MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Machine Structures Group Memo No. 10 November 1965

Techniques for Manipulating Regular Expressions

by

Robert McNaughton

(Presented to the Conference on Systems and Computer Science at the University of Western Ontario, London, Canada, September 1965)

This paper will be a summary of recent work in regular expressions. The story will be told in as non-technical a manner as possible so as to enable those who are not specialists in the Theory of Automata to get a glimpse of the various approaches to the problem of setting up a discipline of regular expressions.

A few years ago Professor Brzozowski (whose conference paper is on something entirely different), wrote a survey paper [1] on regular expressions, which proved to be quite helpful to those, like the readers of the present paper, who wanted a summary account of regular expressions. (Numbers in brackets are references listed at the end of the paper.) The present paper will be concerned with work done since that paper appeared.

To begin with, regular expressions are expressions standing for regu lar events (or regular languages), which are certain sets of words. A wor is a string of symbols over some alphabet, denoted by Σ. Although Σ is general any finite set, very often I will simply use the alphabet whose sym bols are 0 and 1. Occasionally I shall use the more extended alphabet $\Sigma = \{0,1,2\}$.

Regular expressions are made up of letters of the alphabet, and signs standing for certain operators. The three operators are union, concatenation and star (or closure). Union is the ordinary set-theoretic union, whose sign is "$\cup$". Concatenation is written as a dot and sometimes is denoted by mere juxtaposition. The concatenation of two words is obtained by writing the first word, and then the second word following it without any space. The concatenation of two sets of words tends to be thought of as a Cartesian product; however, it is not quite that. Let α and β be two events. $\alpha \cdot \beta$ or $\alpha\beta$, the concatenation of α and β, is the set of all words that can be obtained by a concatenation of a word from α and a word from β in that order. For example, if $\alpha = \{0,01,001\}$ and $\beta = \{1,11\}$ then $\alpha\beta = \{01,011,0111,0011,00111\}$. Note that the word 011 is obtained in two ways: as 0 concatenated with 11, and as 01 concatenated with 1. This example is enough to show that concatenation is not a Cartesian product.

Before explaining the star operator, something should be said about the null word. First, the null word λ and the empty set $\emptyset$ must be distinguished: for the null word is the word of zero length where the empty set is the set that has no words at all as members; these behave quite differently in regular expressions. If the null word is mysterious, just think of a word of length 3. If you throw away one symbol, you are left with a word of length 2. Then if you throw away another symbol, you are left with a word of length 1, a single symbol. When you throw away that symbol you are left with a word of zero length, the null word. The concept of the null word turns out to be important because of the manner in which it concatenates with another word: thus, for any W, $\lambda W = W \lambda = W$. As a consequence, for any set α, $\lambda\alpha = \alpha\lambda = \alpha$. This behavior of the null word is almost its definition.

By the way, we see in the notation "$\lambda\alpha$" a tendency that may tend to dishearten the logical purist, but is convenient in the practice of writing regular expressions. It is the notational identification of an object and its unit set. Thus "λ" in "$\lambda\alpha$" means the unit set of the null word; in the mathematics of regular events a confusion never results from this identification.

On the other hand, the empty set $\emptyset$ is a set-theoretic concept, meaning a set that does not have anything in it. In contrast to the fact that $\lambda\alpha = \alpha\lambda = \alpha$, for any set α, $\alpha\emptyset = \emptyset\alpha = \emptyset$. In order to see this, it is necessary to go back to a literal interpretation of the definition of the concatenation of two sets: $\alpha\emptyset$ is the set of all words obtained by concatenating a word from α and a word from $\emptyset$. But there are no words in $\emptyset$, since $\emptyset$ is the empty set. Hence $\alpha\emptyset = \emptyset$. Similarly, $\emptyset\alpha = \emptyset$.

The star operator can be defined as follows:

$$\alpha^* = \left\{ W : (\exists n)(\exists V_1)\cdots(\exists V_n)(\forall i)\ V_i \in \alpha \ \& \ W = V_1V_2\cdots V_n \right\}.$$

Here n is a non-negative integer. Since n may be 0, $\lambda \in \alpha^*$, for every α. Now the above definition is not a legitimate expression of symbolic logic because of the dots, which are not easily eliminated. Another definition is as follows: α^* is the smallest set containing λ and containing, for every word $W \in \alpha^*$ and $V \in \alpha$, the word WV.

Thus star is the iteration of concatenation. It is sometimes called "closure".

The <u>restricted regular-expression language</u> consists of union, concatenation (denoted by juxtaposition), star, each member of the alphabet (whatever it may be), λ and $\emptyset$. The <u>enlarged regular-expression language</u> has all of this plus intersection and complementation. In this paper we shall be concerned only with the restricted language and the term "regular expression" will refer to an expression of this language.

The language of regular expressions surpasses not only the distinction between the unit set of a word and the word itself, but also the distinction between a letter of the alphabet and a word of length one. Thus "01" stands for both a word and the unit set of that word. And "0" stands for a letter of the alphabet, a word of length one, and the unit set of that word.

Some laws which have been noticed about regular expressions are the following:

(1) $\lambda\alpha = \alpha\lambda = \alpha$

(2) $\emptyset\alpha = \alpha\emptyset = \emptyset$

(3) $\lambda^* = \emptyset^* = \lambda$

(4) $\alpha(\beta\alpha)^* = (\alpha\beta)^*\alpha$

(5) $(\alpha\cup\beta)^* = (\alpha^*\beta^*)^*$

(6) $\alpha(\beta\cup\gamma) = \alpha\beta\cup\alpha\gamma$

(7) $(\alpha^*\beta)^* = \lambda\cup(\alpha\cup\beta)^*\beta$

(8) $\mathrm{Reg}(\alpha,\beta)\subseteq(\alpha\cup\beta)^*$

(9) $\alpha^* = \lambda\cup\alpha\cup\alpha^2\cup\ldots\cup\alpha^{k-1}\cup\alpha^k\alpha^*$

Of these (1) and (2) have already been discussed. (3) can also be verified by going back to a literal interpretation of either of the two definitions of the star operator. (4) can be proved by a general techn[illegible] that could be called "proof by reparsing". That is, consider any word $W\in\alpha(\beta\alpha)^*$. $W = U_0(V_1U_1)(V_2U_2)\ldots(V_nU_n)$, where each $U_i\in\alpha$ [illegible] each $V_i\in\beta$. By reparsing the last expression (using the fact that con- catenation is associative) we establish that $W = (U_0V_1)(U_1V_2)\ldots(U_{n-1}V_n)$ which shows that $W\in(\alpha\beta)^*\alpha$. Since W is arbitrarily selected, this [illegible] that $\alpha(\beta\alpha)^*\subseteq(\alpha\beta)^*\alpha$. Thus (4) is established, by symmetry.

(5) is also established easily by reparsing. (6) is too obvious to require proof. (7) is perhaps least obvious of all, and will be discussed in detail below. In (8), Reg (α,β) means any restricted regular expres[illegible] made up exclusively from α and β; in other words, containing no occurrence of the alphabet or letters thereof outside of α and β. The proof of (8) by reparsing is quite similar to the proof of (5). However, one must be quite careful because (8) is true only for restricted regular expressions. It is no longer true if intersection or complementation is allowed in Reg (α,β).

A good name for (9) is the development law, which is quite obvious [illegible] the definition of the star operator.

There are two main research problems about regular expressions. The first concerns means of proving valid equations (such as the above), while the second is the problem of finding interesting and fruitful equations to prove. The second problem will be discussed only briefly at the end. Most of the remainder of this paper will be concerned with the first problem.

Suppose we are given two regular expressions for comparison. The first question that arises is, are they equal? If not, then is one included in the other? However, there is a familiar and simple technique to handle inclusion as an equation: thus $\alpha \subseteq \beta$ will be true if and only if $\alpha \cup \beta = \beta$. Thus the problem of proving equation is foremost; a systematic and successful approach to this problem could be called a calculus of regular expressions.

At this point it would be appropriate to say something explaining the connection between this paper and the purpose of the conference as a whole. The discipline of regular expressions was originally introduced to describe automata in Kleene's paper [2], which was written way back in 1951 and published in 1956. For many years since, this discipline was interesting to switching theorists. Recently, however, I have discovered that there is considerable interest in this language among people who do advanced programming. It is likely that their interest in the regular-expression language is due to many different reasons: but one important reason is that the regular operators turn up frequently in the advanced study of mechanical languages. It must be said, of course, that regular events (or languages) only form a small subclass of the class of all languages that are interesting to advanced programmers. There are other classes which come closer to being candidates for the class of all programming languages. For example, there is the class of context-free languages. A less likely candidate would be the class of context-sensitive languages, — less likely, because it is too broad, and because most principles of formal language construction seem to be context-free principles. Indeed, one could argue that the class of all context-free languages is too broad. But, in any case, the class of regular languages is certainly too narrow, and the deficiency is made up in the direction of the class of context-free languages. Nevertheless, it is still true that the study of regular events has application to problems coming up in the investigation of the broader class of languages.

In spite of all this speculation about possible applications, I would like to suggest that our proper attitude towards the study of regular expressions (and toward the theory of automata, in general) should be that it is part of pure mathematics. We should attack the problems without worrying about where the application is going to be. In spite of its origin in switching theory (or, more precisely, in nerve-net theory) and present vague relation to problems of advanced programming, these connections are not altogether decisive in formulating valid research objectives. I hope these remarks suffice to explain my opinion about the applicability of regular expressions to the more practical matters in the computer sciences, and, at the same time, to provide a link between the contents of this paper and the conference as a whole.

Now let us return to regular-expression equations. There are several methods of proof, which I shall survey. The first method is the technique of converting each regular expression to a state graph and, then, reducing the state graphs and testing whether the resulting state graphs are isomorphic. We thereby make use of a basic theorem from switching theory, that two state graphs that recognize exactly the same languages have to be isomorphic.

And so there it is, a procedure for testing an equation. This method is foolproof, and works absolutely. That is, given any two regular expressions, one can perform the test mechanically and provide a yes or a no answer to the question of their equality. The only trouble is that it is not insightful; that is, to say, when one actually goes through this process with a given α and β, the mechanical procedure that is used is usually not such as to provide any insight into the nature of these regular expressions. For example, α and β may be unequal but closely related to each other. But the procedure would terminate with a no answer and that would be the end of it; one would not discover any relationship. Even in cases where a yes answer results, after the computation is over there is usually a feeling that one lacks an understanding of why the two expressions are equal. Generally speaking, the reason that the method of test by state graph fails to produce any insight about the regular expressions is that it goes outside of the language of regular expression to test the equality.

The second method of establishing the equality of regular expressions is proof by reparsing. This method was used above to verify that $\alpha(\beta\alpha)^* = (\alpha\beta)^*\alpha$. To illustrate a more involved application it will now be used to show that $(0^*1)^* = \lambda \cup (0 \cup 1)^* 1$. Note that this equation is an instance of law (7) above. It is easy to see that the proof below is adequate to establish the more general law, but the more specific instance is more easily discussed.

To show first that $(0^*1)^* \subseteq \lambda \cup (0 \cup 1)^* 1$, consider an arbitrary word $W \in (0^*1)^*$. Then $W = U_1U_2\ldots U_n$, $n \geqq 0$. If $n = 0$ then $W = \lambda$ and then $W \in \lambda \cup (0 \cup 1)^* 1$. If $n \geqq 1$, then for each i, $1 \leqq i \leqq n$, $U_i \in 0^*1$. We can then write $W = (U_1 U_2\ldots U_{n-1}\ 0\ldots 0)\,1$, where the number of 0's at the end of the parenthesized part is zero or more. Clearly $U_1U_2\ldots U_{n-1}\ 0\ldots 0 \in (0 \cup 1)^*$, since the latter is the set of all words of 0's and 1's. Thus $W \in (0 \cup 1)^* 1$ and therefore $W \in \lambda \cup (0 \cup 1)^* 1$.

To show that $\lambda \cup (0 \cup 1)^* 1 \subseteq (0^* 1)^*$ consider an arbitrary $W \in \lambda \cup (0 \cup 1)^* 1$. If $W \in \lambda$ then $W \in (0^* 1)^*$ by the definition of the star. If $W \in (0 \cup 1)^* 1$ then $W = U1$, where U is any sequence of 0's and 1's. Suppose there are n-1 occurrences of 1. For the sake of perspicacity, and since our primary concern is not for rigor, suppose that n = 5 and suppose U = 00110100001000. Then W = 001101000010001. To show that $W \in (0^* 1)^*$, simply parse W so that each 1 ends a phrase, thus

$$W = (001)(1)(01)(00001)(0001).$$

That this reparsing can be accomplished in general follows from the mere fact that W ends in a 1, which concludes the proof.

I trust that everyone will agree that the above proof by reparsing does provide insight into why $(0^* 1)^* = \lambda \cup (0 \cup 1)^* 1$.

Proof by reparsing, although insightful, tends to be tedious and complicated, if a new proof is required for each new equation. The third and fourth methods are more systematic and practical. The third method is that of a logical system with rules of inference, in which one proves equations in the manner of a formal proof. This method has received some attention recently with interesting positive results. Arto Salomaa has written a paper [3], now in print, in which he put forth an axiom system with three rules of inference, described below, and conjectured that the system is complete (which means that all valid equations can be derived). A few months after he published his work, he was able to prove that his system was complete. Furthermore, the proof of completeness for a similar system, using exactly the same rules with ~~perhaps~~ a slightly different set of axioms was given by

Mr. Stal Aanderaa [4], a graduate student at Harvard. The set of axioms Aanderaa used are similar to Salomaa's. They are simple and obviously valid.

These equations may contain variables ranging over regular sets of words, which will be the later letters of the Roman alphabet, x, y, etc. in this paper. Greek letters will be meta-language variables ranging over regular expressions, for use in describing the system.

The three rules of inference used by both Salomaa and Aanderaa are as follows:

Substitution: Substitute a regular expression for every occurrence of a variable in a given equation. Example, from $(x^*y)^* = \lambda \cup (x \cup y)^* y$ infer $(x^*000)^* = \lambda \cup (x \cup 000)^* 000$.

Replacement: From an equation $\alpha = \beta$, and from an equation in which α occurs as a well formed part, the result of replacing an occurrence of α by β may be obtained. Example: from $(x^*y)^* = (x^*y)^* \cup yyy$ and $(x^*y)^{**} = (x^*y)^*$ infer $(x^*y)^* = (x^*y)^{**} \cup yyy$.

Star introduction: From $\alpha = \alpha\beta \cup \gamma$ derive $\alpha = \gamma\beta^*$, provided $\lambda \notin \beta$. (It is to be noted here that there is an easy syntactic method of noting whether or not any given regular expression contains the null word. E.g., see page 8 of [3].) Example: take $\alpha = (0 \cup 1)^* 1 \cup \lambda$, $\beta = 0^*1$, and $\gamma = \lambda$. Then from $(0 \cup 1)^* 1 \cup \lambda = [(0 \cup 1)^* 1 \cup \lambda] 0^* 1 \cup \lambda$ one can infer $(0 \cup 1)^* 1 \cup \lambda = \lambda (0^*1)^*$, since in this case $\lambda \notin \beta$.

Note that this last example could be a step in the proof that $(0^*1)^* = (0 \cup 1)^* 1 \cup \lambda$ in an axiom system. The earlier portion of the proof would have to establish the lemma that $(0 \cup 1)^* 1 \cup \lambda = [(0 \cup 1)^* 1 \cup \lambda] 0^* 1 \cup \lambda$, and the last few steps would simply utilize the fact that $\lambda (0^*1)^* = (0^*1)^*$.

To give some idea of how simple the set of axioms can be, I shall adapt Aanderaa's axioms to the notation of this paper:

(A1) $x \cup x = x$

(A2) $x \cup y = y \cup x$

(A3) $(x \cup y) \cup z = x \cup (y \cup z)$

(A4) $(xy)\, z = x\, (yz)$

(A5) $x(y \cup z) = xy \cup xz$

(A6) $(x \cup y)\, z = xz \cup yz$

(A7) $x \cup \emptyset = x$

(A8) $x\emptyset = \emptyset$

(A9) $\emptyset x = \emptyset$

(A10) $\lambda = \emptyset^*$

(A11) $x\lambda = x$

(A12) $x^* = \lambda \cup xx^*$

(A13) $x^* = (\lambda \cup x)^*$

To say that the system with these axioms and rules of substitution, replacement and star introduction is complete is to say that all valid equations can be derived from the axioms by means of the rules. Aanderaa's proof of completeness makes rather skillful use of Brzozowski's notion of the derivative, as found in [5].

The simplicity of the axioms is not all that should be hoped for in such a system. Optimally, an axiom system should have only logical rules of inference; the axioms ought to be sufficient to yield all desired valid truths by means of such rules. But in this axiom system, the rule of star introduction expresses a great deal of mathematical content and cannot be justified on the basis of logic alone. Substitution and replacement are all right in this regard; they are justified solely by virtue of the meaning of equality.

Now the Ukrainian Redko [6] has proved that there is no finite set of axioms yielding all valid equations when only the rules of substitution and replacement are allowed. In my opinion, an interesting unsolved problem (although vague) is whether there exists an interesting infinite set of axioms from which all valid equations can be derived using only substitution and replacement. Such an axiom set might be more significant than the simple set listed above. The question comes up here as to what we mean by an <u>interesting</u> infinite set of axioms. Saul Gorn has suggested during the discussion following the oral presentation that the axioms all be instances of a finite number of schemata. This appears to me to be a reasonable suggestion, although it does not remove all of the vagueness from the question until we specify precisely what we mean by "an axiom schema".

The fourth method for proving regular-expression equations is by means of graphs. The earliest paper on the use of the kind of graphs that we shall now discuss (as opposed to the more restricted concept of state graph) is the early paper by Chomsky and Miller [10]. It did not discuss regular expressions, but defined the notion of "finite-state language" as a language generated by a graph with a finite set of nodes. That was 1958 and, although several papers have appeared linking up these graphs with regular expressions, I know of no reference to the proof of regular-expression equations by means of them.

A year ago I was convinced that the graphical method of proving equations had the most to offer in almost every respect. However, since then completeness proofs for the axiom systems have indicated that the axiomatic method is at least as worthy. For it is so much easier to write down a sequence of regular-expression equations than to write down a sequence of graphs.

My present opinion is that even if the axiomatic method is better for proving equations, the graphical approach is better in cases in which we are not certain what it is we want to prove.

To begin our discussion of graphs let us take another look at regular expressions. We can think of a regular expression as a manner of generating words. For example, from $[0(00)^{*} 1 \cup 11]^{*}$ one generates a sequence either with a 0 or a 1. If one selects the 0 then one has the option of writing 00 any number of times and then writing 1; but if one elects to begin with a 1 one must write 1 immediately after. After that one again has the option of writing 0 or 1, etc. Of course, by virtue of the meaning of the star one could have settled for the null word at the very outset. Thus the regular expression can be thought of as a generator, a sequential machine that operates in a non-deterministic manner (in the sense that it makes arbitrary choices at each juncture) to generate a word. The event of such a machine would be the set of all possible words that could be generated by such a device.

Now the concept of non-deterministic machine has little practical significance, but is found very frequently in the theory of automata. Certainly not all machines considered in the Theory of Automata will come into existence. And probably all of the varieties of non-deterministic automata that have been considered are excellent candidates for non-existence. But the prevalence and the obvious theoretical usefulness of non-deterministic concepts cannot be denied; rather, we are left with a challenge to explain why it is that these concepts are useful.

In this case we are interested in the non-deterministic machine only as a way of describing a set, namely the set of all words that could be generated. Thus there is no sense in building such a machine, since the regular expression is as concrete as we have to get. It turns out that a useful alternative to the regular expression is something equally abstract, namely the graph. As Yamada has noted [9], the graph is a concept that arises quite naturally when one takes this point of view towards a regular expression. Thus consider the regular expression $\{0(00)^* 1 \cup 11\}^*$ as a word generator in the manner described above. The same function could be accomplished by the graph in Fig. 1. To generate a word in such a graph one first selects a path

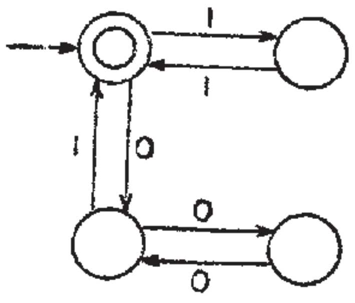


Fig. 1

beginning at the initial node (designated by a single unlabeled arrow pointing to it) and terminating at a terminal node (double circle). The word spelled out by this path by the labels on the branches along the way is in the event. Thus the event is simply the set of words spelled out by the set of all such paths.

Every regular expression can be converted into a graph. There is an algorithm to accomplish this which I shall not describe in detail, since in most cases the construction is obvious, suggested by the relationship between the regular expression $\{0(00)^* 1 \cup 11\}^*$ and Fig. 1. I should mention, however, that there is a pitfall: if one constructs a graph in the most obvious way from a given regular expression one may introduce sneak paths. For example, one might incorrectly render $[10 \cup 0(11)^*]^*$ as the graph of Fig. 2, which contains a sneak path, namely the path spelling out 1011, which is not in the event represented by the regular expression. A correct rendition would be Fig. 3 which has a branch labeled λ (the null word). The significance of

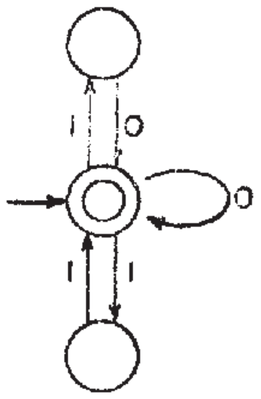


Fig. 2

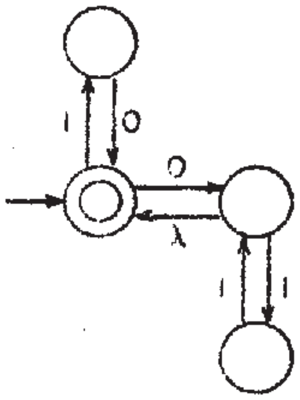


Fig. 3

such a branch is that, from any path using the branch, to determine the word spelled out one ignores the λ. Sneak paths are avoided by this technique because any path using the branch must go in the direction of the arrow.

The totality of these graphs includes the well known state graphs. But not all such graphs are state graphs; in fact, a state graph is a very special kind of graph, in which there are no branches labelled λ and, for every member of the alphabet and for every node, there is exactly one branch labeled with that letter and leaving that node. These general graphs are sometime referred to as "non-deterministic state graphs," although I prefer to call them "transition graphs." An interesting variant on them in which the branches may be labeled with arbitrary regular expressions and to which the techniques of signal flow graphs can be applied has been developed by McCluskey and Brzozowski [7].

Now although every regular expression can be transformed into a graph that has the same structure, the converse is not true. I will not define here precisely what I mean by the structure of a regular expression or graph, and hope that my point is made on the intuitive level. Thus, for example, the structure of $(0^*1)^*$ is very much different from the structure of $(0 \cup 1)^*1 \cup \lambda$, although the two regular expressions are equal. And I hope the reader knows what I mean when I say that the state graph of Fig. 3 has the same structure as the regular expression $[0(11)^* \cup 10]^*$ or (more precisely) $[0(11)^* \lambda \cup 10]^*$. One way of explaining what this means in intuitive terms is to say that the parts of the graph correspond to the parts of the regular expression in such a way that a path through a graph will spell out the same word as a path through the corresponding parts of the regular expression. To be slightly more precise, the part of a regular expression corresponding to a node of a graph would be a point immediately to the left or to the right of one of the characters (such as 0, 1 or λ). And, for further clarification of this point, note that for each well formed part of the regular expression $[0(11)^* \cup 10]^*$ there is a corresponding part of the graph, in such a way that if one well formed part of a regular expression is part of another, the same is true of the corresponding part of the graph.

Hopefully, the notion of structure of regular expressions and graphs is clear. Then the point is that although every regular expression has a graph which represents the same event and has the same, or almost the same, structure, there are some graphs whose structures are very far from the structure of any regular expression. An example of such a graph is given in Fig. 4. Such graphs are graphs in which there is no hierarchy of loops as there

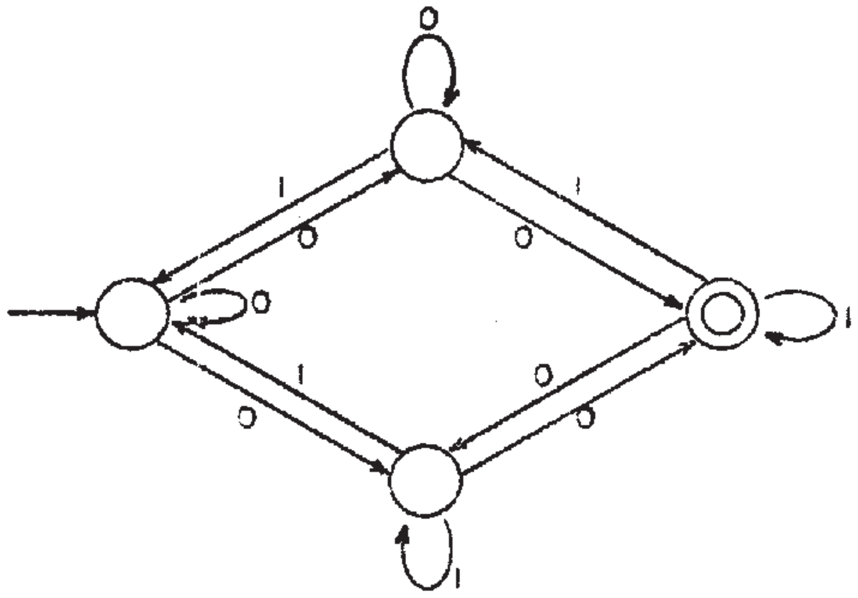
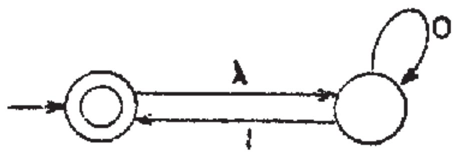


Fig. 4

is in regular expression. A loop in a regular expression is always given by a star; and for every pair of stars, either one is inside the scope of the other or their scopes are completely disjoint. A loop in a graph is any directed path that ends where it begins, and otherwise does not repeat any node. It should be clear that in the graph of Fig. 4, there is no way of imposing a hierarchy on the loops in the manner in which loops must occur in a regular expression. I am afraid that I must leave this remark as a rather vague suggestion, rather than as a precise technical proposition. A precise account of what a hierarchy of loops might be in a graph is beyond the scope of this paper. I trust it is clear at least that the graph of Fig. 4 has a structure which is quite unlike the structure of any regular expression. And it is not difficult to construct many graphs of this kind.

For this reason we can say that the set of graph structures is richer than the set of regular-expression structures. This fact contrasts with the fact that the class of events represented by graphs is exactly the same as the class of events represented by regular expressions. For it is well known that for every graph (including, e.g. that of Fig. 4) there exists a regular expression representing the same event (although it has, in general, a vastly different structure). In fact there is an algorithm to make this conversion. (See [8].)

To show how graphs can be used to prove a regular-expression equation, I will present a graph-theoretic proof of the same equation that I have proved by the other methods, namely $(0^*1)^* = (0 \cup 1)^* 1 \cup \lambda$. This is done in the sequence of graphs in Fig. 5, where the graph of Fig. 5a has the structure of $(0^*1)^*$ and the graph of Fig. 5d has the structure of $\lambda \cup (0 \cup 1)^* 1$. The sequence is such that each step preserves the event precisely; no words are introduced or deleted. Furthermore, each step is the result of deletion or addition of a branch, or a branch and a node. The step leading to Fig. 5b is justified in that any path using the new branch could just as well have



$(0^*1)^*$

Fig. 5a

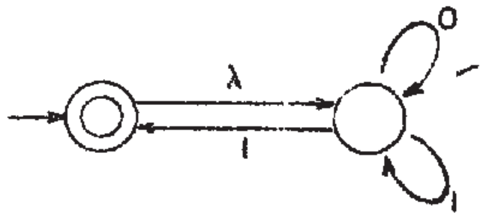


Fig. 5b

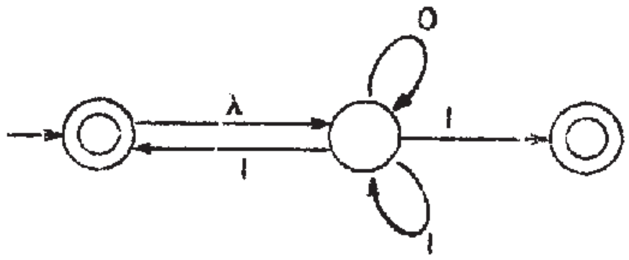
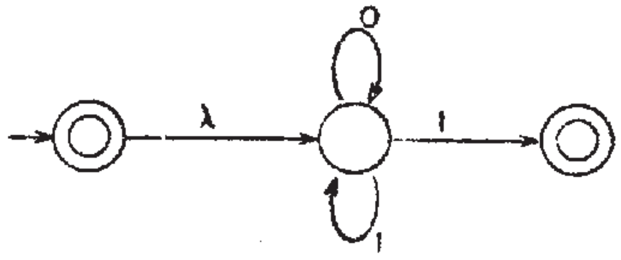


Fig. 5c



$\lambda \cup (0 \cup 1)^* 1$

Fig. 5d

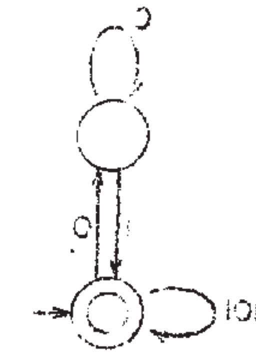
gone via the old branches labeled 1 and λ. The addition of the branch to a new terminal node in Fig. 5c is justified by the fact that any word ending at that new node could just as well have ended at the original terminal node. And the deletion of the branch in Fig. 5d is justified by a slightly more complicated consideration: any use of that branch in a path is either terminal or non-terminal; if it is terminal then it could be replaced by the branch labeled 1 to the right-most node; if it is non-terminal then it must be followed by the branch labeled λ, in which case both items in the path together could be replaced by the loop branch labeled 1. Note finally that λ is in the event represented by the graph of Fig. 5d by virtue of the fact that the initial node is terminal.

It turns out that, probably, the proof by graphs in Fig. 5 is not the most general kind of proof by graphs because (as Mr. Floor pointed out during the discussion following the oral presentation of this paper) all of the graphs in Fig. 5 are structurally similar to regular expressions as follows: (5a) $(0^*1)^*$; (5b) $((0 \cup 1)^* 1)^*$; (5c) $((0 \cup 1)^* 1)^* (\lambda \cup (0 \cup 1)^* 1)$ (5d) $\lambda \cup (0 \cup 1)^* 1$. In general, in such derivations, we can expect graphs which are not structurally similar to any regular expressions. (Examples of such graphs occurring in a derivation are Fig. 7b and Fig. 8 b.)

This will conclude my discussion of proofs by graphs. Several more illustrations located at the end of this paper will be given for the enterprising reader without a full explanation of the steps in each case, although each can be verified by inspection, perhaps after some reflection. Fig. 6 is a proof that $[00^* 1 \cup 101]^* = \lambda \cup (01 \cup 0 \cup 101)^* (01 \cup 101)$; Fig. 7 that $\{[(0 \cup 1)^* 000]^* 001\}^* = (0 \cup 1)^* 000001 (001)^* \cup (001)^*$; Fig. 8 that $[(1^*0)^* 01^*]^* = \lambda \cup 0 (0 \cup 1)^* \cup (0 \cup 1)^* 00 (0 \cup 1)^*$. (These were problems assigned to my class in "The Theory of Automata" at M.I.T., and some parts of these proofs are adapted from their homework papers.) Note that in these graphs branches are labeled with arbitrary words instead of just letters of the alphabet; this practice makes them easier to draw and easier to look at. The reader who studies Figs. 6, 7 and 8 will naturally ask the question, is there a set of formal rules for graph manipulation of this kind? As far as I know, no one has put forth a set of rules which are valid, simple, formally precise and complete. ("Complete" means that for any valid regular expression equation there is a sequence of graphs establishing the equation, where each step is according to one of the rules.) In my opinion this open problem is worth looking into.

My final remark concerns the problem of finding interesting equations to prove. Alternatively, given a regular expression what interesting or (in some significant sense) more simple regular expression is it equal to? A more ambitious question along these lines is that of a canonical form. As far as I know, no one has advanced a method of putting regular expressions into a canonical form, which is anything but just an arbitrary unique regular expression.

My favorite concept along these lines is the concept of star height. The star height of a regular expression is the maximum length of a sequence of stars in the expression, such that each star is in the scope of the star that follows it. Thus the star height of $(0^*1)^* [(000^*10)^* 000]^*$ is 3 by virtue of the sequence consisting of the third, fourth and fifth stars. The star height of a regular expression is a precise explication of the loop complexity. And since it does achieve something to simplify the loop complexity of an event, it is also an achievement to reduce the star height of a regular expression. I will stop now, except to point out that this objective could also explain the choice of equations proved in Figs. 5, 7, and 8. In each case, imagine that one starts with a regular expression whose star height he wishes to reduce. He then draws a graph, reduces the loop complexity of the graph, and ends up with a graph whose corresponding regular expression has a reduced star height. Thus if we consider the problems of reducing the star height of the regular expressions $(0^*1)^*$, $[00^*1 \vee 101]^*$, $\{[(0 \vee 1)^* 000]^* 001\}^*$, and $[(0^*0)^* 01^*]^*$, then Figs. 5, 6, 7 and 8, respectively, are solutions showing that each is reducible to a regular expression with star height 1.



$[00^*1 \cup 101]^*$

Fig. 6a

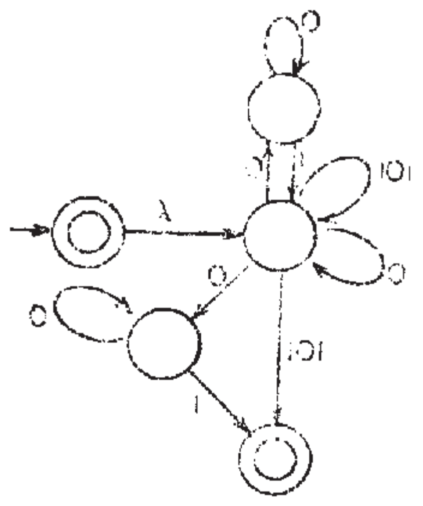


Fig. 6d

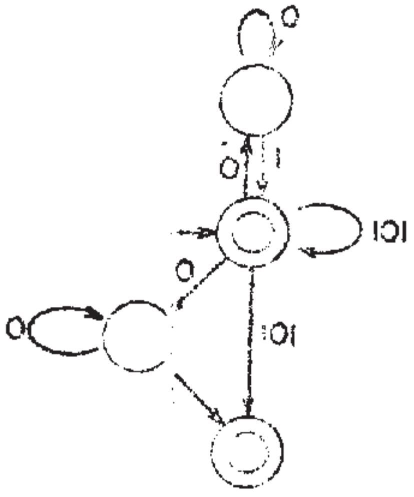
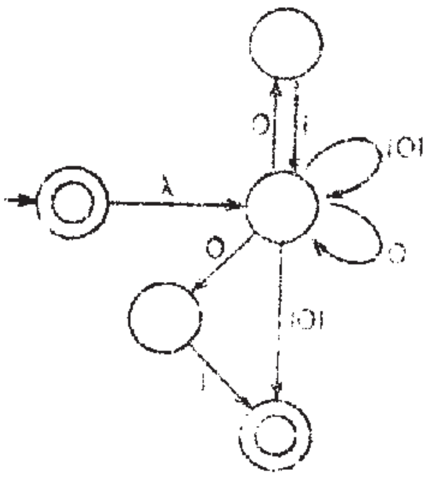


Fig. 6b



$\lambda \cup (01 \cup 0 \cup 101)^*(01 \cup 101)$

Fig. 6e

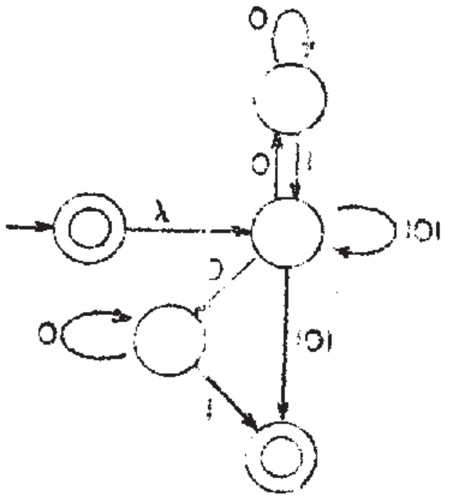
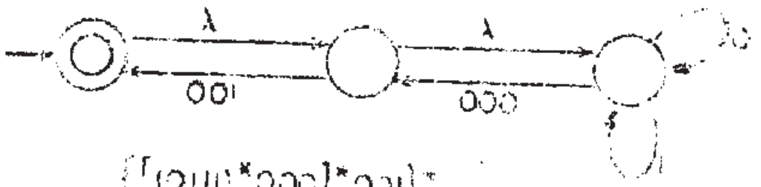


Fig. 6c



$\{[(0\cup 1)^*000]^*001\}^*$

Fig. 7a

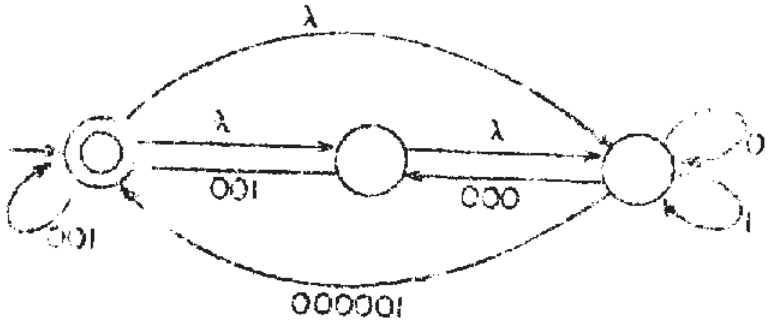


Fig. 7b

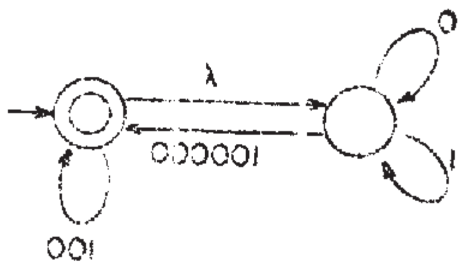


Fig. 7c

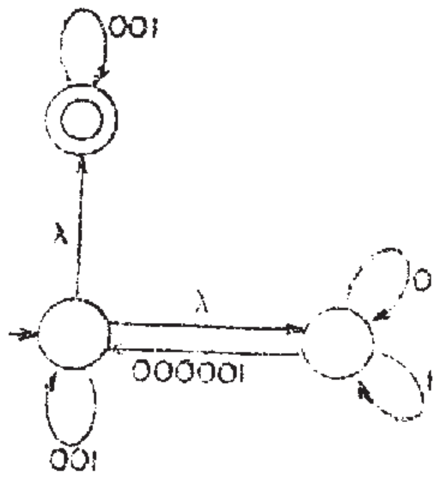


Fig. 7d

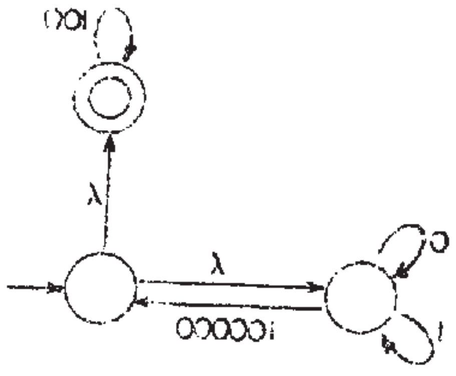


Fig. 7e

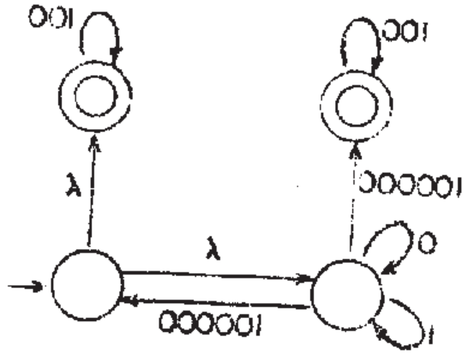


Fig. 7f

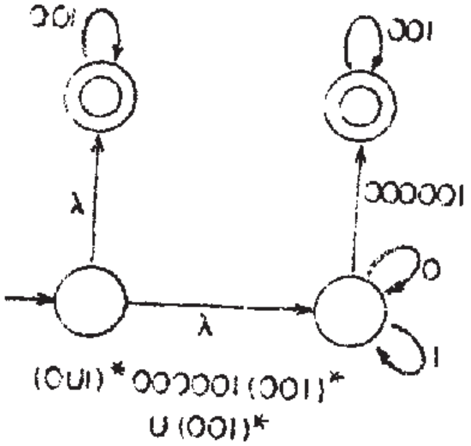
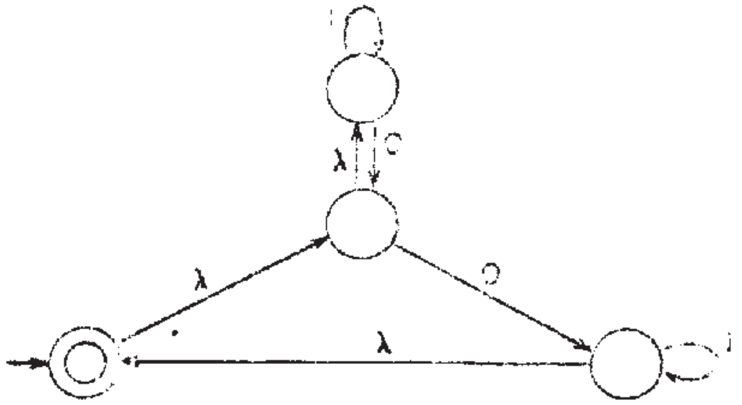


Fig. 7g



$[(1^*01)^*\ 01^*]^*$

Fig. 8a

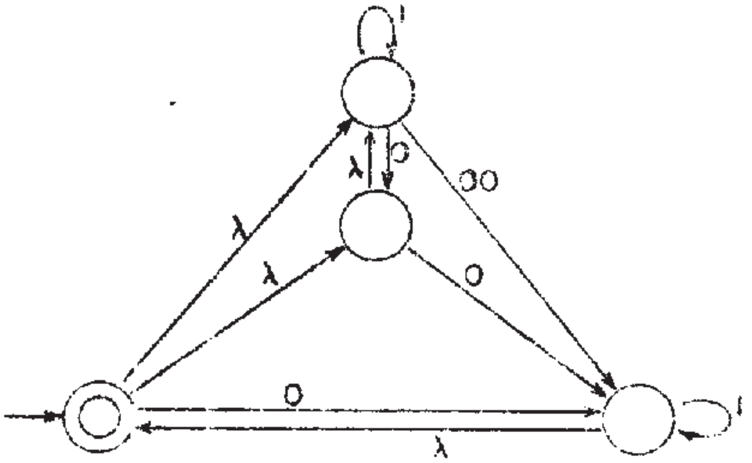


Fig. 8b

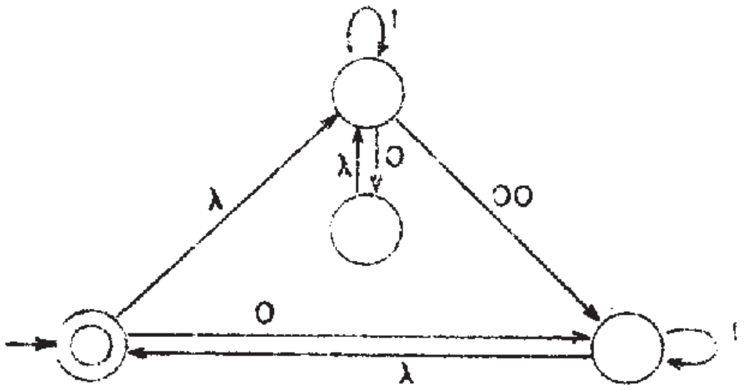


Fig. 8c

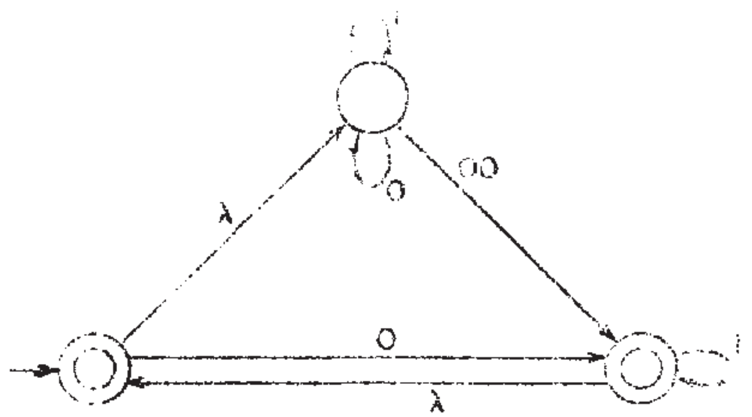


Fig. 8d

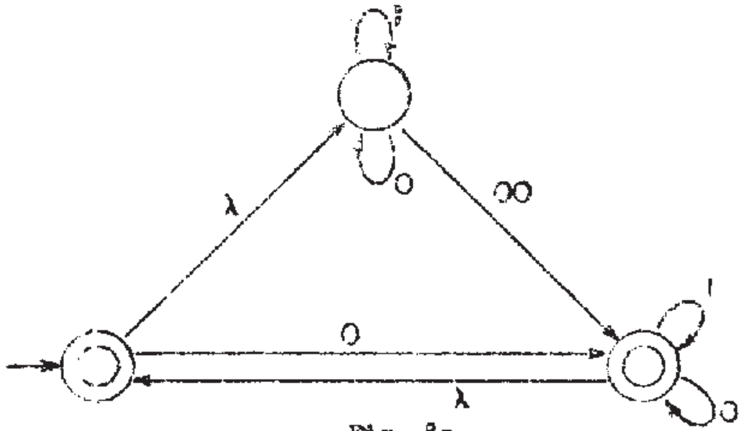
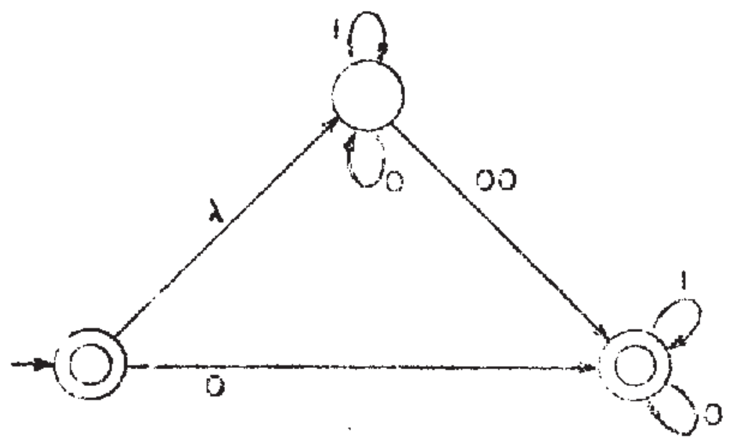


Fig. 8e



$\lambda \cup 0\,(0 \cup 1)^* \cup (0 \cup 1)^*\,00\,(0 \cup 1)^*$

Fig. 8f

REFERENCES

[1] Brzozowski, J. A., A survey of regular expressions and their applications. IRE Trans. on Electronic Computers Vol. EC-11 (1962) pp. 324-335.

[2] Kleene, S. C., Representation of events in nerve nets and finite automata. Automata Studies (C. E. Shannon and J. McCarthy, eds.) pp. 3-42 Princeton University Press, 1956.

[3] Salomaa, A., Axiom systems for regular expressions of finite automata. Turun Yliopiston Julkaisuja (Annales Universitatis Turkuensis) Series A, Number 75 (1964), pp. 5-29

[4] Aanderaa, S., On the algebra of regular expressions. Unpublished.

[5] Brzozowski, J. A., Derivatives of regular expressions. Jour. Assoc. Computing Machinery, vol. 11 (1964), pp. 481-494.

[6] Redko, V. N., About the determining set of relations in the algebra of regular events (in Russian). Ukrainian Mathematical Journal, vol. XVI (1964), pp. 120-126.

[7] Brzozowski, J. A. and McCluskey, E. J., Jr., Signal flow graph techniques for sequential circuit state diagrams. IEEE Trans. on Electronic Computers, vol. EC-12 (1963), pp. 67-76.

[8] Eggan, L. C., Transition graphs and the star-height of regular events. Michigan Math. Jour., vol. 10 (1963), pp. 385-397.

[9] Yamada, H., Disjunctively linear logic nets. IRE Trans. on Electronic Computers, vol. EC-11 (1962), pp. 623-639.

[10] Chomsky, N. and Miller, G. A., Finite state languages. Inf. and Control, vol. 1 (1958), pp. 91-112.