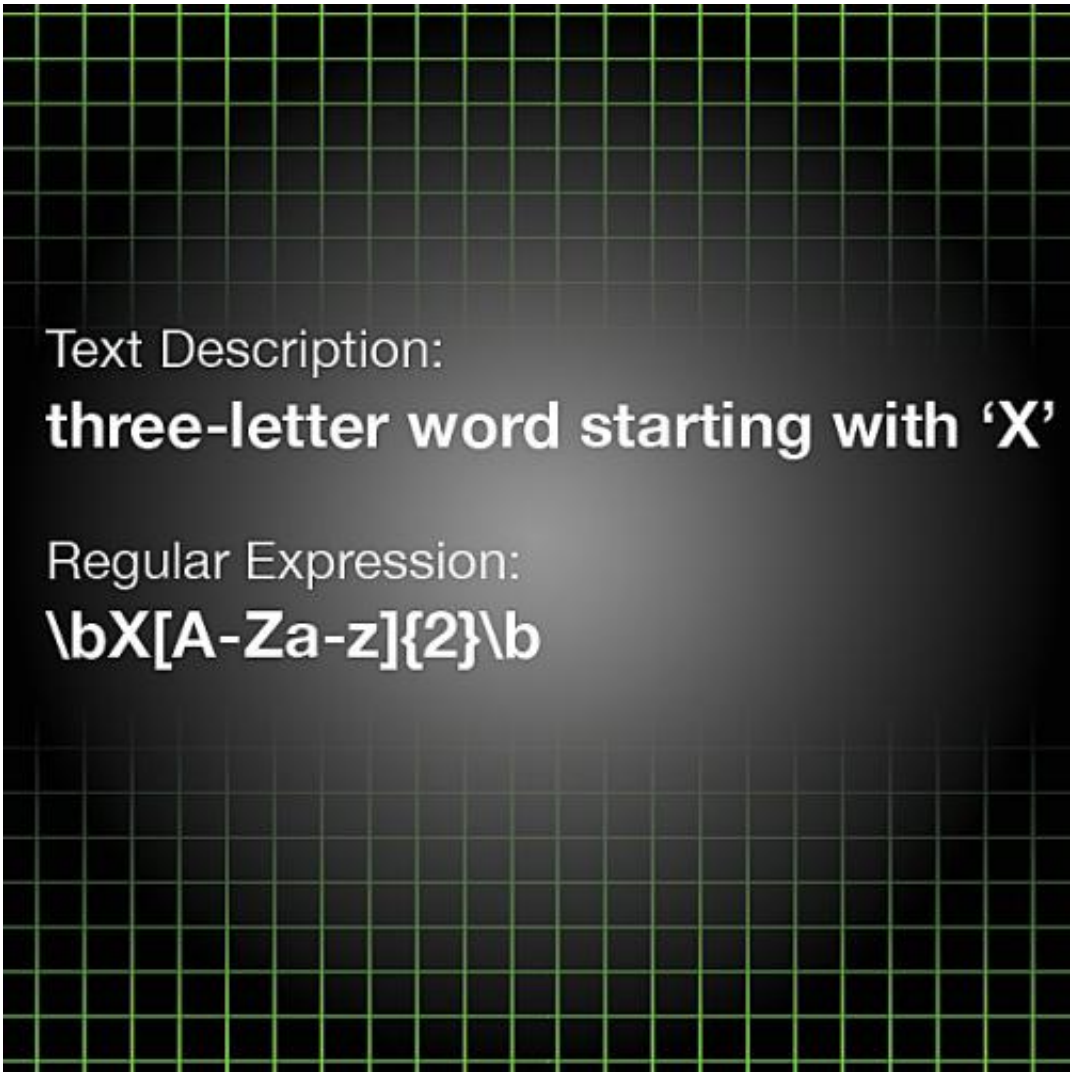
Writing programs using ordinary language

July 11 2013, by Larry Hardesty



A new algorithm can automatically convert natural-language specifications into "regular expressions" — special-purpose combinations of symbols that allow very flexible searches of digital files. Credit: CHRISTINE DANILOFF

In a pair of recent papers, researchers at MIT's Computer Science and Artificial Intelligence Laboratory have demonstrated that, for a few specific tasks, it's possible to write computer programs using ordinary language rather than special-purpose programming languages.

The work may be of some help to programmers, and it could let nonprogrammers manipulate common types of files—like word-processing documents and spreadsheets—in ways that previously required familiarity with programming languages. But the researchers' methods could also prove applicable to other programming tasks, expanding the range of contexts in which programmers can specify functions using ordinary language.

"I don't think that we will be able to do this for everything in programming, but there are areas where there are a lot of examples of how humans have done translation," says Regina Barzilay, an associate professor of computer science and electrical engineering and a co-author on both papers. "If the information is available, you may be able to learn how to translate this language to code."

In other cases, Barzilay says, programmers may already be in the practice of writing specifications that describe computational tasks in precise and formal language. "Even though they're written in natural language, and they do exhibit some variability, they're not exactly Shakespeare," Barzilay says. "So again, you can translate them."

The researchers' recent papers demonstrate both approaches. In work presented in June at the annual Conference of the North American Chapter of the Association for Computational Linguistics, Barzilay and graduate student Nate Kushman used examples harvested from the Web to train a computer system to convert natural-language descriptions into so-called "regular expressions": combinations of symbols that enable file searches that are far more flexible than the standard search functions

available in desktop software.

In a paper being presented at the Association for Computational Linguistics' annual conference in August, Barzilay and another of her graduate students, Tao Lei, team up with professor of electrical engineering and computer science Martin Rinard and his graduate student Fan Long to describe a system that automatically learned how to handle data stored in different file formats, based on specifications prepared for a popular programming competition.

Regular irregularities

As Kushman explains, computer science researchers have had some success with systems that translate questions written in natural language into special-purpose formal languages—languages used to specify database searches, for instance. "Usually, the way those techniques work is that they're finding some fairly direct mapping between the natural language and this formal representation," Kushman says. "In general, the logical forms are handwritten so that they have this nice mapping."

Unfortunately, Kushman says, that approach doesn't work with regular expressions, strings of symbols that can describe the data contained in a file with great specificity. A regular expression could indicate, say, just those numerical entries in a spreadsheet that are three columns over from a cell containing a word of any length whose final three letters are "BOS."

But regular expressions, as ordinarily written, don't map well onto natural language. For example, Kushman explains, the regular expression used to search for a three-letter word starting with "a" would contain a symbol indicating the start of a word, another indicating the letter "a," a set of symbols indicating the identification of a letter, and a set of symbols indicating that the previous operation should be repeated twice.

"If I'm trying to do the same syntactic mapping that I would normally do," Kushman says, "I can't pull out any sub-chunk of this that means 'three-letter.'"

What Kushman and Barzilay determined, however, is that any regular expression has an equivalent that does map nicely to natural language—although it may not be very succinct or, for a programmer, very intuitive. Moreover, using a mathematical construct known as a graph, it's possible to represent all equivalent versions of a regular expression at once. Kushman and Barzilay's system thus has to learn only one straightforward way of mapping natural language to symbols; then it can use the graph to find a more succinct version of the same expression.

When Kushman presented the paper he co-authored with Barzilay, he asked the roomful of computer scientists to write down the regular expression corresponding to a fairly simple text search. When he revealed the answer and asked how many had gotten it right, only a few hands went up. So the system could be of use to accomplished programmers, but it could also allow casual users of, say, spreadsheet and word-processing programs to specify elaborate searches using natural language.

Opening gambit

The system that Barzilay, Rinard, Lei and Long developed is one that can automatically write what are called input-parsing programs, essential components of all software applications. Every application has an associated file type—.doc for Word programs, .pdf for document viewers, .mp3 for music players, and so on. And every file type organizes data differently. An image file, for instance, might begin with a few bits indicating the file type, a few more indicating the width and height of the image, and a few more indicating the number of bits assigned to each pixel, before proceeding to the bits that actually

represent pixel colors.

Input parsers figure out which parts of a file contain which types of data: Without an input parser, a file is just a random string of zeroes and ones.

The MIT researchers' system can write an input parser based on specifications written in natural language. They tested it on more than 100 examples culled from the Association for Computing Machinery's International Collegiate Programming Contest, which includes file specifications for every programming challenge it poses. The system was able to produce working input parsers for about 80 percent of the specifications. And in the remaining cases, changing just a word or two of the specification usually yielded a working parser.

"This could be used as an interactive tool for the developer," Long says. "The developer could look at those cases and see what kind of changes they need to make to the natural language—maybe some word is hard for the system to figure out."

The system begins with minimal information about how written specifications might correspond to parser programs. It knows a handful of words that should consistently refer to particular data types—the word "integer," for instance—and it knows that the specification will probably describe some data structures that are nested in others: An image file, for instance, could consist of multiple chunks, and each chunk would be headed by a few bytes indicating how big it is.

Otherwise, the system just tries lots of different interpretations of the specification on a few sample files; in the researchers' experiments, the samples, too, were provided on the competition website. If the resulting parser doesn't seem to work on some of the samples, the system varies its interpretation of the specification slightly. Moreover, as it builds more and more working parsers, it becomes more adept at recognizing

regularities in the way that parsers are specified. It took only about 10 minutes of calculation on an ordinary laptop for the system to produce its candidate parsers for all 100-odd specifications.

"This is a big first step toward allowing everyday users to program their computers without requiring any knowledge of programming language," says Luke Zettlemoyer, an assistant professor of computer science and engineering at the University of Washington. "The techniques they have developed should definitely generalize to other related programming tasks."

The two paper are titled "From natural language specifications to program input parsers" and "Using semantic unification to generate regular expressions from natural language."

This story is republished courtesy of MIT News (web.mit.edu/newsoffice/), a popular site that covers news about MIT research, innovation and teaching.

Provided by Massachusetts Institute of Technology