PRACTICAL COMPUTATION THEORY*

Charles D. Allison
Utah Valley State College
Orem, UT 84663
801-863-6389
allisoch@uvsc.edu

ABSTRACT

The Theory of Computation is considered essential for all CS undergraduates, yet most of the texts in common use are more suited for graduate-school-bound mathematics majors than today's typical CS student. This paper describes pedagogical techniques that motivate and simplify the presentation of undergraduate topics from the theory of computation.

INTRODUCTION

Once a subject reserved for graduate students, the study of the theory of computation is now entrenched in undergraduate curricula. Many of the textbooks on the subject still belie their graduate school roots by employing a high level of mathematical formalism that is lost on many contemporary CS undergraduates. The preface to one accessible text states:

> "Undergraduate Computer Science majors generally do not speak the language of mathematical symbolism fluently, nor is it important at their level that they do more than try… It is at best a means to an end. To those to whom it is opaque, it is a hindrance to understanding. When this happens it is mathematically dysfunctional and a pedagogical anathema." [1]

This paper catalogs some pedagogical aids that have been found to appeal to the typical CS student that is not comfortable with formal mathematical proofs.

REGULAR LANGUAGES AND FINITE AUTOMATA

It is not uncommon for students to ask early in a course on computation theory, "What is all this good for?" This may be a good time to talk about the usefulness of state machines in problem solving, and how to implement them in a programming language. A motivating

---

example has students design and implement a "state machine with output" to strip C-style comments (`/*…*/`) from source code. Figure 1 shows an automaton with output (Mealy style, with 1-character memory, admittedly taking some notational liberties; drawn with JFLAP [2]) that solves the problem.

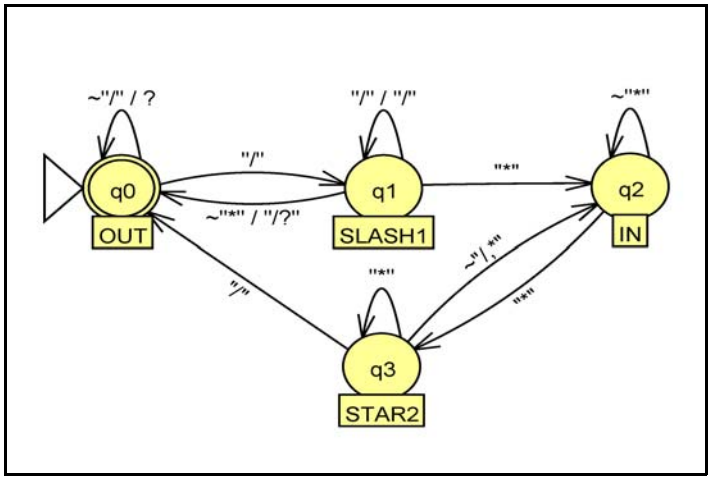

Figure 1 – An Automaton that Filters C-style Comments

Quotes are used around input characters here so that a slash input character is not confused with the conventional slash used to separate input from output on the transition edge. The tilde is used to indicate "anything but" the following character(s), so for example, starting in state q_0 (labeled OUT to mean "outside of a comment"), if the input character is a slash, we go to state q_1, otherwise we stay put and echo the input character (indicated by the meta-character '?'). Similarly, from state q_1, if a slash is read, we output the *previous* slash, because it was not part of the comment. If the input is an asterisk, we move to state q_2, otherwise, we output the original slash that took us to q_1, as well as the current character that takes us back to q_0. A C++ implementation follows in Figure 2.

```
int main() {
    enum State {OUT, SLASH1, IN, ASTERISK2} state;
    state = OUT; // We start outside of a comment
    char c;
    while (cin.get(c))
        switch(state) {
        case OUT:
            if (c != '/')
                cout << c;
            else
                state = SLASH1;
            break;
        case SLASH1:
            if (c == '/')
                cout << c;
            else if (c == '*')
                state = IN;
            else {
                cout << '/' << c;
                state = OUT;
            }
            break;
        case IN:
            if (c == '*')
                state = STAR2;
            break;
```

```
            case STAR2:
                if (c == '/')
                    state = OUT;
                else if (c != '*')
                    state = IN;
                break;
            }
        if (state != OUT)
            cout << "invalid comment syntax\n";
}
```

Figure 2 – A C++ Implementation of the Automaton in Figure 1

TEACHING REGULAR LANGUAGE THEORY

In their seminal 1959 paper [3], Rabin and Scott introduced a technique that can be used to establish many of the theoretical results for regular languages. The idea is to simulate all possible paths through an automaton. For example, one can show that the union of regular languages is regular by constructing the corresponding "combined" automaton. To illustrate, consider the deterministic finite automata (DFA) over the alphabet {a, b} shown in Figure 3

FA_1	A	b
$-x_1$	X_2	X_1
x_2	X_3	X_1
$+x_3$	X_3	x_3

FA_2	A	b
$\pm y_1$	y_3	y_2
Y_2	y_4	y_1
Y_3	y_1	y_4
Y_4	y_2	y_3

Figure 3 – Two Sample Finite Automata

Following Cohen, we use the prefix "-" to indicate the initial state and "+" for accepting states. To construct the union, we form a composite initial state consisting of the set $\{x_1, y_1\}$. At this point we consider where x_1 and y_1 take us with an 'a' as input and then with a 'b', obtaining new composite states $\{x_2, y_3\}$ and $\{x_1, y_2\}$ respectively. Continuing in this fashion, we obtain the result shown in Figure 4.

$FA_1 + FA_2$	A	B
$\pm\{x_1, y_1\}$	$\{x_2, y_3\}$	$\{x_1, y_2\}$
$\{x_2, y_3\}$	$\{x_3, y_1\}$	$\{x_1, y_4\}$
$\{x_1, y_2\}$	$\{x_2, y_4\}$	$\{x_1, y_1\}$
$+\{x_3, y_1\}$	$\{x_3, y_3\}$	$\{x_3, y_2\}$
$\{x_1, y_4\}$	$\{x_2, y_2\}$	$\{x_1, y_3\}$
$\{x_2, y_4\}$	$\{x_3, y_2\}$	$\{x_1, y_3\}$
$+\{x_3, y_3\}$	$\{x_3, y_1\}$	$\{x_3, y_4\}$
$+\{x_3, y_2\}$	$\{x_3, y_4\}$	$\{x_3, y_1\}$
$\{x_2, y_2\}$	$\{x_3, y_4\}$	$\{x_1, y_1\}$
$\{x_1, y_3\}$	$\{x_2, y_1\}$	$\{x_1, y_4\}$
$+\{x_3, y_4\}$	$\{x_3, y_2\}$	$\{x_3, y_3\}$
$+\{x_2, y_1\}$	$\{x_3, y_3\}$	$\{x_1, y_2\}$

Figure 4 – Combining FA_1 and FA_2

The accepting states are those composite states where the **x** or the **y** component was an accepting state to begin with. Since this table represents a simultaneous traversal through the two original automata, their intersection is found by assigning accepting states only where

both the **x** and **y** components accept (in this case, that would be only the composite state **{x_3, y_1}**). This technique constitutes a proof by construction tractable to the majority of students.

The same technique can be used to convert a non-deterministic automaton (NFA) to a deterministic one (DFA). Consider the NFA in Figure 5.

Because of the lambda transition from the initial state, the effective initial state is actually a composite, namely $\{q_0, q_2\}$. Accordingly, we follow the Rabin-Scott technique from $\{q_0, q_2\}$ and see where it takes us. See Figure 6.

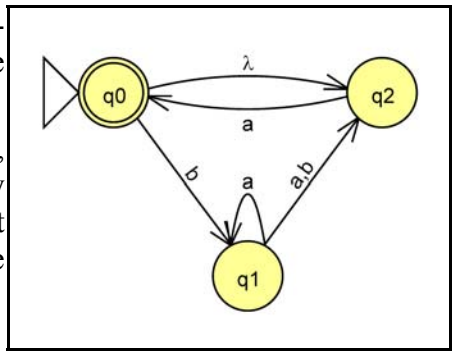

Figure 5 – A Non-deterministic Finite Automaton

	A	B
$\{q_0, q_2\}$	$\{q_0, q_2\}$	q_1
Q_1	$\{q_1, q_2\}$	q_2
$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$	q_2
Q_2	$\{q_0, q_2\}$	φ
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$

Figure 6 – A DFA Equivalent to Figure 5

With this simple technique in hand, it is easy to construct the concatenation or Kleene closure of DFAs. For example, to construct a concatenation of two machines, we merely create a NFA by connecting the accepting states of the first to the initial state of the second and convert the result to a DFA as shown. To summarize, this simple technique of tracing multiple paths through an automaton simultaneously allows us to convert NFAs to DFAs, and to construct the union, intersection, concatenation, and Kleene closure of finite automata, a major component of regular language theory.

TEACHING CONTEXT-FREE LANGUAGE THEORY

The Chomsky Normal Form (CNF) is convenient for establishing a number of theoretical results for context-free languages. To convert a context-free grammar (CFG) to CNF involves removing null production rules. There is a straightforward "algebraic" technique for removing nulls not readily available in the literature. Simply replace each nullable non-terminal, N, say, with (N + λ) and "multiply", dropping any lambdas that remain. To illustrate, consider the following grammar.

S => ASA | aB
A => B | S
B => b | λ

Both A and B are nullable, so we make the indicated substitutions and multiply:

S => (A+λ)S(A+λ) | a(B+λ) => ASA | AS | SA | aB | a
A => (B+λ) | S => B | S
B => b | λ => b

This technique finesses the confusion students often encounter when they attempt to trace all of the possible outcomes "manually."

A central result for context-free languages is that they can be represented either by a CFG or a pushdown automaton (PDA). To find a PDA for a CFG is trivial if you allow non-determinism—just push the start symbol on the stack, and then have transitions that simulate the productions in the grammar. Starting with the non-null version of the grammar above, the PDA (which accepts by empty stack) in Figure 7 obtains.

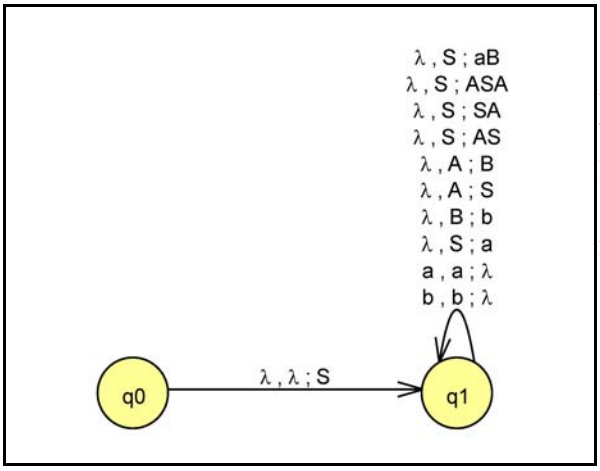

Figure 7 – Mapping a CFG to a PDA

Going from a PDA to a CFG is more difficult, but doable without special techniques such as stack bottom markers by following a procedure of Lewis and Papadimitriou.[4] As this procedure is too lengthy to include here, the reader is referred to an online resource for an explanation. [5]

The well-known CYK algorithm for determining if a given string can be generated by a CFG is effectively conveyed using a triangular matrix as a visual aid. Initially, the rules in a given CNF grammar of the form A => c are laid out on the first diagonal of the matrix as they apply. To illustrate, we will use the following grammar in CNF:

S => XY
X => XA | a | b
Y => AY | a
A => a

The matrix for testing the string "baaaa" is initialized as shown in Figure 8.

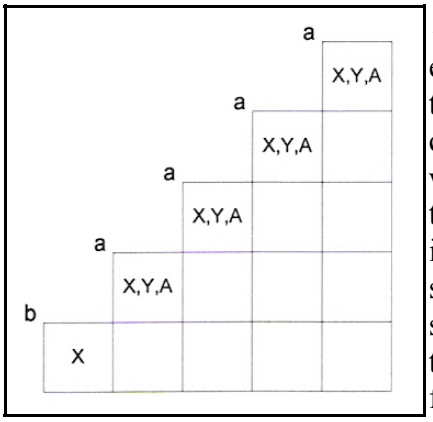

Figure 8 – The First Stage of the CYK Algorithm in Tableau Form

{00021A14-0000-0000-C000-000000000046}For example, since an 'a' can be generated from the non-terminals X, Y, or A, all three symbols appear in the cells corresponding to the letter 'a'. Now the second diagonal will be filled to determine how all substrings of length two in the test string can be generated by using the results in the first diagonal (which, as explained, represent substrings of length one). For example, the leading substring "ba" can come from XX, XY, or XA. The latter two appear in the grammar and themselves originate from S and X respectively, so the lower cell of the second diagonal stores S and X. Figure 9 shows the final tableau.

Filling in the lower cell in the third diagonal corresponds to determining how the substring "baa" is obtained, which, because of CNF, must come from a concatenated pair of non-terminals, so the possibilities are "b" paired with "aa", and "ba" paired with "a". Using entries from previously-computed diagonals, the first pairing (b/aa) could originate from X (on diagonal 1) paired with one of {X,Y,A} (from diagonal 2). Only one of three possible concatenated pairs, XY, is in the grammar, so its source, X, is entered in the corresponding cell. To see how the second pairing could be generated (ba/a), we combine {S,X} from diagonal 2 with {X,Y,A} from diagonal 1, yielding valid pairs XY and XA, which come from S and X respectively, so {S, X} is the final result for the lower cell in the third diagonal. The process continues until the lower right cell is complete. The string is in the grammar if and only if S appears there.

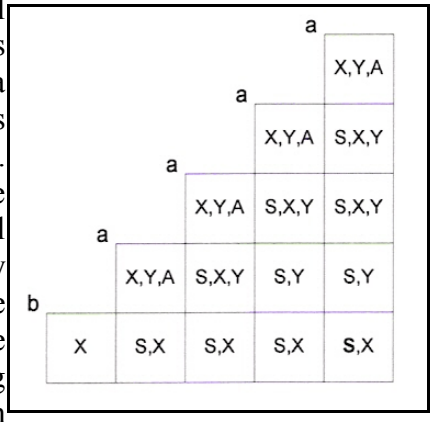

Figure 9 – The Completed CYK Tableau

SUMMARY

Traditional approaches to teaching the theory of computation rely heavily on mathematical formalisms. We have presented some intuitive techniques that appeal to the programming mindset of typical contemporary CS undergraduates.

REFERENCES

[1] Cohen, D., *Introduction to Computer Theory*, 2nd Edition, Wiley, 1997.

[2] Duke University. Available for download at http://www.cs.duke.edu/csed/jflap/.

[3] Rabin, MO; Scott, D (April 1959). *"Finite Automata and Their Decision Problem"*. IBM Journal of Research and Development 4 (2): 114-125

[4] Lewis, H. and Papadimitriou, C, *Elements of the Theory of Computation*, Second Edition, Prentice-Hall, 1998, pp. 139-142.

[5] Allison, C., Procedure for Converting a PDA to a CFG, unpublished. Available for download at http://uvsc.freshsources.com/PDA2CFG.doc