A Colorful Approach to Text Processing by Example

Kuat Yessenov
MIT
Cambridge, MA

Shubham Tulsiani
IIT Kanpur
Kanpur, India

Aditya Menon
UCSD
San Diego, CA

Robert C. Miller
MIT
Cambridge, MA

Sumit Gulwani
Microsoft Research
Redmond, WA

Butler Lampson
Microsoft Research
Cambridge, MA

Adam Kalai
Microsoft Research
Cambridge, MA

ABSTRACT
Text processing, tedious and error-prone even for programmers, remains one of the most alluring targets of Programming by Example. An examination of real-world text processing tasks found on help forums reveals that many such tasks, beyond simple string manipulation, involve latent hierarchical structures.

We present STEPS, a programming system for processing structured and semi-structured text by example. STEPS users *create and manipulate hierarchical structure by example*. In a between-subject user study on fourteen computer scientists, STEPS compares favorably to traditional programming.

Author Keywords
Programming by Example; Text Processing.

ACM Classification Keywords
D.1.2 Programming Techniques: Automatic Programming; H.5.2 Information Interfaces and Presentation: UI

INTRODUCTION

Text processing (TP) is a problem of importance to programmers, data analysts, and other knowledge workers who have to handle data in many formats. Modern programming languages and text processing tools generally use regular expressions, string manipulation primitives, and parser generators. In contrast, the programming by example (PBE) approach (also called programming by demonstration) allows the user to edit example text by hand, and the system produces a program automatically [2, 13, 3, 7]. PBE systems are easier to learn and lack the arcane syntax of programming languages. Even for programmers, a sufficiently-powerful PBE system should have superior usability.

A number of PBE systems address repetitive TP tasks, such as reformatting a bibliography, from short demonstrations. These systems face challenges and advance the state of the art in HCI, Program Synthesis, and Machine Learning aspects of PBE.

A key difficulty, revealed by our examination of tasks from TP user forums, is that many tasks *crucially rely on latent hierarchical structure*. Previous PBE systems, based on patterns for cursor movement, string manipulation, or multiple selections, fail to capture these structures.

Almost every text file has structure. Reformatting a bibliography involves manipulating deeply nested structures including different entry types like books or articles, each with lists of author names that can be further decomposed into surnames, etc. For a simpler motivating example, consider a text file of multi-line records, e.g.:

```
TRB1006:
   Company: Yamaha
   Kind: Bass
   Year: 2006
COX15SA:
   Kind: Guitar
   Company: Yamaha
   Year: 2005
…
```

There is a structure obvious (to humans) here that may or may not be necessary to uncover, depending on the task at hand. For example, consider the following three tasks:

1. Sort the records by year.
2. Capitalize the company names.
3. Delete the year field from records where the company is "Yamaha" (because they have been discovered to be erroneous).

Only the second task may be accomplished without referring to structure. A system like SmartEdit [1] may quickly learn to repeat: *move the cursor to the end of the string "Company:", select to the end of the line, and then capitalize the selection*.

However, for the first and third tasks, repetitive cursor movements with string manipulation are insufficient, and these tasks cannot be implemented by example using prior systems. How can one enable users to expose and manipulate such structure in a simple fashion, by example?

UIST'13, October 8–11, 2013, St. Andrews, United Kingdom.

http://dx.doi.org/10.1145/2501988.2502040

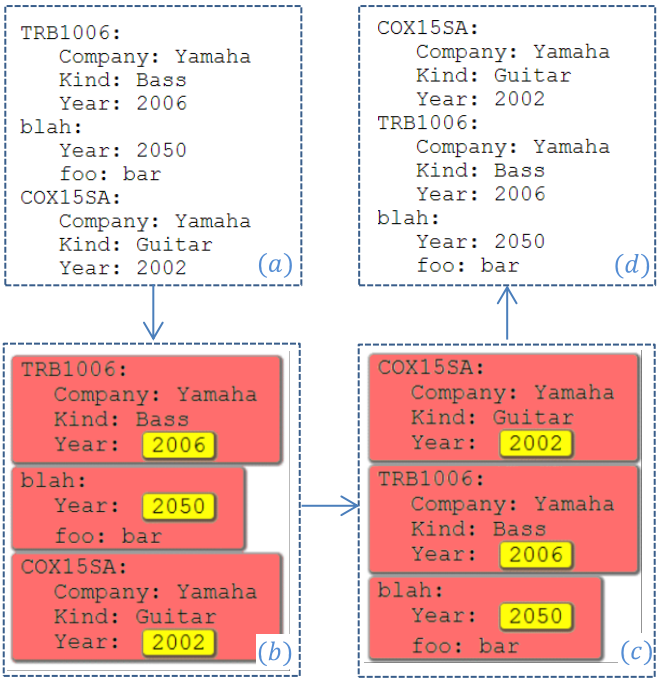


Figure 1. Sorting by year in STEPS. Mock input/output pairs[1] specify each step; nested colored blocks represent structure.

Our work is partly inspired by LAPIS [16], a system for text editing that incorporates "lightweight structure" and PBE. The multiple selection and simultaneous editing features of LAPIS, on the second task above, would enable one to select all company names and then simultaneously capitalize them, all by example. In LAPIS, patterns match arbitrary regions that may even partially overlap, such as the patterns *sentence* and *line of text*. This makes different patterns difficult to simultaneously display, and LAPIS users cannot easily create or manipulate hierarchical structure by example. The TP help forum posts we examined discussed hierarchical structure far more often than overlapping patterns.

STEPS and HSS

We introduce a Sequential Transformations by Example Programming System (STEPS) that, through a sequence of steps, modifies a hierarchically structured string (HSS). This HSS is displayed using nested colored blocks, as in Figure 1.

A STEPS user begins with a possibly-large text file and specifies explicit steps by input/output pairs. On the i^{th} step, the user illustrates a function f_i through an example of input HSS x_i and output HSS y_i such that $f_i\ x_i = y_i$. The system attempts to infer likely candidate functions mapping x_i to y_i. The output of a successful session is the modified text file and a standalone script that may be run on future data.

Figure 1 illustrates the first task in STEPS: first add structure, coloring the records red and the years yellow $(x_1 = a, y_1 = b)$, then sort red blocks by their yellow

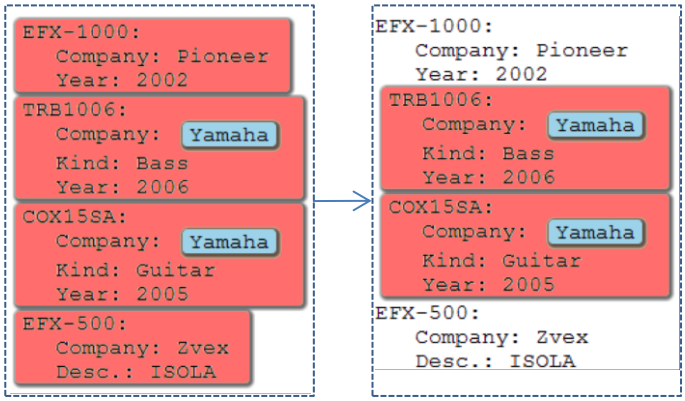


Figure 2. Un-coloring red blocks that do not contain blue blocks. This is a step in deleting year fields from Yamaha records (after coloring records in red and "Yamaha" string in blue).

blocks $(x_2 = b, y_2 = c)$, and finally remove the colors $(x_3 = c, y_3 = d)$.[1] Text and colored blocks are created and manipulated in a fashion similar to standard text editing operations. To color a block, one selects the text to be colored and then clicks on the desired color from a palate. This is similar to highlighting text in a rich text editor, but only nested (nonoverlapping) blocks are permitted.

For the third task, a user might try to delete the yellow blocks from red blocks containing "Yamaha." If the system fails to infer the correct transformation, additional colors and steps provide a recourse, e.g., color records in red, color the word Yamaha in blue, un-color red blocks that do not contain blue blocks (illustrated in Figure 2), and finally color and delete the year lines from the red blocks.

Demonstration vs. (mock) input/output examples

Each step is defined by an input-output pair of HSS's; the keystrokes and mouse movements used to generate this pair are not recorded. In contrast, the original PBE dream is that the system will simply observe the user perform actions and generalize their intention.

Demonstrations can be faster and more natural than the copying and pasting required to provide explicit before and after HSS's, but the ability to *correct* a PBE system is essential as described by Lau's study of why PBE systems fail [11]. Input/output examples ease debugging, since subtle mistakes can be identified and fixed by inspecting and editing a static artifact rather than replaying a trace.

Additionally, interaction through mock examples is a single technique that encompasses a variety of types of operations, unlike demonstration. How do you demonstrate sorting (or removing duplicates or counting) for a large file – do you manually search for the alphabetically first entry? Or suppose a user edits the 1st, 2nd, and 17th records – this may

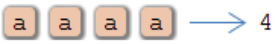
[1] This STEPS program can be displayed succinctly because the output to each step is the input to the next, i.e., $y_i = x_{i+1}$ for all i. In general, this need not be true.

or may not be a signal that the 3rd-16th records should not change. Lau also advises "presenting a model users can understand" [11], and input/output pairs make transparent the information on which the system bases its inference.

Simplified *mock* examples that hide inessential information or convey corner cases are common on TP help forms. Editable input/output pairs similarly enable users to easily construct ad-hoc artificial examples without changing the underlying file. For example, in Figure 1, the second record was created artificially as a response to the fact that the pattern first learned by the system for year was "200?", i.e., a number starting with 200. In a large file, the user may not find the entries where this is not the case. Hence, she creates a mock entry with a year of 2050, and the system infers the pattern of a number following "Year:". Also, since her data format permits arbitrary field order within a record, she places the year first even though the year happened to be last in all records in her file, so that the resulting script will correctly generalize to future data.

Finally, for proficient users, mock examples may provide surprisingly simple input-output pairs, e.g., the operation of Figure 2 may be defined by:

and counting may be illustrated by:



Scope

STEPS is a programming system targeted at an audience of people who program or have once programmed. These users can benefit from such a tool, especially those who use languages such as Java or C++ where TP is notoriously awkward. The scope of STEPS is what one could accomplish with "a short Perl/AWK script," which most programmers interpret to mean a precise syntactic or structural transformation rather than a fuzzy knowledge-based operation. Most text transformations found on TP forums are in the scope of STEPS. One type of exception is tasks that require multiple input files, and extending STEPS to handle multiple files would increase its utility.

More fundamentally, STEPS is poorly suited for probabilistic transformations like spelling correction, which may be more suited for systems such as Google Refine [8]. The types of errors that are rampant in unstructured data entered by humans may require too many steps. For example, in correcting addresses, STEPS would likely require a separate step for correcting each type of error, such as adding "NY" state fields to entries labeled with a city "New York" (or even "New Yrok") but where the state was omitted.

That said, STEPS is easily extensible. If the underlying library of functions included, say, a mapping zip codes to city names, then STEPS programs could easily employ this mapping. STEPS does not acquire and extract rich data from the web. However, the programmers posting on Perl/AWK TP forums rarely expected such fuzzy, probabilistic operations, suggesting that one can convey to programmers the types of tasks appropriate for STEPS.

If the data at hand is tabular in nature, spreadsheet processing software such as Excel, Google Refine [8] or Data Wrangler [9] may be more appropriate. However, the coloring feature of STEPS can be trivially applied to importing spreadsheets, coloring respective cells.

The hope is that systems like ours may also serve as stepping stones towards the dream of popular end-user systems with rich semantic capabilities. However, for the systems built en route to be useful, it is important to establish user expectations that are reliably met.

Contributions

This paper makes the following contributions:

- hierarchical structure coloring and manipulation as a user interface technique for PBE;
- mock input/output examples as a user interface technique for PBE;
- STEPS, a web-based PBE system that incorporates these techniques;
- a Domain Specific Language for manipulating hierarchically structured strings
- a user study of programmers showing that STEPS is faster than conventional programming, on tasks that have been out of reach for other PBE systems.

The remainder of this paper proceeds with a discussion of related work, the tasks used in our user study, a description of the interface, and a short description of the synthesis architecture. Finally, we discuss the user study.

RELATED WORK

A number of PBE systems for simple string manipulation *that do not expose or manipulate structure by example* exist, such as EBE [18], Tourmaline [17], Cima [14], TELS [19], Visual Awk [10], DEED [5], SmartEdit [12], and Flash Fill [6].[2] The lack of exposed structure limits the scope of these systems, as discussed earlier.

Additional tools consider the structure of text documents for purposes other than text processing. The PADS project [4] learns structure for ad-hoc data sources in domain specific languages designed for programmers. It produces very precise format specifications capable of identifying any errors in a data file, at a cost of increased complexity. It does not permit structural manipulations by example.

Data Wrangler extracts data from a text file and imports it into a spreadsheet program that supports some PBE [9]. For

[2] Flash Fill has been released in Microsoft Excel 2013.

Mock Input	Mock Output

TRB1006:
  Company: Yamaha
  Kind: Bass
  Year: 2006
EFX-1000:
  Company: Pioneer
  Year: 2005
COX15SA:
  Company: Yamaha
  Kind: Guitar
  Year: 2002
	COX15SA:
	  Company: Yamaha
	  Kind: Guitar
	  Year: 2002
	EFX-1000:
	  Company: Pioneer
	  Year: 2005
	TRB1006:
	  Company: Yamaha
	  Kind: Bass
	  Year: 2006

System response
FAILED TO FIND PATTERN. Perhaps see an example of sorting.

Figure 3. Though STEPS fails to infer a pattern, it is able to guess that the desired function may be sorting, and it provides context-specific help.

data that is inherently tabular, the powerful spreadsheet manipulations offered provide an alternative to PBE for many tasks. However, the loss of formatting information in the import process and the inability to manipulate hierarchical structure may limit applicability to general TP.

For synthesis and ranking, we use the framework of Menon *et al.* [15]. We also borrow ideas from Gulwani's pattern matching algorithm [6], which in turn borrows ideas from the Version Space Algebra approach to PBE [12].

For a recent overview of Programming by Example interaction paradigms, see [7].

TASKS

Following Gulwani's analysis of Excel help forums [6], we began by examining a large number of (single-file) text-processing tasks from a Unix shell script help forum [1] from which we created over one hundred benchmark tasks. Common practice on forums was to include input/output examples. Real and artificial examples were both prevalent.

Below are some typical scenarios of how STEPS can be used for text processing. These also happen to be the tasks we used to evaluate STEPS, which we describe later.

User Study Task 1: Remove digits (warm up)

Delete digits, as illustrated below.

Al~~123~~
John~~45~~
…

This task can be done by most prior PBE systems and regular-expression search-and-replace in some editors. STEPS requires users to color text before deleting it; the first step is selecting what is to be deleted and the second step is simply deleting it (for which STEPS provides a shortcut).

x_1	y_1
Al123 John45	Al123 John45

x_2	y_2
Al123 John45	Al John

The argument for not allowing direct deletion in one step is that it is surprisingly hard to debug: visually inspecting the result of a deletion is difficult because one does not see what has been deleted, while inspecting the results of coloring is easy. Alternatively, one could modify the system to allow deletion in one step and show deleted text in strikeout, as in earlier systems such as SmartEdit.

User Study Task 2: Replace / with \ in Location field

The data consists of records separated by blank lines, and the goal is to replace / with \ in `Location` fields only, not in other fields such as `Last Modified`. The first two records are:

```
Acrobat 6.0.2 Professional:

Version: 6.0.6
Last Modified: 18/10/2003 00:11
Kind: PowerPC
Location: /Apps/Acrobat/Professional.app

LiveUpdate:

Version: 3.0.1
Last Modified: 17/04/2003 12:00
Location: /Apps/Norton/LiveUpdate.app
```

To accomplish this task, one first adds structure and then performs the replacement. So, if the above data is the first mock input x_1, then a possible first mock output y_1 is:

```
Acrobat 6.0.2 Professional:

Version: 6.0.6
Last Modified: 18/10/2003 00:11
Kind: PowerPC
Location: / Apps / Acrobat / Professional.app
```

```
LiveUpdate:

Version: 3.0.1
Last Modified: 17/04/2003 12:00
Location: / Apps / Norton / LiveUpdate.app
```

After replacing / with \, one removes all color (for which the interface provides a shortcut). Note that adding the three colors above in STEPS would actually require three steps because STEPS currently limits users to adding one new color per step. Throughout this paper, we keep the presentation succinct by adding multiple colors simultaneously.

User Study Task 3: List fonts by style

Each font entry has a style. Sort the styles, and then list fonts with that style in the order they occur, each followed by a semicolon. Sample data:

```
AquaKana:
  Family: .Aqua Kana
  Style: Regular
  Version: 1.0
  Designer: JIYU-KOBO Ltd.
  Embeddable: Yes
Courier:
  Family: Courier
  Style: Regular
  Version: 5.1d1e1
  Embeddable: Yes
Courier-Bold:
  Family: Courier
  Style: Bold
  Version: 5.1d1e1
  Embeddable: Yes
```

Sample output:

```
Bold: Courier-Bold;
Regular: AquaKana; Courier;
```

This task, by far the most difficult, merits a more detailed walk through, which we give in the next section.

The fourth and final user study task involves searching an HTTP log file and extracting URL components that meet multiple criteria. In hindsight, it would have provided more variety to replace this task with one operating on semi-structured data that is not purely record based. We recently used STEPS for such tasks in response to a journal editor who requested a script to format-check journal submission files. The tasks include a number of tests on a latex file, such as verifying that the section titles are in "title case".

AVOIDING "DEAD ENDS"

Every PBE system will inevitably fail in some cases; the key question is how to proceed. There was nothing more frustrating to our users than a failure message with no suggestion of how to make progress, as Lau also reports [11]. *Search failures*, such as in Figure 3, occur when the system fails to find any useful transform mapping x to y that is "reasonable" (i.e., other than the trivial transform that always outputs the constant string y). *Ambiguity failures* occur when the computer finds too many transforms.[3] Users can often address ambiguity failures by providing longer mock examples or weeding through suggested transformations. Furthermore, a system good at *ranking* candidate transforms may also avoid ambiguity failures.

[3] The example of Figure 3 is ambiguous in many ways: it could be sorting records by year (or ID) or reversing records. It is also ambiguous because it will not even be clear how to segment further records.

One participant in our user study reported, "I was surprised that I never reached a dead end." In our experiments, dead ends did occur sometimes, and they were usually caused by search failures. STEPS employs several strategies to avoid dead ends.

First, the STEPS philosophy is to decompose complex transforms into smaller steps. Adding further colored blocks often makes progress.

Second, in the case of search failure, STEPS still attempts to identify the *type* of transform being illustrated, to provide useful feedback in the form of documentation or examples. Some examples of operations that are easier to recognize than to precisely infer include: sorting (mock output is a permutation of mock input), deleting, adding or removing color, extending or shrinking color, and counting (numbers in the mock output but not mock input).

In such cases, context-specific help can be given. For example, in sorting records, suppose that a user first tries to accomplish the entire task in a single step, as in Figure 3. The fact that the lines in the mock input and output are permutations of one another is a clue that sorting may be the primary operation. Hence feedback on how to sort suggests to the user that she might first color records. (Note that if the user is trying to do something besides sorting, such as reversing records, such feedback may still be helpful because she still needs to segment records.) As we shall see, it turns out that the clues of Menon *et al*. [15] perfectly model the problem of guessing the transformation type.

Third, when exact matches to $f\ x\ = y$ are not found, STEPS returns functions f that are approximately correct, i.e., $f\ x\ \approx y$. (Our notion of approximation considers differences in white space to be of minimal importance.) This can address small user typos and also may lead to unanticipated but more productive alternative suggestions.

STEP-BY-STEP WALK THROUGH

Bart, a fictitious user, is asked to write a program for User Study Task 3, namely grouping and listing fonts by style. Bart first pastes the data sample into the system in a text area, and the clicks a "START SCRIPT" button. For the first step, Bart tries:

Mock Input x_1	Mock output y_1
AquaKana:↵ ··Family:·.Aqua·Kana↵ ··Style:·Regular↵ ··Version:·1.0↵ ··Designer:·JIYU-KOBO·Ltd.↵ ··Embeddable:·Yes↵ Courier:↵ ··Family:·Courier↵ ··Style:·Regular↵ ··Version:·5.1d1e1↵ ··Embeddable:·Yes↵	AquaKana:↵ ··Family:·.Aqua·Kana↵ ··Style:·Regular↵ ··Version:·1.0↵ ··Designer:·JIYU-KOBO·Ltd.↵ ··Embeddable:·Yes ↵ Courier:↵ ··Family:·Courier↵ ··Style:·Regular↵ ··Version:·5.1d1e1↵ ··Embeddable:·Yes

Note that the system displays white space (spaces, tabs and newlines) with visible characters. Bart also employs a STEPS feature by which he names the grey color as "font."

After he hits the "Auto-code" button, the system offers Bart only one suggestion of a program with an English description of "Mark from capital letter to 'Yes' as font." It also gives him an opportunity to preview the transform before executing it. Bart examines the data, and it looks good. It appears that all the records end in "Embeddable: Yes." However, Bart, like the participants in our study, is not intimately familiar with the data. He is not sure if "Embeddable" will always be followed by "Yes," so he changes the mock pair so the second record ends in "No." The computer is forced to generalize.

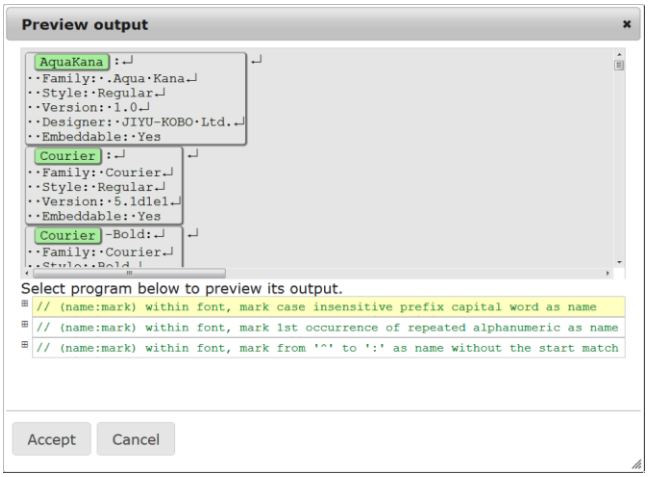
Mock Input x_1	Mock output y_1
AquaKana:↵ ··Family:·.Aqua·Kana↵ ··Style:·Regular↵ ··Version:·1.0↵ ··Designer:·JIYU-KOBO·Ltd.↵ ··Embeddable:·Yes↵ Courier:↵ ··Family:·Courier↵ ··Style:·Regular↵ ··Version:·5.1d1e1↵ ··Embeddable:·No↵	AquaKana:↵ ··Family:·.Aqua·Kana↵ ··Style:·Regular↵ ··Version:·1.0↵ ··Designer:·JIYU-KOBO·Ltd.↵ ··Embeddable:·Yes Courier:↵ ··Family:·Courier↵ ··Style:·Regular↵ ··Version:·5.1d1e1↵ ··Embeddable:·No

The new program generated is, "Mark from capital letter to 'Embeddable: ', word as font." Bart looks at the resulting preview, is satisfied, and continues. Several points worth making are:

1) Bart preferred to create an artificial mock example (actually modify an example) rather than spend the time searching for an example with "Embeddable: No" in the data.
2) In fact, all the data at hand did happen to end in "Embeddable: Yes." However, since Bart wanted a program that generalized well to future examples, his efforts were not in vain.
3) When this task was used in the user study, about half of the study participants examined the data preview and accepted the first program, and about half made the artificial "No" example. Two participants exerted further effort on segmentation because they were concerned about the fact that records may end in a completely different field than "Embeddable."
4) The suggestions and ranking of the system could be better on this example (i.e., perhaps a better pattern would be to recognize that records start on nonindented lines), but nonetheless users succeed with mock examples and additional colors.

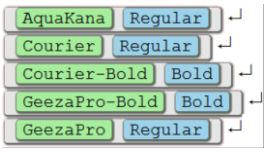
In the second step, Bart introduces a new color, green, which he calls "name." For the mock example, he takes the first two records and colors "AquaKana" and "Courier" in green. The system provides three possible programs along with the ability to preview them in a dialog, shown below.

Each of these programs is presented by its English description alone; the code is collapsed.[4] By clicking on any of these options, Bart sees a preview of its effect. The first



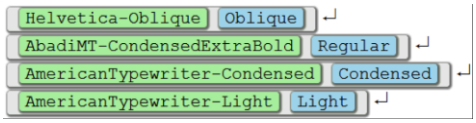
one is clearly wrong, as Bart intended the entire "Courier-Bold" to be green. But the third program is exactly what Bart wanted. He clicks "Accept" and moves on to the next step.

He then introduces a new color called style and marks the styles in a similar fashion. He then simplifies matters by replacing each record with just the name and style. The program generated is, "Within font, keep only name, style" and the resulting data is as follows,



...

Bart then sorts the records by style. For mock input he finds four records that are out of order (two would have sufficed):



The mock output are these four records sorted by their blue style. The generated program is "Sort font by style in alphabetical order."

[4] Moreover, there are typically multiple programs yielding exactly the same output on the data, and these are all collapsed into a single group. In earlier versions, we had them expanded by default, but no user was editing the code or selecting amongst multiple programs that yield the same output.

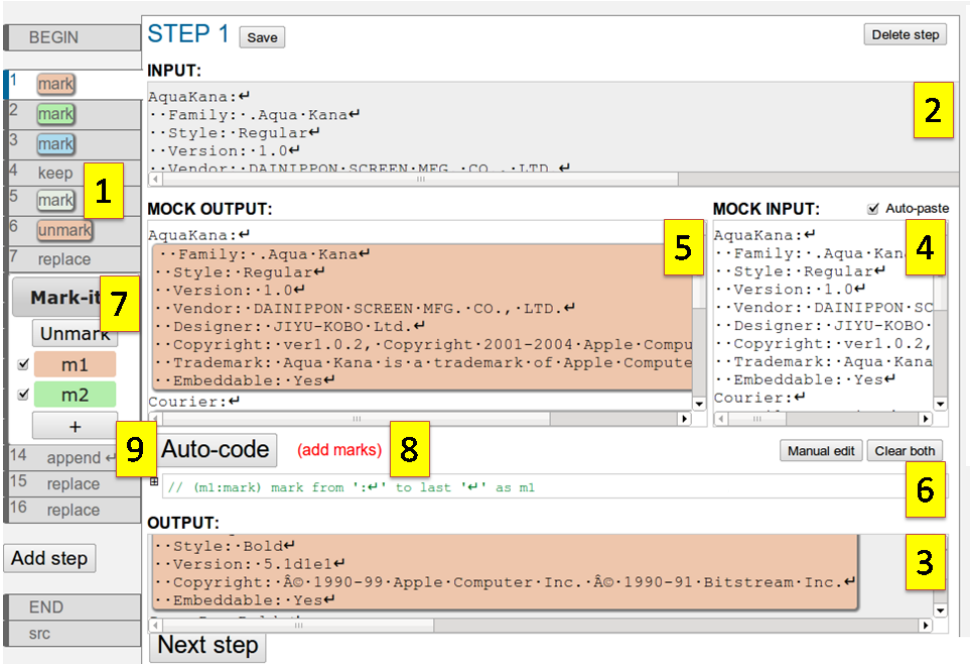
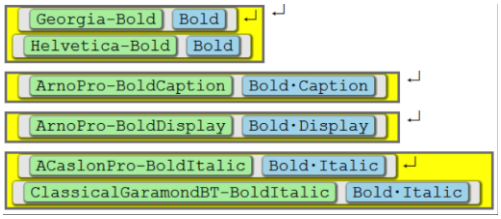


Figure 4. The STEPS interface. Steps are shown in separate tabs. Visible are:

1. List of steps
2. Step input data
3. Step output data
4. Mock input
5. Mock output
6. Editable code
7. Draggable palette
8. Feedback
9. Auto-code button

Grouping

For the crucial step, Bart creates a new color called "styleGroup". He finds six consecutive fonts and groups them into four groups based on equivalent styles.



Above is the mock output (the mock input is the same without the yellow color). The generated program is "Group font by style as styleGroup."

Wrap-up

The rest of the task is straightforward. Bart keeps only the first occurrence of the style within each yellow group, deleting the rest, and then appends semicolons and removes newlines, as was required in the task.

Note that grouping was the most difficult step for most users in the study, as we will discuss. This is probably because the tutorial we had given them had examples of sorting but no examples of grouping. Nonetheless, all but one of the participants tried grouping on their own, and the other participant found an alternative solution. (Also note that no participant renamed any colors or edited the style declarations.)

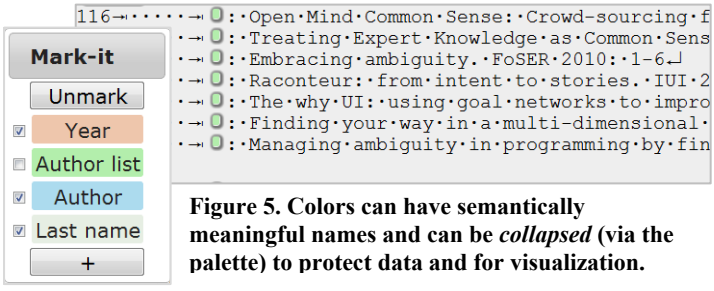


Figure 5. Colors can have semantically meaningful names and can be *collapsed* (via the palette) to protect data and for visualization.

INTERFACE

STEPS is a web application that works inside a browser window. Figure 4 shows a typical session consisting of a list of step tabs shown on the left side (1). The session starts by pasting raw text into the BEGIN tab and ends by copying the output text in the END tab. Step tabs enable the user to revisit and debug steps. Changes to a step automatically propagate throughout all following tabs, and changes to the data in the BEGIN tab propagate all the way to the END tab.

Each step has resizable panes for its *input (2)*, *output (3)*, *mock input (4)*, and *mock output (5)*. Spaces, tabs, and newline characters are displayed with visible glyphs.[5] Our earlier designs, which placed the mock input to the left of the mock output (or above the mock output), led users to mistakenly edit the mock input when they meant to edit the mock output. The code for the step is shown above the output and collapsed to a human-readable comment by default (6).

[5] Using the browser's built-in search functionality, searching for strings that include spaces fails when space characters are replaced with visible characters. Instead, one may create a new font where the standard space character is visible.

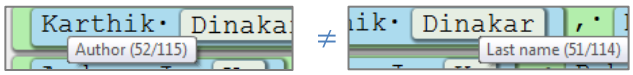


Figure 6. Tooltips appear when you hover over a colored block, indicating region name and block index/total count. Mismatches in numbers can be useful for debugging.

IMPLEMENTATION

We apply the synthesis and ranking approach of Menon *et al* [15]. To instantiate it, we need to define a domain specific language, and a number of *clues*, functions which generate candidate program fragments based the mock input and mock output. A clue might suggest, for example, that if a month name is present, the system should increase the likelihood of date transformations.

Domain Specific Language

We have designed a domain specific language, implemented as a small JavaScript library, which defines a colored string data type together with a small set of operations. A colored string is a string in which some regions are marked with a *tag*, an arbitrary string associated with a color. They must be nested as in XML: each tagged string consists of a sequence of tagged and plain substrings. The library will be made available online and is the only requirement to use the synthesized scripts outside of STEPS.

The core set of operations is shown in Table 1 and makes up the language of STEPS. These operations were chosen because they have natural English descriptions and also are general enough to cover a large number of input-to-output examples. For simplicity, each proposal that our system generates corresponds to exactly one command in the language, which we present in a direct English translation.

Generality is achieved by supplying a variety of options that slightly alter the behavior of the operations. For example, `mark-between` can optionally find the start or the end non-greedily or in nested fashion and drop either the start or the end match; the `sort` command takes an option to specify which comparison to use, etc.

The library also defines a pattern language for tagged strings. This language extends regular expressions with matching by tags and/or containment of another pattern, excluding certain tags by hiding them, and selecting matches by occurrence. The substitution mask for the `replace` operation can refer back to parts of the match to build a new tagged string.

Clue generation

The clue generation is primarily guided by the nature of the operation, e.g., adding colors, removing colors, or editing text. The nature can be determined by a quick inspection of the mock pair.

Then, for each command, STEPS generates a number of candidates for the parameters and the options. For pattern inference, it uses a library of common regular expression patterns, existing text and tags in the mock pair, and their combination. The space of all candidate programs is exhaustively explored, and all solutions that approximately match the mock pair are presented.

Operation	Params	Description
`mark`	p, t	mark matches to pattern p as tag t
`split`	p, t	mark between consecutive matches to pattern p as tag t
`mark-between`	p_1, p_2, t	mark between matches to p_1, p_2 as t
`within`	p, f	apply function f to each match of tag pattern p
`retag`	p, t	re-tag matches to p as t
`untag`	p	un-tag matches to the tag pattern p
`shrink`	t, p	shrink tags t to a match of p
`extend`	t, p	extend tags t to a match of p
`sort`	t, p	sort substrings of tag t by matches to p (which can be a tag itself)
`group`	t_1, t_2, p	consecutive matches to t_1 with the same match to p as t_2
`replace`	p, m	replace matches to p by a substitution mask m
`remove`	p	remove matches to p
`keep`	p	keep only matches to p

Table 1: STEPS operations and their required parameters.

USER STUDY

We performed a between-subject user study on fourteen computer scientists, doctoral students or postdocs, at a major software research laboratory. It would be difficult to get a true sample of "typical" programmers, but as Figure 7 shows, the participants had a variety of levels of programming experience.

The tasks, from the Task section above, were presented through a web-based form. Each task had an input file, and correctness was judged solely on the output produced for that input file (to avoid judging correctness of programs in general). The data varied in length from 200-830 lines.

Submitted answers were automatically judged and participants were informed whether their answer was correct or not and thus had an opportunity to "debug" submissions. (Otherwise it would not be clear when to move to the next problem or double check one's work.)

Seven participants were randomly assigned to each of the two conditions. In the STEPS group, participants were given a 20-minute overview of STEPS including a hands-on tutorial of some of the major types of features. Note that *grouping* was not covered in the tutorial since we wanted to

	Age	Gender	Programming Experience	Task 1	Task 2	Task 3	Task 4
Control	24	M	High	1	2	16	6
	25	M	Medium	12	12	29	12
	29	M	Medium	6	4	20	*
	29	M	Low	11	7	*	*
	28	M	Low	19	10	*	*
	26	M	Low	14	19	*	*
	26	F	High	36	8	*	*
STEPS	25	M	Medium	1	3	14	3
	28	M	High	3	7	17	16
	30	F	Medium	2	5	30	11
	27	M	Medium	3	12	*	13
	27	F	Low	2	5	54	*
	23	M	Low	2	8	56	*
	29	M	Medium	3	8	*	*

Figure 7. User study results. Task times are in minutes.

see if participants could discover how to do it on their own. They were then given a maximum of 70 minutes to work on the tasks.

In the control group, participants were allowed to accomplish the tasks using whatever means they wished, be it programming, manual editing, or using other tools such as word processors or spreadsheets. Prior to the experiment, they were told that they were to be given text-processing tasks and were allowed to prepare a machine of their choice (typically either a personal laptop or a company-provided desktop). They were also given 70 minutes for the tasks.

Participants identified their programming experience as low: "learned to program," medium: "spent a good bit of time programming," or high: "worked on large projects or been have been employed as a programmer."

Results

STEPS users completed **82%** of the tasks compared to **68%** for the control group. Everyone completed the first two tasks. Therefore, we performed an ANOVA of completion times for the first two tasks with task and condition as independent factors. The times were log-transformed to make the distribution closer to normal. We found a significant main effect of condition ($F=5.22$, $p< 0.032$), in which STEPS users were faster than the control on tasks 1 and 2, but no significant effect of task or interaction between task and condition.

Discussion

In the STEPS group, a minority of the steps (the ones that were accepted) were simply prefixes of the data. For task 3, grouping was a key step that six out of the seven participants discovered on their own even though our tutorial did not mention this operation. For this task, the remaining participant used STEPS for partial automation, finding a way to perform what most do in grouping using only five marking steps (though this would not have scaled well to larger data).

Participants were asked to compare STEPS to programming or something else they were familiar with. Several compared STEPS to "playing a game" or "solving a puzzle," one user compared it to using a spreadsheet, and one participant responded that using STEPS was like programming in an unfamiliar language. The latter participant found STEPS frustrating and wished she could just program instead, but the other participants used positive words such as "cool." Several participants requested access to STEPS for their personal use.

On the negative side, participants uniformly expressed frustration at not knowing what the possible primitive operations were, i.e., what it could do and what it couldn't do. In hindsight, this might have been improved at the cost of a longer tutorial, or by better exposition such as is found in LAPIS and SmartEdit.

In the control group, people used a variety of programming languages, partial automation (e.g., for the first task several participants opened the data in a text editor and ran 10 search-and-replace commands, replacing each digit with an empty string), and in some cases even manual editing. Presumably this was done when they thought it would be faster than programming, and hence forcing them to program (or program in a specific language) would present the comparison to STEPS in an even better light.

It is well known that most of the time in programming is spent debugging and understanding one's data. Anecdotally, this was true for our control participants as well. It seems quite likely that the speed-ups we observed were partly due to how STEPS makes it easier to visualize, understand, and debug the data, compared to programming.

CONCLUSION AND FUTURE WORK

For more than a decade, progress has stalled on what used to be considered one of the most interesting challenges in end user programming: performing sophisticated text edits, such as reformatting a bibliography, by example. Examination of tasks found on TP help forums sheds light on the difficulty: many TP tasks require exposing and manipulating *hierarchical structure* present in text files. This paper introduced STEPS, a system for manipulating hierarchical structures by example.

Our user study demonstrates that STEPS is faster than other alternatives, at least for computer science graduate students and postdocs. It also became clear that STEPS can be

improved in terms of the way in which it displays the inferred operations and set of possible operations, both of which have been addressed by previous work.

Looking forward, a central question is to what extent *end users* would adopt a HSS manipulation system like STEPS. In order for this to happen, STEPS inference must be improved to handle single steps that are more complex. Also, ways must be found to teach the end users about key concepts in STEPS. One possible approach is to allow demonstrations alongside input-output pairs, meaning that a user can demonstrate a step on their data while the system generates and displays the corresponding input-output pair. This might be a useful way to introduce the concept of input-output pairs.

A caution raised by Lau [11] and Gulwani [7] is to compare the perceived value of automation with its bottom-line cost, as users may be unwilling to incur the burden of switching out of one application into a separate PBE system. To this end, it may be wise to incorporate such a system into a programming IDE, which may also set certain expectations by explicitly presenting STEPS as an alternative to traditional programming.

REFERENCES

[1] The UNIX and Linux forums: Shell programming and scripting. http://www.unix.com/shell-programming-scripting.

[2] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[3] Allen Cypher, Mira Dontcheva, Tessa Lau, and Jeffrey Nichols. *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[4] Kathleen Fisher and David Walker. The PADS project: an overview. In *ICDT*, 2011.

[5] Yuzo Fujishima. Demonstrational automation of text editing tasks involving multiple focus points and conversions. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI '98)*, pages 101–108, 1998.

[6] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, 317-330, 2011.

[7] Sumit Gulwani. Synthesis from Examples: Interaction Models and Algorithms. In *Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 8-14, 2012.

[8] David Huynh and Stefano Mazzocchi. Google Refine. http://code.google.com/p/google-refine/.

[9] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare, editors, *CHI*, pages 3363–3372, 2011.

[10] Jurgen Landauer and Masahito Hirakawa. Visual AWK: a model for text processing by demonstration. In *Proceedings of the 11th International IEEE Symposium on Visual Languages '95*, pages 267–274, 1995.

[11] Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable AI. *AI Magazine*, 30(4):65–67, 2009.

[12] Tessa Lau, Steven Wolfman, Pedro Domingos, and Daniel Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2), 2003.

[13] H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[14] David Lawrence Maulsby. *Instructible agents*. PhD thesis, Calgary, Alta., Canada, Canada, 1995. UMI Order No. GAXNN-03114.

[15] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. A Machine Learning Framework for Programming by Example. *Proceedings of the 29th International Conference on Machine Learning, 2013*.

[16] Robert C. Miller. *Lightweight structure in text*. CMU PhD thesis, Pittsburgh, PA, USA, 2002..

[17] Brad A. Myers. Watch what I do. chapter Tourmaline: text formatting by demonstration, pages 309–321. MIT Press, Cambridge, MA, USA, 1993.

[18] Robert P. Nix. Editing by example. *TOPLAS*, 7(4):600–621, 1985.

[19] Sun Wu and Udi Manber. Agrep – a fast approximate pattern searching tool. In *Proceedings of the Winter USENIX Technical Conference*, pages 153–162, 1992.