Notes on Sipser: §1.3, “Regular Expressions”
CSC 341, “Automata, Formal Languages, and Computational Complexity”
Department of Computer Science · Grinnell College
August 31, 2020

The “regular expressions” in Perl, Ruby, Python, and similar programming languages, and even in the grep utility, are not quite equivalent to the regular expressions that Sipser describes, because they provide a “back-reference” mechanism that has side effects during the pattern-matching process itself. Pattern matching using classical regular expressions is purely functional in nature.

By exploiting such side effects, a “regular expression” created by a clever Perl programmer may be able to recognize a language (i.e., solve a problem) that no classical regular expression can. That may be good news for Perl programmers, but it means that the theoretical account of Perl's regular expressions, as an application of formal-language theory, is more complicated and less generally useful than Sipser’s account of classical regular expressions.

The regular expressions that are used in lexical analyzers (as Sipser describes briefly on page 66) are the classical ones. Lexical analyzers don’t need back-references, and they would be an unnecessary complication in that context.

•

Experienced Scheme programmers will recognize Definition 1.52 as the definition of a recursive data type with three simple constructors and three recursive constructors (two binary, one unary). We can immediately infer that many of the procedures that we write in Scheme to operate on arguments that are regular expressions will have bodies that are six-way cond-expressions with no recursive calls in the first three cond-clauses, two recursive calls in the fourth and fifth cond-clauses, and one recursive call in the sixth cond-clause.

To a mathematician, the same structural property of this definition suggests immediately that many of our proofs about regular expressions will be structural inductions with three base cases and three “inductive steps” in which the hypothesis of induction says that the proposition holds for the subexpression(s) and the inductive conclusion says that it holds for $R$ as well.

These perspectives are fundamentally identical.

•

In case you were wondering: I don’t know of any use for generalized nondeterministic finite automata apart from the role they play in this proof. They are a kind of data-structure analogue of a helper procedure, designed to play a very specific role as a bridge between two kinds of mathematical entities that are intrinsically not much alike.

In particular, there’s no point in developing an implementation of generalized nondeterministic finite automata that is capable of actually accepting input strings and delivering Boolean results. We don’t need them for actually solving problems or recognizing languages. We need them only to show the equivalence of deterministic finite automata and regular expressions.

•

The key result of this section is Theorem 1.54, which is kind of a surprising result if you think about it. At first glance, there doesn’t seem to be any reason why these two
computational models should be equivalent, in the sense that a language is recognized by some deterministic finite automaton if, and only if, there is a regular expression that describes it. The proof of Theorem 1.54 exposes a deep and non-obvious relationship between the two models.

•

The proof by mathematical induction of Claim 1.65 is not set out quite correctly. Since the induction step is supposed to be a general line of reasoning that can be used to advance from any integer greater than or equal to the base case (which in this proof is 2) to its successor, the induction step should, strictly speaking, use a new variable, say ‘\( i \)’, rather than the universally quantified variable in the statement of the proposition (which in this case is ‘\( k \)’), just as in Sipser’s presentation of mathematical induction on page 23. Moreover, the hypothesis of induction should be “Assume that the claim is true for \( i \) states . . .” (not “\( k - 1 \) states”), and the inductive conclusion should be “the claim is true for \( i + 1 \) states” (not “\( k \) states”). It’s not necessary to change the intermediate line of reasoning at all; we just treat \( G \) as an arbitrary generalized nondeterministic finite automaton with \( i + 1 \) states, where \( i \) can be any integer greater than or equal to 2, and show that \( G \) is equivalent to the \( i \)-state generalized nondeterministic finite automaton \( G' \).