Color profile: Generic CMYK printer profile
Composite Default screen

Chapter Thirteen

SWYN: A Visual Representation for Regular Expressions

Alan F. Blackwell

*Computer Laboratory,
University of Cambridge*

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:47:47 PM

Color profile: Generic CMYK printer profile
Composite Default screen

Abstract

People find it difficult to create and maintain abstractions. We often deal with abstract tasks by using notations that make the structure of the abstraction visible. Programming-by-example (PBE) systems sometimes make it more difficult to create abstractions. The user has to second-guess the results of the inference algorithm and sometimes cannot see any visual representation of the inferred result, let alone manipulate it easily. SWYN (See What You Need) addresses these issues in the context of constructing regular expressions from examples. It provides a visual representation that has been evaluated in empirical user testing and an induction interface that always allows the user to see and modify the effects of the supplied examples. The results demonstrate the potential advantages of more strictly applying cognitive dimensions analysis and direct manipulation principles when designing systems for PBE.

13.1 Introduction

Most programming tasks involve the creation of abstractions that can be broadly grouped into two categories: abstractions over context and abstractions over time. An *abstraction over context* defines some category of situations—objects or data—and allows the programmer to define operations on all members of that category. An *abstraction over time* defines events that will happen in the future as a result of the present actions of the programmer. Both of these are potentially labor-saving devices. A good abstraction can be used as a kind of mental shorthand for interacting with the world.

However, creating abstractions is difficult and risky (Green and Blackwell 1996; Blackwell and Green 1999). This is why PBE seems like such a good idea. It is computationally feasible to derive an abstraction from induction over a set of examples. If the abstraction is over context, the examples might include selections of words within a document or files within a directory structure. If the abstraction is over time, the examples can be demonstrations of the actions that the program ought to carry out in the future.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:47:48 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **246**

Color profile: Generic CMYK printer profile
Composite Default screen

13.1.1 Factors in the Usability of PBE Systems

The consequences for the user that result from this approach to programming can be discussed in terms of Green's framework for usability design of programming languages: the Cognitive Dimensions of Notations (Green 1989; Green and Petre 1996; Green and Blackwell 1998). This chapter will not include an extended presentation of the framework, since many published descriptions are available, but the resulting analysis would include observations of the following kind: PBE offers superb *closeness of mapping* between the programming environment and the task domain, because in PBE the task domain *is* the programming environment. However, PBE imposes severe *premature commitment*—the PBE programmer must specify actions in exactly the same order that the program is to execute them, unlike conventional programming languages. A full cognitive dimensions analysis of PBE systems would be enlightening: they are likely to be *error-prone,* for example, and it is often difficult to apply *secondary notation* such as comments to explain why a particular abstraction was created.

As with all programming languages, most designs for PBE systems have both advantages and disadvantages. No programming language can be the "best" language, because while some tasks are made easier by one language feature, the same feature can make other tasks more difficult (Green, Petre, and Bellamy 1991). In the case of PBE systems, a critical feature of this type is the question of whether a representation of the inferred program should be made visible to the user. This is the cognitive dimension of *visibility*. Some ordinary programming environments make it difficult to see the whole program at once, but in PBE systems the program is completely invisible, on the grounds that the programming process should be completely transparent to the user. These systems create abstractions by induction from examples, but the programmer is unable to see those abstractions. This disadvantage of this approach is that it results in concomitant degradation in other dimensions. An invisible abstraction exhibits high *viscosity* (resistance to change) because it is difficult to change something that you cannot see, and any relationships between different abstractions are certain to create *hidden dependencies* if the abstractions themselves are invisible.

The user's experience of PBE without access to a visible representation of the inferred program might be compared to repairing a loose part inside a closed clock by shaking the clock—you know that everything you do has some effect, but you don't know what that effect has been until you see and hear it working. If the clock is very simple on the inside and you understand how it works, it might be possible to succeed. Unfortunately, the most

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

powerful PBE systems employ sophisticated inference algorithms such that it can be quite difficult to anticipate the effect of adding one more instructional trace. The task of constructing a training set for these algorithms can be difficult for a computer scientist; for an end user, the clock-shaking analogy may be an apt description of the experience of PBE without a program representation. Other chapters in this book, including 3 and 16, have referred to the problem of “feedback” in PBE, but analysis in terms of cognitive dimensions makes it clear that the problem is far more extensive than simply a question of feedback.

13.1.2 A Test Case for Visibility in PBE

This chapter describes an investigation of a very simple experimental case that has been chosen to test the preceding argument. The programming domain is that of the earliest types of PBE system—simple text processing, in which text strings are identified by example, in order to be transformed in a systematic way. The simplest example of such a transformation is a search and replace operation. Even search-and-replace can be regarded as programming, because it is an abstraction-creating activity. The search expression is a transient abstraction over occurrences of some string in a document, although this “program” is usually discarded immediately after it has been executed.

However, straightforward search and replace is not a very interesting task in programming terms. A more interesting case is where the search expression includes wild cards, especially the extended types of wild card matching that are provided by regular expressions. Regular expressions have interesting computational properties, are widely used in powerful programmers’ editors, and are of interest in a machine-learning context because the acquisition of regular expressions from examples is a nontrivial induction problem. Furthermore, regular expressions can be used as the core of a powerful language for specifying text-processing scripts, as in sed, awk, or the Perl language (Wall, Christiansen, and Schwartz 1996).

Regular expressions are also interesting from the perspective of usability. In a system in which regular expressions can be inferred from examples, it is still not clear that users will benefit from being shown the resulting expression. This is not because it is a bad thing for users to see the result of PBE inference but because regular expressions themselves are confusing and difficult to read. They appear to be one of the features of Perl that is most difficult for users; popular Perl texts such as Christiansen and Torkington (1998) or Herrman (1997) preface their chapters on regular expressions with

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:47:48 PM

Color profile: Generic CMYK printer profile
Composite Default screen

grim warnings about the difficulties that are in store for the reader. A brief analysis in terms of Cognitive Dimensions of Notations suggests that the problems with regular expressions may be a result of the notational conventions.

Green (personal communication, 15 June, 1999) points out that the conventional regular expression notation is difficult to parse for a number of reasons:

- some of the symbols are ill chosen (notably / and \);
- the indication of scope by paired elements such as () {} [] is likely to cause perceptual problems;
- the targets to be matched and the control characters describing the match requirements are drawn from the same alphabet; and
- the notation is extremely terse; discriminability is reduced and redundancy is very low, so that in general a small random change produces a new well-formed expression rather than a syntax error.

Furthermore, there is no clear mental model for the behavior of the expression evaluator. If the notation indicated some execution mechanism (Blackwell 1996) or allowed users to imagine executing it themselves (Watt 1998), it could be more easily related to program behavior. These considerations give two potential avenues for improvement of regular expressions; both are tested in the experiment described here.

13.1.3 Summary of Objectives

The system described in this chapter is named See What You Need (SWYN). It is able to infer regular expressions from text examples in a context that improves visibility in several important ways:

- The user is always able to see the set of examples that the inference algorithm is using.
- The user is able to see the regular expression that has been inferred from the examples.
- The regular expression is displayed in a form that makes it easier to understand.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:47:49 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **249**

Color profile: Generic CMYK printer profile
Composite Default screen

- The user is able to see the effect of the inferred expression in the context of the displayed data.

After the next section, which reviews other similar research, the body of the chapter describes three important components of SWYN. The first is the method by which the user selects examples and reviews the current effect of the inferred expression. The second is the induction algorithm that is used to infer and update a regular expression based on those examples. The third is a visualization technique that allows users to review and directly modify the inferred expression.

13.2 Other PBE Systems for Inferring Regular Expressions

As described earlier, the inference of text transformations such as search and replace expressions was one of the earliest applications of programming by example. Nix's (1985) Editing by Example prototype allowed users to define input and output sample texts, from which a general transformation was inferred. The user gave a command to execute the current hypothesis, which resulted in a global search and replace according to the inferred hypothesis. The system provided an undo facility to reverse the command if the hypothesis was incorrect.

Mo and Witten's TELS system (Mo and Witten 1992; Witten and Mo 1993) could acquire complete procedural sequences from examples, including series of cursor movements, insertions, and deletions, in addition to the search-and-replace functionality of Nix's system. If inserted text varied between examples, the inferred program would stop and invite the user to insert the required text, rather than try to infer the text that was required.

Masui and Nakayama proposed the addition of a "repeat" key to a text editor, which would execute dynamically created macros (Masui and Nakayama 1994). Their system continually monitored the user's actions, inferring general sequences. At any time, the user could press the "repeat" key, and the system would respond by repeating the longest possible sequence of actions that had been inferred from the immediately preceding input.

The acquired description of the input text in these systems is in the form of regular expressions (Nix uses the term "gap expression" to describe a regular expression with additional specification of transformed output text). It would be possible to display the inferences to the user in various forms,

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

such as those introduced later in this chapter. However, previous systems that infer text-editing programs from examples have extremely poor *visibility* when considered as programming languages: they effectively hide the completed program from the user. They require that the user work only by manipulating the data, with the option of rejecting incorrect hypotheses after observing the results of executing an undesired inferred program. Several previous PBE systems have recognized this problem and have provided visual representations of the inferred program. Early systems include PURSUIT (Modugno and Myers 1993) and Chimera (Kurlander and Feiner 1993), while SMARTedit, described elsewhere in this book (see Chapter 11), provides a highly expressive program representation language.

Only one example-based text-processing system has addressed the question of how textual inferences should be presented to a user without programming skills. In the Grammex (Grammar by Example) system, also described in this book (see Chapter 12), the user assigns meaningful names to the subexpressions that have been inferred by the system. The result is similar to the process followed when defining BNF grammars for language compilers. No attempt has yet been made to evaluate the usability of the Grammex system, but some of the usability implications can be anticipated on the basis of cognitive dimensions. A system in which the user must identify and name the abstractions being created is *abstraction-hungry,* and this property tends to constitute an initial obstacle for inexperienced users. However, the ability to create names is a simple but effective example of *secondary notation*—allowing users to add their own information to the representation. An even more valuable form of secondary notation would be the ability to add further annotations that could be used to describe intended usage, design rationale, or other notes to future users.

13.3 A User Interface for Creating Regular Expressions from Examples

The usability improvements that SWYN aims to provide over previous demonstration-based systems are

- that the user should be able to predict what inference will result from the selection of examples,
- that the inferred program should be visible to the user,

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:47:49 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **251**

Color profile: Generic CMYK printer profile
Composite Default screen

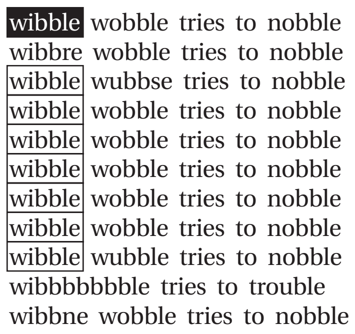
- that the user should be able to anticipate the effects of running the inferred program, and
- that the user should be able to modify the inferred program.

The initial state of the SYWN interface is a simple display of all the text that is a candidate for selection by a regular expression. If integrated into a word processor as an advanced search-and-replace facility, the display could simply be the regular word processor display, and SWYN could be invoked as a search-and-replace mode analogous to the incremental search mode of the EMACS editor.

The user starts to create the regular expression by choosing a string within the displayed text (dragging the mouse over it). The chosen string is highlighted, and every other occurrence of the same string within the text is also highlighted (in a different color), as in Figure 13.1. What the user sees is the set that would be selected when executing the regular expression defined so far. Of course, after choosing only a single example, the regular expression created so far is identical to the example, so all the highlighted strings are the same.

The user can then refine the regular expression by choosing another example, one that is not already selected. The regular expression is modified by induction over the two chosen examples, using the algorithm described in the next section. The highlighted selection set is immediately changed to show the user the effect of this new regular expression, as in Figure 13.2. At this point the selection set will be larger than the initial selection set,

FIGURE 13.1


wibble wobble tries to nobble
wibbre wobble tries to nobble
wibble wubbse tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wubble tries to nobble
wibbbbbbble tries to trouble
wibbne wobble tries to nobble

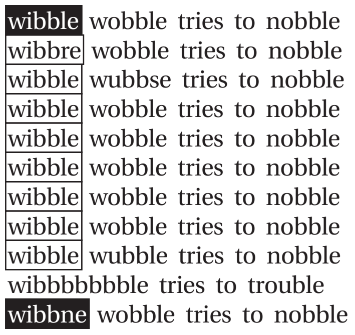
Selection set after choosing “wibble” as an example.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:47:54 PM
TNT Job Number: **[002564]** • *Author:* [**Lieberman**] • *Page:* **252**

Color profile: Generic CMYK printer profile
Composite Default screen

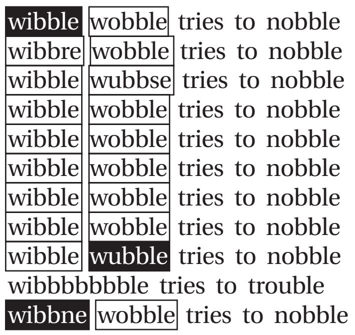
FIGURE **13.2**



wibble wobble tries to nobble
wibbre wobble tries to nobble
wibble wubbse tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wubble tries to nobble
wibbbbbbble tries to trouble
wibbne wobble tries to nobble

Selection set after adding the new example "wibbne".

FIGURE **13.3**



wibble wobble tries to nobble
wibbre wobble tries to nobble
wibble wubbse tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wubble tries to nobble
wibbbbbbble tries to trouble
wibbne wobble tries to nobble

Selection set after adding the new example "wubble".

because the regular expression is more general—it includes both chosen examples, and possibly other strings sharing their common features, as described in the next section.

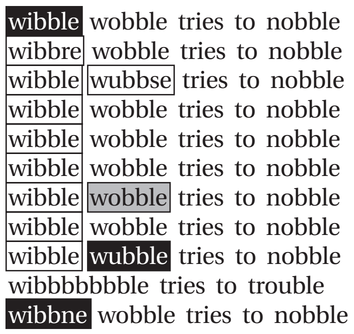
The user can continue to expand the definition of the regular expression by choosing further examples of the kinds of string that should be selected. Every choice of a positive example results in a generalization of regular expression and an increase in the size of the displayed selection set, as in Figure 13.3. However the user can also make the regular expression more specialized by choosing a negative example—a string that should not be

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:02 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **253**

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 13.4



wibble wobble tries to nobble
wibbre wobble tries to nobble
wibble wubbse tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wubble tries to nobble
wibbbbbbble tries to trouble
wibbne wobble tries to nobble

Selection set after choosing a negative example, "wobble".

included in the selection set. Negative examples are chosen by highlighting them in a different color—currently red rather than the green of positive examples (although this brings obvious usability problems for color-blind users). When a negative example is chosen, the regular expression is modified by performing induction on a negative example, which will have the effect of making the regular expression more specialized, as described in the next section. The size of the current selection set will therefore be reduced after choosing a negative example, as shown in Figure 13.4.

The ability to choose negative examples is an extremely valuable way to improve the usability of PBE systems. Much research into the acquisition of programs from examples has concentrated on the theoretical problem of inference from positive examples only (e.g., Angluin 1980). Induction algorithms can be made more efficient and accurate when they have access to negative examples, so training sets can be defined more quickly (Dietterich and Michalski 1984). Furthermore, people naturally describe contextual abstractions in terms of negative exemplars. Human definitions of conceptual categories often employ negative exemplars to describe an excluded category (Johnson-Laird 1983). In the context of SWYN, the ability to work from negative examples also provides an important feature of direct manipulation—the effect of actions should be not only immediately visible but easily reversible (Shneiderman 1983). If I choose an example string that causes the selection set to become too general, it is easy and natural to point to a string that should not have been selected and allow the induction algorithm to correct the problem through specialization of the regular expression.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:06 PM
TNT Job Number: [002564] • Author: [Lieberman] • Page: 254

Color profile: Generic CMYK printer profile
Composite Default screen

Future versions of SWYN will add a further means of choosing examples. The probabilistic induction algorithm described at the end of the next section is able to identify strings that are borderline cases—the user may or may not want them included. If the system knew which the user wanted, this would allow the induction algorithm to remove ambiguities from the regular expression. Borderline cases would be highlighted in a different color from the rest of the selection set, giving a cue to the user of the best way to refine the regular expression. The user can then decide on the appropriate action for each borderline case, simply choosing them as negative or positive examples in the usual way.

13.4 A Heuristic Algorithm for Regular Expression Inference

The current implementation of the inference algorithm used in SWYN has been designed to operate using heuristics whose effects can be anticipated by the user. The approach taken is an extension of the heuristic method proposed by Mo and Witten (1992). Their heuristic approach improved on that of Nix (1985) by defining typed character classes as components of the inferred strings. They suggest that users would normally have some class of characters in mind when selecting examples and that the function of the inference heuristics should be to identify the class that the user intended.

The heuristic algorithm currently implemented in SWYN incrementally modifies the regular expression in response to new examples chosen by the user. A graph reduction algorithm identifies common elements of the examples and produces minimal regular expressions composed of common character classes. This process is illustrated in Figure 13.5, which shows the effects of choosing the first two strings in the figures of the previous section.

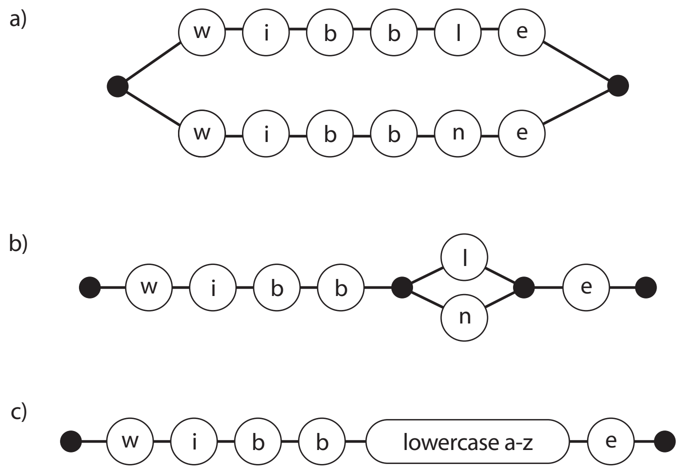
When a new positive example is chosen, it is added to the graph as a complete alternative expression. Alternatives are branches in the regular expression graph, as shown in Figure 13.5(a). This graph is then reduced by merging common elements at the beginning or end of alternative branches. The result of this merging process is shown in Figure 13.5(b). Where the graph reduction produces alternatives that are single characters, these are merged into the smallest general character class, as shown in Figure 13.5(c). The character class can later be refined by choosing negative examples or by directly manipulating the regular expression itself, as described later in the chapter.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:07 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **255**

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 13.5



Regular expressions induction by heuristic graph reduction: (a) addition of new exemplar as an alternative, (b) reduction of common elements in alternative branches, and (c) replacement of single-letter alternatives with a character class.

Repeated elements in the regular expression are also inferred using graph reduction heuristics. Figure 13.6 shows the effect of adding an example that can be explained as a repeated character in the regular expression. Where an alternative branch consists solely of repeated elements (either a repeated single character or repeated subexpressions), these are identified as a repeated component of the regular expression, as in Figure 13.6(c). Repeated branches are then merged with any occurrences of the repeat before or after the branch, as in Figure 13.6(d).

13.4.1 Probabilistic Algorithm

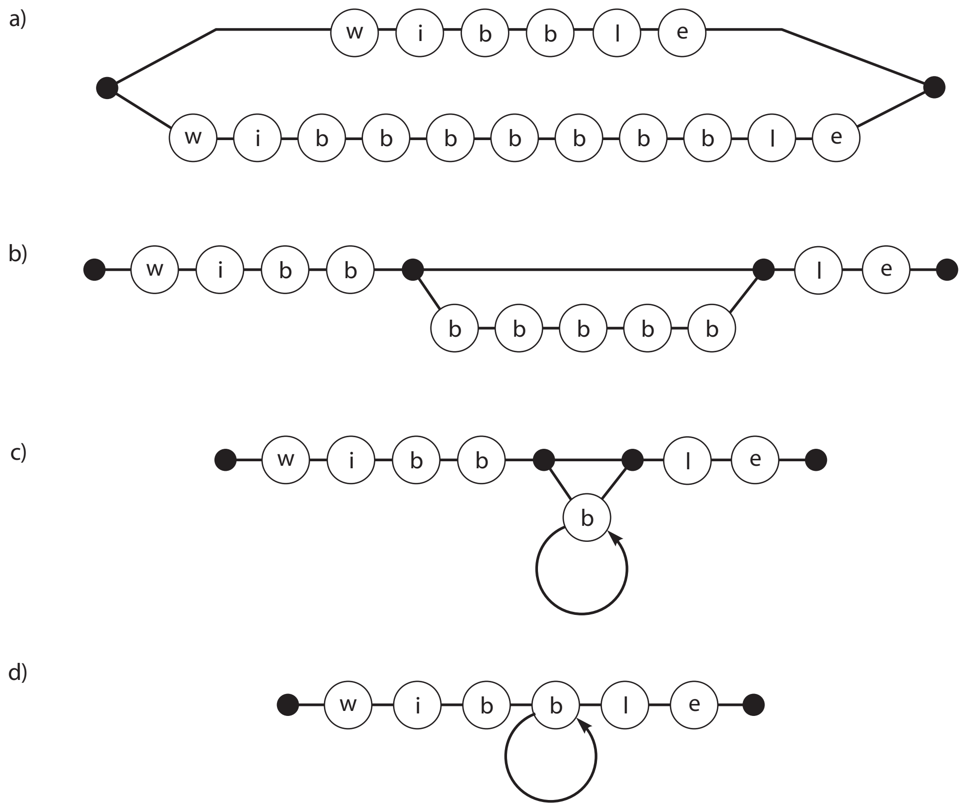
The heuristic algorithm described earlier is both deterministic and predictable, but it may not always result in optimal regular expressions. In the context of SWYN, this is completely intentional. It is better that the user should be able to imagine the result of choosing new examples than that the results be optimal. Just as Mo and Witten made some assumption about the classes of characters that would most likely be intended by the user, this heuristic

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:11 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **256**

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE **13.6**



Heuristics for inferring repeated sections of regular expression.

graph reduction approach makes some assumptions about the types of regular expression structure that are most likely intended by the user. This usability feature does result in some loss of generality, but it is always possible for the user to optimize the expression by direct manipulation, as described later in the chapter.

However, an alternative is to use a probabilistic algorithm, in which the examples chosen have more influence on the intended expression. Current work on SWYN is replacing the simple graph heuristics described earlier with a probabilistic model based on stochastic context-free grammars, in which alternative grammars can be assigned probabilities on the basis of

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:16 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **257**

Color profile: Generic CMYK printer profile
Composite Default screen

the examples that the user has chosen. A great advantage of this new approach is that text not yet chosen by the user can also be assigned probabilities according to different interpretations of the grammar. A string that may be matched by one possible interpretation but not by another can then be classed as ambiguous and brought to the user's attention as a priority for his or her next decision.

13.5 A Visual Notation for Regular Expressions

The previous sections have described the activity of creating a regular expression from examples, as though the user might be able to see and modify not only the examples and resulting selection set but the expression itself. It would certainly be a good thing if that were possible, for reasons described earlier. In fact, the previous section did provide a kind of visual representation of the regular expression for the benefit of you, the reader. The graphs drawn in the discussion of the graph reduction algorithm are very useful in understanding how the algorithm works, and they included some *ad hoc* syntactic elements that provide clues about how one might represent regular expressions visually—a loop with an arrow represented repetition of a character, and a solid black circle represented the beginning and end of alternate subexpressions.

It is difficult to develop usable new visual representations from purely theoretical considerations. The design of visual representations is partly a craft skill and partly a question of cognitive science, in which experimental evidence can be used to assess alternatives. The SWYN project is based on cognitive research into reasoning with diagrammatic representations (Blackwell et al., in press) and has taken the second approach to the design of visual notation.

This section reports an experiment that evaluated four potential representations of regular expressions: conventional regular expressions and three alternatives. Altogether, two of the four alternatives presented the regular expressions in declarative form, while two suggested an explicit order of evaluation. Furthermore, two of the alternative notations used only conventional characters from the ASCII character set, while two used graphical conventions in a way that might be described as a "visual" regular expression. The design of the experiment allowed the effects of these factors to be compared.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:16 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **258**

Color profile: Generic CMYK printer profile
Composite Default screen

13.5.1 Experiment: Evaluation of Alternative Representations

The four notations used in this experiment expressed equivalent information in very different forms. The first of these was that of conventional regular expressions, although a slightly constrained subset was used. One example is that the "+" command was used rather than the "*" command, because the latter often mystifies novices when it matches null strings. Figure 13.7 shows a typical example of a regular expression that was used in the experiment: this example matches a range of English telephone numbers: 01223 613234, 0044 1223 356319, 0044 1223 354319, and so forth. Note that it also matches some strings that are *not* valid English phone numbers, such as 0044 1223 303. This is intentional—it is typical of the problems encountered by novices when using regular expressions, and the experiment specifically tested whether users were able to recognize valid matches even when they were inconsistent with environmental knowledge.

The second alternative notation is still textual, but it defines a strict order of evaluation that can be followed by the user. It also replaces the cryptic control characters of the regular expression with English instructions. An example, logically identical to Figure 13.7, is given in Figure 13.8.

The third alternative notation is declarative, as in conventional regular expressions, but it uses graphical cues in place of control characters. These cues are easily distinguished from the characters in the search expression, both visually and semantically. Some of the cues—spatial enclosure, for example—are so familiar that an explanation seems redundant. Nevertheless, participants in the experiment were provided with a legend defining the meaning of each graphical element. An example of this notation is shown in Figure 13.9(a), and the explanatory legend is in Figure 13.9(b).

The final alternative notation is both graphical and procedural. It might be regarded as a state transition diagram, typical of those used in computer science classes in which regular expressions are taught in terms of finite state automata, or for teaching language grammars. Participants in this experiment, however, treated the notation as an imperative flowchart. An

FIGURE **13.7**

```
(0|0044)1223 [356][0–9]+
```

Regular expression defining a set of phone numbers.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:21 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **259**

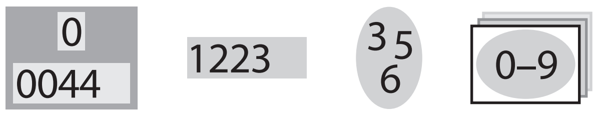
Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE **13.8**

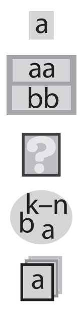
Find one of the following:
  a) either the sequence "0" or
  b) the sequence "0044"
followed by the sequence "1223"
followed by any one of these
characters: "3" or "5" or "6"
followed by at least one, possibly more,
of the following:
  -any one of these characters: any
  one from "0" to "9"

Procedural expression defining the same set as that in Figure 13.7.

FIGURE **13.9**



(a)



a	boxes group sequences together
aa / bb	means either the sequence aa, or the sequence bb can go here
?	means any character can go here
k–n b a	means that one of the characters a, b or k..n (k,l,m,n) can go here
a	means that "a" must occur at least once but possibly more times

(b)

Visual declarative expression defining (a) the same set as that in Figure 13.7, and (b) legend defining notation C.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:29 PM
TNT Job Number: **[002564]** • *Author:* [**Lieberman**] • *Page:* **260**

Color profile: Generic CMYK printer profile
Composite Default screen

example of the procedural graphical notation is shown in Figure 13.10(a), and the explanatory legend in Figure 13.10(b).

13.5.2 Method

The participants in the evaluation experiment were thirty-nine postgraduate students from Oxford and Cambridge Universities, studying a wide range of arts and science disciplines. None were previously familiar with regular expressions, but this population clearly has a high level of intelligence and might be expected to learn principles of programming more quickly than average.

Each participant was given an instruction booklet. The first two pages described the experiment and presented the four different notations. This introduction did not refer to programming, which has been found to intimidate participants in previous experiments on programming notations. Instead, it described the expressions as being experimental formats for use in Internet search.

The following twelve pages in the experiment booklet presented twelve different tasks, all chosen to represent typical regular expressions that might be constructed to represent user abstractions. The tasks included identification of post codes, telephone numbers, market prices, food ingredients, car license numbers, examination marks, email addresses, and Web pages. Each page presented a single regular expression in one of the four formats and five candidate text strings that might or might not match the expression. The participant was asked to mark each of the five candidates with a tick or a cross to show whether it would be matched by this regular expression. Participants also used a stopwatch to record the amount of time that they spent working on each page.

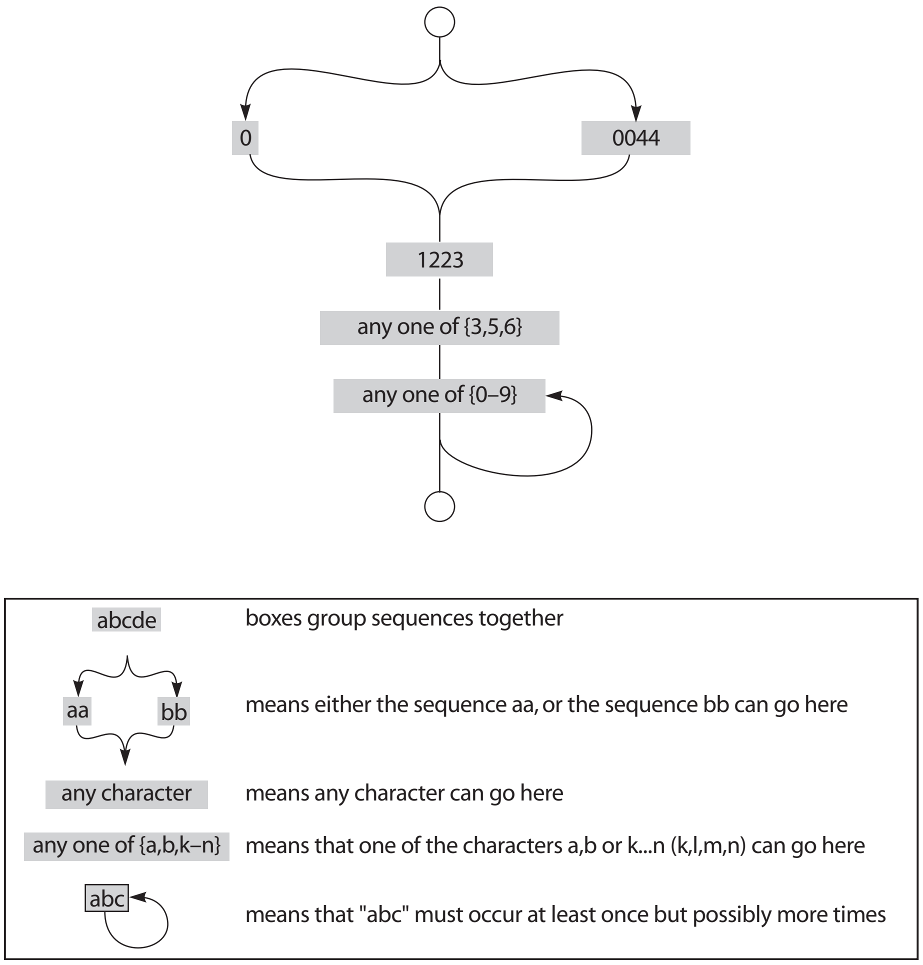
The twelve tasks were divided into six pairs. Within each pair the structures of the two expressions were made as similar as possible, then disguised by using examples from different fields (e.g., post codes vs. car registrations). A different notation format was used for each half of the pair. The six pairs of tasks thus allowed direct comparison of all combinations of the four regular expression notations: format A with format B in one pair, A-C in another, A-D, B-C, B-D, and C-D. For each participant in the experiment it was therefore possible to compare their performance on similar tasks using each pair of alternative notations. Performance comparisons were made according to two measures: the completion time for each page and the accuracy of responses for that page.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:30 PM

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 13.10


0
0044
1223
any one of {3,5,6}
any one of {0–9}
abcde
boxes group sequences together
aa
bb
means either the sequence aa, or the sequence bb can go here
any character
means any character can go here
any one of {a,b,k–n}
means that one of the characters a,b or k...n (k,l,m,n) can go here
abc
means that "abc" must occur at least once but possibly more times

Visual procedural expression defining (a) the same set as that in Figure 13.7 and (b) legend defining notation D.

The assignment of notational formats to pairs and to individual tasks was varied across all participants, as was the presentation order for the different formats. Each participant carried out three tasks using each of the four notations, but every participant had a different assignment of notations to tasks.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:34 PM
TNT Job Number: [002564] • Author: [Lieberman] • Page: 262

Color profile: Generic CMYK printer profile
Composite Default screen

TABLE 13.1

Overall performance results.

	Time (s)	*N errors*
Conventional regular expression	117.6	60
Procedural text	106.7	48
Declarative graphic	86.1	48
Procedural graphic	86.2	38

13.5.3 Results

The notational format used to carry out the tasks had a highly significant effect on performance, *F*(38,1) = 26.8, $p < .001$. As shown in Table 13.1, all the alternative notations were completed more quickly on average than conventional regular expressions. Furthermore, the two graphical formats resulted in fewer errors.

More detailed analysis shows that the use of a graphical format has a significant effect on completion time: the average completion time for all graphical notations was 86.1 versus 112.1 seconds for the two textual notations, *F*(38,1) = 26.0, $p < .001$. In contrast, the mean difference between the two declarative (101.8 s) and the two procedural (96.4 s) notations was nonsignificant.

An investigation of the individual pairings across all participants confirmed that there were statistically significant improvements in performance, first when using the procedural graphical format rather than the procedural text format and, second when using the declarative graphic format rather than conventional regular expressions, *t*(39) = 2.57, $p < .02$ and *t*(39) = 5.18, $p < .001$, respectively.

With notational conventions such as these, it is reasonable to ask whether more verbose notations like the procedural text might be appropriate for novices because they are easier at first sight, even though their diffuseness might make them inconvenient after more practice (Green and Petre 1996). In fact, the terse notations of languages such as C or Perl are justified by the complementary argument—that the verbose prompts needed by novices are not appropriate for expert users. A further analysis therefore compared performance on the first task encountered by each participant and on the last six tasks, to test whether any notation provides disproportionate early advantages while being slower with practice. The

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

TABLE 13.2

Performance for first and later tasks.

	Mean time on first task (s)	*Percentage wrong in first task*	*Mean time on last six tasks*	*Percentage wrong in last six tasks*
Conventional regular expression	198	64	104	47
Procedural text	207	44	82	37
Declarative graphic	110	25	77	47
Procedural graphic	123	9	71	27

results in Table 13.2 show that procedural text suffers an even greater disadvantage in speed when it is encountered first, and it is still not as accurate as the graphical alternatives. Furthermore, a comparison of performance speed relative to the experimental presentation order found that the most highly correlated improvement in performance over the course of the experiment was for the procedural graphic notation, $r = .41$, $p < .001$. This format was thus the most accurate initially, almost the fastest initially, and still provided the greatest improvement in performance with further practice. The declarative graphical format, on the other hand, appears to have been more error prone toward the end of the experiment.

13.5.4 Discussion

It is clear that graphical notations provide a large improvement in usability over conventional regular expressions for typical comprehension tasks. Clearer text formats that use typographic devices such as indenting and have interpretative information included in the notation may perform slightly better overall. But this slight advantage does not reduce the number of errors, and there is no clear advantage for first-time users.

In fact, the format that is the least error-prone overall also provides the greatest improvements in usability with practice. It has the disadvantage common to many graphical notations: it requires far more screen space than conventional regular expressions. For this reason, the declarative graphical format may be more effective in practical programming applications. It still provides large improvements in usability over the conventional notation, and it is sufficiently compact that it can be used *in situ,* in place of

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:35 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **264**

Color profile: Generic CMYK printer profile
Composite Default screen

conventional regular expressions. The last part of this chapter describes a prototype text editor using this notation.

13.6 An Integrated Facility for Regular Expression Creation

This section describes an approach toward integrating all of these elements in a text-processing environment such as a word processor. It applies the declarative graphical format that was evaluated in the previously described experiment, and it integrates this into the user's environment so that regular expressions can easily be created from examples. The graphical representation of the regular expression is displayed continuously and is updated in response to each selection of a positive or negative example. The regular expression is overlaid on the text window, so that the direct correspondence between the regular expression and the most recently selected string can be indicated via superimposed graphical links. The simple syntax of the representation means that it can be made partially transparent, so that it is completely integrated into the task context.

13.6.1 Visual Integration with Data

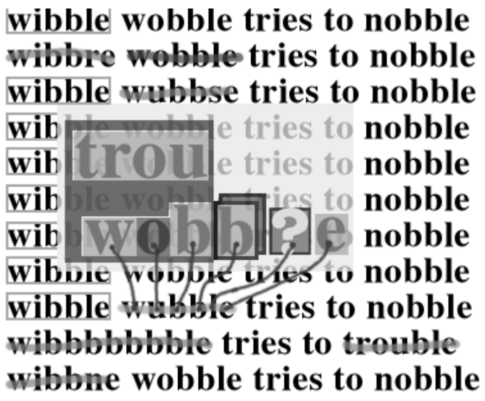
This integration is further enhanced by a simple correspondence between the color of selected example strings in the task domain and coloring of the elements of the visual representation. Required parts of the regular expression are colored green, and parts that must not occur (e.g., excluded character sets) are colored red. This creates a visual link to the green and red colors that are used to highlight positive and negative examples in the text and also to the green outline displayed around all members of the current selection set.

The resulting visual appearance for SWYN is shown in Figure 13.11. The structure of the displayed regular expression is indicated by simple blocks of color, and alternate subexpressions are linked by a containing colored region. Only two special syntactic elements are used: a wild card character to represent potential character sets and a style of decorative border to indicate repetition. Both use existing conventions—the wild card element is represented by a question mark, and the repetition border uses the conventional visual cue of a "stack" of cards.

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 13.11



wibble wobble tries to nobble
wibbre wobble tries to nobble
wibble wubbse tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wobble tries to nobble
wibble wubble tries to nobble
wibbbbbbble tries to trouble
wibbne wobble tries to nobble

User interface for specification and display of regular expressions. The word "wobble" on the second row from the top is annotated in red, and the box around the letter "o" in the center of the screen is red. (This greyscale reproduction obscures the red and green annotations. See color plate XX in the color insert for full color image.)

The regular expression is also directly related to the user's most recent action by drawing correspondence lines between the letters of the most recently selected example string and the elements of the visual representation. This allows the user to see immediately what effect each new example has had on the inferred result.

13.6.2 Modification of the Regular Expression

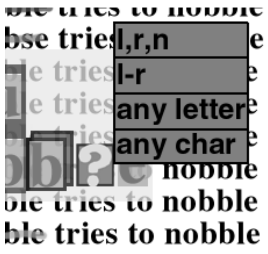
In addition to refining the regular expression by selecting further positive and negative examples, the SWYN visual expression supports two more specialized ways to modify the regular expression. The first of these is by direct manipulation of the visual expression itself. The most important direct manipulation facility supported by the currently implemented algorithms is the ability to select elements of the regular expression and redefine them. As described earlier, the induction algorithm assumes very general character sets when reducing the expression graph. However, the actual characters that were used to infer the graph are recorded as annotations to the graph nodes. If the user wishes to review any character set, he or she can click on that element in the expression in order to see a list of possible interpretations that could be drawn from the original examples. This list could be

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:36 PM
TNT Job Number: [002564] • Author: [Lieberman] • Page: 266

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 13.12



Modifying the expression directly by selecting an intended character class.

presented as a pop-up menu, as shown in Figure 13.12, so the user can select the desired interpretation. Any direct modification of the regular expression will, of course, result in an immediate update of the current selection set within the main text display.

Once the probabilistic inference algorithm described earlier has been incorporated into SWYN, the system will also support active learning by identifying boundary cases and marking them for the user's attention. The user will then be free to refine the current inferred expression by classifying the boundary case, directly modifying the elements of the expression, or simply proceed on the basis of current selections. The result should be both powerful and natural to use, clearly showing the advantages of integrating principles of visual design and direct manipulation into a PBE system. Future work on SWYN will include an empirical investigation of the usability of the novel interaction techniques described here. This will consider both the selection of positive and negative examples to construct a regular expression and the modification of that expression to refine specific boundary conditions or intended character classes.

13.7 Conclusions

The SWYN project aims to help users create powerful abstractions through programming by example. Rather than emphasizing sophisticated inference algorithms, it has applied a relatively simple algorithm for inference of regular expressions from examples but combined it with thorough design for usability. This has taken into account both Green's Cognitive

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:37 PM

Color profile: Generic CMYK printer profile
Composite Default screen

Dimensions of Notations framework and also the application of direct manipulation principles to the domain of abstraction creation.

The consequences for the system design have been that the results of the inference are made visible to the user—the inferred abstraction and the effects of the abstraction within the task domain. The inference results are displayed using a novel visual formalism that is both motivated by sound theoretical principles and verified in experimental evaluation.

This visualization, the approach to identifying examples in the user interface, and the heuristic algorithms used for inference mean that all user actions have incremental effects whose results are immediately visible. Users can both predict and observe the results of their actions, and either refine their abstractions or correct them accordingly. The result is a tool that, although it has a rather specialized purpose, exemplifies many important future emphases for the development of PBE systems.

Acknowledgments

Kim Marriott first suggested regular expressions as an experimental topic for investigating direct manipulation of visual expressions in programming. Thomas Green, Jonathan Pfautz, and Kerry Rodden have given useful feedback on the form of the regular expression visualizations and also on experiment design. Participants in the experiment were volunteers from the choirs of Darwin College, Cambridge, and Wolfson College, Oxford. This research is funded by the Engineering and Physical Sciences Research Council under EPSRC grant GR/M16924, "New Paradigms for Visual Interaction."

References

Angluin, A. 1980. Inductive inference of formal languages from positive data. *Information and Control* 45: 117–135.

Blackwell, A. F. 1996. Metaphor or analogy: How should we see programming abstractions? In *Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group* (January), ed. P. Vanneste, K. Bertels, B. De Decker, and J.-M. Jaques.

Blackwell, A. F., and T. R. G. Green. Investment of attention as an analytic approach to cognitive dimensions. In *Collected papers of the 11th Annual Workshop of the*

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:37 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **268**

Color profile: Generic CMYK printer profile
Composite Default screen

Psychology of Programming Interest Group (PPIG-11), ed. T. Green, R. Abdullah, and P. Brna.

Blackwell, A. F., K. N. Whitley, J. Good, and M. Petre. (in press). Cognitive factors in programming with diagrams. *Artificial Intelligence Review* (special issue on thinking with diagrams).

Christiansen, T., and N. Torkington. 1998. *Perl cookbook.* Sebastopol, Calif.: O'Reilly.

Dietterich, T. G., and R. S. Michalski. 1984. A comparative review of selected methods for learning from examples. In *Machine learning: An artificial intelligence approach,* ed. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. Palo Alto, Calif.: Tioga.

Green, T. R. G. 1989. Cognitive dimensions of notations. In *People and computers V,* ed. A. Sutcliffe and L. Macaulay. Cambridge: Cambridge University Press.

Green, T. R. G., and A. F. Blackwell. 1996. Ironies of abstraction. Presentation at 3rd International Conference on Thinking, British Psychological Society, London, August.

———. 1998. *Design for usability using Cognitive Dimensions.* Invited tutorial at HCI'98, Shefield, U.K., September.

Green, T. R. G., and M. Petre. 1996. Usability analysis of visual programming environments: A "cognitive dimensions" approach. *Journal of Visual Languages and Computing 7:* 131–174.

Green, T. R. G., M. Petre, and R. K. E. Bellamy. 1991. Comprehensibility of visual and textual programs: A test of superlativism against the "match-mismatch" conjecture. In *Empirical studies of programmers: Fourth workshop,* ed. J. Koenemann-Belliveau, T. G. Moher, and S. P. Robertson. Norwood, N.J.: Ablex.

Herrman, E. 1997. *Teach yourself CGI programming with Perl 5 in a week.* Indianapolis: Sams.

Johnson-Laird, P. N. 1983. *Mental models.* Cambridge, Mass.: Harvard University Press.

Kurlander, D., and S. Feiner. 1993. A history-based macro by example system. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Lieberman, H., B. A. Nardi, and D. Wright. 1999. Training agents to recognize text by example. In *Proceedings of the Third ACM Conference on Autonomous Agents* (Seattle, May).

Masui, T., and K. Nakayama. 1994. Repeat and predict—Two keys to efficient text editing. In *Proceedings of Human Factors in Computing Systems, CHI '94.*

Mo, D. H. and I. H. Witten. 1992. Learning text editing tasks from examples: A procedural approach. *Behaviour and Information Technology 11,* no. 1: 32–45.

Modugno, F., and B. Myers. 1993. Graphical representation and feedback in a PBD system. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

Nix, R. P. 1985. Editing by example. *ACM Transactions on Programming Languages and Systems 7,* 4 (October): 600–621.

Shneiderman, B. 1983. Direct manipulation: A step beyond programming languages. *IEEE Computer 16,* no. 8 (August): 57–69.

Wall, L., T. Christiansen, and R. L. Schwartz. 1996. *Programming Perl,* 2d ed. Sebastopol, Calif.: O'Reilly.

Watt, S. 1998. Syntonicity and the psychology of programming. In *Proceedings of the Tenth Annual Meeting of the Psychology of Programming Interest Group* (Milton Keynes, U.K., January), ed. J. Domingue and P. Mulholland.

Witten, I. H. and D. Mo. 1993. TELS: Learning text editing tasks from examples. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

S
R
L

V:\002564\002564.VP
Wednesday, December 20, 2000 2:48:37 PM