CHAPTER

Twelve

Training Agents to Recognize Text by Example

HENRY LIEBERMAN
Media Laboratory, Massachusetts Institute of Technology

BONNIE A. NARDI
AT&T Labs West

DAVID J. WRIGHT
Apple Computer

S
R
L

Abstract

12.1 Introduction

An important function of an agent is to be "on the lookout" for bits of information that are interesting to its user, even if these items appear in the midst of a larger body of unstructured information. But how to tell these agents which patterns are meaningful and what to do with the result?

Especially when agents are used to recognize text, they are usually driven by parsers that require input in the form of textual grammar rules. Editing grammars is difficult and error-prone for end users. Grammex (Grammars by Example) is the first direct manipulation interface designed to allow nonexpert users to define grammars interactively. The user presents concrete examples of text that he or she would like the agent to recognize. Rules are constructed by an iterative process, in which Grammex heuristically parses the example and displays a set of hypotheses, and the user critiques the system's suggestions. Actions to take upon recognition are also demonstrated by example.

12.2 Text Recognition Agents

One service that agents can provide for their users is helping them deal with *semistructured information,* information that contains nuggets of semantically meaningful and syntactically recognizable items embedded in a larger body of unstructured information. Since vast amounts of interesting information are already contained in Web pages, application files, and windows, recognition could prove valuable even if that recognition is only partial (Bonura and Miller 1998). The agent can automatically recognize and extract the meaningful information, and take action appropriate to the kind of information found. Existing agents of this kind are generally preprogrammed with a recognition procedure and can only be extended with difficulty by the end user. The aim of this work is to allow users to teach the agent interactively how to recognize new patterns of data and take actions.

Our approach is to let users specify what they want by example, since learning and teaching by example is easier for people than writing in abstract formalisms. The focus in this chapter will be on agents that recognize text in interactive desktop applications, but the general approach is

FIGURE **12.1**

Apple data Detectors.

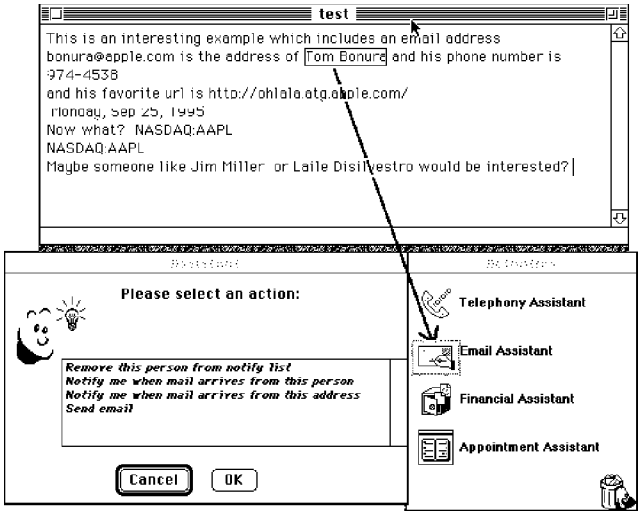
applicable even where the data concerned is graphical or numerical rather than text.

The recent advent of the World Wide Web has sparked renewed interest in text-parsing technology. Parsers are also beginning to be deployed as an integral part of the text-editing facilities available across all computer applications. Examples are Apple Data Detectors (Nardi, Miller, and Wright 1998; see Fig. 12.1) and the Intel Selection Recognition Agent (Pandit and Kalbag 1997). These facilities allow automatic recognition of simple, commonly occurring text patterns such as email addresses, URLs, or date formats. Whether they occur in electronic mail, spreadsheets, or Web pages, URLs can be automatically piped to Web browsers, telephone numbers to contact managers, or meeting announcements to calendars, without explicit cut-and-paste operations. LiveDoc and DropZones (Miller and Bonura 1998; Bonura and Miller 1998) go further, making recognition by the agent more automatic, highlighting recognized objects in place, permitting drag and drop of recognized objects, and allowing actions to operate on more than one object (see Fig. 12.2).

Typically, the set of patterns recognized by the parser is to be programmed by a highly expert user, a grammar writer skilled in computational linguistics. The end user is merely expected to invoke the parser and use its results. However, no set of patterns supplied by experts can be complete. Chemists will want to recognize chemical formulas, stockbrokers will want to recognize ticker symbols, and librarians will want to recognize ISBN numbers, even if these abbreviations have little interest to those outside their own group. Individuals might invent their own idiosyncratic "shorthand" abbreviations for specific situations. We are interested in enabling ordinary end users to define their own text patterns. S

We also expect that many text patterns will be programmed by "gardeners," users who have above-average interest and skill in using applications R L

FIGURE 12.2



LiveDoc and DropZones.

but are still not full-time computer experts or necessarily even have programming skills. Gardeners often serve as informal consultants for a local group of users (Nardi 1993).

12.3 Writing Conventional Grammars as Text

Grammars are the traditional means of expressing a pattern in a stream of text to be identified by a parser. The usual means of defining grammars is by a text file containing rules specified in a Backus-Naur Form (BNF) syntax. The parser takes the grammar file and a target text and returns a tree of symbols used in the grammar and a correspondence between substrings of the text and each symbol. Grammar files sometimes also serve as input for grammar compilers, which output a recognizer that can subsequently be used to parse text with a single grammar.

First, we'll present a very simple example to establish the methodology. Later, we'll return to questions of complexity and scalability. Let's consider trying to teach the computer to understand the format of an electronic mail address. Examples of electronic mail addresses are

```
lieber@media.mit.edu
```

and

```
nardi@apple.com.
```

This pattern is expressed in BNF as a set of context-free rules, each of which tells the computer how to recognize a certain grammatical *category,* or *nonterminal,* as a sequence of specific strings, lexical categories such as "Word" or "Number," or other nonterminals.

Below, `E-Mail-Address` is expressed in terms of `Person` and `Host`. We assume `Word` is a primitive token recognized by the parser.

```
<E-Mail-Address> := <Person> @ <Host>
<Person> := <Word>
<Host> := <Word> | <Word> . <Host>
```

Grammars are difficult for end users for many reasons. Users do not want to learn the syntax of BNF itself, and it is very easy to make mistakes. When the grammar does contain mistakes, parsers generally offer little help for determining which rule was responsible or what interaction between rules caused the problem. Grammar categories themselves can seem very abstract to users, and the effect of a set of grammar rules on concrete examples is not always clear.

Despite the difficulty of writing and editing grammars in BNF text form, users may well be comfortable with the idea of a grammar itself. If you asked a typical user, "What does an email address look like?" you'd likely get the answer, "An email address is the person's name followed by an @, followed by a host." To the further question "And what does a host look like?" would come the answer "A host is any number of words, with periods between the words." This indicates that it is the grammar format and syntax itself, and the complexities of applying them in concrete cases, rather than conceptual difficulties surrounding grammars themselves that are the problem.

12.4 Programming Grammars by Example for More Accessibility

S
Though abstraction is a source of power for grammars, abstraction is also R
what makes grammars difficult for end users. Because people have limited L

short-term memory, they find it difficult to keep track of how abstract concepts map to specific instances when systems grow large. People are simply much better about thinking about concrete examples than they are about abstractions such as grammar rules.

Our solution for dealing with the complexity of grammar definition is to define grammars *by example*. The need for a new text pattern will often become apparent to the user when he or she is examining some text that already contains one or several examples of the pattern. Our approach is to let the user use an example that arises naturally in their work as a basis for defining a grammar rule. Abstraction is introduced incrementally, as the user interacts with the system to provide a description of each example.

The idea for the interface is to have the user interact with a display that shows both the text example and the system's interpretation of that example (either simultaneously or at most one mouse click away). At any time, the user can direct the system to make a new interpretation of an example, or to apply the interpretations it has already learned, and display the result. By keeping a close association between the grammar categories and their effects in concrete examples, the user can always see what the effect of the current grammar is and what the effect of incremental modifications will be.

12.5 Grammex: A Demonstrational Interface for Grammar Definition

Grammex is the interface we have developed for defining grammars from examples. It consists of a set of Grammex rule windows, each containing a single text string example to be used as the definition of a single grammar rule. Text may be cut and pasted from any application. The user's task is to create a description of that example in terms of a grammar rule.

Grammex parses the text string according to the current grammar and makes mouse-sensitive the substrings of the example that correspond to grammar symbols in its interpretation. Clicking on one of the mouse-sensitive substrings brings up a list of heuristically computed guesses of possible interpretations of that substring. The user can select sets of adjacent substrings to indicate the scope of the substring to be parsed. At any time, a substring can be designated as a new example, spawning a new Grammex rule window, supporting a top-down grammar definition strategy.

There is also an overview window, containing an editable list of the examples and rules defined so far. The overall structure of the interface was inspired by the Tinker programming-by-example system (Lieberman 1993).

12.6 An Example: Defining a Grammar for Email Addresses

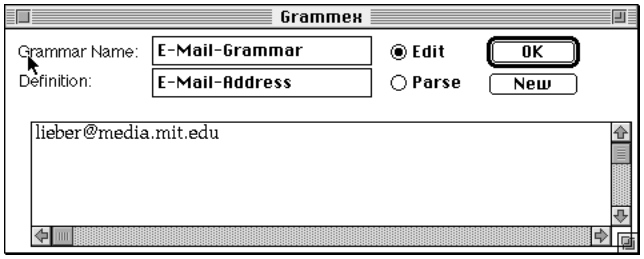
We start defining the pattern for *E-Mail-Address* by beginning with a new example that we would like to teach the system to handle. We get a new Grammex rule window and type in the name for our grammar, *E-Mail-Grammar;* the name of the definition, *E-Mail-Address;* and the example text *lieber@media.mit.edu* (Fig. 11.4).

The Grammex window has two modes: *Edit* and *Parse*. In *Edit* mode, the bottom view functions as an ordinary text editor; the user can type in it or cut and paste text from other windows as the source of the example.

In *Parse* mode, Grammex tries to interpret the text in the example view, and the user can interactively edit the interpretation. Grammex makes pieces of the text mouse-sensitive. Initially, *lieber*, *@*, *media*, ., *mit*, ., and *edu* are identified as separate pieces of text, using the parser's lexical analysis. Each displays a box around it. Clicking on a piece of text brings up a pop-up menu with Grammex's interpretations of that piece of text. In Figure 12.5, the user clicks on *mit*.

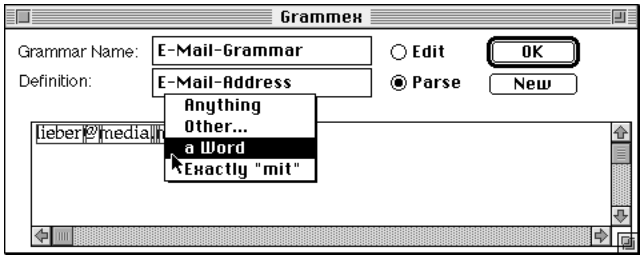
Grammex displays several interpretations of the chosen text, "mit". In the context of an email address, "mit" could be described as being exactly the string "m", followed by "i", then "t", or as an example of any word (string of alphanumeric characters), or as anything, meaning that any string could take the place of "mit". The default interpretation for a string depends first of all on its lexical category. In *Figure 12.5,* the string *mit* is a sequence of alphabetic characters, so the default interpretation is *a Word*. For the punctuation character @, the default interpretation is exactly that string. The user may also select a preexisting grammar to use for the default interpretation. The *Other* option leads to a dialogue box that allows options specific to the

FIGURE 12.3



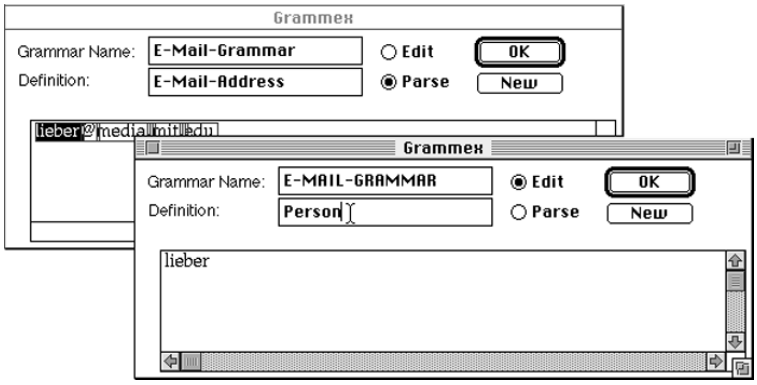
"lieber@media.mit.edu is an example email address.

FIGURE 12.4



Interpretations of the string mit.

FIGURE 12.5



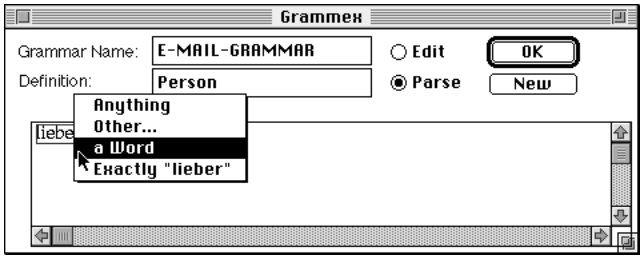
lieber as an example of a Person.

kind of object recognized (e.g., for numbers one can specify a range, for strings a length, etc.).

12.6.1 Top-Down Definition

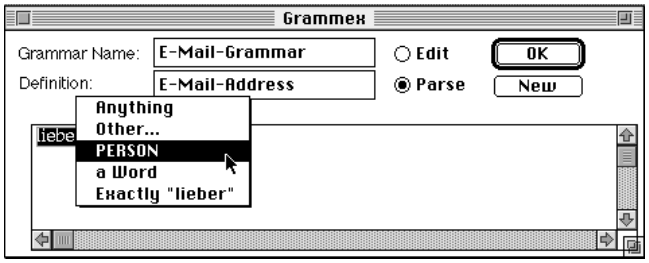
We start explaining to Grammex the structure of an email address by explaining the meaning of the string "lieber," which represents the *Person* part of an email address (Fig. 12.6). We shift-select the string *lieber,* which highlights it, then select the *New* button, which spawns a new window

FIGURE 12.6



A Person *as* a Word.

FIGURE 12.7



Verifying that lieber is recognized.

containing just the string "lieber" and the name of our grammar, *E-Mail-Grammar.* We type in the name of the new definition, *Person.*

For *Person,* we accept the default interpretation of *a Word* (Fig. 12.7). This tells the system that any word can be interpreted as being a *Person.* This illustrates a *top-down* style of grammar definition. Underneath the general goal of explaining to the system how to understand the text "lieber@media.mit.edu," we establish the subgoals of explaining "lieber" as a *Person* and, later, "media.mit.edu" as a *Host.* Alternatively, we could also adopt a bottom-up style of definition, starting with *Person* and *Host,* and only then passing to the full email address.

Now, when we return to defining an email address, "lieber" has an additional possible interpretation, that of a *Person* (Fig. 12.8).

12.7 Rule Definitions from Multiple Examples

The concept of a Host is more complex than that of a person, because we can have hosts that are simply names, such as the machine named "media", or we can have hosts that consist of a path of domains, separated by periods, such as "media.mit.edu". Thus, the definition for Host requires two examples: one of each important case.

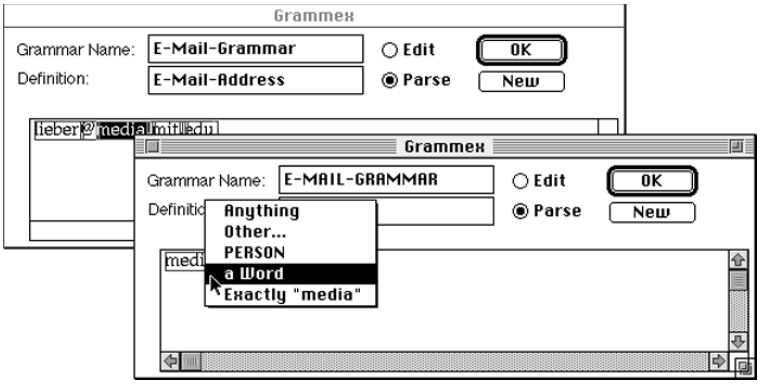
We describe "media" as being an example of a *Host* being a single word, in the same way we did for "lieber" as a *Person*. Note in Figure 12.9 that when we choose the word "media," the possible interpretation (plausible, but wrong) of "media" being a *Person* crops up.

Note that if we wanted to describe a domain as an enumerated type ("edu", "com", "mil", "org", etc.), we could type in each of these as examples and choose the *Exactly* option for each one, such as "Exactly "edu"."

12.7.1 Definition of Recursive Grammar Rules

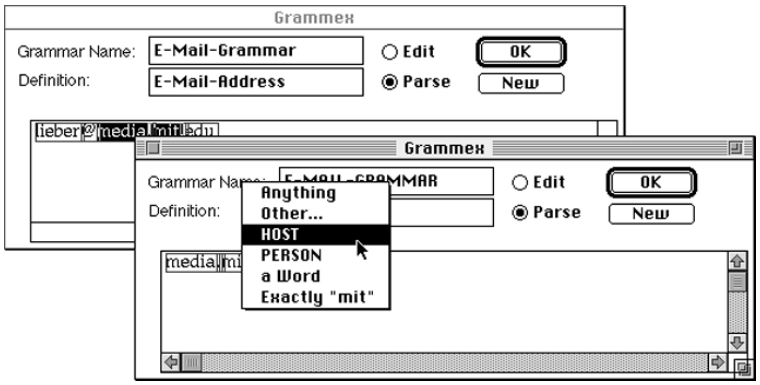
The second example for a *Host* describes the case where there is more than one component to the host name—for example, "media.mit". We select the substring "media.mit" from our original example, "lieber@media.mit.edu", and invoke *New*.

FIGURE 12.8



"media" as a Host.

FIGURE **12.9**



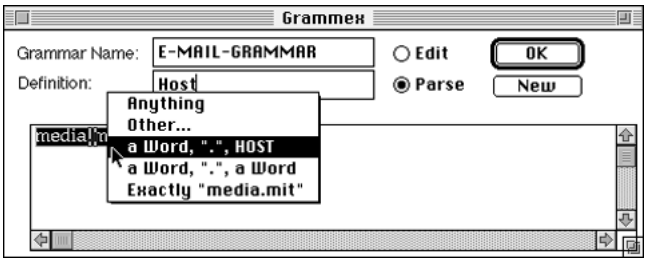
Recursively defining a Host.

The default interpretation of "media.mit" would be as *a Word,* followed by a period, followed by another *Word*. However, while this is a possible interpretation, it does not describe the general case in such a way as could accommodate any number of host components. For that, we need to express the idea that following a period we could then have another sequence of a word, then another period, then a word; that is, we could have another *Host*. In our example, then, we need to change the interpretation of "mit" to be a *Host* rather than a word, so that we could have not just "media.mit" but also "media.mit.edu", "media.mit.cambridge.ma.us", and so forth. This is done by simply selecting "mit" and choosing the interpretation *Host* from the pop-up menu (Fig. 12.10).

The result is now that if we ask what the interpretation of "media.mit" is, we get *a Word,* then ".", then a *Host* (Fig. 12.11). This is an important and subtle idea, the concept of defining a *recursive* grammar definition through multiple examples. One might question whether this idea might not be difficult for "ordinary end users" to grasp.

While it is unlikely that the concept of defining recursive grammars might occur spontaneously to an untrained user, we believe that Grammex offers users a gentle introduction to the concept of recursion. Initially, a user might have to be shown an example, such as defining the host as in our example. Grounding the definition process in concrete examples gives the user a way to motivate the concept and check his or her understanding at each step. Then, the user should be able to easily acquire the skill for

FIGURE 12.10



Verifying the interpretation of media.mit.

defining similarly recursive definitions such as URLs and Unix file paths. Experience with the Logo programming language has shown that even young children can grasp the idea of recursion without difficulty if it is introduced properly.

We also intend to provide a shortcut, the two choices *Optional* and *Repeating,* which will cover the two most common cases where recursive definitions are needed. With this shortcut, one could proceed directly to the example "media.mit" and mark "." and "mit" as Repeating. This will generate two definitions automatically, one for "media" and one for "media.mit", with the same effect as the steps presented earlier.

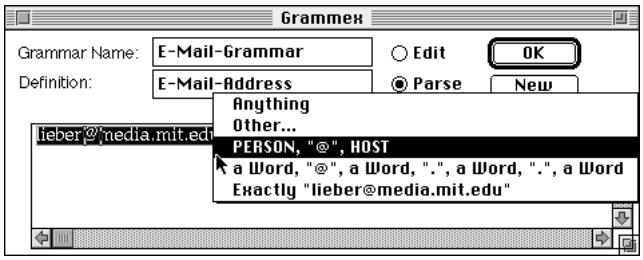
Finally, note that no great harm is done even if the user simply accepts the initial interpretation of "media.mit" as a Word, ".", then a Word. That definition will only be good for two-component Host names. Then "media.mit.edu" could be presented as a separate example for three-component names. Two-, three- and four-component names probably constitute 95 percent of host names, which might be good enough for most users that the general case might not be worth bothering about.

Returning to our original example, we can now express the description of "lieber@media.mit.edu", verifying that it has been successfully recognized by Grammex as a *Person@Host* email address (Fig. 12.12).

12.7.2 Managing Sets of Rule Definitions

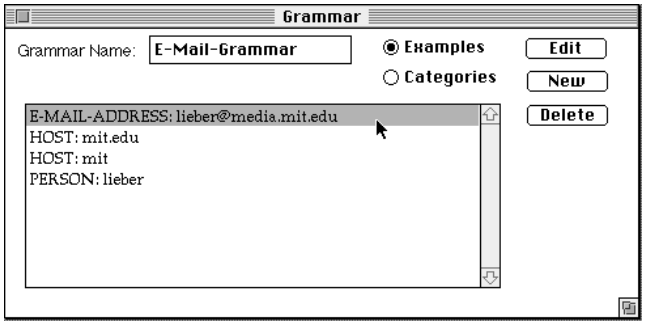
Grammex's Grammar Window holds a list of definitions for the current grammar. Each definition defines a grammar category and has an example of the category. Figure 12.13 presents a grammar window for the email grammar.

FIGURE 12.11



Verifying lieber@media.mit.edu.

FIGURE 12.12



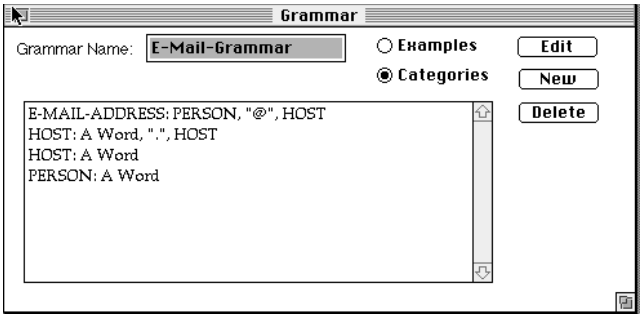
Email grammar, Examples *view.*

In the *Categories* view, each definition is represented by its sequence of categories, rather than by its example (Fig. 12.14). Seeing both views shows the relationship between the example and its description. In either view, double-clicking on an entry results in editing that entry.

12.7.3 Complexity and Scalability

Although the email address example presented is very simple, the methodology itself is very general and can define grammars of arbitrary complexity. The grammar formalism used by Grammex is equivalent to context-free grammars, which are quite general and can be used to describe a wide

FIGURE 12.13



Email grammar, Categories *view.*

variety of text patterns. However, released versions of Apple Data Detectors and Intel's Selection Recognition Agent currently use regular expression recognizers that do not have the full generality of context-free grammars, and some patterns require context-sensitive grammars. Defining grammars is sometimes tricky, and finding exactly the right way to describe examples to cover minor variations in formats sometimes requires some subtlety. Grammex's incremental and iterative approach means that complex grammars can be built up little by little. When an example is found that doesn't yield the desired result, that example can always be fed as input to define or modify a rule.

12.7.4 Defining Actions by Example

To make the paradigm of training agents by example complete, we would also like to define by example the actions to be taken upon recognition of a certain pattern. Previous recognition agents such as Data Detectors and Selection Recognition Agent only allow choosing from a fixed set of actions, provided in advance by someone with expertise in programming. In the case of Data Detectors, actions are programmed in the scripting language AppleScript. LiveDoc's DropZones provides for more sophisticated computation in action selection, but actions still must be programmed in advance. We would like to define the actions just at the time when the user demonstrates examples of the text to be recognized. S

Our approach is to use ScriptAgent (Lieberman 1998), a programming-by-example system for the scripting language AppleScript. When the user R L

demonstrates an example to be recognized, the system goes into a "recording" mode in which the user can demonstrate actions using other applications. For example, after we show how to recognize the email address *lieber@media.mit.edu,* the system pushes the recognized text onto the system's clipboard, and we can then demonstrate what to do with it. For example, we can enter the email program, select the *send message* operation, and paste *lieber@media.mit.edu* into the *To:* field of the message window. The system generates an AppleScript program to represent the action, generalized so it can work on any example, not just the one presented.

The clipboard interface, while convenient for actions that simply use the entire recognized text, does not allow direct access to the recognized subcomponents (e.g., user name and host name). We are experimenting with ways of allowing this, but the problem is to find one that integrates well with current Macintosh interface conventions.

The biggest problem with recording actions by example is that it depends on the aspect of applications being "recordable" (reporting user actions to the agent), and to date, very few Macintosh applications are fully recordable.

An end run around the recordability problem is to use the technique of *examinability* (Lieberman 1998), in which the agent polls the state of the applications and uses similarity-based learning to infer user actions. In Lieberman (1998), we discuss an application that achieves recordability of several specific applications (a calendar program and a spreadsheet program) by polling the states of each application (current calendar displayed and current spreadsheet displayed, respectively) and comparing states to induce the operations performed.

12.8 Future Work: Using Grammar Induction to Speed Up the Definition Process

Automatic induction of grammar rules from examples has been a well-studied topic in machine learning (see Langley 1987 for a survey and techniques). Natural language researchers study the topic to try to come to an understanding of how children learn the grammar of English by only hearing examples of adult speech and only occasionally hearing negative examples or receiving explicit instruction in grammar. What is amazing is both the difficulty of the problem and the success rate achieved!

Grammar induction is also studied in finite-state automata, data compression, and data mining applications. Typically, though, the algorithms assume that the examples are processed in a "batch" mode, rather than

considering how a user might want to interact with the grammar definition process on the fly.

We have initially been quite conservative in our use of inference techniques, but using Grammex as a "front end" for some of these induction techniques holds promise for speeding up the grammar definition process. Essentially, a grammar induction algorithm could be used to generate and narrow down more intelligently the choices presented in Grammex's pop-up menus. The user need present fewer examples, and the system could even generate examples for the user's approval. Cima, discussed in the next section, represents a step in this direction.

12.9 Related Work

To our knowledge, Grammex is the first direct-manipulation interface to address the question of interactive definition and editing of grammars for end users. The closest work to this is David Maulsby's Cima (Maulsby 1994; Maulsby and Witten 1995), whose goal was also to generate grammatical patterns from presentation of concrete textual examples. Maulsby's work concentrated on developing a more sophisticated inference procedure for determining which features of the text are relevant. However, the proposed user interface for Cima consisted of simply presenting a sequence of examples, and it did not allow the interactive generalizing and specializing of parts of the rules and substrings, as does Grammex. Cima's inference techniques could also profitably be applied within Grammex's user interface framework. Tourmaline (Myers 1993) induced stylistic patterns of text formatting (e.g., fonts, sizes) rather than grammar rules, from examples.

Grammex's approach is strongly related to a similar approach for generating procedures in a programming language, called *programming by example* or *programming by demonstration* (Cypher 1993). In programming by example, the user demonstrates a sequence of operations in an interactive interface, and the system records these steps and the objects they were performed on. Machine learning techniques are used to generalize the program so that it works on examples analogous to the ones originally presented.

In our approach, the examples are sample text strings, and the programs are grammar rules. Many programming-by-example systems use some form of heuristic inference to infer generalized descriptions of the operations used to compute the examples. Unlike programming systems, though, grammar rules themselves can be thought of as generalized descriptions of

the examples. We solve the inference problem by heuristically "running the grammar backward" to suggest possible generalizations of a given example. The user chooses from a set of plausible generalizations or proposes his or her own generalizations.

12.10 Conclusion

This chapter has presented Grammex, a demonstrational interface for defining rules for simple context-free grammars from concrete examples. A new rule is defined by directing the system to guess an interpretation of an example string, based on the current grammar, and then modifying this guess using operations that generalize or specialize parts of the rule. By keeping a close association between the grammar rules and their consequences in specific examples of interest to the user, we can give the user the power of grammar and parsing formalisms, without the abstraction and syntactic complications that prevent them from being easy to use.

Acknowledgments

Bonnie Nardi was at Apple Computer at the time of this project, and Henry Lieberman was a consultant to Apple. Jim Miller deserves thanks for providing support for the work on this project. Tom Bonura provided valuable insights and contributed code for the editor. Bob Strong contributed the code for the parser. Lieberman's research was sponsored in part by Apple Computer, British Telecom, IBM, Exol spA, the European Community, the National Science Foundation, the Digital Life Consortium, the News in the Future Consortium, and other sponsors of the MIT Media Laboratory.

References

Bonura, T., and J. Miller. 1998. Drop Zones: An extension to LiveDoc. *SigCHI Bulletin* 30, no. 2 (April): 59–64.

Cowie, J. and W. Lehnert. 1996. Information extraction. *Communications of the ACM* 39, no. 1 (January).

Cypher, A., ed. 1993. *Watch what I do: Programming by demonstration.* Cambridge, Mass.: MIT Press.

Langley, P. 1987. *Machine learning and grammar induction. Machine Learning Journal,* 2: 5–8.

Lieberman, H. 1993. Tinker: A programming by demonstration system for beginning programmers. In *Watch what I do: Programming by Demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

———. 1998. Integrating user interface agents with conventional applications. In *ACM conference on intelligent user interfaces* (San Francisco, January).

Maulsby, David. 1994. Instructible agents. Ph.D. diss. University of Calgary, Calgary, Alberta Canada.

Maulsby, David, and Ian H. Witten. 1995. Learning to describe data in actions. In *Proceedings of the Programming by Demonstration Workshop, Twelfth International Conference on Machine Learning* (Tahoe City, Calif., July).

Miller, J., and Bonura, T. 1998. From documents to objects: An overview of LiveDoc. *SigCHI Bulletin* 30, no. 2 (April): 53–59.

Myers, B. Tourmaline: Text formatting by demonstration. In *Watch what I do: Programming by Demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Nardi, B. 1993. *A small matter of programming perspectives on end user computing.* Cambridge, Mass.: MIT Press.

Nardi, B., J. Miller, and D. Wright. 1998. Collaborative, programmable intelligent agents. *Communications of the ACM* (March).

Pandit, M., and Kalbag, S. 1997. The selection recognition agent: Instant access to relevant information and operations. In *Proceedings of Intelligent User Interfaces '97.* New York: ACM Press.

S
R
L