Using Real Examples to Motivate Automata Theory[*]

David P. Bunde
Computer Science Department
Knox College
Galesburg, IL 61401
dbunde@knox.edu

Abstract

Many institutions have a course on automata theory that studies the limits of computation by examining successive computational models, including deterministic finite automata (DFAs), context-free grammars (CFGs), and Turing machines (TMs). Some students resist this material because they see it as overly theoretical, but much of it has important practical applications. In this position paper, we discuss some of these and advocate a view of the course with applications emphasized to provide motivation.

1 Introduction

In the automata theory class included in many CS degree programs, the emphasis is on fundamental questions such as "What is computing?" and "What is computable?". The subject also asks about the power of non-determinism, which is a key question in polynomial-time complexity and the study of algorithms. These are important questions, but students sometimes struggle to see the motivation for this abstract material.

With the issue of motivation in mind, we propose a different approach to teaching automata theory, one which foregrounds applications and practical

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

concerns even as it covers the standard material. This approach recognizes that the material also has important uses in text processing (using regular expressions) and parsing. By emphasizing these applications, we attempt to make the course more accessible to students with an applied focus while still teaching abstract topics like computability and non-determinism.

This paper presents our efforts to follow this approach; we have taught using it several times and endeavor to increase the number and prominence of applications each time. Due to the course's evolving nature and the relative infrequency with which we have taught it (roughly every other year), we do not have formal evidence of the effectiveness of our approach. Thus we present our ideas as a position paper.

The remainder of the paper discusses our ideas roughly in the order in which they are used in a standard course, followed by brief discussions of the context of our course, related work, and possible future directions.

2 Regular languages

We think of an automata theory course as organized into 3 parts, each corresponding to a different class of languages and machines. The first part covers regular languages, represented by regular expressions, deterministic finite automata (DFAs), and non-deterministic finite automata (NFAs). Thus, we begin by presenting applications related to this material.

Practical notation for regular expressions. Regular languages are actually easy to motivate because many programming languages implement a version of them, though with a slightly different notation than in standard automata textbooks. The latter define a regular expression using three base cases: $\varnothing$ (the empty set), ϵ (the empty word), and a single character. Then, regular expressions can be combined using operators for choice ($+$) and repetition ($*$).

Unfortunately, this notation is somewhat different than that used in regular expressions as implemented in practical programming languages. This forms an unnecessary barrier for students who have already used regular expressions and also makes it harder for students learning about them in the class to take their skills outside. Thus, we change the standard notation to something closer to that used in languages such as Python. Specifically, we adopted $|$ rather than $+$ for a choice, but added ϵ and $\varnothing$ since the languages $\{\epsilon\}$ and $\{\}$ are otherwise difficult or impossible to represent. Having introduced regular expressions as a practical tool, it then becomes natural to show that other common regular expression notation is syntactic sugar. For example, other types of repetition available in Python can be represented using our limited set of operators:

Symbol	Meaning	Example	Equivalent to
?	0 or 1 times	a?	a | ϵ
+	1 or more times	a+	aa*
{n}	n times	a{5}	aaaaa
{n,m}	n to m times	a{1,3}	a | (aa) | (aaa)

These become examples done in class or they could be used as homework problems.

Another useful type of syntactic sugar are character classes. For example, [a-z] matches any single lowercase letter and [A-Za-z] matches any letter. Students are quick to see that these expressions can easily be created using a list of the options, but appreciate the concise representation provided by the character class. More puzzling, but equivalent in a finite alphabet, are character classes with negations: [^a-z] matches any single character that is not a lowercase letter. Again, all of these can be used as examples or exercises while giving students more powerful regular expression notation for use either in class or in their programs.

Practical regular languages. Although the syntactic sugar discussed above does not technically increase the power of regular expressions, using it (particularly character classes) does make it reasonable to express regular languages of practical importance. An obvious possibility is programming language identifiers, which have rules like needing to start with a letter. Limited ranges of numbers are also natural and constitute a nice challenge; creating a regular expression for a number with a fixed number of digits (like 2) is easy, but making one for a range of values like 0–255 (one part of an IPv4 address) is surprisingly tricky since the values allowed for each digit depend on the values of the previous digits. One regular expression for numbers in this range (without leading 0s) is

([1-9]?[0-9]) | (1[0-9][0-9]) | (2[0-4][0-9]) | (25[0-5])

The separation into multiple terms prevents (i) leading 0s and (ii) a ones digit above 5 in three digit numbers if the first two digits are 25. Obviously, there are other ways to create regular expressions for this language, but all of them seem to be surprisingly complicated. This is something for the instructor to be careful about; the first time the author used numbers as an example, he asked for the range of a `short` in C (-32,768 to 32,767), which requires a very complicated expression.

Code counting. Our favorite example of using a DFA is writing a program to count lines of code (LOC). Integrated editors will provide a line count, as will utilities such as Unix's `wc`, but their counts include lines that are blank or contain only comments. Instead, we ask students to write a program or give a DFA with output that counts lines while excluding blank lines and Java comments (`//` to ignore the rest of the line and `/*` to ignore until the next `*/`). Consider the following Java code:

```
 1  public class TestProgram \{
 2      public static void main(String[] args) { //comment
 3          /* Comment at head of line */ int var = 0;
 4
 5          /* Hard case //with nested comments and break
 6         mid-comment */ System.out.println("Hello!");
 7
 8          //Here the comments nest the other way /*
 9          var++;
10      }
11  }
```

In this example, lines 5 and 8 contain only comments, while lines 4 and 7 are blank. This leaves 7 lines of actual code. Note that lines of code can contain either kind of comment and that `//` can appear inside a `/*` comment and vice versa. Because of these complications, it is tricky to write a code-counting program without the idea of states. Code counting is not a highly practical problem, but it is a familiar idea for students and the program's output does provide a primitive measure of program complexity. (Despite the flaws of LOC as a measure, the author was asked to write a program to count it in a summer internship; we share this with the students to help motivate the task.)

3 Context-free languages

The second part of an automata course covers context-free languages, represented by context-free grammars (CFGs) and pushdown automata (PDAs). Unlike regular expressions, these are not commonly integrated into programming languages. Grammars are, however, commonly used to represent and parse programming languages. Thus, the examples for this material tend to be drawn from programming constructs.

Printf calls. One of the canonical examples of a language that is context-free but not regular is 0^n1^n. The difficulty of this language comes from ensuring that the numbers of 0s and 1s are equal, something which regular expressions and finite automata are unable to do. The language 0^n1^n distills this key limitation into a simple example, but it is hardly motivating without any context or rationale. Instead, we use a programming-based example: invocations of `printf` where the number of conversion specifications (limited to `%d` for simplicity) in the format string must match the number of other arguments. Again, the key aspect of this language is basically to make a 1-to-1 correspondence between a feature of the first part of the word (occurrences of `%d`) and a feature of the end of the word (the other arguments). Using `printf` makes the example messier since there are other characters to consider (the word "`printf`", parentheses, commas, the variable names, etc), but also much more interesting since checking the number of arguments is a task that the compiler actually has to perform. (Of course, this is not how `printf` is parsed in practice; the parser would simply identify the function name and argument list, with the matching done afterwards.)

Non-regular "regular expressions". For another example, we return to regular expressions. Some implementations of these provide features that are more than syntactic sugar, actually expanding the set of languages that can be represented. For example, Python provides a matching feature; putting part of the expression in parentheses binds whatever matches that part of the expression to a numerically-named variable. These can be accessed using \1, \2, . . . later in the regular expression. Thus, the expression

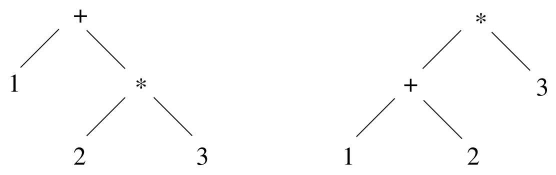
([0-9])x\1

matches words of the form dxd, where d is a decimal digit. Since the parentheses can contain an arbitrary regular expression, it is easy to show that this feature allows Python's "regular expressions" to match at least some languages that are not even context-free. (The basis for a nice homework problem...)

Parsing ambiguity. The next application is parsing arithmetic expressions. A naive way to generate expressions with the four basic operators is with a single rule:

$$E \longrightarrow E + E \mid E - E \mid E * E \mid E \;/\; E \mid \text{number}$$

This rule generates the desired expressions, but can generate parse trees that do not respect operator precedence and associativity. For example, the expression $1 + 2 * 3$ can yield either of the following trees:



The left tree evaluates as $1+(2*3)$ while the right evaluates as $(1+2)*3$. Obviously, the left tree is correct for the standard precedence order. Even without precedence differences, one order is preferable due to operator associativity; the expression $1-2-3$ should evaluate to $(1-2)-3$ rather than $1-(2-3)$. Standard precedence order and associativity can be established by changing the grammar as follows:

$$\begin{array}{lcl} E & \longrightarrow & E+A \mid E-A \mid A \\ A & \longrightarrow & A*B \mid A\ /\ B \mid B \\ B & \longrightarrow & \text{number} \end{array}$$

This is a standard example, with solutions in textbooks and on the web. To a lesser degree, the same is true of the "dangling else" problem (when `if` statements are chained and the compiler needs to select one of them to associate with a trailing `else`). Thus, these examples are shown in class, while homework features an equivalent but less-popular example such as exponentiation (which is right associative) or boolean operators (where left associativity is needed for correct short circuiting).

Some automata theory textbooks discuss this issue under the name *ambiguity*, but typically devote only a few pages to the topic. In many ways it is a programming language topic, but a very low-level one. Note that in a practical context, precedence and associativity are likely to be established by means other than the grammar; compiler generation tools such as flex and bison [3] provide directives to specify them without changing the grammar. That said, they can be established in the grammar itself and doing so illustrates the connection between interpreting computer code and grammars.

Assignment and equality. Considering the need to parse expressions also helps motivate the use of separate symbols for assignment and testing for equality, such as `=` and `==` in C/C++/Java or `:=` and `=` in Pascal. Both assignment and equality testing can be expressed with the English word "equals" and confusion between `=` and `==` has been found to be one of the most common errors by novice programmers learning Java [1], but looking at them from a parsing perspective quickly motivates the use of different symbols.

4 Turing machines

Finally, the third part of a typical automata course uses Turing machines (TMs) to cover recursive and recursively enumerable languages as well as complexity topics such as NP-completeness and the polynomial hierarchy. This is very abstract stuff, but there are still some practical connections to be drawn.

Undecidability of program analysis. A particularly abstract part of this material is reductions, particularly those proving undecidability. A common structure for these reductions involves describing a TM whose input is itself the representation of a TM. The TM being described manipulates this representation and then provides it as input to another TM being simulated. This is confusing on several levels. One source of confusion is just the idea of TMs taking the representations of TMs as input. We have found it helpful to point out that students use something like this all the time, namely a compiler, which is a program that takes the description of a program as input.

Building on this, we try to focus on undecidable problems of practical interest. The Halting problem, deciding whether a program will ever halt, is often presented in practical terms. We take this further with other undecidable problems that come from analyzing programs such as identifying dead code (does any input cause a TM to enter a particular state?).

Historical background for programming terminology. In addition, this material provides historical context for things they may have seen in other CS settings. We include a couple of days on lambda calculus as another attempt to answer to the question "what can be computed?" and its equivalence to TMs while being so different as support for the Church-Turing Thesis. On one hand, lambda calculus is clearly an automata theory topic, albeit one with a different flavor than the rest of the material. On the other hand, "lambdas" are listed as exciting new features of both C++11 and Java 8. Students with experience in functional programming have likely also seen the word there as well as function calls formatted in the same way as lambda expressions.

5 Our Experiences

The genesis of our work on a practically-oriented automata course was as a new assistant professor assigned to teach "Automata theory and programming languages", a course whose creation was motivated by the use of automata in parsing, but which no one had previously taught. The first offering of this course did not include many lectures explicitly combining the two topics, but gradually the perspective discussed above was developed and utilized. Later, the department decided to create separate courses in programming languages and automata theory in order to give more coverage to each area. The new automata-only course has been taught once.

Both versions of the course were taught as upper-division electives, of which 3 are required for the CS major and one for the CS minor. The courses were both taught every other year and typically taken by majors as one of their last courses, though the official prerequisites were just Discrete Mathematics and

CS 2 (Data Structures). Class sizes were 5–12, with an average just below 7.

The combined course was typically taught without a textbook because of the lack of books that covered both the automata and programming languages material. For the automata-only offering, we used a book by Webber [5] which does not directly take our approach, but which is an accessible treatment of the standard automata material. The focus on applications and practical concerns we advocate came through in lecture and assignments. The most awkward change was the use of different notation for regular expressions, but this is mitigated by the students being upperclassmen.

Because the approach described in this paper evolved gradually in a course taught only every other year to relatively few students, we do not have any kind of formal assessment comparing it to a more traditional automata course. Our sense is that explicitly connecting to practical regular expressions is helpful, particular since a couple of students in the class often speak up about how useful regular expressions are when the topic is first introduced. Similarly, the applications of regular expressions seem to be well received. Ambiguity is a somewhat challenging topic since it asks students to relate a grammar to the set of parse trees it creates. Undecidability also remains a challenge, though we believe that mentioning the compiler and focusing on languages that can be interpreted as code analysis tasks gives students a slightly more concrete frame of reference.

6 Discussion

Given the breadth of automata-related applications, we are not the first to notice many of them. Several books explicitly introduce some of these applications. Hopcroft et al. [2] use Unix regular expressions as an example of practical regular expressions. They also include sections on parse trees (including the use of `yacc`, a predecessor to `bison`) and ambiguity. Rich [4] includes nearly 200 pages on applications in the appendices as well as sections on parsing, parse trees, and ambiguity. Neither of these texts fully integrates applications throughout the course as we envision or make changes such as using the practical regular expression syntax, but these books would support the kind of approach we propose here if the practice-oriented material were included in the course. They are also resources that instructors could use to identify interesting applications to include in courses based on other textbooks.

Obviously, much can be done to integrate applications into textbooks and course materials as well as finding other ways to motivate the material. In addition, we are very interested in an assessment of this approach: Do students find our examples more interesting than concise language descriptions such as 0^n1^1? Does this lead them to enjoy the course more or learn more from it?

References

[1] N.C.C. Brown and A. Altadmri. Investigating novice programming mistakes: Educator beliefs vs student data. In *Proc. 10th Ann. Conf. Intern. Computing Education Research (ICER)*, pages 43–50, 2014.

[2] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introdution to automata theory, languages, and computation.* Addison Wesley, 2nd edition, 2001.

[3] J. Levine. *flex & bison.* O'Reilly Media, 2009.

[4] E. Rich. *Automata, computability, and complexity.* Pearson Education, 2008.

[5] A.B. Webber. *Formal language: A practical introduction.* Franklin, Beedle & Associates, Inc., 2011.