REGPACK, AN INTERACTIVE PACKAGE FOR REGULAR LANGUAGES AND FINITE AUTOMATA*

by

E. Leiss

Research Report CS-77-32

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

October 1977

* This work was supported by the National Research Council of Canada under Grant No. A-1617.

REGPACK, AN INTERACTIVE PACKAGE FOR REGULAR LANGUAGES AND FINITE AUTOMATA

by

E. Leiss

Abstract

REGPACK is a package of programmes written in SPITBOL for interactive use to facilitate dealing with regular expressions and finite automata. It contains procedures for some frequently used operations such as constructing a deterministic automaton from a given nondeterministic one or from a given regular expression, reducing a deterministic automaton, and constructing the syntactic monoid of a regular language.

This report on REGPACK consists of three parts: The first part gives some comments on the implementation of the algorithms, in the second part performance and limitations of the programme are discussed, and the third part presents one of the algorithms in more detail.

A certain familiarity with regular languages and their representations is assumed throughout the report.

INTRODUCTION

Despite the fact that the class of regular languages is one of the best understood classes of formal languages there are still various open problems concerning regular languages. To study these problems one often would like to test certain conjectures on examples. However, to carry this out for nondegenerate examples usually involves a great amount of (highly error-prone) hand computation. On the other hand, several of the algorithms used in these computations can be programmed without great difficulty. While we are aware that several such programmes have been written [10, 13, 14, 15] we submit that their use is feasible only in cases where hand computation does not present too much of a problem either, whereas in cases where the amount of work involved is indeed prohibitive for hand computation - computation of the syntactic monoid of a regular language which happens to have more than 300 elements, for example - these programmes typically cannot be used because they need too much time and/or space.

For these reasons the package REGPACK was written which can be used to solve the following problems: To construct a deterministic automaton from either a nondeterministic one or from an (unrestricted) regular expression, to reduce a deterministic automaton, to test whether a regular language is noncounting, and to construct its syntactic monoid. This package was written in the programming language SPITBOL to be used interactively under the environment of the IBM Virtual Machine Facility/370. It is very fast - due to a great extent to the use of specific features of SPITBOL - and can be used for rather large examples. Furthermore, problems due to space requirements are alleviated since it runs on a virtual machine.

This report gives some information pertaining to REGPACK. It has three major parts. In part I, we give some comments on the implementation of the algorithms without, however, presenting any code. In part II, we discuss the performance of the programme and its limitations (intrinsic ones as well as those of the implementation). In part III, we describe one of the algorithms in more detail.

The reader is expected to be familiar with regular languages, their representations, and operations on them.

I. THE PROGRAMME

REGPACK is a package of programmes to facilitate dealing with regular expressions and finite automata. It contains subprogrammes for the following tasks:

1. Constructing a deterministic automaton from a given unrestricted regular expression, in two versions
   (A) using the method described in III (RGX),
   (B) using derivatives (DERIV).
2. Constructing a deterministic automaton from a given nondeterministic one (MAKE_DET).
3. Reducing a given deterministic automaton (REDUCE).
4. **Constructing the monoid of a given deterministic automaton (MON).** This programme includes various other features (see description).
5. Testing whether a reduced automaton is permutation-free (PERMUTATION).
6. Reversing a given deterministic automaton (REVERSE).

These routines are tied together by an interface which makes it possible, for instance, to obtain the syntactic monoid of a language given by a **regular expression; note that this involves RGX or DERIV, REDUCE, and MON.**

The programme was written for interactive use under the environment of the IBM Virtual Machine Facility/370 as it exists at the Computing Center of the University of Waterloo (circa July 1977). The programming language used is SPITBOL, which is a compiler version for the IBM 360/370 of SNOBOL4 ([6], see also [4]).

In the following we will describe important features of the implementation; we will not, however, give the actual code or comments on it (the source listing is about 40 pages long).

1. <u>RGX</u>

A detailed description of the algorithm can be found in Section III. It may be advisable to refer to that section.

Four procedures form the core of RGX:

UN : given nonreturning finite automata $\underset{\sim}{A}$ and $\underset{\sim}{B}$, UN constructs a nonreturning finite automaton $\underset{\sim}{C}$ such that $L(\underset{\sim}{C}) = L(\underset{\sim}{A}) \cup L(\underset{\sim}{B})$ (UNion);

CON: given nonreturning finite automata $\underset{\sim}{A}$ and $\underset{\sim}{B}$, CON constructs a nonreturning finite automaton $\underset{\sim}{C}$ such that $L(\underset{\sim}{C}) = L(\underset{\sim}{A})L(\underset{\sim}{B})$ (CONcatenation);

ST : given a nonreturning finite automaton $\underset{\sim}{A}$, ST constructs a nonreturning finite automaton $\underset{\sim}{C}$ such that $L(\underset{\sim}{C}) = (L(\underset{\sim}{A}))^*$ (STar);

COMPL:given a nonreturning finite automaton $\underset{\sim}{A}$, COMPL constructs a nonreturning deterministic finite automaton $\underset{\sim}{C}$ such that $L(\underset{\sim}{C}) = \overline{L(\underset{\sim}{A})}$ (COMPLement).

UN, CON, and ST are straightforward implementations, COMPL first finds a deterministic finite automaton $\underset{\sim}{B}$ such that $L(\underset{\sim}{A}) = L(\underset{\sim}{B})$ and then obtains $\underset{\sim}{C}$ by complementing the set of final states of $\underset{\sim}{B}$.

Any regular expression which is to be used in RGX is first preprocessed (in PREPROCESS) as follows:

(a) An internal binary operator (@) for concatenation is inserted.

(b) The expression is fully parenthesized.

(c) Intersection, difference, and symmetric difference are expressed in terms of union and complement.

(d) Multiple brackets and redundant complementations are removed.

Then the basis for the inductive algorithm is established (in BASIS) as follows:

(a) For each element a of the alphabet which occurs in the expression, as well as for I and λ, an automaton accepting $\{a\}$, I, $\{\lambda\}$ respectively, is generated.[†]

(b) For each a, I, λ a token is substituted which refers to the corresponding automaton. (Note that this token refers to the same automaton for various occurrences of the same letter.)

Now, the subprogrammes ST, COMPL, CON, and UN can be applied to the resulting string. Note that the pattern matching facilities of SNOBOL4 make this process rather efficient. In particular whenever an automaton for a certain subexpression has been found the whole expression is scanned for another occurrence of this particular subexpression. If one is found it is also replaced by the token referring to the corresponding automaton. To save memory space tokens which become obsolete can be reused as reference to some other automaton; this mechanism employs a queue of tokens which can be reused (procedures FEEDQUEUE and NEXTAUT). The final result of RGX is then output and stored in a global variable AUT, which can be used later on as input to REDUCE or MON.

[†] I denotes the full monoid A^*, λ the empty word.

2. DERIV

The programme is a fairly direct implementation of the algorithm as described in [3]. At its centre is a procedure (DRIV) to take the derivative of a regular expression (PREFIX_STRING) with respect to a letter (LETTER). This procedure is recursively used and outlined below. As the name of the variable suggests the internal representation of the regular expression is in prefix notation. This is obtained by a recursive procedure (POL) which is applied after an internal operator for concatenation (@) has been inserted. The programme keeps a queue to record the derivatives which still need to be processed. A derivative D is considered to be processed if either D is similar to a processed derivative or the derivative of D with respect to all letters in the alphabet has been constructed. The subroutine DELTA is used to determine recursively whether the empty word λ is contained in the language denoted by a regular expression.

To test for similarity (actually somewhat stronger than in [3]) a standardization procedure is employed (STANDARD) in addition to a simple-minded test for certain equalities (SIMPLE). STANDARD basically rearranges a given derivative so that the operands of every binary commutative operator are in lexicographical order (they are simply treated as strings), and then tests whether two adjacent operands are equal in which case the expression is simplified. STANDARD uses first a simple bucket sort (ORDER), which in turn employs an exchange sort (PRIM_ORD) if necessary. STANDARD and SIMPLE together guarantee that the empty derivative will never appear as a proper subexpression of any derivative.

Computation of DELTA (PREFIX_STRING)

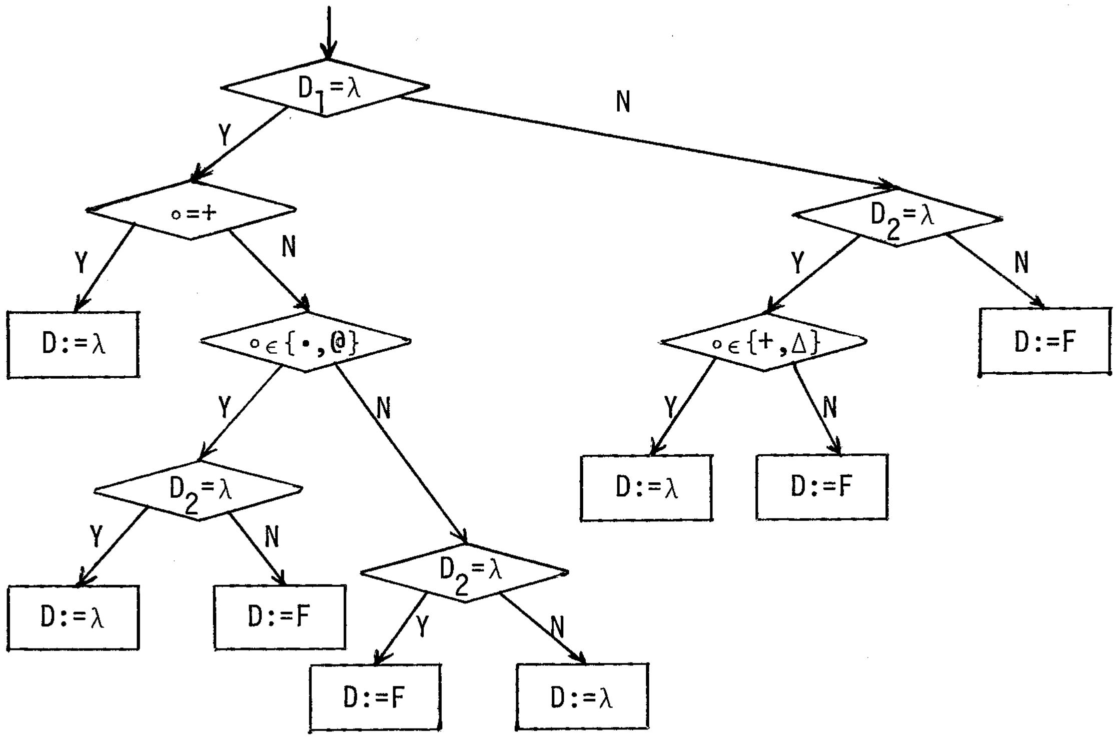
For brevity let S denote PREFIX_STRING. PREFIX_STRING is a simplified and standardized derivative in prefix notation. DELTA(S) returns λ if $\lambda \in L(S)$, otherwise F, the empty derivative.

(a) If S is an atom then DELTA(S) = λ iff S = λ or I (an atom being any letter a in the alphabet, or λ , or I).

(b) If $S = \overline{S_1}$ then DELTA(S) = λ iff DELTA(S_1) = F .

(c) If $S = S_1^*$ then DELTA(S) = λ .

(d) If $S = S_1 \circ S_2$ where $\circ$ is a binary operator, i.e. union (+), intersection ($\cdot$), concatenation (@), difference (-), or symmetric difference (Δ), then the result of DELTA(S) is given in the following flow chart, where D = DELTA(S) , D_i = DELTA(S_i) , i = 1, 2:

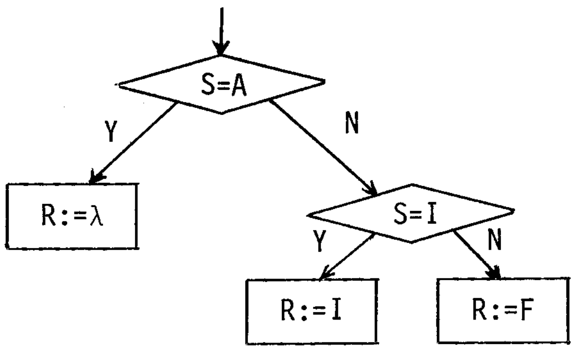


Clearly, for any S DELTA(S) is either λ or F .

<u>Computation of DRIV(LETTER, PREFIX_STRING)</u>

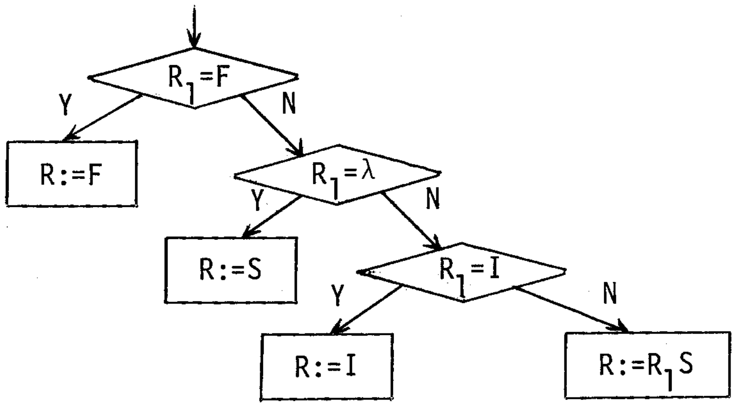
For brevity let S denote PREFIX_STRING, A denote LETTER. S is a simplified standardized derivative in prefix notation. DRIV(A, S) is the derivative of S with respect to the letter A . Let R denote DRIV(A, S) , R_i = DRIV(A, S_i) , i = 1,2 .

(I) S is an atom (i.e. a letter a , or I , or λ)

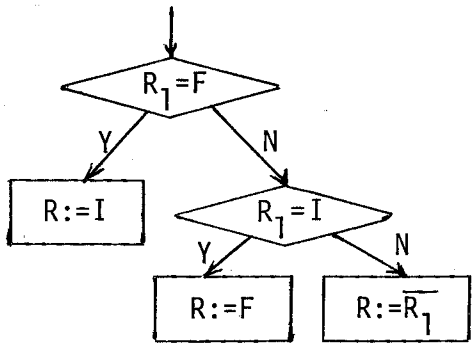


(II) S contains operators

(1) star S = S_1^* ; DRIV(A, S) = DRIV(A, S_1)S

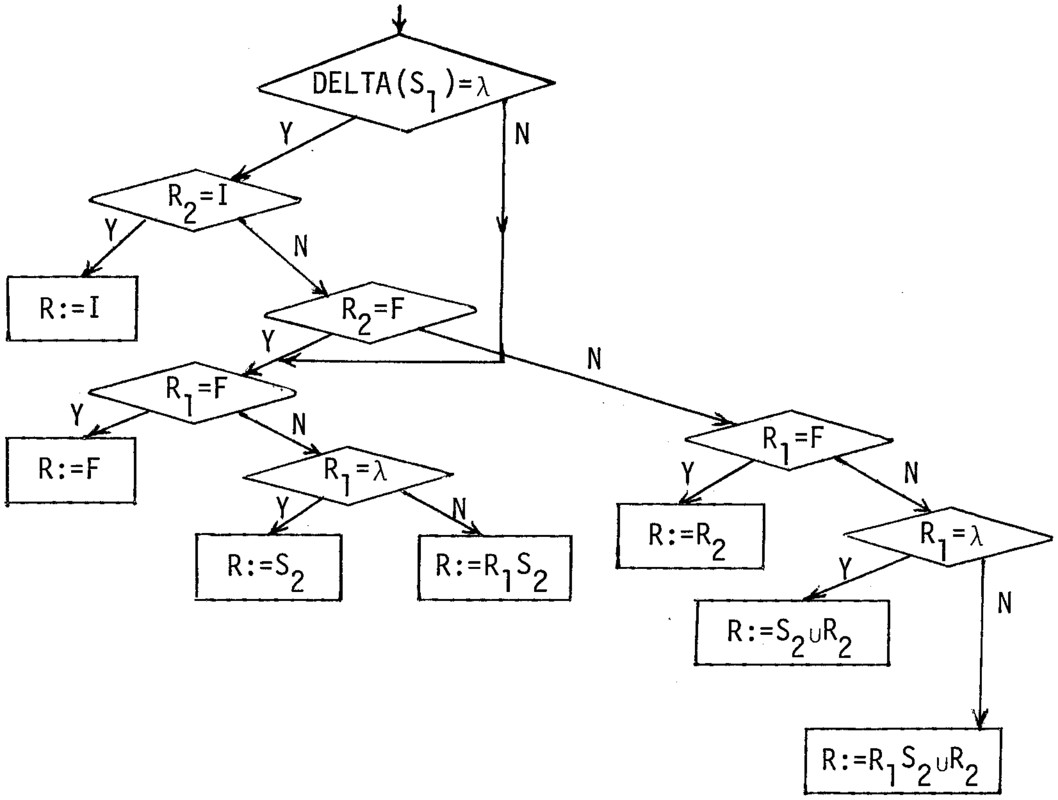


(2) Complement $S = \overline{S_1}$; $DRIV(A, S) = \overline{DRIV(A, S_1)}$



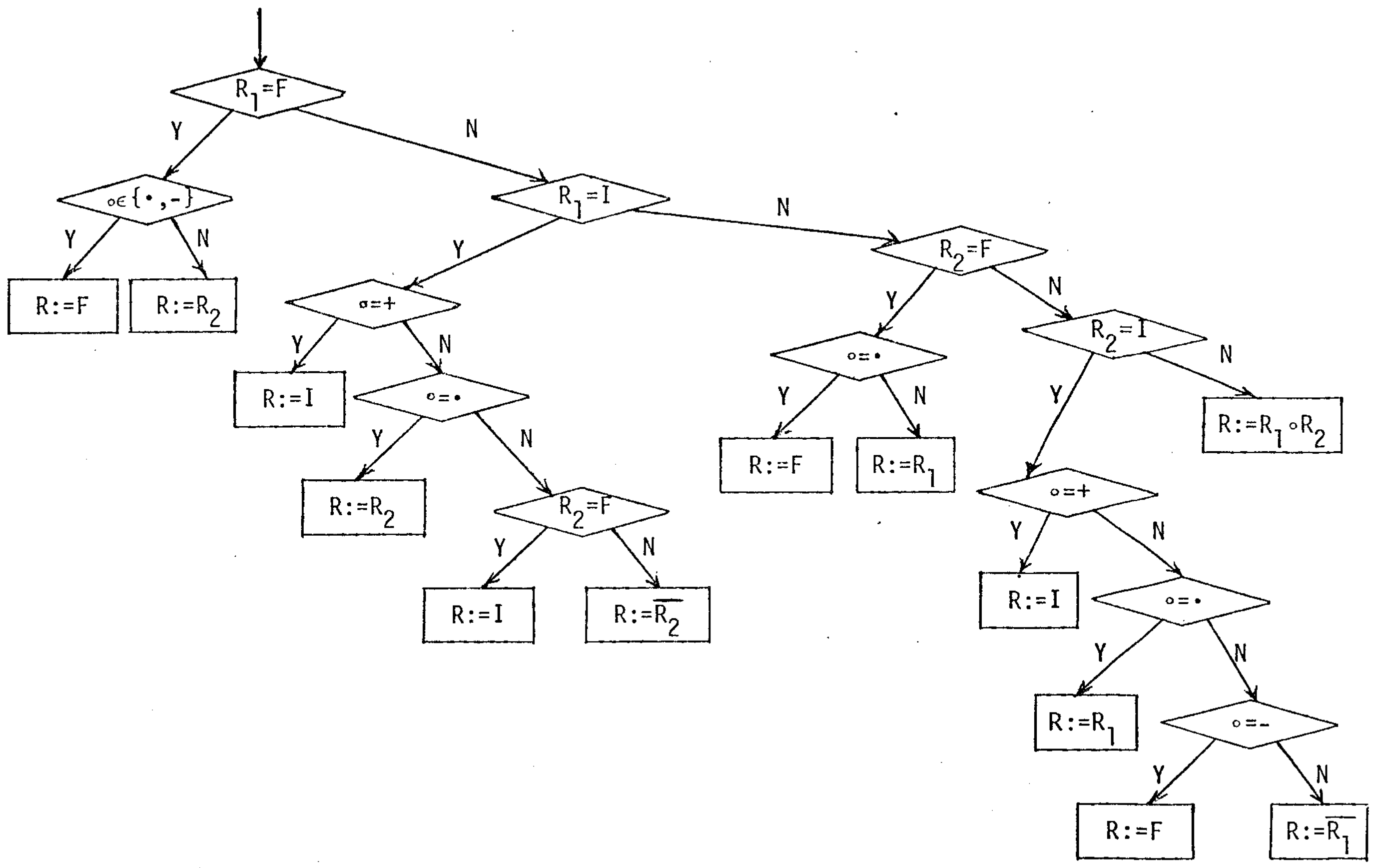
(3) Concatenation $S = S_1S_2$;

$DRIV(A, S) = DRIV(A, S_1)S_2 \cup DELTA(S_1)DRIV(A, S_2)$



(4) Binary boolean operators $S = S_1 \circ S_2$, $\circ \in \{+, \cdot, -, \Delta\}$;

$\mathrm{DRIV}(A, S) = \mathrm{DRIV}(A, S_1) \circ \mathrm{DRIV}(A, S_2)$.



This concludes the computation of DRIV(A, S) . It is easily verified that DRIV(A, S) never contains F as a proper subexpression. Using these derivatives DERIV then constructs a deterministic automaton which can be referenced by AUT for further use by REDUCE or MON.

3. MAKE_DET

This procedure uses the global variable NAUT which refers to a nondeterministic automaton. It creates an equivalent deterministic automaton which can be referenced by the global variable AUT. It employs as primitive a procedure DET which basically performs the subset construction. More specifically it starts with the initial state and puts it into a queue of subsets not yet processed. For each subset in the queue it computes the result of that subset under a letter, for all letters in the alphabet. This result - which clearly is again a subset - is ordered. Then it is checked whether this subset - actually a sequence - was already encountered during the computation so far. If this is not the case the sequence is appended to the queue. DET employs two procedures (MERGE and ORDR) to obtain ordered sequences and uses the built-in tables of SPITBOL to test whether a certain sequence has been already encountered during the process (SYN).

4. REDUCE

This procedure operates on the global AUT and constructs the minimal automaton accepting the same language. It uses the usual reduction algorithm*) i.e. successive refinements of equivalence classes on the set of states starting with the sets of accepting and rejecting states until the partition is consistent with the transition function of AUT (see any standard textbook on automata theory or alg. 2.2 in [2]). The result is then stored in the global variable MIN for possible further use in MON.

---

*) This is a quadratic algorithm. At the moment we do not intend to supply an implementation of Hopcroft's algorithm [7].

5. MON

This procedure operates on the global variable MIN. It assumes that the automaton referenced by MIN does not have more than 26 states because it renames them so that 1 becomes A, 2 becomes B, ... ,26 becomes Z. Let this renamed automaton be $\underset{\sim}{A} = (A, Q, M, q_0, F)$. MON constructs the monoid (or the semigroup) of this automaton. Since it is irrelevant for this construction whether a state is accepting or rejecting this information is ignored for the time being. The method is the usual one; it starts with the n-tuple consisting of the first n letters, ABC ... N (n is the number of states in MIN). If the monoid is to be constructed then this is considered to be the first state of it, if the semigroup is to be constructed only its successors are considered states. The successor of a state under a letter a of the alphabet is the n-tuple (M(A,a), M(B,a), ... ,M(N,a)). This new n-tuple is then also considered a state. Each state coming up in the process which had not been encountered up to this point is placed into a queue. For all states in the queue their successors under each letter of the alphabet are computed.

In addition to the choice between semigroup and monoid, MON also offers partial computations. This is to be understood as follows: Rather than starting the above process with the n-tuple ABC ... N the user can input a nonempty string w over the alphabet and take as the starting n-tuple (M(A,w), ... ,M(N,w)) (again this works for semigroups and monoids). The reason for this is that in some cases the information derived from such a partial semigroup or monoid suffices to determine the structure of the whole semigroup or monoid while requiring considerably less work.

This process of taking successors will eventually terminate; the result yields an automaton, the (partial) semigroup or monoid automaton. This automaton can be printed as well as the multiplication table of the (partial) semigroup or monoid (the latter only if the number of elements is less than 40 - otherwise the table would not fit on one page).

Upon completion of this process of enumerating the elements of the (partial) semigroup S or monoid M the following special features are available. Features flagged with "table" require the multiplication table, similarly for "full monoid" and "full semigroup".

1. The eSe tables for all idempotents e (table), and the tests for definiteness (full semigroup) and for local testability (full semigroup, table) of the input language.
2. The correspondence of the resulting elements to the states of the input automaton.
3. Input of a set of elements of the monoid (semigroup); the programme determines the smallest submonoid (subsemigroup) containing this set (table).
4. Principal ideals fM, Mf, MfM for $f \in M$, using the procedures P_IDEAL and REACHABLE (full monoid).
5. $M_f = \{g \in M \mid f \in MgM\}^*$ using procedure M_SUB_F (full monoid), and analogous tests as in 1. (Hence, * denotes the monoid closure.)

The tests mentioned in 1 are as follows:

$$\text{The language is } \left\{\begin{array}{l}\text{finite/cofinite}\\ \text{definite}\\ \text{reverse definite}\\ \text{generalized definite}\end{array}\right\} \text{ iff } \left\{\begin{array}{l} eS \cup Se = e\\ Se = e\\ eS = e\\ eS \cap Se = e\end{array}\right\} \text{ for all}$$

idempotents e of its semigroup S. (Note that $eS \cap Se = eSe$.) Furthermore the language is locally testable iff all eSe monoids are idempotent (ff=f) and commutative (fg=gf).

The significance of the eSe monoids is amply demonstrated in the literature; for example see [24], [22], [23].

The corresponding tests in 5 are:

Tests for $eM_e \cup M_e e = e$, $eM_e = e$, $M_e e = e$, and $eM_e \cap M_e e = e$ for all idempotents e of the monoid M, and further tests whether the $eM_e e$ ($= eM_e \cap M_e e$) monoids are idempotent and commutative ([21], [22]). We make use of the fact that $g \in M_f$ iff $g \in MaM$ for some $a \in M$ whose corresponding congruence class of A* contains a letter of A. This considerably reduces the amount of work to be done. Furthermore the list of elements in MaM for all such $a \in M$ is saved to avoid duplicating computations.

MON also uses a procedure (TRANSFORMATION) to compute the starting sequence if a partial semigroup or monoid is needed.

6. PERMUTATION

This is a procedure to determine whether a given reduced automaton $\underset{\sim}{A} = (A,Q,M,q_0,F)$ is permutation-free, i.e. for all nonempty subsets $\{p_1,\ldots,p_r\} \subseteq Q$ and all words $x \in A^*$ $(M(p_1,x),\ldots,M(p_r,x))$ is either the identity permutation of $(p_1,\ldots,p_r)$ or no permutation at all. This condition is known to be equivalent to the following statements:

(1) $L(\underset{\sim}{A})$ can be denoted by a star-free expression - a regular expression which does not use the star operator

(2) $L(\underset{\sim}{A})$ is noncounting

(3) The syntactic monoid of $L(\underset{\sim}{A})$ is group-free; i.e. doesn't have nontrivial subgroups.

The test whether $\underset{\sim}{A}$ is permutation-free is far cheaper than any of the other tests. This algorithm can be found in [19].

PERMUTATION operates as follows: Starting with the sequence $T_\lambda = (1,\ldots,n)$ it computes in depth-first fashion T_{wa} which is defined as follows for $w \in A^*$, $a \in A$: If $T_w = (p_1,\ldots,p_r)$ then $T'_{wa} = (M(p_1,a),\ldots,M(p_r,a))$. If T_w has as many different elements as T'_{wa} then $T_{wa} = T'_{wa}$, otherwise T_{wa} is the sequence obtained from T'_{wa} by discarding repetitions and ordering it. If T_{wa} has either only one element or is a sequence already encountered there is no need to consider T_{wau} for any $u \in A^*$. If T_{wa} is a permutation of a sequence T_v previously encountered and v is a prefix of wa then $\underset{\sim}{A}$ is not permutation-free (note that we compute T_w in depth-first fashion i.e. if $A = \{0,1\}$ we compute all possible T_{0^i}'s <u>before</u> we compute T_{0^j1}, and so on).

8. The Interface

The interface constitutes the main programme from which the functions described above are called. In addition to these it also uses functions IN_AUT and OUT_AUT for input and output of finite automata (not necessarily deterministic). It makes use of three global variables: AUT, NAUT, and MIN. AUT references a deterministic automaton obtained either by input via IN_AUT or from RGX or DERIV as result, NAUT references a nondeterministic automaton obtained by input via IN_AUT, and MIN references a reduced automaton obtained as result from REDUCE. All automata are represented in the same way namely as a table C , where C<-1> contains the cardinality of the underlying alphabet, C<0> contains the number of states of the automaton, and C<X,L> is the result of state X under letter L of the alphabet.

Although NAUT, AUT, and MIN are global variables, the information they refer to is erased after the full cycle of possible operations is exhausted. Typical computation cycles might be:

IN_AUT, MAKE_DET, REDUCE, PERMUTATION or

DERIV, REDUCE, MON.

While it usually is sufficient to keep the information just for one cycle sometimes one might want to store some information for several cycles. Therefore the interface allows for saving resulting automata for further use. It creates unique names for them which then can be used as tokens in regular expressions in RGX. If they are not needed any longer they can be released (collectively or selectively).

7. REVERSE

REVERSE is a procedure which finds a deterministic automaton accepting the reverse of the language accepted by the given automaton. The algorithm is described in [20] and can be briefly stated as follows. We first constructed an intermediate nondeterministic automaton with the same set of states as the given automaton. Its set of initial states consists of the final states of the given automaton, its set of final states consists of the initial state of the given automaton, and its transition function is the inverse of the given transition function. Then the result of REVERSE is simply the result of DET (see MAKE_DET) applied to the intermediate automaton. This automaton will always be reduced.

9. IN_AUT

This procedure is used to input finite automata. It is the only way to initialize NAUT. The states of any automaton are the consecutive numbers 1 through n where $1 \leq n \leq 99$. Any alphabet throughout the programme - this also applies to DERIV and RGX - is the set of consecutive numbers 0 through ℓ where $0 \leq \ell \leq 9$; therefore $\ell+1$ is the cardinality of the alphabet. The automaton is represented as described in 8. Furthermore in $C<X, \ell+1>$ it is recorded as λ if state X in automaton C is accepting; otherwise $C<X, \ell+1> = F$. If NAUT is to be initialized the result of state X under letter a has to be written as a sum of states e.g. 3+4+12 without blanks. For the empty sum the symbol F should be typed. This also applies to subsequent corrections.

10. OUT_AUT

OUT_AUT is a procedure to output a finite automaton. This is done by printing the transition table with the understanding that 1 is always the initial state. Since it also outputs nondeterministic automata the width of the columns is variable, this number being determined as the length of the longest sum plus three. It should be noted that this can result in not very readable output if the alphabet as well as the width is large since there are only 120 positions in a line. However, it is hoped that these occasions are rare.

This concludes the section dealing with the programme itself. The next section will be devoted to performance and limits of the programme.

II. PERFORMANCE AND LIMITATIONS

It is known ([12], [17]) that of the 4 main problems listed at the beginning of the preceding section only the reduction of a given deterministic automaton has a computationally efficient solution, in the sense that there is an algorithm bounded by a polynomial in time and space ([7]). While the user should be aware that some computations are not possible or at least not feasible in the case of the other three problems, this should not be a deterrent from using the known methods at all. Practice shows that in almost all cases a solution can be found.

However before we investigate performance and limits of the package we will give - as warning - some unpleasant classes of problems.

In Section III it is shown that for regular expressions involving only union, concatenation, and star (restricted regular expressions) the number of states of the corresponding deterministic automaton is bounded by 2^n+1 if n is the number of letters in the expression. One might hope for an improvement of this bound, however it turns out that one cannot get rid of the exponential nature of any bound for this problem. This is indicated by the following class of expressions:

$$\alpha_n = ([(10*)^{n-1}1]*[01*01*]*)* \quad .$$

Clearly, $s_1(\alpha_n) = n+2$, hence using RGX_1' we obtain the bound $2^{n+2}+1$ on the number of states of the automaton accepting $L(\alpha_n)$ (for the definitions of s_1 and RGX_1' see Section III). Furthermore α_n is accepted by the following finite automaton

$\underset{\sim}{A}_{\alpha_n}$	0	1	
$\rightarrow A_o$	B	A_1	λ
A_1	A_1	A_2	ϕ
$\vdots$			$\vdots$
A_i	A_i	A_{i+1}	ϕ
$\vdots$			$\vdots$
A_{n-1}	A_{n-1}	A_n	ϕ
A_n	B	A_1	λ
B	C	B	ϕ
C	B	C,A_1	λ .

The following recurrence relation for the number N_n of states of the reduced automaton $\underset{\sim}{B}_n$ accepting $L(\alpha_n)$ is suggested by the first 5 numbers:

$$N_2 = 8 \ , \quad N_{n+1} = 2N_n - (n-1) \ , \quad n \geq 2 \ .$$

Indeed, the following numbers have been obtained from the programme

$$N_2 = 8$$
$$N_3 = 15$$
$$N_4 = 28$$
$$N_5 = 53$$
$$N_6 = 102 \ .$$

From the recurrence relation one obtains the following formula

$$N_n = 2^{n+1} - \sum_{j=0}^{n-3} 2^j(n-j-2) \ , \ n \geq 2 \ \text{(for } n = 2 \text{ the sum is empty)}$$

and from this formula one easily deduces

$$N_n \geq (1.9)^{n+1} \quad \text{for all} \quad n \geq 2 .$$

However, most expressions with which the programme was tested did not show such an undesirable behaviour.

Similar examples can be exhibited for MAKE_DET and MON. First the conversion from nondeterministic to deterministic automaton. The following class of automata $\underset{\sim}{A}_n = (\{0,1,2\}, \{1,2,\dots,n\}, M_n, 1, F)$ has the property that applying the subset construction to them yields deterministic automata with 2^n-2 states,

M_n	0	1	2
→ 1	2	2	1
2	3	1	2
3	4	3	3
4	5	4	4
⋮	⋮	⋮	⋮
i	i+1	i	i
⋮	⋮	⋮	⋮
n-1	n	n-1	n-1
n	1	n	1,2

(final states need not be specified). The proof can be sketched as follows: The first 2 columns can be viewed as permutations of $1,\dots,n$. These two permutations generate the symmetric group on n elements. The last column is simply used to obtain a subset of $i+1$ states when given one with i states, for $i < n-1$. This shows that any nonempty subset of $\{1,\ldots,n\}$ different from $\{1,\ldots,n\}$ can be obtained, which proves the claim.

A similar construction applies to MON. Given a deterministic automaton with n states one easily derives n^n as bound on the number of states of the transformation monoid. Recall that each state is a sequence of n states of the original automaton. Since repetitions of the same state in the sequence may occur this shows n^n is a bound. The following class of automata $\underset{\sim}{B}_n = (\{0,1,2\}, \{1,2,\ldots,n\}, M_n, 1, F)$ attains this bound,

M_n :	0	1	2
$\rightarrow 1$	2	2	1
2	3	1	1
3	4	3	3
4	5	4	4
⋮	⋮	⋮	⋮
i	$i+1$	i	i
⋮	⋮	⋮	⋮
$n-1$	n	$n-1$	$n-1$
n	1	n	n

(final states again need not be specified). The proof is similar to the previous one; however, rather than obtaining a subset of $i+1$ elements from one with i elements, the last column is used to obtain an n-tuple with $i+1$ repetitions from one with i repetitions. This shows that any

188181818178.

Yet PERMUTATION spent only about 1.6 sec to perform this test.

<u>MAKE_DET</u>:

(a) To compute a deterministic equivalent automaton for $\underset{\sim}{A}_{\alpha_n}$ for $n = 6$ (using the subset construction) MAKE_DET spent about 5 sec (the result has 166 states).

(b) To compute MAKE_DET($\underset{\sim}{A}_n$) for $n = 4,5,6,7$ the programme used about .4 sec, .9 sec, 2 sec, and 4.7 sec (the results have 14, 30, 62, and 126 states, respectively).

Rather than listing results of RGX and DERIV we will compare the performance of the 2 routines for various expressions.

The following expressions were used for the comparison, the alphabet always being {0, 1} :

(1) $\{0[01(01)^* \cup 10I \cup \lambda] \cup 1[10(10)^* \cup 01I]\} \cap (010 \cup 101)^*$

(2) $\overline{\overline{\overline{\overline{0}101}01}01}01$

(3) $(0 \cup 1^*)0\overline{\{00^*\overline{[0 \cup 01I \cup (00 \cup 11^*0)[00^*11^*0]^*(\lambda \cup 1I)]}\}}$

(4) $01^*[(01^*01^*)^* \cup 1^*] \,\Delta\, (\overline{\{10^*\overline{[(01^*01^*)^* \cup 1^*]}\}} \cup \lambda)$

(5) $[010(11)^* \cup 00(010)^* \cup (00)^*11]^*$

(6) $[11(000 \cup 101)^* \cup 10(111 \cup 010)^*1]^*[00 \cup 11]^*$

(7) $[0(00 \cup 01)^* \cap 0(10 \cup 11)^*] \cap (101010)^*$

The results are listed in the following table; column "sec" gives the number of seconds to compute an automaton for the expression in question, column "states" gives the number of states this automaton has, column "sec(r)" gives the number of seconds it took to reduce this particular sequence consisting of elements of {1,...,n} can be obtained, and therefore proves the claim.

Clearly the above examples show some of the inherent limits of any programme dealing with these problems. However, as mentioned before, it should be kept in mind that these are <u>exceptions</u> and not average cases.

On the other hand we believe that this programme is among the fastest available. This must be attributed to a large extend to the choice of SPITBOL (the compiler version of SNOBOL4 for IBM 360/370 machines) as programming language and the advantageous use of its pattern matching abilities as well as other features of this language.

But let the results speak for themselves.

<u>MON</u>:

(a) To compute the monoid automaton for

	0	1	2
1	2	1	1
2	4	2	1
3	1	4	3
4	3	3	4

which has 256 states (a similar example as the ones given above). MON spent about 12 sec.

(b) To compute the monoid automaton for

	0	1	2
1	1	3	2
2	4	5	2
3	3	2	5
4	2	2	4
5	1	5	4

which has 367 states MON spent about 24 sec.

PERMUTATION: This procedure is much faster than MON (note that from the syntactic monoid one can also retrieve whether an automaton is permutation-free). For instance, consider the following reduced automaton

	0	1	2	3	4	5	6	7	8	
1	1	1	1	1	1	1	1	1	2	λ
2	1	2	2	2	2	2	2	2	3	ϕ
3	3	2	3	3	3	3	3	3	4	ϕ
4	4	4	3	4	4	4	4	4	5	ϕ
5	5	5	5	4	5	5	5	5	6	ϕ
6	6	6	6	6	5	6	6	6	7	ϕ
7	7	7	7	7	7	6	7	8	8	ϕ
8	8	8	8	8	8	8	7	7	1	ϕ

The monoid of this automaton has 8^8 states (16777216; for a proof consider columns 0, 7, 8), thus its computation would present a bit of a problem. Furthermore it is unfairly slated against PERMUTATION which starts its depth-first search with 0 . Thus the first word enacting a permutation (on (1, 3)) is

The reason why RGX had problems with (1) and (7) is simply the fact that in both cases a certain redundancy (bias) is built in. DERIV soon discovers in (1) that it is enough to consider $(010I \cup 101I) \cap (010 \cup 101)^*$ while RGX has to compute the whole expression. Expression (7) is even worse; since everything in the first part starts with a 0 and in the second part with a 1 the intersection must be empty. This is easily detected by DERIV whereas RGX has to compute everything. That RGX "fails" with (4) has its reason in the fact that RGX rewrites $\alpha \Delta \beta$ as $\overline{\bar{\alpha} \cup \beta} \cup \overline{\bar{\beta} \cup \alpha}$ which is far more complicated while DERIV can handle Δ directly.

Turning to DERIV's problems the user is advised not to use it if the expression to be dealt with contains stars within the scope of other stars. This usually yields very long expressions for the derivatives making the computation very costly. Expression (6) is an example for this effect - a rather harmless one, though; it would have been not too difficult to give an example where DERIV runs out of memory space or uses too much time to finish at all (any of the α_n's mentioned in the beginning for $n \geq 4$ will do).

In summary DERIV should be used if built-in redundancies are expected and no complicated starred subexpressions occur within the scope of other stars. Furthermore DERIV must be used if one needs expressions for the states of the resulting automaton which are structurally related to the original expression. In the other cases it is probably cheaper to use RGX, especially for very large expressions. Furthermore, it appears that the sequential operations, i.e. concatenation and star, are better automaton, and column "states(r)" gives the number of states of the reduced automaton for the expression in question (this clearly is the same for both methods), column "total" finally gives sec + sec(r).

RGX					DERIV				expression
total	sec	states	sec(r)	states(r)	sec(r)	states	sec	total	
4.0	2.9	30	1.1	7	0.2	11	1.5	1.7	(1)
2.0	1.2	12	0.8	12	0.5	12	6.2	6.7	(2)
8.8	4.8	79	4.0	6	0.5	22	12.0	12.5	(3)
6.3	5.6	20	0.7	5	0.2	12	2.3	2.5	(4)
1.7	1.4	13	0.3	5	0.3	15	6.5	6.8	(5)
5.1	3.1	40	2.0	35	1.7	37	33.5	35.2	(6)
1.9	1.9	15	-	1	-	2	0.3	0.3	(7)

From these results (and from further experience) it can be concluded that RGX tends to compute automata with more states which subsequently need more time to be reduced (expression(5) is rather an exception). However, this is usually done faster. The main reason is that RGX reuses automata for subexpressions which occur more than once; also important is that it only operates on automata therefore considerations such as the length of a derivative - which is very crucial for DERIV - are irrelevant. On the other hand it should be noted that DERIV actually computes considerably more than just an automaton. In fact it computes a regular expression for each state of the automaton which is structurally very closely related to the original expression.

handled by RGX while the boolean operators are better handled by DERIV. Thus if a large expression involves many boolean operators but very few concatenations and stars, the use of DERIV might be advisable.

One last advice: If efficiency is an important issue when using RGX it should be kept in mind that RGX parenthesizes the input expression. Therefore the programme might not recognize common subexpressions.

For example, the input 00000000 takes .46 seconds while ((00)(00))((00)(00)) takes only .36 seconds, because RGX transforms 00000000 into ((((((00)0)0)0)0)0)0 and there are no common subexpressions. Therefore proper parenthesization can help saving execution time! (This does not apply to DERIV.)

III. SOME REMARKS ON ALGORITHMS TO TRANSLATE EXPRESSIONS INTO AUTOMATA

This part of the report will be devoted to some comments on, and explanations of, algorithms to translate regular expressions into finite automata in general; in particular, we will discuss some aspects of the derivative method (which is described in [3]) and present the algorithm (in fact class of algorithms) underlying RGX. First, however, we will define the basic terms.

Let A be an alphabet; (unrestricted) regular expresssions (r.e.) over A are defined inductively:

(1) (Basis) The empty word λ and the empty set ϕ are r.e.'s denoting the languages $\{\lambda\}$ and ϕ, respectively. For all $a \in A$, a is an r.e. denoting the language $\{a\}$.

(2) (Induction step) If α, β are r.e.'s denoting the languages $L(\alpha)$, $L(\beta)$, respectively, then so are

$$\alpha \cup \beta \ , \quad \alpha \cap \beta \ , \overline{\alpha} \ , \quad \alpha \cdot \beta \ , \quad \alpha^*$$

denoting the languages

$L(\alpha) \cup L(\beta)$ (union), $L(\alpha) \cap L(\beta)$ (intersection), $\overline{L(\alpha)}$ (complement), $L(\alpha) \cdot L(\beta)$ (concatenation), and $(L(\alpha))^*$ (star), respectively.

Every r.e. can be obtained by a finite number of applications of rules (1) and (2).

A restricted regular expression is an r.e. involving only union, concatenation and star as operations. Any regular expression (restricted or unrestricted) denotes a regular language.

A finite automaton $\underset{\sim}{A}$ is a quintuple

$$\underset{\sim}{A} = (A, Q, M, q_0, F)$$

where A is the input alphabet, Q is the finite nonempty set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and M is the transition function, defined by

$$M : Q \times A \to P_0(Q)$$

where $P_0(Q)$ denotes the set of nonempty subsets of Q. Note, that for convenience we do not allow the empty states. If $M(q,a)$ consists of one element for all $q \in Q$, $a \in A$, $\underset{\sim}{A}$ is called deterministic. As usual, M is extended to $M : P_0(Q) \times A^* \to P_0(Q)$ as follows:

$$\begin{array}{ll} M(P, a) = \bigcup_{q \in P} M(q, a) & \text{for } P \in P_0(Q) \text{ , } a \in A \text{ ,} \\ M(q, \lambda) = q & \text{for } q \in Q \text{ , and} \\ M(q, wa) = M(M(q, w), a) & \text{for } q \in Q, w \in A^*, a \in A \text{ .} \end{array}$$

A word $w \in A^*$ is accepted by $\underset{\sim}{A}$ iff $M(q_0, w) \cap F \neq \phi$. $L(\underset{\sim}{A})$ is the set of words accepted by $\underset{\sim}{A}$, $L(\underset{\sim}{A}) = \{w \in A^* \mid M(q_0, w) \cap F \neq \phi\}$; it is always a regular language. For any finite automaton there exists a unique deterministic automaton accepting the same language, which has the minimal number of states; this automaton is called reduced.

A regular expression is commonly considered to be a very neat representation of a regular language. So why would one want to obtain an automaton accepting this language? Well, one of the drawbacks of regular expressions is that it is usually quite difficult to determine whether a given word is in the corresponding regular language. Take for instance the word $w = 001010100$ and the expression α over $\{0, 1\}$,

$$\alpha = \overline{\overline{\overline{\overline{\overline{0}0}0}0}0} \text{ .}$$

Is $w \in L(\alpha)$? Hard to say. But it certainly helps, if one knows that $L(\alpha)$ is accepted by $A_\alpha = (\{0, 1\}, \{1,2,3,4,5,6,7\}, M, 1, \{1,3,5,7\})$,

M being given by

	0	1
1	2	3
2	3	3
3	4	3
4	5	3
5	6	3
6	7	3
7	7	3

It is very simple to compute $M(1, w) = 5 \in F$ which implies $w \in L(\alpha)$. Moreover, there is a less tangible benefit derived from $\underset{\sim}{A}_\alpha$ namely that we can obtain a "simpler" expression for the same language. For we might realize that $L(\underset{\sim}{A})$ consists of all words $\neq 0,000,00000$, which do not end in 10 or 1000;therefore,abbreviating $I = \{0, 1\}^*$, we can write

$$\alpha' = \overline{I10(\lambda \cup 00) \cup 0 \cup 000 \cup 00000} \ .$$

Clearly α' is a more convenient expression for $L(\alpha)$ than α itself if we want to test for membership, since $w \in L(\alpha')$ iff $w \notin L(\overline{\alpha'})$, and the last condition is easily tested.

Having illustrated the desirability of a method to obtain a deterministic automaton for a given regular expression, we will review some of the algorithms.

A very commonly used method involves taking derivatives; it is usually referred to as derivative method (see [3]) and can be described as follows: Define functions D_w from regular expressions (over A) to regular expressions, for all $w \in A^*$, by:

(1) $D_\lambda(\alpha) = \alpha$ for all regular expressions α

(2) $D_{va}(\alpha) = D_a(D_v(\alpha))$ for $v \in A^*$, $a \in A$

(3) (a) $D_a(\phi) = \phi$, $D_a(\lambda) = \phi$, $D_a(b) = \phi$ for $b \in A - \{a\}$, and $D_a(a) = \lambda$

(b) $D_a(\alpha \cup \beta) = D_a(\alpha) \cup D_a(\beta)$

$D_a(\alpha \cap \beta) = D_a(\alpha) \cap D_a(\beta)$

$D_a(\overline{\alpha}) = \overline{D_a(\alpha)}$

$D_a(\alpha \cdot \beta) = D_a(\alpha) \cdot \beta \cup \delta(\alpha) \cdot D_a(\beta)$

$D_a(\alpha^*) = D_a(\alpha) \cdot \alpha^*$

where δ is a function from regular expressions onto the set consisting of the two regular expressions λ and ϕ, defined as follows:

$\delta(\lambda) = \lambda$, $\delta(\phi) = \phi$, $\delta(a) = \phi$ for $a \in A$,

$\delta(\alpha \cup \beta) = \delta(\alpha) \cup \delta(\beta)$, $\delta(\alpha \cap \beta) = \delta(\alpha) \cap \delta(\beta)$, $\delta(\overline{\alpha}) = \lambda$ iff $\delta(\alpha) = \phi$, $\delta(\alpha \cdot \beta) = \delta(\alpha) \cdot \delta(\beta)$, and $\delta(\alpha^*) = \lambda$.

Then $D_w(\alpha)$ is called the derivative of α with respect to w. It can be shown that every regular expression has a finite number of dissimilar derivatives where similarity is an equivalence relation on regular expressions defined as follows:

Two regular expressions α and β are similar iff β can be obtained from α by use of the following equalities:

$\gamma\phi = \phi\gamma = \phi$, $\phi \cup \gamma = \gamma$, $\gamma\lambda = \lambda\gamma = \gamma$, $\gamma \cup \gamma = \gamma$, $\gamma \cup \gamma' = \gamma' \cup \gamma$, $\gamma \cup (\gamma' \cup \gamma'') = (\gamma \cup \gamma') \cup \gamma''$.

Clearly, α similar to β implies $L(\alpha) = L(\beta)$, but the converse is not true, as the example at the beginning shows. It should be noted that these rules can easily be applied mechanically.

Now, we can summarize the derivative method :

Let $A^* = \{w_1, w_2, \ldots\}$ such that $|w_i| \leq |w_{i+1}|$ for all $i = 1,2,\ldots$ ($|w|$ denotes the length of w). Generate $D_{w_i}(\alpha)$ for $i = 1,2,\ldots$ where α is the given regular expression and check whether $D_{w_i}(\alpha)$ is similar to any of the $D_{w_j}(\alpha)$ for $j < i$. If this is the case then remove all w's from A^* where w_i is a prefix of w . Since there are only finitely many dissimilar derivatives this process terminates. Then define the automaton

$$\underset{\sim}{A}_{\alpha} = (Q, A, M, q_0, F)$$

as follows: Q is the set of equivalence classes, $Q = \{[D_w(\alpha)] \mid w \in A^*\}$ (clearly Q is finite), $q_0 = [D_\lambda(\alpha)] = [\alpha]$, F is the set of all those $[D_w(\alpha)]$ for which $\delta(D_w(\alpha)) = \lambda$, and M is defined by

$$M([D_w(\alpha)], a) = [D_{wa}(\alpha)] \quad w \in A^*, \quad a \in A .$$

It follows that $L(\alpha) = L(\underset{\sim}{A}_{\alpha})$.

It should be clear that by introducing additional equations in the definition of similarity the algorithm can be improved upon, in the sense that the resulting automaton will have fewer states, in general. In fact, any equation will do as long as it implies equality of the languages involved. Now, the obvious idea is to introduce "enough" equations in order to strengthen similarity such that α similar to β is equivalent to $L(\alpha) = L(\beta)$ - the ultimate goal, for then we would obtain a "smallest", i.e. the reduced automaton. Yet, this is impossible since it is known that there is no finite set of equations such that this can be achieved ; this even holds for restricted regular expressions [16] (see also [5]). We therefore are in the somewhat puzzling situation that by introducing arbitrarily many equations we can get an arbitrarily "good" result but not the optimal one - which nevertheless does exist, namely the reduced automaton for α .

This algorithm is implemented in REGPACK (DERIV); for more details of the algorithm see [3], some remarks on the implementation can be found in Section I.

While the derivative method is a very general algorithm, various methods have been developed for the special case of restricted regular expressions. Common to all of them is that they implicitly use the fact that finite automata are "closed" under union, concatenation, and star, precisely the operations involved in restricted regular expressions. Closure of a class $\mathcal{C}$ of finite automata under an m-ary operation b_m , $m \geq 1$, will here be used in the following sense: Given automata $\underset{\sim}{A}_i \in \mathcal{C}$, $i=1,..,m$, there is a "direct" way to construct an automaton $\underset{\sim}{A} \in \mathcal{C}$ such that

$$L(\underset{\sim}{A}) = b_m(L(\underset{\sim}{A}_1),\ldots,L(\underset{\sim}{A}_m)) \quad ,$$

"direct" meaning that the construction of $\underset{\sim}{A}$ is of time complexity linear in the sum of the number of states of the $\underset{\sim}{A}_i$. The algorithms we have in mind are those in [1 (ch. 9.1), 5 (ch. 4), 8, 9, 11, 17]. Rather than describing any of these algorithms we will give an extendible generalization of the algorithm by Mirkin ([11]). The simplest version of this algorithm forms the basis of RGX, the second general algorithm to translate regular expressions into deterministic finite automata.

A finite automaton $\underset{\sim}{A} = (Q,A,M,q_0,F)$ in which no transition goes back to the initial state q_0 (that is $M(q_0,x) \cap \{q_0\} = \phi$ for all $x \in A \cdot A^*$) is called nonreturning. Clearly, if $\underset{\sim}{A}$ does not satisfy this property then $\underset{\sim}{A}' = (Q \cup \{q_0'\}, A, M', q_0', F')$ does, where $q_0' \notin Q$, $M'(p, a) = M(p, a)$ for $p \in Q$, and $M'(q_0', a) = M(q_0, a)$, $F' = F$ if $q_0 \notin F$, otherwise $F' = F \cup \{q_0'\}$. Furthermore $L(\underset{\sim}{A}) = L(\underset{\sim}{A}')$.

Assume now that $\underset{\sim}{A}_i$, $i=1,2$, are nonreturning finite automata, $\underset{\sim}{A}_i = (Q_i, A, M_i, q_0, F_i)$, where $Q_1 \cap Q_2 = \{q_0, \square\}$, and $\square$ is a rejecting state such that $M_i(\square, a) = \square$ for all $a \in A$, $i = 1,2$. (If there is no such state in Q_i for some $i \in \{1, 2\}$ then $Q_1 \cap Q_2 = \{q_0\}$.) We will construct nonreturning finite automata $\underset{\sim}{B}_1, \underset{\sim}{B}_2, \underset{\sim}{B}_3$, directly from $\underset{\sim}{A}_1$ and $\underset{\sim}{A}_2$, such that $L(\underset{\sim}{B}_1) = L(\underset{\sim}{A}_1) \cup L(\underset{\sim}{A}_2)$, $L(\underset{\sim}{B}_2) = L(\underset{\sim}{A}_1)^*$, $L(\underset{\sim}{B}_3) = L(\underset{\sim}{A}_1) \cdot L(\underset{\sim}{A}_2)$.

(a) Union: $\underset{\sim}{B}_1 = (Q_1 \cup Q_2, A, N_1, q_0, G_1)$ where $G_1 = F_1 \cup F_2$, N_1 as follows:

$$N_1(q, a) = \begin{cases} M_1(q_0, a) \cup M_2(q_0, a) & \text{if } q = q_0 \\ M_i(q, a) & \text{if } q \in Q_i - \{q_0\} \text{ for some } i \end{cases}$$

(b) Star: $\underset{\sim}{B}_2 = (Q_1, A, N_2, q_0, F_1 \cup \{q_0\})$, N_2 as follows:

$$N_2(q, a) = \begin{cases} M_1(q, a) & \text{if } q \in Q_1 - F_1 \\ M_1(q, a) \cup M_1(q_0, a) & \text{if } q \in F_1 \end{cases} .$$

(c) Concatenation: $\underset{\sim}{B}_3 = (Q_1 \cup Q_2, A, N_3, q_0, G_3)$ where $G_3 = F_2$ if $q_0 \notin F_2$, otherwise $G_3 = F_1 \cup (F_2 - \{q_0\})$, and N_3 as follows:

$$N_3(q, a) = \begin{cases} M_1(q, a) & \text{if } q \in Q_1 - F_1 \\ M_1(q,a) \cup M_2(q_0,a) & \text{if } q \in F_1 \\ M_2(q, a) & \text{if } q \in Q_2 \end{cases} .$$

The proofs are easily supplied. Clearly if $\underset{\sim}{A}_i$ has n_i states different from $\square$ then $\underset{\sim}{B}_1$ and $\underset{\sim}{B}_3$ have n_1+n_2-1 such states, and $\underset{\sim}{B}_2$ has n_1 such states.

These constructions give rise to an inductive method for obtaining a finite automaton for a given restricted regular expression provided we can find a suitable basis to start the induction. To this end it suffices to supply a nonreturning finite automaton for each of the expressions ϕ, λ, a for all $a \in A$:

$\underset{\sim}{A}_\phi = (\{1,\square\}, A, M_\phi, 1, \phi)$ where $M_\phi(q, a) = \square$ for $q \in Q$, $a \in A$.

$\underset{\sim}{A}_\lambda = (\{1,\square\}, A, M_\lambda, 1, \{1\})$, $M_\lambda = M_\phi$.

$\underset{\sim}{A}_a = (\{1,2,\square\}, A, M_a, 1, \{2\})$ where $M_a(1, a) = 2$, $M_a(q, b) = \square$ for all $q \in Q$, $b \in A$ such that $q \neq 1$ or $b \neq a$.

Therefore each of the regular expression $\phi, \lambda, a \in A$ is accepted by a nonreturning finite automaton with at most two states $\neq \square$. Thus to every restricted regular expression containing m occurrences of elements in $\{\phi,\lambda\} \cup A$ there exists a nonreturning finite automaton with at most $m+1$

states different from $\square$.

More formally, let s_0 be a function from restricted regular expressions to natural numbers, defined as follows:

(a) (Basis) $s_0(\lambda) = s(\phi) = 0$, $s_0(a) = 1$ for $a \in A$.

(b) (Induction Step) $s_0(\alpha \cup \beta) = s_0(\alpha \cdot \beta) = s_0(\alpha) + s_0(\beta)$, $s_0(\alpha^*) = s_0(\alpha)$ for α, β restricted regular expressions.

Now we can state: For every restricted regular expression α there exists a nonreturning finite automaton $\underset{\sim}{A}_\alpha$ with $s(\alpha)+1$ states $\neq \square$. Clearly in a restricted regular expression α $(\neq \phi)$ we can always get rid of ϕ. If λ occurs as an operand of a concatenation, we can drop it. If λ occurs as an operand of a union and the automaton for the operand has m states, we obtain an m state automaton for the union by making the initial state accepting. Finally, the star can be reduced to the two previous problems, since $\lambda^* = \lambda$. Therefore the definition $s_0(\phi) = s_0(\lambda) = 0$ correctly reflects the situation.

We will denote this algorithm for restricted regular expressions by RGX_0' . Basically, it is Mirkin's method ([11]), and also in principle similar to the methods in [2], [8], [17]. In particular, all the algorithms (including the one in [1]) are based on structural induction, rely on closure properties, and start the induction with $(\underset{\sim}{A}_\phi,)$ $\underset{\sim}{A}_\lambda$, $\underset{\sim}{A}_a$, $a \in A$. This basis, however, is not the only possible one, in fact, it turns out to be somewhat wasteful.

Recall that in order to obtain the bound $s_0(\alpha)+1$ on the number of states $\neq \square$ of $\underset{\sim}{A}_\alpha$ all we used was the induction step - which works for all nonreturning finite automata - and the fact that for each element $b \in \{\phi, \lambda\} \cup A$ we have a nonreturning automaton $\underset{\sim}{A}_b$ with at most two states $\neq \square$. However, such automata can accept more than just $\{L(b) \mid b \in \{\phi, \lambda\} \cup A\}$. For one easily verifies that to any expression γ of the form $\gamma = F \cdot G^*$ or $\gamma = F \cdot G^* \cup \lambda$, $F,G \subseteq A$, there exists a nonreturning finite automaton $\underset{\sim}{A}_\gamma$ with at most two states $\neq \square$ such that $L(\gamma) = L(\underset{\sim}{A}_\gamma)$, and conversely, given such an automaton $\underset{\sim}{A}$ there exists an expression of the above form describing $L(\underset{\sim}{A})$.

Therefore we define s_1 as follows:

(a) (Basis) $s_1(\phi) = s_1(\lambda) = 0$, $s_1(\gamma) = 1$ for all $\gamma \in S_1 - \{\phi, \lambda\}$, $S_1 = \{F \cdot G^*, F \cdot G^* \cup \lambda \mid F,G \subseteq A\}$.

(b) (Induction) $s_1(\alpha \cup \beta) = s_1(\alpha \cdot \beta) = s_1(\alpha) + s_1(\beta)$, $s(\alpha^*) = s(\alpha)$ for all restricted regular expressions α , β .

Clearly $s_1(\alpha) \leq s_0(\alpha)$ for all α . For example, consider $A = \{0,1,2\}$ and $\alpha_1 = (0 \cup 1)2^*1(0 \cup 1 \cup 2)^*$. Clearly $s_0(\alpha_1) = 7$ while $s_1(\alpha_1) = 2$. In general, if k is the cardinality of A

$$\frac{1}{2k} s_0(\alpha) \leq s_1(\alpha) \leq s_0(\alpha) \quad \text{for all } \alpha.$$

So far we dealt with a basis consisting of automata with at most two states $\neq \square$. However, this can be carried over to automata with more than two states $\neq \square$. Obviously, this can only improve the bound on the number of states of the resulting nondeterministic finite automaton constructed from a given restricted regular expression α . Thus, let S_k be a certain subset of the set of all restricted regular expressions γ_k such that $L(\gamma_k)$ is accepted by a nonreturning finite automaton with $k+1$ states $\neq \square$, $k \geq 1$; furthermore let $S_0 = \{\phi,\{\lambda\}\}$.

Then define

(a) (Basis) $s_m(\gamma_k) = k$ for all $\gamma_k \in S_k - S_{k-1}$, $k = 1,2, \ldots ,m$.

(b) (Induction) $s_m(\alpha \cup \beta) = s_m(\alpha\beta) = s_m(\alpha) + s_m(\beta)$, $s_m(\alpha^*) = s_m(\alpha)$ for all restricted regular expressions α , β .

There are two basic difficulties with this approach.

The first problem is the definition of the set S_k . Clearly in general the set of different restricted regular expressions for the same regular language is infinite, as indicated by $\bigcup_{i=1}^{t} 0^i \cup 0^{t+1}0^*$, for all $t \geq 1$. Since no normal form is known for restricted regular expressions which can be defined by an equivalence relation induced by a finite set of equations, the problem to decide which expressions should be in S_k is not easy. Usually, this decision will be quite arbitrary. In the above case (that is for S_1) we insisted on a certain structural relation between automata and expressions. The second problem is the definition of s_m , for s_m need not be a function anymore. For instance, take $A = \{0,1,2\}$ and $\alpha = 01^*20^*01^*2$. Now, $\alpha = \alpha_1\alpha_2$, $\alpha_1 = 01^*20^*$, $\alpha_2 = 01^*2$, therefore $s_2(\alpha) = s_2(\alpha_1) + s_2(\alpha_2) = 2 + 2 = 4$ if we define S_2 appropriably. On the other hand, $\alpha = \beta_1\beta_2\beta_3$, $\beta_1 = 01^*2$, $\beta_2 = 0^*$, $\beta_3 = 01^*2$, and $s_2(\alpha) = s_2(\beta_1) + s_2(\beta_2) + s_2(\beta_3) = s_2(\beta_1) + s_1(\beta_2) + s_2(\beta_3) = 2 + 1 + 2 = 5$. However, this problem can be avoided by defining $s_m(\alpha)$ to be the minimum of all possible values. Note, that this depends crucially upon S_m .

Here, again, we are in the strange situation that by choosing larger and larger m and admitting more and more expressions to S_m we can get as close to the optimal result as we want (namely, always obtaining the reduced automaton) - but will never be able to reach this goal. It appears that this is an inherent property of all algorithms which translate expressions into automata.

At any rate, having chosen S_m sensibly - at least satisfying $S_i \supsetneq S_{i-1}$ for all $i \leq m$ - we obtain an algorithm which we will denote by RGX'_m . Obviously, for all $m \geq 0$ RGX'_{m+1} is "better" than RGX'_m in the sense that for some input, RGX'_{m+1} will construct a smaller automaton that RGX'_m .

We conclude by remarking that a nonreturning finite automaton $\underset{\sim}{A} = (A,Q,M,q_0,F)$ with $n+1$ states different from $\square$ has a nonreturning deterministic counterpart $\underset{\sim}{B} = (A,P,N,\{q_0\},G)$ such that $L(\underset{\sim}{A}) = L(\underset{\sim}{B})$, and $\underset{\sim}{B}$ has at most 2^n+1 states. Note, that this is completely independent of how $\underset{\sim}{A}$ was obtained. This remark follows immediately from the subset construction and the following observations: If $q_0 \in p \in P$ then $p = \{q_0\}$, for $\underset{\sim}{A}$ is nonreturning. Whenever $\square \in p$ and $p \neq \{\square\}$ we can remove $\square$ from p without affecting anything. Therefore we need to consider only the nonempty subsets of $Q - \{\square, q_0\}$. There are 2^n-1 of them; adding to this 1 for $\{\square\}$ and 1 for $\{q_0\}$ gives precisely the bound 2^n+1 . This shows that for any restricted regular expression α there exists a deterministic finite automaton with $2^{s_m(\alpha)}+1$ states, for all $m \geq 0$. This bound for $m = 0$ is also obtained in [9] by a slightly different method.

While finite automata are not closed under complementation, deterministic automata do have this property. This suggests the following general method:

(1) Given an unrestricted regular expression α , express all boolean operators in α in terms of union and complement; let this be α' .

(2) Isolate all maximal restricted regular subexpressions of α' and construct finite automata for them using RGX'_m .

(3) If β is a proper maximal restricted subexpression of α' then $\overline{\beta}$ is a subexpression of α' . If β is not proper then $\beta = \alpha'$; in this case goto (4). Make $\underset{\sim}{A}_\beta$ deterministic using the subset construction and complement the final states of the resulting automaton. Replace all occurrences of $\overline{\beta}$ in α' by a token β' which is considered to be a restricted regular expression associated with the automaton for $\overline{\beta}$. Call this modification of α' again α' and goto (2).

(4) Construct a deterministic automaton $\underset{\sim}{B}$ for the result and stop. $\underset{\sim}{B}$ accepts precisely $L(\alpha)$.

This method is implemented in REGPACK (as RGX) using as subalgorithm for restricted regular expressions RGX'_0 , and the basis $\{\phi, \lambda\} \cup A$.

For some remarks on the implementation see Section I; for a discussion of the performance of RGX see Section II.

Acknowledgement

I wish to express my gratitude to Professor J.A. Brzozowski whose interest and suggestions helped shape REGPACK and this report on it. Thanks are also due to Faith Fich and Denis Thérien who have critically read this report.

BIBLIOGRAPHY

[1] Aho, A.V., Hopcroft, J.E., Ullman, J.D. The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

[2] Aho, A.V., Ullman, J.D. The Theory of Parsing, Translation, and Compiling, Vol. I, Prentice Hall (1972).

[3] Brzozowski, J.A. Derivatives of regular expressions, JACM 11, 481-494 (1964).

[4] Dewar, R.B.K. Spitbol version 2.0, Document S4D23, Illinois Institute of Tech., Chicago, Ill. (1971).

[5] Ginzburg, A. Algebraic Theory of Automata, Academic Press (1968).

[6] Griswold, R.E., Poage, J.F., Polonsky, I.P. The Snobol4 Programming Language, Prentice-Hall (1971).

[7] Hopcroft, J.E. An n log n Algorithm for Minimizing States in a Finite Automaton, in Theory of Machines and Computations (Z. Kohavi and A. Paz, eds.) Academic Press, 189-196 (1972).

[8] Johnson, W., Porter, J., Ackley, S., Ross, D. Automatic Generation of Efficient Lexical Processors Using Finite State Techniques, CACM 11:12, 805-813 (1968).

[9] McNaughton, R., Yamada, H. Regular expressions and state graphs for automata, in Sequential Machines: Selected Papers, (E.F. Moore, ed.), 157-174, Addison-Wesley (1964).

[10] McNaughton, R., Papert, S. Counter-free Automata, MIT Press, Cambridge, Mass. (1971).

[11] Mirkin, B.G. An algorithm for constructing a base in a language of regular expressions (in Russian), Iz. Akad. Nauk SSSR, Techn. Kibernet. No. 5, 113-119 (1966), Engl. translation in Engineering cybernetics, No. 5, 110-116 (1966).

[12] Meyer, A.R., Stockmeyer, L.J. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space, 13th Annual IEEE Symp. on Switching and Automata Theory, 125-129 (1972).

[13] Perrot, J.F. Utilisation d'APL pour calculer des monoïdes finis, Institut de Programmation, Université de Paris VI (1972).

[14] Perrot, J.F., Cousineau, F.G., Rifflet, J.M. APL Programs for Direct Computation of a Finite Semigroup, Institut de Programmation, Université de Paris VI (1973), also in APL '73, Actes du Congrès, Copenhague, August 1973, North-Holland.

[15] Piatkowski, T.F. Computer Programs Dealing with Finite-State Machines I, II, SEL Technical Reports 11, 20, Department of Electrical Engineering, University of Michigan, Ann Arbor (1967).

[16] Redko, V.N. On defining relations for the algebra of regular events, Ukrain. Mat. Z. 16, 120-126 (1964).

[17] Stockmeyer, L.J., Meyer, A.R. Word Problems Requiring Exponential Time: Preliminary Report, 5th Annual ACM Symp. on Theory of Computing, 1-9 (1973).

[18] Thompson, K. Regular expression search algorithm, CACM 11:6, 419-422 (1968).

[19] Bierman, E. Realization of Star-Free Events, M.A.Sc. Thesis, Dept. of Elect. Eng., University of Waterloo, Waterloo, Ontario, 1971.

[20] Brzozowski, J.A. Canonical Regular Expressions and Minimal State Graphs for Definite Events, Mathematical Theory of Automata, New York, 1962, 529-561, Brooklyn, Polytechnic Institute of Brooklyn, 1963 (Symposia Series 12).

[21] Brzozowski, J.A. A Generalization of Finiteness, Semigroup Forum Vol. 13 (1977), 239-251.

[22] Brzozowski, J.A. Hierarchies of Aperiodic Languages, Revue Française d'Automatique, Informatique et Recherche Opérationelle, Vol. 10, n. 8 (1976), 33-49.

[23] Eilenberg, S. Automata, Languages, and Machines, Vol. B, New York, Academic Press, 1976.

[24] Simon, I., Hierarchies of Events with Dot-Depth One, Ph.D. Thesis, Department of Applied Analysis and Computer Science, University of Waterloo, Waterloo, Ontario, 1972.