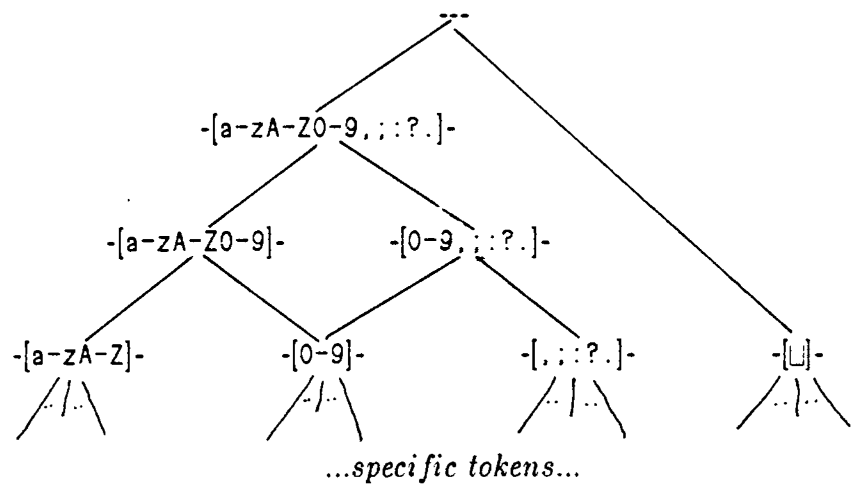
-*1*:[0-9]- ⊔ -*2*:[0-9]- ⊔ -*3*:[0-9]- ⊔ ⇒ -*1*- - -*2*- - -*3*- ⊔.

The virtue of this character class program is that the corresponding gap program

-*1*- ⊔ -*2*- ⊔ -*3*- ⊔ ⇒ -*1*- - -*2*- - -*3*- ⊔

will match any group of three non-blank strings separated by a blank. Although the gap program will correctly transform "203 228 6750 " to "203-228-6750 ", it will change "New Haven, CT " to "New-Haven,-CT " as well. The character class pattern imposes a tighter filter on the text transformed by the program.

The EBE system can synthesize gap programs that make use of the character class notation in much the same way that it performs the gap bounding heuristic: by synthesizing a standard gap pattern and transforming that pattern to a character class pattern through a post-processing step. For example, if all of the characters matched by a gap are digits, then the gap could be changed to -[0-9]-. On the other hand, if the two strings that match the gap are "foo" and "bar", then while the system could adopt -[abfor]- as its hypothesis for the text filling the gap, this is probably not the restriction that the user had in mind. The most useful character class that contains these characters would probably be found with reference to a fixed heterarchy of character class notations something like



The generalization best fitting the text occupying a gap would be the most restrictive class in the heterarchy that matches the text. In the case of "foo" and "bar", this would be -[a-zA-Z]-, and if a particular gap matched "375.23" and "3,264.44" in two input samples, then the system could reclassify that gap as -[0-9,;:?.]-. Such a character class program synthesis procedure is purely heuristic; it is not guaranteed to find a character class gap program if one exists, and the ones that it finds do not satisfy any interesting optimality conditions. Still, it seems to be a useful heuristic.

6.1.2 Generalizing tokenization

Character class patterns can be viewed as an extension of the tokenization heuristic presented in Section 5.11. The tokenization heuristic groups the characters of the samples into tokens in an *ad hoc* fashion; that is, it reads two samples

```
212 779 5061
415 494 4000
```

and decides to arbitrarily group them as the tokens

bol 212 ␣ 779 ␣ 5061 *eol*
bol 415 ␣ 494 ␣ 4000 *eol*

When analyzing these tokens, the pattern synthesis algorithm views two tokens as equal only if they are composed of the same basic characters, which allows it to come up with the descriptive gap pattern

bol --- ␣ --- ␣ --- *eol*

rather than the descriptive gap pattern

bol --- 1 --- ⊔ --- 9 --- ⊔ --- 0 --- *eol*

that would have been yielded by a character-at-a-time analysis.

The tokenization heuristic can be extended by introducing the idea of *generic tokens* that define whole classes of strings. For example, the generic token <*integer*> can be defined to match any integer; it would match the same strings as the character class gap -[0-9]-. Many other generic tokens would be useful: <*number*>, <*word*>, <*non-blank*>, <*whitespace*>, <*line*>, <*paragraph*>, <*time*>, <*date*>, <*phone-number*>, <*zip-code*>, <*dollars*>, <*human-name*>, <*user-id*>, <*filename*>, <*town-in-Connecticut*>. The generic tokens also form a heterarchy; for example, <*zip-code*> is a subclass of <*integer*>, which is a subclass of <*number*>.

The system can synthesize a generic token program in the same way that it creates character class programs: it first finds a standard gap pattern, and then applies a heuristic that transforms each gap to the most specific generic token in the token heterarchy that matches all of the text matched by the gap. An analysis of the phone number example above would yield the generic token program:

bol $<integer_1>$ ⊔ $<integer_2>$ ⊔ $<integer_3>$ ⊔ ⇒ -*1*- - -*2*- - -*3*- ⊔

6.1.3 Multi-token programs

It is easy for a program synthesis system to synthesize a program in which each gap is changed to a *single* character class expression or a token; all it does is apply a post-processing step that examines the text that fills each gap and changes that gap to the most specific character class or token in the heterarchy that can match the same text. The function of this kind of post-processing is to make the program's pattern be more selective about the text that it matches; the resulting patterns do not expand the transformational capabilities of their programs in any way.

However, these capabilities can be expanded if patterns that contain more than one character class or generic token between every pair of constants are permitted. For example, there is no gap pattern that will serve to isolate two words that are separated by a variable number of blanks, and so there is no gap program that can make the transformation

```
John   Ellis      ⇒     Ellis, John
Nat Mishkin       ⇒     Mishkin, Nat
Steve   Wood      ⇒     Wood, Steve
```

in one step. This transformation can be effected by a character class program:

<word$_1$> <whitespace$_2$> <word$_3$> ⇒ *-3-* ,␣ *-1-*

Note that there is no trailing constant in the pattern; the token *<word$_3$>* is defined to match as many alphabetic characters as it can, and thus no terminating constant is needed to make the text matched by the pattern be well defined.

A scheme for synthesizing such programs could work by analyzing the samples in the standard way, and then re-analyzing the text matched by the gaps at a different, more generic, level of tokenization. This reanalysis of the text within the gaps would continue until either a program that can transform the inputs to the outputs is found, or until the process runs out of different tokenizations to try. For example, the first phase of analysis of the examples yields the pseudo-gap pattern

-1- ␣ *-2-*

which is not a legal gap pattern because of the trailing gap. The text matching the first gap can be re-tokenized at a more generic level, which views all of the strings as *<word>*s. The pattern synthesis algorithm has no trouble finding the pattern *<word>* common to all of them. The text matching the second gap can be retokenized as

<whitespace> <word>
<word>
<whitespace> <word>

Which is analyzed to yield the gap pattern

--- *<word>*

The gap pattern matching the entire set of strings is now

<word> ␣ --- *<word>*

Which parses the input samples so that they may be formed into the output. A post-processing step with special knowledge of space characters could turn the "␣ ---" fragment into a single *<whitespace>* token.

One of the problems with this scheme is that it generates a large number of low-quality hypotheses. For example, the *<non-blank>* and *<whitespace>* tokens are ubiquitous, and an analysis of a gap might conclude that it is composed of at least 14

<*non-blank*>s and 13 occurrences of <*whitespace*>; not an interesting pattern. The problem is that there are just too many ways of fleshing out the internal structure of each gap, and there is not enough of a reason for preferring one way to another. The root of the problem is that the technique of synthesizing the pattern independent of the replacement breaks down as the pattern language becomes more expressive. Gap patterns that allow several generic tokens to fit into each gap are expressive enough that many different patterns can fit the same set of samples, and thus the chances of choosing one of the few that happens to lead to a replacement expression is small. It is probably best to consider synthesizing such gap programs using an algorithm that unifies the synthesis of the pattern and replacement expressions. We rejected this technique for gap programs because we showed that an algorithm that *always* promised to create a gap program if one existed would be too inefficient; for this same reason, we would probably do well to develop an algorithm that does not try to deliver on such promises.

Section 6.2.1 presents an extension to gap replacement expressions that results in a related extension to gap programs.

6.1.4 Context-free gap substitute grammars

Gap substitute grammars that are more powerful than regular expressions are useful when the user is manipulating text that contains embedded recursive structures, such as a Lisp program, or a Pascal program, or even just text with matched sets of parentheses. In this section we propose a scheme for synthesizing gap programs that manipulate such text.

We describe the needs of such a system using an example that manipulates a T program [71, 72]. In this example, all occurrences of the IF construct

```
(IF predicate consequent alternate)
```

are to be changed to COND:

```
(COND (predicate
        consequent)
       (T
        alternate))
```

This transformation cannot be performed by gap programs because each of the *predicate*, *consequent*, and *alternate* expressions may be arbitrary symbolic expressions (s-expressions) that may be parenthesized to arbitrary depth. A program to perform such a transformation must be able to match and skip over s-expressions, and a gap program that searches for the next ")" will fail miserably.

Given knowledge of T's syntax, or in other words, given knowledge of the fixed gap substitute grammar describing s-expression syntax, the gap pattern synthesis algorithm can be modified to compute a class of s-expression transformations that includes this one. The idea is that given the input samples

```
    (IF (<= X 1) 1 (* X (FACTORIAL (- X 1))))
    (IF (NULL? X) () (APPEND (REVERSE X) (CONS X NIL)))
```

and the corresponding output samples

```
    (COND ((<= X 1)
           1)
          (T
           (* X (FACTORIAL (- X 1))))
    (COND ((NULL? X)
           1)
          (T
           (APPEND (REVERSE X) (CONS X NIL))))
```

the s-expression program synthesis algorithm analyzes the parse tree representation level-by-level, starting with a zeroth level view of the input samples in which each of the s-expressions is viewed as being a single token.

```
    (IF⊔(<=⊔X⊔1)⊔1⊔(*⊔X⊔(FACTORIAL⊔(-⊔X⊔1))))
    (IF⊔(NULL?⊔X)⊔()⊔(APPEND⊔(REVERSE⊔X)⊔(CONS⊔X⊔NIL)))
```

This tokenization does not yield a gap pattern that can match both of the samples, so the samples are tokenized, or *holophrasted*, one level deeper

```
    ( : IF : (<=⊔X⊔1) : 1 : (*⊔X⊔(FACTORIAL⊔(-⊔X⊔1))) : )
    ( : IF : (NULL?⊔X) : () : (APPEND⊔(REVERSE⊔X)⊔(CONS⊔X⊔NIL)) : )
```

In these tokenized samples the symbol ":" is used to denote an s-expression boundary. There is a gap pattern common to these two samples, and it yields the gap program

```
    ( : IF : -1- : -2- : -3- : ) ⇒
    (COND (-1-
           -2-)
          (T
           -3-))
```

which can compute the transformation (although it cannot handle the change of indentation). If the procedure had not been able to find a program at this level of holophrasting, it would have tried one level deeper, and so on until the full depth of the examples had been plumbed.

This program is defined to search for an s-expression that begins with an IF and is followed by exactly three other s-expressions, when it finds such an expression, it replaces it with the desired COND. If this function were scanning text, then it would resume its search after the closing parenthesis of the IF; however, programs are recursively structured, and it would be more useful to have the default control structure conduct a depth-first search that visits the s-expressions contained in the gaps -*1*-, -*2*-, and -*3*- before continuing on to search the rest of the program.

This analysis would be most easily performed if the editor had already gone to the trouble of maintaining the code in parse-tree form. On the other hand, there are many other transformations in which the view of the code as a string of ASCII text would be more natural. The engineering tradeoffs involved in the design of program-oriented editors are not yet clear [98].

The s-expression gap programs that we have just described are not very powerful, and do not serve to describe the full range of manipulations that might usefully be performed on a parse tree. Gap programs were designed for manipulations carried out on the "flat" structures common in text, and thus the s-expression gap programs suffice only for expressing "flat" s-expression manipulations. A serious attempt at a program-oriented EBE system would probably do well to expand on the techniques used by the list processing program synthesis systems mentioned in Chapter 2.

6.1.5 Testing for equality among fields

As soon as we assigned names to the gaps in a pattern, there was probably an inclination on the reader's part to write patterns that used the same name more than once, with the intent of requiring that each of the instances of the gap so named match the same text. Adding an equality test to gap programs is certainly an interesting extension from the point of view of defining a vehicle for computation. On the other hand, this extension would probably not be widely used in text processing programs; about the only use that comes to mind is in computing the join of some textually represented database relations.

Angluin [4] studied a similar problem for a slightly different class of patterns, and found that finding a maximal length pattern that makes use of equality tests is NP-hard. The obvious heuristic for creating gap patterns that use an equality test is to examine the pattern for all pairs of gaps that match equal text on the input samples, and to give all such pairs the same name. This heuristic is probably adequate for the limited use that we would make of this construct.

6.2 Replacement expression extensions

The role of the replacement expression of a gap program is to produce a replacement for the text that matches the program's pattern. The replacement expressions that we have considered so far produce simple concatenations of constant strings together with the text matching the gaps of the program's gap pattern. In this section, we propose four other classes of replacement expressions that do not stray far from this framework. The first class is an extension that allows replacement expressions to insert fixed substrings of the text matching the gaps of the gap pattern; the second two allow the replacement expressions to compute special-case functions on the text matching the gaps, and the fourth adds a facility for formatting the output text in columns.

6.2.1 Substrings of gaps

The administrators of one of our favorite timesharing systems form user-ids by concatenating the first three letters of the user's last name with the first three letters of their first name and their middle initial.

```
ELLIS, JOHN R.       ⇒     ELLJOHR
MUSHLEUT, MAT W.     ⇒     MUSMATW
WOOD, STEVEN R.      ⇒     WOOSTER
```

This transformation cannot be performed by a gap program because there is no gap pattern that will isolate ELL, MUS, and WOO from the characters that follow them.

An extension that makes this transformation expressible is to allow gap replacement expressions to select the first k characters of the text matched by a gap. For example, the notation -*1*(1..3)- defines a *substring selector*, or *prefix expression*, that when used in a replacement expression will insert the first three characters of the text that matches the first gap. Using this notation, the program

-1- .⊔ *-2-* ⊔ *-3-* .   ⇒   *-1*(1..3)- *-2*(1..3)- *-3-*

would suffice to create the user-ids from the list of names.

This extension turns out to be easy to implement. The first phase of the gap replacement synthesis algorithm (Algorithm 11) computes a representation of all possible ways that the text of a gap in a particular input can occur in the corresponding output; the second phase intersects these representations to find those replacement expressions that work simultaneously for all of the sample pairs. To find prefix expressions, the first phase is modified so that it instead computes a representation of all possible ways that every prefix of the text of a gap can occur in the output; the second phase remains the same. If each of the n output samples has l symbols, and there are $|g(G)|$ gaps in the gap pattern that matches the input, then the algorithm has a worst case running time bounded by $O(|g(G)|^n l^{2n})$. In practice, the actual running time would be much better, for the same reason that Algorithm 11 performs well in practice.

The prefix-extracting algorithm can be modified to extract suffixes as well, with the same order-of-magnitude cost in running time. Arbitrary fixed-range substrings may be found at a greater cost bounded by $O(|g(G)|^n l^{3n})$.

6.2.2 Fixed functions of gaps

Users of text editors often apply functions to their text that transform it in a way that does not involve copying. Most of these functions are completely domain specific, and a subclass of those will be considered in the next section. However, there is a small set of functions that are commonly applied to text that are not domain specific, or rather are specific to the domain of text. These functions involve manipulations on text that change its attributes: capitalization, conversion to lower case, capitalizing the first letter of each word, and other changes of formatting style.

For example, a certain programmer has many fetishes about the format of his T code; one particular fetish is apparent in the form of each function definition

```
(define (choose n m)
    ...)
```

which *must* begin with a peculiar style of comment

```
;*** (CHOOSE N M)
;*** ==========================================================
;*** Computes the number of ways of choosing M things from N.
;***
(define (choose n m)
    ...)
```

Automatically generating the comments themselves is beyond the scope of this thesis, but the synthesis of the comment template, which consists of the comments beginning "`;***`", the capitalized function call form, and the line of "`===...`", does lie within the framework. The only aspect of generating this template that cannot be handled by the standard replacement expressions is producing the text "`(CHOOSE N M)`" from the text "`(choose n m)`". If a capitalization function were in the gap program's repetoire, then the following program could perform the transformation:

```
bol (define ( -1- ) eol ⇒
;***␣( capitalize(-1-) ) eol
;***␣========================================================== eol
;***␣eol
(define␣( -1- ) eol
```

It is easy to extend the gap program synthesis procedure to accomodate a limited set of such transformation functions as *capitalize*, *lowercase*, or *capitalize-first-letter*. To implement this extension, the replacement synthesis algorithm is modified so that the first phase of the algorithm finds all of those places in each output sample that are equal to the image of one of the transformation functions applied to the text of one of the gaps in the corresponding input sample. If all of the sample pairs account for the text in this same way, then that will be the transformation performed in the synthesized program.

6.2.3 Tabularly specified functions

Many functions on text are completely idiosyncratic; for example, in the transformation

```
5/2/58     ⇒    May 2, 1958
8/19/59    ⇒    August 19, 1959
6/14/80    ⇒    June 14, 1980
```

the month name produced from 5 is `May`, from 8 is `August`, and so on. Any algorithm that generalizes this transformation must simply know, in some way, that the fifth month is May. Whenever it sees 5 in the input, and `May` magically appears in the output, then it should know that the user is making use of the *month-number-to-name* function. We call such functions ***tabularly specified functions.***

If the system already knows of the table representing a particular tabularly specified function, then it can synthesize such functions in the same way as in the previous section. That is, whenever some gap matches text that is a valid input, or index, for one of the tables, then the system scans the corresponding output sample for the occurence of the table entry corresponding to that text. If the output of the function appears somewhere, then it is tagged as possibly being a result of the tabular function performed on the input gap; if all of the input/output samples account for the text in this same way, then that is the transformation being performed.

If the table is not known to the system, then the user could specify its contents by giving examples. Such a user interface would be basically the same as for the editing by example system, with the user selecting input examples and the corresponding outputs. It would differ in that the system would *not* try to generalize the samples, but would simply record them for later use.

6.2.4 Layout functions

Another important class of commonly applied functions are those that perform manual text formatting, particularly arranging data in columns. For example, the ragged list

```
Nat Mishkin, Judy Mishkin, 777-5562
John Levine, Lydia Spitzer, 864-9650
John Ellis, Ann Ambassador, 865-6438
```

might be easier to read if it were aligned in columns

```
Nat Mishkin         777-5562
Judy Mishkin        777-5562
John Levine         864-9650
Lydia Spitzer       864-9650
John Ellis          865-6438
Ann Ambassador      865-6438
```

Such tabular layout functions are not expressible as gap programs because the amount of whitespace between the entries in each row varies among the rows. For example, assuming that the layout is produced by inserting a certain number of spaces, there are 9 spaces between `Nat Mishkin` and his phone number, but only 7 spaces between

Lydia Spitzer and hers. The transformation could be performed by a program that makes use of an absolute column positioning function called *tab*:

> *-1-* ,␣ *-2-* ,␣ *-3-* *eol* ⇒
> *-1-* *tab*(21) *-3-* *eol*
> *-2-* *tab*(21) *-3-* *eol*

Synthesizing such functions could proceed as follows. First, the standard gap replacement synthesis algorithms are applied, but to an output in which all sequences of blanks have been replaced by a generic <*whitespace*>, or <*w*>, token. In this case the input and output above would be tokenized as:

```
Nat <w> Mishkin , <w> Judy <w> Mishkin , <w> 777 - 5562 eol
John <w> Levine , <w> Lydia <w> Spitzer , <w> 864 - 9650 eol
John <w> Ellis , <w> Ann <w> Ambassador , <w> 865 - 6438 eol
```

and

```
Nat <w> Mishkin <w> 777 - 5562 eol
Judy <w> Mishkin <w> 777 - 5562 eol
John <w> Levine <w> 864 - 9650 eol
Lydia <w> Spitzer <w> 864 - 9650 eol
John <w> Ellis <w> 865 - 6438 eol
Ann <w> Ambassador <w> 865 - 6438 eol
```

A program is then found to transform this "unaligned" text:

> *-1-* <*w*> *-2-* , <*w*> *-3-* <*w*> *-4-* , <*w*> *-5-* - *-6-* *eol* ⇒
> *-1-* <*w*> *-2-* <*w*> *-5-* - *-6-* *eol*
> *-3-* <*w*> *-4-* <*w*> *-5-* - *-6-* *eol*

In the next step, the system replaces all of those <*w*> tokens that contain the same number of space characters in all samples with that particular number of spaces

> *-1-* ␣ *-2-* ,␣ *-3-* ␣ *-4-* ,␣ *-5-* - *-6-* *eol* ⇒
> *-1-* ␣ *-2-* <*w*> *-5-* - *-6-* *eol*
> *-3-* ␣ *-4-* <*w*> *-5-* - *-6-* *eol*

The system then examines the numerical relationships that hold among the columnar positions of the text following the remaining <*w*> tokens. In this case, it finds that the text following both <*w*> tokens in the output string starts in column 21 in all output samples, yielding the following gap program:

> *-1-* ␣ *-2-* , ␣ *-3-* ␣ *-4-* , ␣ *-5-* - *-6-* *eol* ⇒
> *-1-* ␣ *-2-* *tab*(21) *-5-* - *-6-* *eol*
> *-3-* ␣ *-4-* *tab*(21) *-5-* - *-6-* *eol*

Which after pattern reduction becomes

$$\begin{array}{l} \textit{-1- } ,\sqcup \textit{ -2- } ,\sqcup \textit{ -3- eol} \quad \Rightarrow \\ \textit{-1- } \textit{tab}(21) \textit{ -3- eol} \\ \textit{-2- } \textit{tab}(21) \textit{ -3- eol} \end{array}$$

6.3 Extensions to the data collected

The systems that we have proposed and developed in this thesis base their hypotheses on positive examples of the input/output behavior of the target function. There are many other sources of information that would be naturally available in the system, and in this section we consider schemes for taking advantage of traces, negative data, queries, assertions, and other sources of information.

6.3.1 Traces

In Section 3.2.1 we argued that traces, which are the record of the commands that the user employed while finding and transforming his example text, were too unreliable a source of information about the user's target program to serve as the principal basis for generalization in the EBE system. We also pointed out that a system that used traces would have to have some knowledge of the semantics of each editor command if it wanted to generalize the trace. It was for these reasons that we rejected traces in favor of making use of the sample input/output behavior.

On the other hand, traces are an effectively useable source of information in those situations in which the trace is simple, but the input/output behavior is not. For example, suppose that the user wants to fill and justify the ragged margins of every paragraph in a long document. His editor has a command that will perform this operation on one paragraph, and he would like to apply the command to all of the paragraphs in the document. This function's input/output behavior is inscrutable, and trying to synthesize such a function from input/output behavior is out of the question.

However, it is not too difficult to envision a hybrid EBE system that uses the appearance of a few input samples to synthesize a pattern that can locate the next piece of text to transform, but that replays the user's commands to actually transform the text that it finds. For example, if each paragraph is defined to be a sequence of lines ending in a blank line, the hybrid program *bol -1- eol eol* $\Rightarrow$ *fill-and-justify(bol -1- eol eol)*

might be the result of the system's analysis.

It is not necessary that such a hybrid system generalize the sequence of commands; the system would still be useful if it simply replayed the literal sequence of commands given by the user in transforming one instance. If the user discovers a new class of transformations that he would like to perform by example, but cannot because they are not expressible with a fixed sequence of commands, then he might be able to extend the system's capabilities by adding the necessary command or commands to his editor. For example, if the user finds himself always wanting to sort some text, he would do well to add a sort function to his editor.

6.3.2 Negative data

Gold's work showed that inductive inference systems that work from positive and negative data are more powerful than systems that restrict their attention to positive data [32].

There are two relevant kinds of negative data available for gap programs: negative data that shows shortcomings in the pattern, and negative data that shows that the transformation has been incorrectly done. The negative data that shows shortcomings in the pattern can be further divided into two classes: text that the pattern should have matched, but did not, and text that the pattern should not have matched, but did. The negative data imposes constraints on the programs synthesized by the system by prohibiting the synthesized programs from exhibiting behavior that has been disallowed.

The exact gap pattern synthesis algorithm developed for Theorem 17, the one that runs in $O(l^{3n})$ time, can be adapted so that it finds a gap pattern that will match the strings in a positive sample set without matching any of the strings in a negative sample set. That algorithm works by first constructing representations of all possible gap patterns that match each sample, and then intersecting these representations together to find all of the gap patterns that match all samples simultaneously. The algorithm can be extended to handle negative samples by having it first construct a representation of all possible gap patterns that match each negative sample, and then intersect the complement of each of these sets with the intersection of the positive samples. This construction would yield a representation of all possible gap patterns that can match all of the positive samples without matching any of the negative samples. The most descriptive such pattern can be recovered by applying the technique described in Theorem 18.

This algorithm is not practical. It runs in $O(l^{3n+1} \log l)$ time, and it is not clear how to formulate a heuristic that makes it practical. The descriptive gap pattern synthesis heuristic already does pretty well for negative data, in that it tries to find the pattern that maximally differentiates the strings in the positive sample set from all other strings. But if the descriptive gap pattern matches a negative sample, then it is unlikely that the pattern could be patched to not match the sample. The reason for this is that a descriptive pattern matches a minimal set containing the positive samples, and all changes to the pattern will probably either make the set of strings that it matches larger, or will result in it no longer matching one of the positive samples that it should. However, negative data would be useful in restricting the application of the pattern reduction heuristic, which tends to turn descriptive gap patterns into less-than-descriptive gap patterns.

The problem with extending the system to handle negative data is that gap patterns are not powerful enough to draw subtle distinctions between positive and negative samples. The control structure extensions presented in Section 6.4 offer a more appropriate mechanism for keeping the synthesized programs from matching a particular class of samples.

6.3.3 Queries, assertions, and other user interactions

The EBE system could ask the user questions about the program that it is trying to synthesize, perhaps as Shapiro's Algorithmic Debugging System queries the user about the validity of steps in the computation of the target function [76, 79]. The problem is to find the right things to ask about; a system that asks the user about the validity of internal steps of its computation would probably not be well received. We have not yet found an algorithmic use for queries in the synthesis of text processing programs.

Another source of information could be the user's assertions about the kind of editing program that he would like the EBE system to synthesize. For example, the user might be manipulating a Lisp program, and he could tell the system that his manipulations are in terms of the Lisp program's parse tree. Such an assertion could allow the system to narrow in on a particular class of programs that have special knowledge of the syntactic structure of Lisp. The user could also provide more specific information about his target program, perhaps by providing the system with code fragments that could be incorporated into the program.

6.4 Control structure extensions

One of the obvious flaws of gap programs is that they do not provide general-purpose control structures. There are two reasons for this oversight: control structure generalization from examples is very hard, and the rigid set of control structures provided by the EBE system user interface work well enough in practice that solving this hard problem was not a requirement for building a useful EBE system. However, expanding the system's repetoire of control structures is a requirement for any real expansion in its capabilities.

One approach to providing a full range of control structures is to provide the EBE system user with an entry into a full-fledged programming language. For example, the user might want to write something like the following program to apply two different search-and-transform rules to some text:

```
(define (apply-transformation context)
    (cond ((end-of-context? context)
           ())
          ((matches? pattern_1 context)
           (transform_1 context)
           (apply-transformation (rest-of context)))
          ((matches? pattern_2 context)
           (transform_2 context)
           (apply-transformation (rest-of context)))
          (t
           (apply-transformation (rest-of context)))))
```

The program sketched searches a context for parts of the context that match one of two patterns, $pattern_1$ or $pattern_2$, and applies either $transform_1$ or $transform_2$ to the parts of the context that match one of the patterns.

We have not developed facilities that can automatically synthesize such complicated control structures from examples, and judging by the experience gained from research on program synthesis, it is unlikely that we would be able to come up with a system that would be able to reliably synthesize a wide range of control structures from example information. However, while the user could write this entire program manually, we have developed facilities that can automatically synthesize the components $pattern_1$, $transform_1$, $pattern_2$, and $transform_2$ from examples. To take advantage of these facilities, we could provide an interface that allows the user to manually create the control structures of the program, but provides him with facilities for writing the pattern matching and transformation components through examples.

Even the control structures would not have to specified in a completely manual fashion. For example, certain various special case control structures schemas, such as the recursive conditional expression in the example above, could be available through menus. The user could select one of these program schemas, and then fill in the details of the patterns and transformations by giving examples for each.

The system just sketched bears a resemblance to Lieberman and Hewitt's Tinker system [57]. Perhaps the difference in domain, text processing vs. list processing, will be enough to make the text processing version of such a system be useful.

6.5 Future work

In this chapter we have sketched extensions to the EBE system that increase its power. We intend these extensions to be incorporated into a user interface to writing text processing programs that allows some program fragments to arise as the result of the system's analysis of examples, some to come from transcripts of the actions carried out by the user, and others to be written manually or specified through menus. The realization of such a user interface is a subject for further research.

Chapter 7

CONCLUSION

This dissertation presented the design of a system for specifying text processing programs by example. As a first stab at building the system, we decided to concentrate on synthesizing a class of text processing programs called *gap programs*, and we decided to base the synthesis on examples of the program's input/output behavior. Guided by results characterizing the computational complexity of various aspects of gap program synthesis, we developed an efficient heuristic procedure for synthesizing gap programs from examples. We showed that this procedure, even though it used heuristics, was still guaranteed to find gap programs in the limit from positive data.

We went on to evaluate how well the gap program synthesis heuristic performed on the text encountered in practice. This evaluation led to the development of several heuristics that act both to improve the quality of the hypotheses proposed by the system and to reduce the number of examples required to converge to a target program. The result is a gap program synthesis heuristic that can usually synthesize a target gap program from two or three input examples and a single output example. The heuristic has been implemented within a working text editor as the core of an editing by example system.

The primary contribution of this dissertation 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; as a result, this thesis presents one of the few instances of a programming by example system that performs interesting generalizations of its examples.

Another contribution of this dissertation lies in providing an application for the techniques of inductive inference, an area of research 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 towards 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 useable until this project was nearly complete. 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. While the extensions proposed in Chapter 6 increase the capabilities of the system, it is clear that 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.

Appendix I. Gap program synthesis code

This section presents the code for the system described in Chapters 4 and 5, in hopes that it will serve as a starting point for the implementation of editing by example systems in other text editors. Only the code for the gap program synthesis heuristic is reproduced here, the code that implements the user interface and the gap program interpreter has not been included. The system is written in T, a dialect of Lisp; the interested reader should consult the T manual for definitions of the programming constructs used here [71, 72].

This part of the system is organized into 6 parts:

1. Routines for approximating the LCS of a set of samples.
2. Routines for inserting gaps into the LCS to make a gap pattern.
3. Utilities for replacement expression synthesis.
4. A replacement expression synthesis algorithm.
5. The gap bounding heuristic.
6. The pattern reduction heuristic.

The various structures manipulated by the gap program synthesis code have the following representations:

Tokens:

Tokens are atomic items; in U, tokens are either characters, strings, or numbers. For efficiency, a pair of tokens are treated as being equal if and only if they answer true to T's pointer equality test (`eq?`). A tokenization package, not reproduced here, makes sure that all equal tokens are referenced by equal pointers.

Gaps:

Individual gaps, like *-1-*, are represented by data structures that contain room for recording gap bounds as well as starting and stopping positions for gap matching. The accessors for these gap structures are the procedures beginning `EBE-Gap-`.

Samples:

Each input or output sample is represented as a vector of tokens. Groups of input/output samples are represented as a pair of parallel lists of input samples and of output samples.

Gap patterns:

A gap pattern is represented by a pair of vectors. One vector, usually C or CS, is a vector of tokens that gives the constants of the pattern. The other vector, usually G, is of the same length as C and has a gap in slot *i* if there is a gap preceding the *i*'th constant in C, and () in all of the other slots of G.

Replacement expressions:

A replacement expression is a list of tokens and gaps.

Replacement automata:

A replacement automaton is a finite state machine that is the working representation of a replacement expression. These finite state machines are acyclic. They are represented by a vector of lists, where each list is an A-list in which each member is a pair whose car is a token and whose cdr is a state to proceed to after producing that token. The zeroth entry in the vector is the start state, and the last is the sole accepting state. For example, the vector

```
#(((-2- . 1) (-1- .  2)) ((c . 2)) ((d . 3)) ())
```

defines an automaton that accepts the two strings "-*2*- cd" and "-*1*- d".

The system described in this paper should be fairly easy to port to other editors. I would be very pleased to hear reports of experience with EBE system implementation and use.

```
;*** ======================================================================
;*** ======================================================================
;*** An algorithm for approximating the longest common subsequence of a list
;*** of samples.
;*** ======================================================================
;*** ======================================================================


;*** (LCS SS)
;*** ======================================================================
;*** LCS computes an approximation to the longest common subsequence of the
;*** vectors in the list SS.  The first step of the approximation is to sort the
;*** members of SS in order of increasing length.  Then, the exact LCS of the
;*** first two (shortest two) vectors is computed, and then the exact LCS of
;*** that with the third vector is computed, and then the LCS of that with the
;*** fourth, ...  and so on.
;***
(define (lcs ss)
    (let ((slen (length ss)))
      (cond ((fx= slen 0)
             nil)
            ((fx= slen 1)
             (copy-vector (car ss)))
            (t
             ;*** Sort in increasing order of length.
             (set ss
                  (sort ss (lambda (x y) (fx< (vector-length x) (vector-length y)))))
             (loop (initial (l1 (make-vector (vector-length (car ss))))
                            (l2 (copy-vector l1))
                            (scr (copy-vector l1))
                            (xlcs (list->vector
                                        (lcs-2 (cadr ss) (car ss) l1 l2 scr))))
                   (for x in (cddr ss))
                   (do (or (common-subsequence? xlcs x)
                           (set xlcs (list->vector (lcs-2 x xlcs l1 l2 scr)))))
                   (result xlcs))))))


;*** (LCS-2 A B L1 L2 SCR)
;*** ======================================================================
;*** Sets up Hirschberg's linear space algorithm for computing the longest
;*** common subsequence of a pair of strings.  A and B are the two vectors for
;*** which you're computing the LCS.  L1, L2, and SCR are scratch vectors of
;*** length at least |A|.  This returns a list giving the exact longest common
;*** subsequence of A and B.
;***
(define (lcs-2 a b l1 l2 scr)
      (lcs-c a 0 (fx- (vector-length a) 1) b 0 (fx- (vector-length b) 1)
             l1 l2 scr))


;*** (LCS-C A ML MR B NL NR L1 L2 SCR)
;*** ======================================================================
;*** LCS-C, LCS-B, and LCS-B-REV implement Hirschberg's linear space algorithm
;*** for computing the longest common subsequence of a pair of vectors.
```

```
;***
;*** The running time of this implementation is:
;***   100 micro-seconds * n * m, where n and m are the lengths of the vectors.
;***   This timing applies on a 10MHz Apollo DN400 MC68000, with cache.
;***   Compiled under an early version of the T compiler.
;***
(define (lcs-c a ml mr b nl nr l1 l2 scr)
    (cond ((or (fx> nl nr) (fx> ml mr))
           nil)
          ((fx= ml mr)
           (loop (initial (the-a (vref a ml)))
                 (step j from nl to nr)
                 (until (eq? the-a (vref b j)))
                 (result
                    (if (fx<= j nr) (cons (vref b j) nil) nil))))
          ((eq? (vref a ml) (vref b nl))
           (loop (initial (l (cons (vref a ml) nil)))
                 (step i from (fx+ ml 1) to mr)
                 (step j from (fx+ nl 1) to nr)
                 (while (eq? (vref a i) (vref b j)))
                 (do (set l (cons (vref a i) l)))
                 (result
                    (append! (reverse! l)
                             (lcs-c a i mr b j nr l1 l2 scr)))))
          ((eq? (vref a mr) (vref b nr))
           (loop (initial (l (cons (vref a mr) nil)))
                 (decr i from (fx- mr 1) to ml)
                 (decr j from (fx- nr 1) to nl)
                 (while (eq? (vref a i) (vref b j)))
                 (do (set l (cons (vref a i) l)))
                 (result
                    (append! (lcs-c a ml i b nl j l1 l2 scr)
                             l))))
          (t
           (let ((i (fx+ ml (fx- (fx\/ (fx+ (fx- mr ml) 1) 2) 1))))
             (lcs-b a ml i b nl nr l1 l1 scr)
             (lcs-b-rev a (fx+ i 1) mr b nl nr l2 l2 scr)
             (let ((m (vref l2 nl))
                   (m-index (fx- nl 1)))
               (loop (step j \.in nl to nr)
                     (do (let ((tm (fx+ (vref l1 j) (vref l2 (fx+ j 1)))))
                           (cond ((fx> tm m)
                                  (set m tm)
                                  (set m-index j))))))
               (if (fx> (vref l1 nr) m)
                   (set m-index nr))
               (append!
                  (lcs-c a ml i b nl m-index l1 l2 scr)
                  (lcs-c a (fx+ i 1) mr b (fx+ m-index 1) nr l1 l2 scr)))))))


;*** (LCS-B A ML MR B NL NR LL K0 K1)
;*** ================================================================
;*** Implements procedure B of Hirschberg's linear space LCS algorithm.
```

```
;***
(define (lcs-b a ml mr b nl nr ll k0 k1)
    (loop (step j from nl to nr)
          (do (vset k1 j 0)))
    (loop (initial (the-a nil))
          (step i from ml to mr)
          (do (set the-a (vref a i))
              (let ((tk k0))
                (set k0 k1)
                (set k1 tk))
              (if (eq? the-a (vref b nl))
                  (vset k1 nl 1)
                  (vset k1 nl (vref k0 nl)))
              (loop (step j from (fx+ nl 1) to nr)
                    (do (if (eq? the-a (vref b j))
                            (vset k1 j (fx+ (vref k0 (fx- j 1)) 1))
                            (let ((x1 (vref k1 (fx- j 1)))
                                  (x2 (vref k0 j)))
                              (if (fx< x1 x2)
                                  (vset k1 j x2)
                                  (vset k1 j x1))))))))
    (or (eq? ll k1)
        (loop (step j from nl to nr)
              (do (vset ll j (vref k1 j))))))


;*** (LCS-B-REV A ML MR B NL NR LL K0 K1)
;*** ====================================================================
;*** Implements procedure B of Hirschberg's linear space LCS algorithm,
;*** only does it for reversed strings.
;***
(define (lcs-b-rev a ml mr b nl nr ll k0 k1)
    (loop (decr j from nr to nl)
          (do (vset k1 j 0)))
    (loop (initial (the-a nil))
          (decr i from mr to ml)
          (do (set the-a (vref a i))
              (let ((tk k0))
                (set k0 k1)
                (set k1 tk))
              (if (eq? the-a (vref b nr))
                  (vset k1 nr 1)
                  (vset k1 nr (vref k0 nr)))
              (loop (decr j from (fx- nr 1) to nl)
                    (do (if (eq? the-a (vref b j))
                            (vset k1 j (fx+ (vref k0 (fx+ j 1)) 1))
                            (let ((x1 (vref k1 (fx+ j 1)))
                                  (x2 (vref k0 j)))
                              (if (fx< x1 x2)
                                  (vset k1 j x2)
                                  (vset k1 j x1))))))))
    (or (eq? ll k1)
        (loop (decr j from nr to nl)
              (do (vset ll j (vref k1 j))))))
```

```
;*** =====================================================================
;*** =====================================================================
;*** Routines for inserting gaps into a common subsequence to make a gap pattern.
;*** =====================================================================
;*** =====================================================================


;*** (COMMON-SUBSEQUENCE?  C S)
;*** =====================================================================
;*** Returns T if the vector C occurs as a subsequence of the vector S.
;***
(define (common-subsequence? c s)
    (loop (initial (j 0) (endj (vector-length c)))
          (step i \.in 0 to (vector-length s))
          (until (fx= j endj))
          (do (cond ((eq? (vref c j) (vref s i))
                     (set j (fx+ j 1)))))
          (result (fx= j endj))))


;*** (GAP-CONSTANTS-MATCH? SAMPLE S-FROM C C-FROM C-TO)
;*** =====================================================================
;*** Returns T if the vector slice starting at S-FROM in SAMPLE
;*** is equal to the constants C[C-FROM..C-TO).
;***
(define (gap-constants-match? sample s-from c c-from c-to)
    (loop (step i \.in s-from to (vector-length sample))
          (step j \.in c-from to c-to)
          (while (eq? (vref sample i) (vref c j)))
          (result (fx= j c-to))))


;*** (FIND-END-OF-GAP G C-FROM)
;*** =====================================================================
;*** Returns the index of the next gap in G at or after C-FROM.
;***
(define (find-end-of-gap g c-from)
    (loop (step i \.in c-from to (vector-length g))
          (until (vref g i))
          (result i)))


;*** (GAP-MATCHES? SAMPLE S-FROM C G C-FROM)
;*** =====================================================================
;*** Returns T iff the suffix of the gap pattern represented by C and G
;*** starting at C-FROM matches the vector SAMPLE starting at S-FROM.
;***
(define (gap-matches? sample s-from c g c-from)
(and (fx< c-from (vector-length g))
     (let ((eofg (find-end-of-gap g c-from)))
       (cond ((and (not (vref g c-from))
                   (not (gap-constants-match? sample s-from c c-from eofg)))
              nil)
             (t
            (set s-from (fx+ s-from (fx- eofg c-from)))
            (loop (initial (ci eofg)
                           (ce ci)
                           (clen (vector-length c)))
                  (while (fx< ci clen))
                  (do (set ce (find-end-of-gap g (fx+ ci 1)))
                      (loop (initial (first-c (vref c ci)))
                            (step j \.in s-from to (vector-length sample))
                            (until (and (eq? first-c (vref sample j))
                                        (gap-constants-match?
                                             sample (fx+ j 1) c (fx+ ci 1) ce)))
                            (result (set s-from j))))
                  (while (fx< s-from (vector-length sample)))
                  (do (set s-from (fx+ s-from (fx- ce ci)))
                      (set ci ce))
                  (result (and (fx>= s-from (vector-length sample))
                               (fx>= ci (vector-length c)))))))))


;*** (LEFTMOST-GAP-INSERTION SAMPLE C G)
;*** ====================================================================
;*** Implements the leftmost match gap insertion heuristic.
;***
(define (leftmost-gap-insertion sample c g)
    (loop (initial (j 0)
                   (k 0)
                   (endgaps (vector-length g))
                   (endsample (vector-length sample)))
          (while (fx< k endsample))
          (while (fx< j endgaps))
          (until (gap-matches? sample k c g j))
          (do (cond ((eq? (vref c j) (vref sample k))
                     (set j (fx+ j 1))
                     (set k (fx+ k 1)))
                    (t
                     (vset g j (Make-EBE-Gap))
                     (set k (fx+ k 1)))))
          (result (and (fx< k endsample) (fx< j endgaps)))))



;*** (INSERT-GAPS-IN-COMMON-SUBSEQUENCE SAMPLES C)
;*** ====================================================================
;*** The driver for the leftmost match gap insertion heuristic.
;***
(define (insert-gaps-in-common-subsequence samples c)
    (let ((g (vector-fill (make-vector (vector-length c)) nil)))
      (loop (step i \.in 0 to (length samples))
            (for x in samples)
            (while (leftmost-gap-insertion x c g))
            (result (if (fx= i (length samples)) g nil)))))


;*** (GAP-MATCHES-ALL? SAMPLES C G)
```

```
;*** =====================================================================
;*** Returns T if the gap pattern represented by C and G matches all of
;*** the samples in the list SAMPLES.
;***
(define (gap-matches-all? samples c g)
    (loop (initial (failed? nil))
          (for x in samples)
          (do (set failed? (not (gap-matches? x 0 c g 0))))
          (until failed?)
          (result (not failed?))))


;*** (REDUCE-GAPS-IN-PATTERN SAMPLES C G)
;*** =====================================================================
;*** Deletes gaps from the pattern represetned by C and G as long as the
;*** samples still match.
;***
(define (reduce-gaps-in-pattern samples c g)
    (loop (decr i from (fx- (vector-length g) 1) to 0)
          (do (cond ((vref g i)
                     (let ((gp (vref g i)))
                       (vset g i nil)
                       (cond ((not (gap-matches-all? samples c g))
                              (loop (initial (lj i))
                                    (decr j from (fx- i 1) to 0)
                                    (until (vref g j))
                                    (do (vset g j t)
                                        (if (gap-matches-all? samples c g)
                                            (set lj j))
                                        (vset g j nil))
                                    (result (vset g lj gp))))))))))


;*** (WORKING-C-G->GAP-PATTERN C G)
;*** =====================================================================
;*** A representation conversion utility that converts the gap pattern
;*** represented by the vectors C and G to a list in which the gaps and
;*** constants are interspersed.
;***
(define (working-c-g->gap-pattern c g)
    (loop (initial (l nil))
          (step i \.in 0 to (vector-length c))
          (do (if (vref g i)
                  (set l (cons (vref g i) l)))
              (set l (cons (vref c i) l)))
          (result (reverse! l))))
```

```
;*** ======================================================================
;*** ======================================================================
;*** Utilities for the gap replacement synthesis algorithm.
;*** ======================================================================
;*** ======================================================================

;*** (GAP-SKIP-TO-VECTOR V FROM END FIND FFROM FEND)
;*** ======================================================================
;*** Locates the first occurrence of FIND[FFROM..FEND) in V[FROM..END).
;*** Returns the index if it finds that slice of FIND, otherwise returns ().
;***
(define (gap-skip-to-vector v from end find ffrom fend)
    (if (fx= (fx- fend ffrom) 0)
        (and (fx< from end) from)
        (loop (initial (found? nil)
                       (first (vref find ffrom)))
              (step i \.in from to end)
              (do (if (eq? (vref v i) first)
                      (loop (step j \.in (fx+ ffrom 1) to fend)
                            (step k \.in (fx+ i 1) to end)
                            (while (eq? (vref find j) (vref v k)))
                            (after (if (fx= j fend)
                                       (set found? t))))))
              (until found?)
              (result (and found? i)))))


;*** (PARSE-INPUT-INTO-GAPS CS G V)
;*** ======================================================================
;*** Parses the input represented in vector V using the gap pattern represented
;*** by CS and G.  Side-effects the EBE-Gap-Start and EBE-Gap-End slots of the
;*** non-null entries of G with the bounds where the gap pattern matched V.
;*** Returns non-() if CS and G match V.
;***
(define (parse-input-into-gaps cs g v)
    (let ((firstg 0)
          (from 0)
          (end (vector-length v)))
      (cond ((not (vref g firstg))
             (let* ((endg (find-end-of-gap g firstg))
                    (loc (gap-skip-to-vector v from end cs firstg endg)))
               (cond ((and loc (fx= from loc))
                      (set from (fx+ from endg))
                      (set firstg endg))
                     (t
                      (set from nil))))))
      (loop (initial (endg)
                     (endm))
            (while from)
            (while (fx< from end))
            (while (fx< firstg (vector-length g)))
            (do (set (EBE-Gap-Start (vref g firstg)) from)
                (set endg (find-end-of-gap g (fx+ firstg 1)))
                (set endm (gap-skip-to-vector v from end cs firstg endg))
              (cond (endm
                     (set (EBE-Gap-End (vref g firstg)) endm)
                     (set from (fx+ endm (fx- endg firstg)))
                     (set firstg endg))
                    (t
                     (set from nil))))
          (result (and from
                       (fx= from (vector-length v))
                       (fx= firstg (vector-length g)))))))

;*** (EXTRACT-GAPS G V)
;*** =====================================================================
;*** Returns a vector of the pieces of the vector V that are matched by each of
;*** each of the gaps of G.
;***
(define (extract-gaps g v)
    (loop (initial (l nil))
          (step i \.in 0 to (vector-length g))
          (do (cond ((vref g i)
                     (set l (cons (sub-vector v (EBE-Gap-Start (vref g i))
                                              (fx- (EBE-Gap-End (vref g i))
                                                   (EBE-Gap-Start (vref g i))))
                                  l)))))
          (result (list->vector (reverse! l)))))


;*** (SUB-VECTOR V FROM LEN)
;*** =====================================================================
;*** Utility for extracting the slice V[FROM..FROM+LEN) of V.
;***
(define (sub-vector v from len)
    (let ((vx (make-vector len)))
      (loop (step i \.in from to (fx+ from len))
            (step j \.in 0)
            (do (set (vref vx j) (vref v i)))
            (result vx))))


;*** (PARSE-INPUTS-INTO-GAPS INPUT-LIST CS G)
;*** =====================================================================
;*** Matches the gap pattern represented by CS and G against the input
;*** samples in INPUT-LIST, and returns a list of vectors in which
;*** each vector represents the parse of the corresponding input.
;***
(define (parse-inputs-into-gaps input-list cs g)
    (loop (initial (l nil)
                   (failed? nil))
          (for v in input-list)
          (do (cond ((parse-input-into-gaps cs g v)
                     (set l (cons (extract-gaps g v) l)))
                    (t
                     (set failed? t))))
          (until failed?)
          (result (if failed? nil (reverse! l)))))
```

```
;*** ====================================================================
;*** ====================================================================
;*** The gap replacement synthesis algorithm.
;*** ====================================================================
;*** ====================================================================


;*** (CONSTRUCT-REPLACEMENT-MACHINES OUTPUTS PARSES)
;*** ====================================================================
;*** Implements the replacement expression synthesis algorithm.  Takes as input
;*** a list of output samples OUTPUTS and a corresponding list of parses of the
;*** input samples PARSES, and returns a representation of a finite state
;*** machine that encodes all possible replacement expressions that produce the
;*** output samples from the inputs.  Returns () if there is no replacement
;*** expression that will do the job.
;***
(define (construct-replacement-machines outputs parses)
    (cond ((all-null-vector? outputs)
           (make-vector 0))
          (t
           (loop (initial (m (and (car parses)
                                  (construct-replacement-machine (car outputs)
                                                                 (car parses)))))
                 (for o in (cdr outputs))
                 (for parse in (cdr parses))
                 (while (fx> (vector-length m) 0))
                 (do (let ((mt (construct-replacement-machine o parse)))
                       (set m (intersect-replacement-machines m mt))))
                 (result (and (fx> (vector-length m) 0) (remove-self-loops m)))))))


;*** (CONSTRUCT-REPLACEMENT-EXPRESSION M)
;*** ====================================================================
;*** Transforms a machine into a replacement expression.  A replacement
;*** expression is a list of strings, characters and integers; the characters
;*** and strings insert themselves, and the integers insert the gap from the
;*** input bearing that number.  The strategy used in building this single
;*** expression from a machine (which may yield more than one expression) is to
;*** traverse the machine (which has been minimized) from the start state to the
;*** final state taking the biggest jump possible at each step.  That is, if
;*** you're in state 3 with outgoing symbols #\A to state 4, -1- to state 9, and
;*** -2- to state 7, then take the -1- path and insert that gap into the
;*** replacement expression.  Of course, this strategy assumes that the state
;*** numbers imply something about the number of characters being skipped over,
;*** and it's quite possible that they don't.
;***
(define (construct-replacement-expression m)
    (loop (initial (l nil)
                   (s 0)
                   (final (fx- (vector-length m) 1)))
          (while (fx< s final))
          (do (let ((cs (vref m s)))
                (cond ((or (and (fixnum? (caar cs)) (fx> (caar cs) 0))
                      (cdr cs))
                  (loop (initial (farthest-state -1)
                                 (farthest-gap -1))
                        (for x in cs)
                        (do (cond ((and (fixnum? (car x)) (fx> (car x) 0))
                                   (cond ((fx> (cdr x) farthest-state)
                                          (set farthest-state (cdr x))
                                          (set farthest-gap
                                               (fx- (car x) 999)))))))
                        (result
                           (set s farthest-state)
                           (set l (cons farthest-gap l)))))
                 (t
                  (set l (cons (caar cs) l))
                  (set s (cdar cs))))))
        (result (cond ((null? l) (cons "" nil))
                      (t (reverse! l))))))


;*** (REMOVE-SELF-LOOPS M)
;*** ====================================================================
;*** Removes those state transitions in the finite state machine M that are
;*** self-loops.  Self-loops are the only kinds of loops that occur in the
;*** finite automata having to do with replacement expression synthesis -- they
;*** correspond to a gap that matches a null-string in the input.  All other
;*** state transitions are not involved in cycles.  This operation is applied
;*** as a post-processing pass after all of the intersections have been done.
;***
(define (remove-self-loops m)
    (loop (step i \.in 0 to (vector-length m))
          (do (vset m i (list-subset (vref m i) (lambda (x) (fxN= (cdr x) i))))))
    (minimize-one-more-time m))


;*** (MINIMIZE-ONE-MORE-TIME M)
;*** ====================================================================
;*** This is a minimization procedure in which self loops don't count.  The
;*** other minimization procedure would do fine here, except that we've changed
;*** representations, and its not a lot of code anyway.
;***
(define (minimize-one-more-time m)
    (loop (initial (final (fx- (vector-length m) 1)))
          (decr i from final to 0)
          (do (vset m i (list-subset (vref m i)
                                     (lambda (x) (or (fx= (cdr x) final)
                                                     (vref m (cdr x))))))))
    m)


;*** (LIST-SUBSET L PREDICATE)
;*** ====================================================================
;*** Destructively modifies a list L, leaving only those elements that answer
;*** true to the PREDICATE.
;***
```

```
(define (list-subset l predicate)
   (loop (while (and l (not (predicate (car l)))))
         (do (set l (cdr l))))
   (loop (initial (prev l))
         (while (cdr prev))
         (do (if (not (predicate (cadr prev)))
                 (set (cdr prev) (cddr prev))
                 (set prev (cdr prev)))))
   l)


;*** (CONSTRUCT-REPLACEMENT-MACHINE OV PARSE)
;*** ======================================================================
;*** Constructs a finite automaton that represents all possible replacement
;*** expressions that can create the output sample OV using the input parse
;*** PARSE.
;***
(define (construct-replacement-machine ov parse)
   (let* ((l (vector-length ov))
          (m (make-vector (fx+ l 1))))
     (vector-fill m nil)
     (loop (decr i from (fx- (vector-length parse) 1) to 0)
           (do (loop (initial (pv (vref parse i))
                              (j 0))
                     (while (fx< j l))
                     (do (let ((start (gap-skip-to-vector ov j l
                                                          pv 0 (vector-length pv))))
                           (cond (start
                                  (vset m start
                                        (cons (cons (fx+ i 1000)
                                                    (fx+ start (vector-length pv)))
                                              (vref m start)))
                                  (set j (fx+ start 1)))
                                 (t
                                  (set j l))))))))
     (loop (step i \.in 0 to l)
           (do (vset m i (cons (cons (vref ov i) (fx+ i 1))
                               (vref m i)))))
     m))


;*** (INTERSECT-REPLACEMENT-MACHINES M1 M2)
;*** ======================================================================
;*** Intersects two finite state machines M1 and M2, returning a machine that M
;*** that encodes those replacement expressions that can simulaneously produce
;*** those outputs handled by both M1 and M2.  Takes advantage of the fact that
;*** M1 and M2 are both acyclic (with the exception of self-loops), and that
;*** every transition from state i in M1 (and M2) is to some state j numbered
;*** j >= i.  The intermediate representation of M is as a sparse matrix, a
;*** vector of lists of column elements; the final call to CANONICALIZE-MACHINE
;*** converts M back to the same form as M1 and M2.
;***
(define (intersect-replacement-machines m1 m2)
   (cond ((fx> (vector-length m1) (vector-length m2))
        (let ((t1 m1)) (set m1 m2) (set m2 t1))))
    (let ((m (vector-fill (make-vector (vector-length m1)) nil)))
      (create-state m 0 0)
      (loop (step i \.in 0 to (vector-length m))
            (do (loop (initial (l (vref m i))
                               (cs nil))
                      (while l)
                      (do (set cs (car l))
                          (set (cdr cs) (merge-states m m1 i m2 (car cs))))
                      (do (set l (cdr l))))))
      (minimize-machine m (fx- (vector-length m1) 1) (fx- (vector-length m2) 1))
      (canonicalize-machine m)))


;*** (MERGE-STATES M M1 I M2 J)
;*** ========================================================================
;*** A utility used by the intersection procedure that takes state I from M1 and
;*** state J from M2 and merges them together to form state [I,J] in M.  For
;*** example, if state I has output transitions on symbols A and B to states 43
;*** and 77, respectively, then M1[I] would contain the list ((A .  43) (B .
;*** 77)).  If state J in M2 goes on B and C to states 11 and 17, then M2[J]
;*** would contain ((B .  11) (C .  17)).  State [I,J] in M2 would contain
;*** something like ((B .  (77 .  11))), where (77 .  11) is a temporary marker
;*** for a state in M that will be made into a regular state number by the
;*** machine canonicalization procedure.
;***
(define (merge-states m m1 i m2 j)
    (loop (initial (l nil)
                   (l1 (vref m1 i))
                   (l2 (vref m2 j)))
          (for x in l1)
          (do (loop (for y in l2)
                    (do (cond ((eq? (car x) (car y))
                               (set l (cons (cons (car x)
                                                  (create-state m (cdr x) (cdr y)))
                                            l)))))))
          (result (reverse! l))))


;*** (CREATE-STATE M I J)
;*** ========================================================================
;*** A utility used by MERGE-STATES to create state M[ (I . J) ].
;***
(define (create-state m i j)
    (let ((l (vref m i)))
      (cond ((or (null? l) (fx< j (caar l)))
             (vset m i (cons (cons j nil) l))
             (car (vref m i)))
            (t
             (loop (while (cdr l))
                   (while (fx>= j (caadr l)))
                   (do (set l (cdr l))))
             (cond ((fx= j (caar l))
                    (car l))
```

```
                (t
                 (set (cdr l) (cons (cons j nil) (cdr l)))
                 (cadr l)))))))


;*** (MINIMIZE-MACHINE M FINAL-I FINAL-J)
;*** ====================================================================
;*** Minimizes the state machine M that is constructed by the intersection
;*** procedure.  M has only one accepting state, state [FINAL-I, FINAL-J], and
;*** if this state is not present, then the resulting machine is the empty
;*** machine.  This procedure then takes advantage of the fact that all
;*** transitions in M are from lower-numbered states to higher-numbered states
;*** and minimizes M by simply traversing the states from highest-numbered to
;*** lowest and deleting those states in M that have no non-failing successors.
;***
(define (minimize-machine m final-i final-j)
    (loop (initial (got-to-final? nil))
          (for x in (vref m final-i))
          (do (cond ((fx= (car x) final-j)
                     (set (cdr x) (cons (cons 'final x) (cdr x)))
                     (set got-to-final? t))))
          (after
             (cond ((not got-to-final?)
                    (vector-fill m nil)))))
    (loop (decr i from (fx- (vector-length m) 1) to 0)
          (do (cond ((vref m i)
                     (if (cdr (vref m i))
                         (set (vref m i) (reverse! (vref m i))))
                     (loop (for x in (vref m i))
                           (do (loop (initial (prev x))
                                     (while (cdr prev))
                                     (do (cond ((null? (cddadr prev))
                                                (set (cdr prev) (cddr prev)))
                                               (t
                                                (set prev (cdr prev))))))))
                     (if (cdr (vref m i))
                         (set (vref m i) (reverse! (vref m i))))))))


;*** (CANONICALIZE-MACHINE M)
;*** ====================================================================
;*** Converts a minimized state machine from the sparse-matrix form produced by
;*** the intersection procedure to the canonical form.  Performs a breadth-first
;*** traversal of the machine from the start state, assigning state numbers, and
;*** then produces a canonical finite state machine with that numbering.
;***
(define (canonicalize-machine m)
    (let ((number-states 0))
      (loop (step i \.in 0 to (vector-length m))
            (do (loop (for x in (vref m i))
                      (do (cond ((cdr x)
                                 (set (car x) number-states)
                                 (set number-states (fx+ 1 number-states))))))))
      (loop (step i \.in 0 to (vector-length m))
          (do (loop (for x in (vref m i))
              (do (loop (for y in (cdr x))
                        (do (set (cdr y) (cadr y))))))))
    (loop (initial (final-m (make-vector number-states)))
          (step i \.in 0 to (vector-length m))
          (do (loop (for x in (vref m i))
                    (do (cond ((cdr x)
                               (vset final-m (car x) (cdr x)))))))
          (result final-m))))
```

```
;*** =====================================================================
;*** =====================================================================
;*** The gap bounding heuristic.
;*** =====================================================================
;*** =====================================================================

;*** (BOUND-GAPS I-PARSE DGAP)
;*** =====================================================================
;*** Performs the gap bounding heuristic, counting the maximum number of NEWLINE
;*** characters that occur in the text matched by each of the gaps in the
;*** pattern.
;***
(define (bound-gaps i-parse dgap)
    (loop (initial (gnum -1))
          (for x in dgap)
          (do (cond ((ebe-gap? x)
                     (set gnum (fx+ 1 gnum))
                     (loop (initial (max-lines 0))
                           (for y in i-parse)
                           (do (loop (initial (v (vref y gnum))
                                              (nlcnt 0))
                                     (step j \.in 0 to (vector-length v))
                                     (do (if (eq? (vref v j) #\NEWLINE)
                                             (set nlcnt (fx+ 1 nlcnt))))
                                     (result (if (fx> nlcnt max-lines)
                                                 (set max-lines nlcnt)))))
                           (result
                               (set (ebe-gap-bound x)
                                    (fx\/ (fx* max-lines 3) 2)))))))))
```

```
;*** ======================================================================
;*** ======================================================================
;***  The pattern reduction heuristic.
;*** ======================================================================
;*** ======================================================================

;*** (GAPS-IN-COMPATIBLE-POSITIONS? G-OCC NG-OCC SEP)
;*** ======================================================================
;*** A predicate for testing if a pair of gaps -1- CONSTANT -2- are being
;*** moved as a block in the replacement expression.
;***
(define (gaps-in-compatible-positions? g-occ ng-occ sep)
    (and (fx= (length g-occ) (length ng-occ))
         (loop (initial (failed? nil))
               (for x in g-occ)
               (for y in ng-occ)
               (do (cond ((fx<= (length x) (fx+ sep 1))
                          (set failed? t))
                         (t
                          (set failed? (not (eq? y (nthcdr x (fx+ sep 1))))))))
               (until failed?)
               (result (not failed?)))))


;*** (REPL-CONSTANTS-EQUAL? CS N ENDG REPL G-OCC)
;*** ======================================================================
;*** A predicate for testing if a pair of gaps -1- CONSTANT -2- are being
;*** moved as a block in the replacement expression.
;***
(define (repl-constants-equal? cs n endg repl g-occ)
    (loop (initial (failed? nil))
          (for x in g-occ)
          (do (loop (step i \.in n to endg)
                    (for y in (cdr x))
                    (do (set failed? (not (eq? (vref cs i) y))))
                    (until failed?)))
          (until failed?)
          (result (not failed?))))


;*** (MERGE-GAP-PAIR-IN-PATTERN CS G N ENDG)
;*** ======================================================================
;*** Utility for modifying the gap pattern when merging a pattern fragment -1-
;*** CONSTANT -2- into -1-.
;***
(define (merge-gap-pair-in-pattern cs g n endg)
    (loop (step i \.in endg to (vector-length cs))
          (step j \.in n)
          (do (set (vref cs j) (vref cs i))
              (set (vref g j) (vref g i)))
          (after
              (set (vector-length cs) (fx- (vector-length cs) (fx- endg n)))
              (set (vector-length g) (fx- (vector-length g) (fx- endg n))))))
```

```
;*** (MERGE-GAP-PAIR-IN-REPL REPL GAPDEL G-OCC NG-OCC)
;*** ====================================================================
;*** Utility for modifying the replacement expression when merging a pattern
;*** fragment -1- CONSTANT -2- into -1-.
;***
(define (merge-gap-pair-in-repl repl gapdel g-occ ng-occ)
    (loop (for x in g-occ)
          (for y in ng-occ)
          (do (set (cdr x) (cdr y))))
    (loop (initial (r repl)
                   (one-deleted gapdel))
          (while r)
          (do (if (and (fixnum? (car r))
                       (fx> (car r) one-deleted))
                  (set (car r) (fx- (car r) 1))))
          (next (r (cdr r)))))


;*** (MERGE-GAP-PAIR CS G REPL N ENDG GAPNUM NEXTNUM)
;*** ====================================================================
;*** Tentatively merges a pattern fragment like -1- CONSTANT -2- into a single
;*** gap -1-, if they are used as a block in the replacement expression REPL.
;***
(define (merge-gap-pair cs g repl n endg gapnum nextnum)
    (let ((g-occ ())
          (ng-occ ()))
      (loop (initial (r repl))
            (while r)
            (do (cond ((eq? (car r) gapnum)
                       (set g-occ (cons r g-occ)))
                      ((eq? (car r) nextnum)
                       (set ng-occ (cons r ng-occ)))))
            (next (r (cdr r))))
      (cond ((and (gaps-in-compatible-positions? g-occ ng-occ (fx- endg n))
                  (repl-constants-equal? cs n endg repl g-occ))
             (merge-gap-pair-in-pattern cs g n endg)
             (merge-gap-pair-in-repl repl nextnum g-occ ng-occ)
             t)
            (t
             nil))))


;*** (VECTOR-POPULATION G)
;*** ====================================================================
;*** Counts the number of non-null elements of the vector G.
;***
(define (vector-population g)
    (loop (initial (pop 0))
          (step i \.in 0 to (vector-length g))
          (do (if (vref g i) (set pop (fx+ pop 1))))
          (result pop)))
```

```
;*** (GAP-REDUCTION! CS G REPL)
;*** ==========================================================================
;*** Modifies the gap program represented by CS, G, and REPL to a reduced
;*** gap program.
;***
(define (gap-reduction! cs g repl)
    (loop (initial (previ)
                   (i (fx- (vector-length g) 1))
                   (gapnum (vector-population g)))
          (before
              (loop (while (fx> i 0))
                    (until (vref g i))
                    (do (set i (fx- i 1)))))
          (while (fx> i 0))
          (do (set previ (fx- i 1))
              (loop (while (fx>= previ 0))
                    (until (vref g previ))
                    (do (set previ (fx- previ 1)))))
          (until (fx< previ 0))
          (do (merge-gap-pair cs g repl previ i (fx- gapnum 1) gapnum)
              (set gapnum (fx- gapnum 1))
              (set i previ))))


;*** (GAP-REDUCTION CS G REPL)
;*** ==========================================================================
;*** Performs the pattern reduction heuristic on the gap program represented
;*** by CS and G.
;***
(define (gap-reduction cs g repl)
    (let ((cs (copy-vector cs))
          (g (copy-vector g))
          (repl (copy-list repl)))
      (gap-reduction! cs g repl)
      (list cs g repl)))


;*** (GAP-PROGRAM-DOESNT-WORK  CS G REPL IL OL)
;*** ==========================================================================
;*** This program accepts a gap program encoded as a constant sequence,
;*** a gap sequence, and a replacement expression; a list of inputs;
;*** and a parallel list of outputs.  It returns false if the gap program
;*** can compute the function described by the inputs and outputs,
;*** a positive index i if the pattern fails to match the i'th input,
;*** and a negative index if the program fails to transform the -i'th
;*** input to the -i'th output.
;***
(define (gap-program-doesnt-work cs g repl il ol)
    (loop (initial (failure nil))
          (step j from 0)
          (for i in il)
          (for o in ol)
          (do (cond ((parse-input-into-gaps cs g i)
                     (cond (o
                            (cond ((input-maps-to-output? g i repl o))
                                  (t
                                   (set failure j))))))
                    (t
                     (set failure (fx- 0 j)))))
          (until failure)
          (result failure)))


;*** (INPUT-MAPS-TO-OUTPUT? G I REPL O)
;*** =======================================================================
;*** Returns T if the replacement expression REPL can transform all of the
;*** inputs in I parsed by G to the outputs in O.
;***
(define (input-maps-to-output? g i repl o)
    (loop (initial (failed? nil)
                   (j 0)
                   (endo (vector-length o)))
          (while (fx< j endo))
          (while repl)
          (do (cond ((and (fixnum? (car repl))
                          (fx> (car repl) 0))
                     (let ((gp (retrieve-gap g (car repl))))
                       (or (gap-constants-match? o j i (EBE-Gap-Start gp)
                                  (EBE-Gap-End gp))
                           (set failed? t))
                       (set j (fx+ j (fx- (EBE-Gap-End gp) (EBE-Gap-Start gp))))))
                    ((eq? (car repl) (vref o j))
                     (set j (fx+ j 1)))
                    (t
                     (set failed? t)))
              (set repl (cdr repl)))
          (until failed?)
          (result (and (not failed?)
                       (fx= j endo)
                       (null? repl)))))


;*** (RETRIEVE-GAP G NUM)
;*** =======================================================================
;*** Returns the NUM'th gap of G.
;***
(define (retrieve-gap g num)
    (loop (step i \.in 0 to (vector-length g))
          (do (cond ((vref g i)
                     (set num (fx- num 1)))))
          (until (fx= num 0))
          (result
              (cond ((fx>= i (vector-length g))
                     (error "Illegal number in repl: ~s~&" num))
                    (t
                     (vref g i))))))
```

References

[1] A.V. Aho, B.W. Kernighan, and P.J. Weinberger.
AWK - A pattern matching and scanning language.
Software -- Practice and Experience 9(4):267-280, April, 1979.

[2] A.V. Aho.
Pattern matching in strings.
In *Formal Language Theory*, pages 325-347. Academic Press, 1980.

[3] D. Angluin.
On the complexity of minimum inference of regular sets.
Information and Control 39:337-350, 1978.

[4] D. Angluin.
Finding patterns common to a set of strings.
Journal of Computer and System Sciences 21:46-62, 1980.

[5] D. Angluin and C. Smith.
A survey of inductive inference: theory and methods.
Technical Report 250, Yale University, Department of Computer Science, October, 1982.

[6] D. Angluin.
A note on the number of queries needed to identify regular languages.
Information and Control 51(1):76-87, October, 1982.

[7] D. Angluin.
Inference of reversible languages.
Journal of the ACM 29(3):751-765, 1982.

[8] R. Balzer.
Automatic programming.
Technical Report 1, USC/ISI, September, 1972.

[9] M.A. Bauer.
Programming by examples.
Artificial Intelligence 12(1):1-21, May, 1979.

[10] A.W. Biermann and J.A. Feldman.
On the synthesis of finite-state machines from samples of their behavior.
IEEE Transactions on Computers C-21:592-597, 1972.

[11] A.W. Biermann.
On the inference of Turing machines from sample computations.
Artificial Intelligence 3:181-198, 1972.

[12] A.W. Biermann and J.A. Feldman.
A survey of results in grammatical inference.
In *Frontiers of Pattern Recognition*. Academic Press, N.Y., 1972.

[13] A.W. Biermann and R. Krishnaswamy.
Constructing programs from example computations.
IEEE Transactions on Software Engineering SE-2(3):141-153, September, 1976.

[14] A.W. Biermann.
Regular LISP programs and their synthesis from examples.
Report CS-1976-12, Computer Science Department, Duke University, 1976.

[15] L. Blum and M. Blum.
Toward a mathematical theory of inductive inference.
Information and Control 28:125-155, 1975.

[16] R.S. Boyer and J.S. Moore.
A fast string searching algorithm.
Communications of the ACM 20(10):262-272, October, 1977.

[17] V. Chvatal, D.A Klarner and D.E. Knuth.
Selected combinatorial research problems.
Technical Report STAN-CS-72-292, Stanford University Computer Science Department, June, 1972.

[18] S. Crespi-Reghizzi, M.A. Melkanoff and L. Lichten.
The use of grammatical inference for designing programming languages.
Communications of the ACM 16:83-90, 1973.

[19] S. Crespi-Reghizzi, G. Guida and D. Mandrioli.
Noncounting context-free languages.
Journal of the ACM 25:571-580, 1978.

[20] Gael Curry.
Programming by Abstract Demonstration.
PhD thesis, University of Washington, 1977.
also appeared as Computer Science Department Tech. Rep. 77-08-02.

[21] T. Dietterich, R. London, K. Clarkson, and R. Dromey.
Learning and inductive inference.
In P. Cohen and E. Feigenbaum (editors), *The Handbook of Artificial Intellegence*, pages 323-512. William Kaufman, Inc., 1982.

[22] A. Ehrenfeucht and P. Zeiger.
Complexity measures for regular expressions.
Journal of Computer and System Sciences 12:134-146, 1976.

[23] John R. Ellis, Nathaniel Mishkin, Robert P. Nix, and Steven R. Wood.
A BLISS programming environment.
Research report 231, Yale University, Department of Computer Science, June, 1982.

[13] A.W. Biermann and R. Krishnaswamy.
Constructing programs from example computations.
IEEE Transactions on Software Engineering SE-2(3):141-153, September, 1976.

[14] A.W. Biermann.
Regular LISP programs and their synthesis from examples.
Report CS-1976-12, Computer Science Department, Duke University, 1976.

[15] L. Blum and M. Blum.
Toward a mathematical theory of inductive inference.
Information and Control 28:125-155, 1975.

[16] R.S. Boyer and J.S. Moore.
A fast string searching algorithm.
Communications of the ACM 20(10):262-272, October, 1977.

[17] V. Chvatal, D.A Klarner and D.E. Knuth.
Selected combinatorial research problems.
Technical Report STAN-CS-72-292, Stanford University Computer Science Department, June, 1972.

[18] S. Crespi-Reghizzi, M.A. Melkanoff and L. Lichten.
The use of grammatical inference for designing programming languages.
Communications of the ACM 16:83-90, 1973.

[19] S. Crespi-Reghizzi, G. Guida and D. Mandrioli.
Noncounting context-free languages.
Journal of the ACM 25:571-580, 1978.

[20] Gael Curry.
Programming by Abstract Demonstration.
PhD thesis, University of Washington, 1977.
also appeared as Computer Science Department Tech. Rep. 77-08-02.

[21] T. Dietterich, R. London, K. Clarkson, and R. Dromey.
Learning and inductive inference.
In P. Cohen and E. Feigenbaum (editors), *The Handbook of Artificial Intellegence*, pages 323-512. William Kaufman, Inc., 1982.

[22] A. Ehrenfeucht and P. Zeiger.
Complexity measures for regular expressions.
Journal of Computer and System Sciences 12:134-146, 1976.

[23] John R. Ellis, Nathaniel Mishkin, Robert P. Nix, and Steven R. Wood.
A BLISS programming environment.
Research report 231, Yale University, Department of Computer Science, June, 1982.

[24] John R. Ellis, Nathaniel Mishkin, Mary-Claire van Leunen, and Steven R. Wood.
Tools: An environment for timeshared computing and programming.
Research report 232, Yale University, Department of Computer Science, June, 1982.
To appear in Software, Practice & Experience.

[25] W.S. Faught, D.A. Waterman, S. Rosenschein, D. Gorlin and S. Tepper.
EP-2: A prototype exemplary programming system.
Report R-2411-ARPA, Rand Corporation, 1979.

[26] W.S. Faught.
Applications of exemplary programming.
In *AFIPS Conference Proceedings, 1980 National Computer Conference,* pages 459-464. AFIPS Press, 1980.

[27] J.A. Feldman.
First thoughts on grammatical inference.
Technical Report 55, Stanford University Artificial Intelligence, 1967.

[28] J.A. Feldman, J. Gips, J.J. Horning, S. Reder.
Grammatical complexity and inference.
Technical Report, Stanford University, Computer Science Department, 1969.

[29] E. Fredkin.
Techniques using LISP for automatically discovering interesting relations in data.
In *The Programming Language LISP,* pages 108-124. MIT Press, 1964.

[30] K.S. Fu and T.L. Booth.
Grammatical inference: introduction and survey, parts 1 and 2.
IEEE Transactions on Systems, Man, and Cybernetics SMC-5:95-111 and 409-423, 1975.

[31] B.A. Galler and A.J. Perlis.
A View of Programming Languages.
Addison-Wesley, Reading, Mass., 1970.

[32] E.M. Gold.
Language identification in the limit.
Information and Control 10:447-474, 1967.

[33] E.M. Gold.
Complexity of automaton identification from given data.
Information and Control 37:302-320, 1978.

[34] C.C. Green, R.J. Waldinger, D.R. Barstow, R. Elschlager, D.B. Lenat, B.P. McCune, D.E. Shaw, and L.I. Steinberg.
Progress report on program understanding systems.
Technical Report Stan-CS-74-444, Stanford University, Computer Science Department, 1974.

[35] C.C. Green.
The design of the PSI program synthesis system.
In *Proceedings of the Second International Conference on Software Engineering*, pages 4-18. San Francisco, 1976.

[36] C.C. Green.
A summary of the PSI program synthesis system.
In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 380-381. Cambridge, Massachusetts, August, 1977.

[37] R.E. Griswold, J.F. Poage and I.P. Polonsky.
The SNOBOL4 Programming Language.
Prentice Hall, 1971.

[38] D.C. Halbert.
An Example of Programming by Example.
Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1981.
Also an internal report of Xerox Office Products Division, Palo Alto CA, 1981.

[39] S. Hardy.
Synthesis of LISP functions from examples.
In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 240-245. Tbilisi, USSR, September, 1975.

[40] D. Hatfield.
Formatting and Formats by Template (By Example) and What-You-See-Is-What-You-Get Programming.
1980.
An internal report of the IBM Cambridge Scientific Center, Cambridge MA, October 1980 (draft).

[41] David Hirschberg.
A linear space algorithm for computing maximal common subsequences.
Communications of the ACM 18(6):341-343, June, 1975.

[42] J.E. Hopcroft and J.D. Ullman.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, Reading, Mass., 1979.

[43] James W. Hunt and Thomas G. Szymanski.
A fast algorithm for computing longest common subsequences.
Communications of the ACM 20(5):350-353, May, 1977.

[44] E. T. Irons and F. M. Djorup.
A CRT editing system.
Communications of the ACM 15(1):16-20, January, 1972.

[45] K.P. Jantke and H.R. Beick.
Combining Postulates of Naturalness in Inductive Inference.
1980.
preprint, Humboldt Universitat zu Berlin.

[46] S. C. Johnson.
YACC: Yet another compiler compiler.
Technical Report 32, Bell Telephone Laboratories, 1978.

[47] J-P. Jouannaud, G. Guiho and J-P. Treuil.
SISP/1: An interactive system able to synthesize functions from examples.
In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 412-417. Cambridge, Massachusetts, August, 1977.

[48] J-P. Jouannaud and Y. Kodratoff.
Characterization of a class of functions synthesized by a Summers-like method using a B.M.W. matching technique.
In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 440-447. 1979.

[49] J-P. Jouannaud and Y. Kodratoff.
An automatic construction of LISP programs by transformations of functions synthesized from their input-output behavior.
International Journal of Policy Analysis and Information Systems 4:331-358, 1980.

[50] W.N. Joy, O. Babaoglu, R.S. Fabry, K. Sklower (editors).
Unix Programmer's Manual, Seventh Edition, Virtual VAX-11 Version
Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1980.

[51] Brian W. Kernighan and D. M. Richie.
The C Programming Language.
Prentice Hall, 1978.

[52] D.E. Knuth, J.H. Morris, and V.R. Pratt.
Fast pattern matching in strings.
SIAM Journal of Computing 6(2):323-350, 1977.

[53] D.E. Knuth.
TEX and METAFONT, New Directions in Typesetting.
Digital Press, Bedford, Massachusetts, 1979.

[54] Y. Kodratoff and J. Fargues.
A sane algorithm for the synthesis of LISP functions from example problems.
In *Proceedings of the AISB/GI Conference on Artificial Intelligence*, pages 169-175. AISB and GI, 1978.

[55] Y. Kodratoff.
A class of functions synthesized from a finite number of examples and a LISP program scheme.
International Journal of Computer and Information Sciences 8:489-521, 1979.

[56] M.E. Lesk.
LEX - A lexical analyzer generator.
Technical Report 39, Bell Telephone Laboratories, 1975.

[57] H. Lieberman and C. Hewitt.
A session with Tinker: interleaving program testing with program design.
In *Conference Record of the 1980 LISP Conference*, pages 90-99. Stanford University, 1980.

[58] D. Maier.
The complexity of some problems on subsequences and supersequences.
Journal of the ACM 25:322-336, 1978.

[59] Z. Manna and R. Waldinger.
Artificial Intelligence Series: Studies in Automatic Programming Logic.
North-Holland, 1977.

[60] Z. Manna and R. Waldinger.
The logic of computer programming.
IEEE Transactions on Software Engineering SE-4:199-229, May, 1978.

[61] Z. Manna and R. Waldinger.
Synthesis: dreams --> programs.
IEEE Transactions on Software Engineering SE-5:294-328, July, 1979.

[62] N. Meyrowitz and A. van Dam.
Interactive editing systems: Part II.
Computing Surveys 14(3):353-415, 1982.

[63] R. Michalski, J. Carbonell, and T. Mitchell.
Machine Learning - an artificial intelligence approach.
Tioga Publishing Company, Palo Alto, California, 1983.

[64] Jim Morris and Eric Schmidt.
Poplar Language Manual.
1978.
Xerox Palo Alto Research Center, 1978.

[65] James H. Morris, Eric Schmidt, and Philip Walder.
Experience with an applicative string processing language.
In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 32-46. Association for Computing Machinery, January, 1980.

[66] R.P. Nix and N.W. Mishkin.
U Editor User's and Programmer's Manual
Yale University, Department of Computer Science, 1983.
In preparation.

[67] T.W. Pao and J.W. Carr III.
A solution of the syntactical induction-inference problem for regular languages.
Computer Languages 3:53-64, 1978.

[68] S. Persson.
Some Sequence Extrapolating Programs: A Study of Representation and Modeling in Inquiring Systems.
PhD thesis, Stanford University, Computer Science Department, 1966.

[69] M. Pivar and E. Gord.
The LISP program for inductive inference on sequences.
In *The Programming Language LISP*, pages 260-289. MIT Press, 1964.

[70] M. Pivar and M. Finkelstein.
Automation, using LISP, of inductive inference on sequences.
In *The Programming Language LISP*, pages 125-136. MIT Press, 1964.

[71] Jonathan A. Rees and Norman I. Adams IV.
T: a dialect of Lisp or, lambda: the ultimate software tool.
In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming.* Association for Computing Machinery, August, 1982.

[72] Jonathan A. Rees and Norman I. Adams IV.
T User's Manual
Yale University, Department of Computer Science, 1982.

[73] Brian K. Reid and Janet H. Walker.
Scribe User Manual
Computer Science Department, Carnegie-Mellon University, 1978.

[74] Brian K. Reid.
A high-level approach to computer document formatting.
In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 24-31. ACM, January, 1980.

[75] D. M. Ritchie and K. Thompson.
The Unix time-sharing system.
Communications of the ACM 17(7):365-375, July, 1974.

[76] E. Shapiro.
Inductive inference of theories from facts.
Technical Report 192, Yale University, Department of Computer Science, 1981.

[77] E. Shapiro.
A general incremental algorithm that infers theories from facts.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 446-451. Vancouver, 1981.

[78] E. Shapiro.
Algorithmic program diagnosis.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*. 1982.

[79] Ehud Yehuda Shapiro.
Algorithmic Program Debugging.
PhD thesis, Yale University, 1982.
Also appeared as Yale Computer Science Department Research Report #237, and as a book in the ACM Distinguished Dissertation Series, MIT Press, 1983.

[80] D.E. Shaw, W.R. Swartout and C.C. Green.
Inferring LISP programs from examples.
In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 260-267. Tbilisi, USSR, September, 1975.

[81] T. Shinohara.
Polynomial time inference of pattern languages and its application.
In *Proceedings of the 7th IBM Symposium on Mathematical Foundations of Computer Science.* 1982.

[82] T. Shinohara.
Polynomial time inference of extended regular pattern languages.
In *Proceedings of Software Science and Engineering.* Kyoto, Japan, 1982.

[83] D.C. Smith.
Pygmalion: A Computer Program to Model and Stimulate Creative Thought.
PhD thesis, Stanford University, Computer Science Department, 1975.
Also available as AIM-260, Jun 1975, and as a book from Birkhauser Verlag, 1977.

[84] D.R. Smith.
A survey of the synthesis of LISP programs from examples.
In *Proceedings of the Symposium on Program Construction.* Bonas, France. INRIA, 1980.

[85] R.M. Stallman.
EMACS, the extensible, customizable, self-documenting display editor.
In *Proceedings of the ACM SIGPLAN-SIGOA Symposium on Text Manipulation*, pages 147-160. 1981.
The conference proceedings appeared as SIGPLAN Notices, Volume 16, Number 6, June 1981.

[86] L.J. Stockmeyer and A.R. Meyer.
Word problems requiring exponential time: preliminary report.
In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pages 1-9. 1973.

[87] P.D. Summers.
Program Construction from Examples.
PhD thesis, Yale University, Department of Computer Science, December, 1975.
Also available as IBM T.J. Watson Research Center Report RC-5637, September 1975.

[88] P.D. Summers.
A methodology for LISP program construction from examples.
Journal of the ACM 24(1):161-175, January, 1977.

[89] G.J. Sussman.
Artificial Intelligence Series. Volume 1: *A Computational Model of Skill Acquisition.*
American Elsevier, 1975.

[90] William R. Tanner.
Industrial Robots — Volume 1: Fundamentals.
Society of Manufacturing Engineers, Dearborn, Michigan, 1979.

[91] K. Thompson.
Regular expression search algorithm.
Communications of the ACM 11:419-422, June, 1968.

[92] B.A. Trakhtenbrot and Y.M. Barzdin.
Finite Automata.
North-Holland, Amsterdam, 1973.

[93] R.A. Wagner and M. J. Fischer.
The string-to-string correction problem.
Journal of the ACM 21:168-173, 1974.

[94] D.A. Waterman.
Exemplary programming in Rita.
In D.A. Waterman and F. Hayes-Roth (editors), *Pattern Directed Inference Systems,* . Academic Press, 1978.

[95] R.M. Wharton.
Grammar enumeration and inference.
Information and Control 33:253-272, 1977.

[96] P. Wiener.
Linear pattern matching algorithm.
In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1-11. 1973.

[97] I.H. Witten.
Programming by example for the casual user: a case study.
In *Proceedings of the Seventh Conference of the Canadian Man-Computer Communications Society.* Waterloo, Ontario, June, 1981.

[98] S.R. Wood.
Z: The 95% program editor.
In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 1-7. Association for Computing Machinery, June, 1981.

[99] M.M. Zloof.
Query by example.
In *AFIPS Conference Proceedings, 1975 National Computer Conference*, pages 431-438. AFIPS Press, 1975.

[100] M.M. Zloof.
Query-by-example: a data base language.
IBM Systems Journal 16(4):324-343, 1977.

[101] M.M. Zloof and S.P. deJong.
The system for business automation (SBA): programming language.
Communications of the ACM 20(6):385-396, June, 1977.

[102] M.M. Zloof.
QBE/OBE: a language for office and business automation.
Computer 14(5):13-22, May, 1981.

[103] M.M. Zloof.
Office-by-example: a business language that unifies data and word processing and electronic mail.
IBM Systems Journal 21(3):272-304, 1982.