

From Regular Expressions to Finite Automata*

J.-M. Champarnaud J.-L. Ponty D. Ziadi

L.I.F.A.R.

Université de Rouen, Faculté des Sciences et des Techniques,
76821 Mont-Saint-Aignan Cedex, France

E-mails: {champarnaud, ponty, ziadi}@dir.univ-rouen.fr

October 13, 1998

Abstract

There are three classical algorithms to compute a finite automaton from a regular expression. The Brzozowski algorithm yields a deterministic automaton, the Glushkov algorithm a nondeterministic one, and the general step by step method generally yields a NFA with ε -transitions. Berry and Sethi have adapted Brzozowski's algorithm to compute the Glushkov automaton of an expression. We describe a variant of the step by step construction which associates standard and trim automata to regular languages. We show that the automaton constructed by this variant and the Glushkov automaton (computed by Berry-Sethi algorithm) are isomorphic.

Introduction

The aim of this paper is to show that a variant of step by step construction [18, 33] to compute an automaton from a regular expression leads to the same result as the Glushkov algorithm [19, 20] up to a renaming of states. The construction implemented in the Automate package [15] to compute an

¹This work is a contribution to the Automate software development project carried on by A.I.A. Working Group (Algorithmics and Implementation of Automata) of LIFAR. Contacts: {jmc, ziadi}@dir.univ-rouen.fr.

automaton from an unrestricted expression reduces to this variant of step by step method when expressions are simple ones.

Section 1 recalls basic definitions in automata theory. Section 2 describes several (old and new) papers dealing with the problem of converting a regular expression into an automaton. Section 3 presents automata constructions we use in our variant of step by step method. Section 4 describes the Berry–Sethi algorithm which computes the Glushkov automaton of an expression. In Section 5, we prove that our construction and the Berry–Sethi one yield identical automata up to a renaming of states. Section 6 deals with implementation and complexity features.

1 Definitions

Let A be a finite alphabet. An *automaton* over A is a 4-tuple $\mathcal{A} = (Q, I, T, E)$ where Q is a set of *states*, I is a subset of Q whose elements are the *initial states*, T is a subset of Q whose elements are the *final states* (or *terminal states*), E is a subset of the cartesian product $Q \times A \times Q$ whose elements are the *edges*.

Let $\mathcal{A} = (Q, I, T, E)$ be an automaton. An edge (q, a, q') goes from a head q to a tail q' . A *path* of \mathcal{A} is a sequence (q_i, a_i, q_{i+1}) , $i = 1, \dots, n$, of consecutive edges. Its *label* is the word $w = a_1 a_2 \dots a_n$. A word $w = a_1 a_2 \dots a_n$ is *recognized* by the automaton \mathcal{A} if there is a path with label w such that $q_1 \in I$ and $q_{n+1} \in T$. The language *recognized* by the automaton \mathcal{A} is the set of words which it recognizes. The automaton \mathcal{A} is *trim* if for all $q \in Q$ there is at least one path through q beginning at an initial state and ending at a final state. The automaton \mathcal{A} is *deterministic* if there is only one initial state and if for all $(q, a) \in Q \times A$ there is at most one state q' such that $(q, a, q') \in E$. \mathcal{A} is said to be a *DFA* if it is deterministic and a *NFA* otherwise. An automaton is *standard*, or, following Leiss [22], *non-returning* if there is only one initial state and if there is no edge having the initial state as tail. We shall write $\mathcal{A} = (Q, i, T, E)$ for such an automaton with a unique initial state i .

Let A be an alphabet. An *atomic expression* (on A) is either a letter $a \in A$, or the empty word denoted by 1. A (*simple*) *regular expression* is obtained by applying recursively the following operators to the atomic expressions: if r and r' are two regular expressions, one defines their *union* $r \cup r'$, their *product* rr' , the *star* of r written r^* .

To each regular expression r there corresponds a language $L(r)$, that is

to say a subset of A^* recursively defined as follows: the single element subset $L(a) = \{a\}$ is associated to a , for every letter $a \in A$; if r and r' are two regular expressions, the associated languages of which are already defined, then the language $L(r \cup r') = L(r) \cup L(r')$, the union of the languages $L(r)$ and $L(r')$, is associated to the union $r \cup r'$, the product language of $L(r)$ and $L(r')$, i.e. $L(r)L(r') = \{uv \mid u \in L(r), v \in L(r')\}$, is associated to the product rr' , and the Kleene closure of $L(r)$, that is the language $(L(r))^* = \cup_{n \geq 0} [L(r)]^n$, is associated to the star r^* . A language $L \subset A^*$ is *regular* if and only if there is a regular expression r such that $L = L(r)$.

The *unrestricted regular expressions* are obtained by adding new operators such as:

- *plus operator*: to the expression r^+ one associates the language $L(r^+) = \bigcup_{n > 0} [L(r)]^n$.
- *intersection operator*: to the intersection $r \cap r'$ of two expressions one associates the language $L(r \cap r') = L(r) \cap L(r')$, the intersection of the languages $L(r)$ and $L(r')$.
- *difference operator*: to the difference $r \setminus r'$ of two expressions one associates the language $L(r \setminus r') = L(r) \setminus L(r')$, the set difference of the languages $L(r)$ and $L(r')$.
- *left quotient operator*: to the left quotient $r^{-1}r'$ of the expression r' by the expression r one associates the language $L(r^{-1}r') = \{v \in A^* \mid \exists u \in L(r) \text{ such that } uv \in L(r')\}$.
- *right quotient operator*: to the right quotient $r'r^{-1}$ of the expression r' by the expression r one associates the language $L(r'r^{-1}) = \{v \in A^* \mid \exists u \in L(r) \text{ such that } vu \in L(r')\}$.
- *shuffle product operator*, written W : to the shuffle product rWr' of two expressions one associates the language $L(r)WL(r')$, the shuffle of the languages $L(r)$ and $L(r')$ i.e. the set defined by $L(r)WL(r') = \{u_1v_1u_2v_2 \dots u_nv_n \mid u_1u_2 \dots u_n \in L(r) \text{ and } v_1v_2 \dots v_n \in L(r')\}$, where u_i and v_i are arbitrary words.

The number of symbol occurrences of a regular expression will be called the alphabetic width [2] of the expression.

2 Algorithms to convert a regular expression into an automaton

During the last forty years, automata synthesis, which is nowadays rather called conversion of regular expressions into automata, has aroused a lot of interesting research works. Watson recently published a very comprehensive taxonomy on this topic [32].

Before reviewing the various algorithms, let us remark that their complexity classically depends either on the size of the expression, which is the length of the string representing the expression, or on the alphabetic width of the expression, which is the number of symbols occurring in the expression. These two parameters are linearly dependent as far as sequences of star operators and occurrences of the empty word are carefully handled.

Conversion algorithms can be classified into three categories: language equations systems, position sets and step by step constructions.

2.1 Language equations systems

Let L be a regular language over the alphabet A , and let $\mathcal{A} = (Q, \{0\}, T, E)$ be the finite automaton recognizing the language L . Let us assume that $A = \{a_1, \dots, a_m\}$ and that $|Q| = n + 1$.

For $i \in Q$, the language recognized by the automaton $(Q, \{i\}, T, E)$ is denoted $L(i)$. We define $L(i)$ inductively as follows:

$$L(0) = L. \tag{1}$$

$$L_{ij} = \bigcup_{(i, a_j, k) \in E} L(k).$$

$$L(i) = \bigcup_{j=1}^m a_j L_{ij} \cup \text{Null}(L(i)), i \in [0, n]. \tag{2}$$

If \mathcal{A} is a deterministic automaton, we replace the equation (2.1) by the following one:

$$L_{ij} = L(k), \text{ with } (i, a_j, k) \in E. \tag{3}$$

Brzozowski's *word derivatives* method [10, 11] works on such a deterministic system, in which each language $L(k)$ is replaced by an equivalent expression. It yields a deterministic automaton and its time complexity may be exponential. Similar results are stated by Spivak [29, 30], who introduces the notion of *base in the languages of regular expressions*.

Mirkin [25, ?] generalizes this notion and makes use of a *prebase* to compute a nondeterministic automaton with a nice property: the number of states is at most the alphabetic width of the expression.

More recently, Antimirov [2] presents the concept of *partial derivative* as a generalization of word derivative, gives a constructive definition of partial derivatives, and reports a prototype implementation of NFA construction in $O(n^4)$ time.

Let us mention that Champarnaud and Ziadi [36] have proved that Mirkin and Antimirov algorithms yield identical NFA automata. They also provide a more efficient implementation of this algorithm.

2.2 Position sets

In this category, algorithms first linearize the expression: each symbol is replaced by its position in the expression, and is associated with a state in the nondeterministic result. This construction was developed by Glushkov [19, 20] and McNaughton and Yamada [24]. We shall call a Glushkov automaton the result of this construction. The number of states of the Glushkov automaton of an expression is one more than the alphabetic width of the expression.

A naïve implementation of this construction, as in AMoRE [23] and AG [12] yields $O(n^3)$ time complexity. Brüggeman-Klein [8, 9] provides an $O(n^2)$ time algorithm based on the notion of *star normal form*. Chang and Paige [17] report a similar complexity result with a *lazy evaluation* of the edges of the automaton. Champarnaud *et al.* [34, 35, 26, 27] compute a linear space and time representation of the automaton, called the *ZPC-structure*, and deduce the transition table in quadratic time. The design of the ZPC-structure and some algorithmic improvements based on the use of this representation have been studied by Champarnaud *et al.* [28, 27, 16].

Let us mention that Glushkov automata have interesting properties. Berry and Sethi [4] have studied their relation to word derivatives. Caron and Ziadi [13] have characterized the Glushkov automata in terms of graphs and of strongly connected components.

2.3 Step by step construction

The algorithms of this category first compute the syntax tree of the expression and then realize the operations on the expressions by the way of operations on the automata. Many variants of such algorithms have been described. For

example, the classical Thompson method [31] builds a ε -NFA in linear space and time on the size of the syntax tree, whereas Leiss method [22] computes a nondeterministic automaton without ε -transition.

The algorithm implemented by Champarnaud [15] in Automate turns out to be similar to the Leiss method, as far as simple expressions are concerned. This paper was motivated by the 1985 implementation of Automate, since automata displayed by Automate software looked like Glushkov automata.

3 A variant of the step by step construction

The step by step method consists in representing languages by automata and realizing operations defined on languages with the help of constructions on these automata [3, 18]. There exist many variants of this general method, according to the properties of the automata they build. For instance, the variant described by Thompson [31] makes use of ε -transitions, whereas the one presented by Berstel [5] works on normalized automata. In the variant that we consider automata associated with languages are characterized by the property of being standard and trim. We first formulate the algorithms of the regular operations (union, concatenation and Kleene closure) adjusted to such automata, and then present the general algorithm for converting a regular expression into an automaton.

Let $\mathcal{C}_1 = (Q_1, i_1, T_1, E_1)$ and $\mathcal{C}_2 = (Q_2, i_2, T_2, E_2)$ be standard and trim automata. We shall assume that sets Q_1 and Q_2 are disjoint; this can be always achieved by renaming the states in one of them.

3.1 Algorithm for union operation

The union $\mathcal{C}_1 \cup \mathcal{C}_2$ of two automata \mathcal{C}_1 and \mathcal{C}_2 is the automaton $\mathcal{C} = (Q, i, T, E)$ defined by the following algorithm:

If $i_1 \in T_1$, then i_1 is the initial state of the union and i_2 is removed. More precisely, we have:

$$\begin{aligned} Q &= Q_1 \cup (Q_2 \setminus \{i_2\}), \\ i &= i_1, \\ T &= \begin{cases} T_1 \cup (T_2 \setminus \{i_2\}) & \text{if } i_2 \in T_2, \\ T_1 \cup T_2 & \text{otherwise,} \end{cases} \\ E &= E_1 \cup \{(q, a, q') \in E_2 \mid q \neq i_2\} \cup \{(i, a, q') \mid (i_2, a, q') \in E_2\}. \end{aligned}$$

Otherwise i_2 is the initial state of the union and i_1 is removed. More precisely, we have:

$$\begin{aligned} Q &= (Q_1 \setminus \{i_1\}) \cup Q_2, \\ i &= i_2, \\ T &= T_1 \cup T_2, \\ E &= \{(q, a, q') \in E_1 \mid q \neq i_1\} \cup \{(i, a, q') \mid (i_1, a, q') \in E_1\} \cup E_2. \end{aligned}$$

The following properties hold:

1. If the automata \mathcal{C}_1 and \mathcal{C}_2 recognize the languages L_1 and L_2 , the automaton $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$ recognizes the language $L_1 \cup L_2$.
2. If the automata \mathcal{C}_1 and \mathcal{C}_2 are standard, then \mathcal{C} is standard. The initial state of one of the automata \mathcal{C}_1 or \mathcal{C}_2 must be removed to make \mathcal{C} trim.
3. If i_1 is a final state in \mathcal{C}_1 , it is chosen as the initial state of \mathcal{C} . Otherwise i_2 is chosen. Therefore \mathcal{C} recognizes the empty word if at least one of the two automata \mathcal{C}_1 or \mathcal{C}_2 recognizes it.

3.2 Algorithm for concatenation operation

The concatenation $\mathcal{C}_1 \cdot \mathcal{C}_2$ of two automata \mathcal{C}_1 and \mathcal{C}_2 is the automaton $\mathcal{C} = (Q, i, T, E)$ defined by the following algorithm:

$$\begin{aligned} Q &= Q_1 \cup (Q_2 \setminus \{i_2\}), \\ T &= \begin{cases} T_1 \cup (T_2 \setminus \{i_2\}) & \text{if } i_2 \in T_2, \\ T_2 & \text{otherwise,} \end{cases} \\ E &= E_1 \cup \{(q, a, q') \in E_2 \mid q \neq i_2\} \cup \{(q, a, q') \mid q \in T_1 \text{ and } (i_2, a, q') \in E_2\}. \end{aligned}$$

The following properties hold:

1. If the automata \mathcal{C}_1 and \mathcal{C}_2 recognize the languages L_1 and L_2 , the automaton $\mathcal{C} = \mathcal{C}_1 \cdot \mathcal{C}_2$ recognizes the language $L_1 \cdot L_2$.
2. If the automata \mathcal{C}_1 and \mathcal{C}_2 are standard, then \mathcal{C} is standard. The initial state of the automaton \mathcal{C}_2 must be removed to make \mathcal{C} trim.

3.3 Algorithm for the Kleene closure operation

Let $\mathcal{C} = (Q, i, T, E)$ be a standard and trim automaton. The Kleene closure \mathcal{C}^* of the automaton \mathcal{C} is the automaton $\mathcal{C}^* = (Q', i', T', E')$ defined by the following algorithm:

$$\begin{aligned} Q' &= Q, \\ i' &= i, \\ T' &= \begin{cases} T & \text{if } i \in T, \\ T \cup \{i'\} & \text{otherwise,} \end{cases} \\ E' &= E \cup \{(q, a, q') \mid q \in T \text{ and } (i, a, q') \in E\}. \end{aligned}$$

The following properties hold:

1. If \mathcal{C} recognizes the languages L , the automaton \mathcal{C}^* recognizes the language L^* .
2. If \mathcal{C} is standard and trim, then \mathcal{C}^* is standard and trim.

3.4 Converting a regular expression into an automaton

Let \mathcal{C}_e be the automaton constructed from the expression e . \mathcal{C}_e is recursively defined by the following formulae:

$$\begin{aligned} \mathcal{C}_1 &= (\{1\}, 1, \{1\}, \emptyset), \\ \mathcal{C}_a &= (\{1, 2\}, 1, \{2\}, \{(1, a, 2)\}) \text{ for } a \in A, \\ \mathcal{C}_{e \cup e'} &= \mathcal{C}_e \cup \mathcal{C}_{e'}, \\ \mathcal{C}_{e \cdot e'} &= \mathcal{C}_e \cdot \mathcal{C}_{e'}, \\ \mathcal{C}_{e^*} &= \mathcal{C}_e^*. \end{aligned}$$

The basic automata \mathcal{C}_1 and \mathcal{C}_a for $a \in A$ are standard and trim. Algorithms for union, concatenation and Kleene closure that we have described preserve these properties.

Example 1: $e = (ab + b)^*ba$.

Figure 1 illustrates the recursive computation of the NFA $\mathcal{C}_{(ab+b)^*ba}$ constructed by our variant of step by step method. Notice that a word (such as ba) is a lexical unit for Automate lexical analyser and is (directly) evaluated by its deterministic minimal automaton. This implementational improvement does not affect our results.

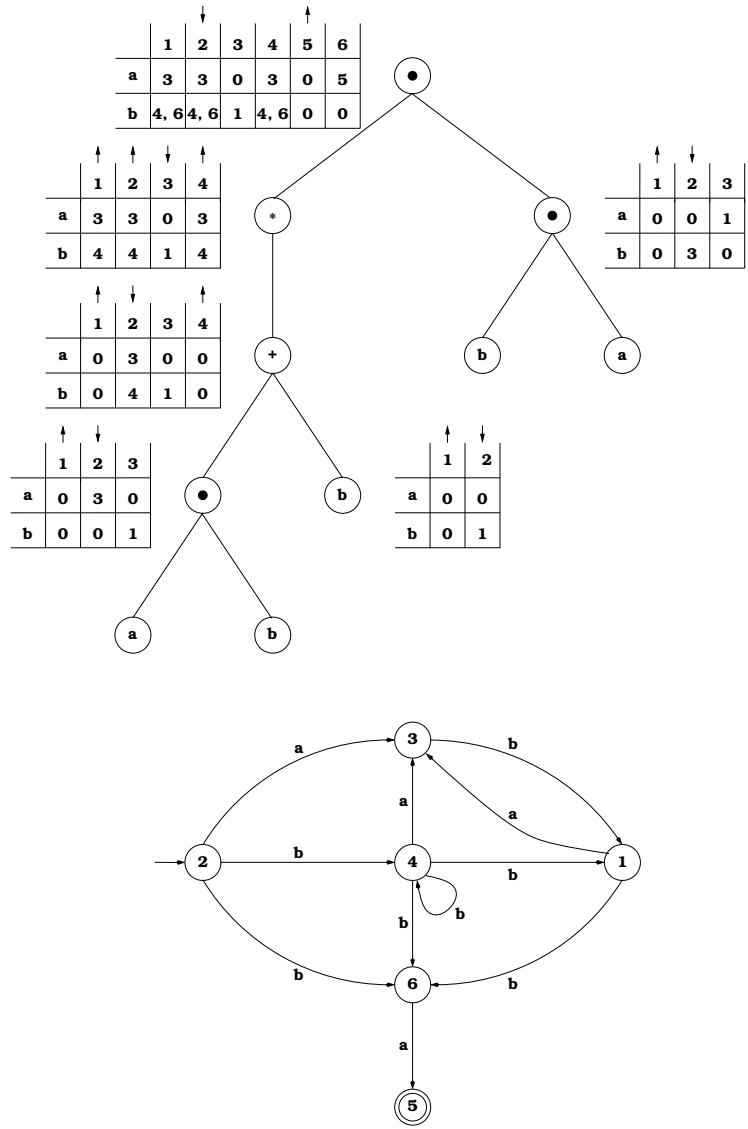


Figure 1: Step by step variant on $e = (ab + b)*ba$.

4 The Berry–Sethi algorithm

A regular expression e is *linear* if all letters in e are distinct. For instance, the expression $(a_1 \cup a_2)^* a_3 a_4 a_5$ is linear. Any regular expression may be *linearized* by substituting a distinct symbol to each of its letters. For instance, the last expression is a linearization of the expression $(a \cup b)^* abb$.

The Berry–Sethi algorithm for converting a simple expression e over the alphabet A into a nondeterministic automaton can be stated as follows:

1. Consider a linearized version e' of e over the alphabet A' .
2. Construct a deterministic automaton \mathcal{G}' recognizing $L(e')$.
3. Replace each letter of A' labelling an edge of \mathcal{G}' by the corresponding letter of A in order to produce a nondeterministic automaton \mathcal{G} which recognizes $L(e)$.

The algorithm of derivatives, designed by Brzozowski [11], computes a deterministic (and not necessarily minimal) automaton from a regular expression. Berry and Sethi have adapted this algorithm to the case of linear expressions. More generally, their algorithm works when the language defined by the expression is *local*. We shall follow the presentation of Berstel and Pin [6] for this algorithm.

A language $L \subset A^*$ is *local* if there exists two subsets P and S of A and a subset N of A^2 such that

$$L \setminus \{1\} = (PA^* \cap A^*S) \setminus A^*NA^*$$

The sets P , S and N have the following meaning: $L \setminus \{1\}$ is the set of words of A^* whose first letter is in P , whose last letter is in S , and whose factors of length two do not belong to N .

Let L be a local language. The sets P , S and N associated with L are defined by:

$$\begin{aligned} P(L) &= \{a \in A \mid aA^* \cap L \neq \emptyset\}, \\ S(L) &= \{a \in A \mid A^*a \cap L \neq \emptyset\}, \\ N(L) &= \{x \in A^2 \mid A^* \times A^* \cap L = \emptyset\}. \end{aligned}$$

For instance, the language $((ac)^* \cup (bc)^*)^*$ over the alphabet $A = \{a, b, c\}$ is local, since we have:

$$((ac)^* \cup (bc)^*)^* = \{1\} \cup [(\{a, b\}A^* \cap A^*c) \setminus A^*\{aa, ab, ba, bb, cc\}A^*]$$

We shall denote by $F(L)$ the complement of $N(L)$ with respect to A^2 .

A deterministic automaton $\mathcal{G} = (Q, i, T, E)$ is *local* if and only if, for every $a \in A$, edges labeled with a have the same tail.

Proposition 1 (Berstel-Pin [6]) The following conditions are equivalent:

1. L is local.
2. L is recognized by a local automaton.

Proof. (1 \Rightarrow 2) Let L be a local language, defined by the sets P, S and F . Let us denote by $\mathcal{G}(P, S, F)$ the automaton (Q, i, T, E) computed from the sets P, S and F as follows:

$$\begin{aligned} Q &= A \cup \{1\}, \\ i &= 1, \\ T &= \begin{cases} S \cup \{1\} & \text{if the empty word belongs to } L, \\ S & \text{otherwise,} \end{cases} \\ E &= \{(1, a, a) \mid a \in P\} \cup \{(a, b, b) \mid ab \in F\}. \end{aligned}$$

It can easily be checked that $\mathcal{G}(P, S, F)$ is local and recognizes L .

(2 \Rightarrow 1) Let $\mathcal{G} = (Q, i, T, E)$ be a local automaton over the alphabet A . Without loss of generality we can suppose \mathcal{G} is trim. Every non initial state of \mathcal{G} is thus the tail of at least one edge. As \mathcal{G} is local, there is a bijection from the set of edge tails onto the alphabet A . Therefore \mathcal{G} has $1 + |A|$ states which may be renamed in the following way: the initial state is renamed 1, and a non initial state is renamed with the letter which labels all the edges entering this state. Thus, every edge of \mathcal{G} can be written either $(1, a, a)$ or (a, b, b) , with $a, b \in A$.

One easily checks that \mathcal{G} recognizes the local language defined by the sets P, S and F computed as follows:

$$\begin{aligned} P &= \{a \mid (1, a, a) \in E\}, \\ S &= T, \\ F &= \{ab \mid (q, a, a) \in E, q \in Q \text{ and } (a, b, b) \in E\}. \square \end{aligned}$$

Proposition 2 Let L be a local language over the alphabet A and $\mathcal{G}(P, S, F)$ the local automaton associated with it. Let $\mathcal{A}' = (Q', i', T', E')$ be a trim and local automaton, on the same alphabet. If \mathcal{A}' recognizes the language L , then

\mathcal{A}' and $\mathcal{G}(P, S, F)$ are identical up to a renaming of states.

Proof. As mentioned in Proposition 1, we use the set $\{1\} \cup A$ to rename the states of \mathcal{A}' . As \mathcal{A}' recognizes L , we have:

$$\begin{aligned} P &= \{a \mid (1, a, a) \in E'\}, \\ S &= T', \\ F &= \{ab \mid (q, a, a) \in E', q \in Q' \text{ and } (a, b, b) \in E'\}. \end{aligned}$$

We observe that E' is computed in the same way as E from the sets P and F . Moreover, $T = T' = S$. Thus, the automata \mathcal{G} and \mathcal{A}' are identical, up to a renaming of states. \square

If the local language L is defined by a regular expression e , the empty word test $null(e)$ and the sets P , S and F can be computed by means of the following recursive procedures:

$$\begin{aligned} null(1) &= \mathbf{true}, \\ null(a) &= \mathbf{false}, \text{ for } a \in A, \\ null(e \cup e') &= null(e) \mathbf{or} null(e'), \\ null(e \cdot e') &= null(e) \mathbf{and} null(e'), \\ null(e^*) &= \mathbf{true}. \end{aligned}$$

$$\begin{aligned} P(1) &= \emptyset, \\ P(a) &= \{a\}, \text{ for } a \in A, \\ P(e \cup e') &= P(e) \cup P(e'), \\ P(e \cdot e') &= \begin{cases} P(e) \cup P(e') & \text{if } null(e), \\ P(e) & \text{otherwise,} \end{cases} \\ P(e^*) &= P(e). \end{aligned}$$

$$\begin{aligned} S(1) &= \emptyset, \\ S(a) &= \{a\}, \text{ for } a \in A, \\ S(e \cup e') &= S(e) \cup S(e'), \\ S(e \cdot e') &= \begin{cases} S(e) \cup S(e') & \text{if } null(e'), \\ S(e') & \text{otherwise,} \end{cases} \\ S(e^*) &= S(e). \end{aligned}$$

$$\begin{aligned}
F(1) &= \emptyset, \\
F(a) &= \emptyset, \text{ for } a \in A, \\
F(e \cup e') &= F(e) \cup F(e'), \\
F(e \cdot e') &= F(e) \cup F(e') \cup S(e)P(e'), \\
F(e^*) &= F(e) \cup S(e)P(e).
\end{aligned}$$

Proposition 3 (Berstel-Pin [6]) The language defined by a linear expression is local.

This result is used in Berry–Sethi [4] algorithm which makes calls to procedures similar to the ones we have described to compute the local automaton of the language defined by a linearized expression and deduce the Glushkov automaton of the initial expression.

Example 2: $e = (ab + b)^*ba$.

Let $e' = (a_1b_2 + b_3)^*b_4a_5$ be the linearized expression of e . We have:

$$\begin{aligned}
P(e') &= \{a_1, b_3, b_4\}, \\
S(e') &= \{a_5\}, \\
F(e') &= \{a_1b_2, b_2a_1, b_2b_3, b_2b_4, b_3a_1, b_2b_3, b_3b_4, b_4a_5\}, \\
null(e') &= \text{false}.
\end{aligned}$$

We deduce the Glushkov automaton $\mathcal{G}_{(ab+b)^*ba}$ given in Fig. 2.

5 Step by step construction and the Berry–Sethi algorithm

Proposition 3. The automata constructed by our variant of step by step construction and by the Berry–Sethi algorithm are identical, up to a renaming of states.

Proof. Let e be a regular expression over an alphabet A , and e' be the linearized version of e , over the alphabet $A' = \{a_1, a_2, \dots, a_n\}$. Let \mathcal{C} and \mathcal{C}' be the automata constructed by our variant of the step by step construction for the expressions e and e' , respectively.

We first observe that there is a bijection from the set of states of \mathcal{C}' onto the set $\{1\} \cup A'$. This is a consequence of the properties of the definition of the operations for union, concatenation and Kleene closure on standard and trim automata. These operations do not introduce new states, and may only

remove states which are initial in the original automata. Thus, the set of non initial states of the result of our construction is exactly the set of the final states of the automata representing the letters of the expression. Therefore, these states can be renamed by the corresponding letters, the initial state being renamed 1. Subsequently, the states of \mathcal{C}' are assumed to be renamed with the set $\{1, a_1, a_2, \dots, a_n\}$.

We now show that the automaton \mathcal{C}' is local. For every letter a_i of e' , an edge $(1, a_i, a_i)$ is computed when the automaton representing a_i is built. Moreover, the algorithms of union, concatenation and Kleene closure operations, compute new edges whose tail and label must be the same as the tail and label of some edge going from the initial state of one of the original automata. Consequently, all the edges labeled a_i have state a_i for tail. Thus, the automaton \mathcal{C}' is local.

By construction \mathcal{C}' is also trim, and it recognizes the language defined by the linear expression e' . Following Proposition 2, \mathcal{C}' is identical, up to a renaming of states, to the local automaton of e' .

By construction, the automaton \mathcal{C} is derived from \mathcal{C}' by replacing the labels a_1, a_2, \dots, a_n by the corresponding letters of A . Thus, \mathcal{C} is identical, up to a renaming of states, to the automaton given by the Berry–Sethi algorithm for the expression e . \square

Example 3: $e = (ab + b)^*ba$.

The automata computed by our variant of the step by step method and by the Berry–Sethi algorithm are identical up to the renaming of states: $1 \rightarrow 2$, $2 \rightarrow 0$, $3 \rightarrow 1$, $4 \rightarrow 3$, $5 \rightarrow 5$ and $6 \rightarrow 4$.

6 Complexity and implementation

The step by step construction and the Glushkov algorithm are both implemented by first computing a parse tree for the expression. The step by step construction builds an automaton for every node of the tree. In every automaton, states are usually labeled $\{1, 2, \dots, n\}$. Whenever a union or concatenation operation is performed on two automata, the states (and thus the edges) of one of the arguments need to be renamed. Renaming an automaton is an $O(v)$ operation, where v is the number of edges. The Glushkov algorithm eliminates state renaming: every state of the result is labeled once, by its corresponding letter. If n is the number of positions in the expression, the

resulting automaton has at most as many edges as a deterministic automaton with $n + 1$ states on an alphabet of n letters, hence a complexity of $O(n^3)$ in a naïve implementation and a (worst-case optimal) complexity of $O(n^2)$ in Brüggemann-Klein, Chang and Paige or Champarnaud *et al.* algorithms.

Our variant of the step by step construction works on the transition tables of the intermediary automata, so it cannot be implemented in $O(n^2)$ time. However, it can be implemented more efficiently by labeling the final states of the automata for every symbol occurring in the expression with the corresponding letter in the linearized expression. The reason why it is still implemented with state renaming in the Automate package [14] is that such a classical implementation handles unrestricted expressions, unlike the Glushkov algorithm. Let us notice that the classical implementation makes it possible to deal with automaton variables occurring in expressions, which are evaluated from transitions tables. It is not so easy with Glushkov algorithm, since an arbitrary automaton is not necessarily a Glushkov one, and computing an equivalent expression by classical algorithms usually yields very large expressions.

Conclusion

We have used the Berry–Sethi formulation of the computation of the Glushkov automaton of a regular expression to show that this automaton can also be produced by a variant of the step by step construction, such that standard and trim automata are associated to regular languages involved by the successive steps of the construction. A nice feature of Glushkov automata is the small number of their states. On the other hand the main advantage of our variant is that it handles unrestricted expressions.

Acknowledgements

The first author would like to warmly thank the referee who made many constructive criticisms and remarks on a first version of this paper some years ago. The authors would like to thank D. Wood for helpful discussions and thoughtful comments.

References

- [1] A. V. Aho, R. Sethi and J. D. Ullman, Compilers, *Addison-Wesley*, 1986, 113–158.
- [2] V. Antimirov, Partial Derivatives of Regular Expressions and Finite Automaton construction, *Theoret. Comput. Sci.* **155** (1996) 291–319.
- [3] J.-M. Autebert, Langages algébriques, *Masson*, Collection Etudes et Recherches en Informatique, Paris, 1987.
- [4] G. Berry and R. Sethi, From Regular Expressions to Deterministic Automata, *Theoret. Comput. Sci.* **48** (1986) 117–126.
- [5] J. Berstel, Finite Automata and Rational Languages, an introduction, in: J.-E. Pin, ed., *Formal Properties of Finite Automata and Applications*, Lecture Notes in Computer Science, Vol. 386 (1987) 2–14.
- [6] J. Berstel and J.-E. Pin, Local Languages and the Berry–Sethi Algorithm, *Theoret. Comput. Sci.* **155** (1996), 439–446.
- [7] R. Book, S. Even, S. Greibach and G. Ott, Ambiguity in Graphs and Expressions, *IEEE Trans. on Computers* Vol. C-20 No. 2 (1971) 149–153.
- [8] A. Brüggemann-Klein, Regular Expressions into Finite Automata, *Theoret. Comput. Sci.* **120** (1993), 197–213.
- [9] A. Brüggemann-Klein and D. Wood, Deterministic Regular Languages, in: A. Finkel and M. Jantzen, eds., Proc. *STACS'92*, Lecture Notes in Computer Science, Vol. 577 (1992) 173–184.
- [10] J. A. Brzozowski and E. J. McCluskey, Jr., Signal Flow Graph Techniques for Sequential Circuit State Diagrams, *IEEE Trans. on Electronic Computers* Vol. EC-12 No. 2 (1963).
- [11] J. A. Brzozowski, Derivatives of Regular Expressions, *J. ACM* **11**(4) (1964) 481–494.
- [12] P. Caron, AG: A Set of Maple Packages for manipulating Automata and Semigroups, *Software–Practice & Experience* 27(8) (1997) 863–884.
- [13] P. Caron and D. Ziadi, Characterization of Glushkov Automata, to appear in *Theoret. Comput. Sci.*.

- [14] J.-M. Champarnaud, Automate: Un Système de manipulation des Automates Finis, *Research report LITP 85-48*, 1985.
- [15] J.-M. Champarnaud and G. Hansel, Automate: a Computing Package for Automata and Finite Semigroups, *J. of Symb. Comp.* **12** (1991) 197–220.
- [16] J.-M. Champarnaud, D. Ziadi and J.-L. Ponty, Determinization of Glushkov automata, Workshop on Implementing Automata, *WIA'98*, Rouen, 1988, to appear in *Lecture Notes in Computer Science*.
- [17] C.-H. Chang and R. Paige, From Regular Expressions to DFAs using NFAs, in: A. Apostolico, M. Crochemore, Z. Galil and U. Manber, eds., *Proc. CPM'91*, *Lecture Notes in Computer Science*, Vol. 664 (1992) 90–110.
- [18] S. Eilenberg, Automata, Languages, and Machines, vol. A, *Academic Press*, New York, 1974.
- [19] V. M. Glushkov, On a Synthesis Algorithm for Abstract Automata, (in russian), *Ukr. Matem. Zurnal* Vol. 12 No 2 (1960) 147–156.
- [20] V. M. Glushkov, The Abstract Theory of Automata, *Russian Mathematical Surveys* 16 (1961) 1–53.
- [21] J. E. Hopcroft and J. D. Ullman, Introduction to Automata Theory, Languages and Computation, *Addison-Wesley*, 1979.
- [22] E. Leiss, The Complexity of Restricted Regular Expressions and the Synthesis Problem of Finite Automata, *J. Computer and System Sciences* Vol. **23** No. 3 (1981) 348–354.
- [23] O. Matz, A. Miller, A. Potthoff, W. Thomas and E. Valkena, Report on the Program AMoRE, *Research report*, Institut für informatik und praktische mathematik, Christian-Albrechts Universität, Kiel, 1995.
- [24] R. McNaughton and H. Yamada, Regular Expressions and State Graphs for Automata, *IRE Trans. on Electronic Computers* Vol. EC-9 (1960) 39–47.
- [25] B. G. Mirkin, An Algorithm for constructing a Base in a Language of Regular Expressions, (in russian), *Izv. Akad. Nauk SSSR, Techn. Kibernet.* **5** (1966) 113–119. English translation in *Engineering Cybernetics* **5** (1966) 110–116.

- [26] J.-L. Ponty, D. Ziadi and J.-M. Champarnaud, A new Quadratic Algorithm to convert a Regular Expression into an Automaton, in: D. Raymond and D. Wood, eds., Proc. *WIA'96*, Lecture Notes in Computer Science, Vol. 1260 (1997) 109–119.
- [27] J.-L. Ponty. *Algorithmique et Implémentation des Automates, Contribution au Développement du Logiciel AUTOMATE*. PhD thesis, LIR, Université de Rouen, France, 1997. Rapport LIR TH97.03.
- [28] J.-L. Ponty, An Efficient ϵ -free Procedure for Deciding Regular Language Membership, in: D. Wood and S. Yu, eds., Proc. *WIA'97*, Lecture Notes in Computer Science, Vol. 1436 (1988) 159–170.
- [29] M. A. Spivak, A new Algorithm for the Abstract Synthesis of Automata, (in russian), *Materiali Nauchnich Seminarov po Teoreticheskim I Prikladnim Voprosam Kiberneteki, Teoria Avtomatov, Kiev* 3 (1963).
- [30] M. A. Spivak, An Algorithm for the Abstract Synthesis of Automata for an Extended Language of Regular Expressions, (in russian), *Izv. Akad. Nauk SSSR, Techn. Kibernet.* 1 (1965) 51–57. English translation in *Engineering Cybernetics*.
- [31] K. Thompson, Regular Expression Search Algorithm, *Comm. Assoc. Comput. Mach.* 11 (1968) 419–422.
- [32] B. Watson, Taxonomies and Toolkits of Regular Languages Algorithms, PhD thesis, Eindhoven University of Technology, The Netherlands, 1995.
- [33] D. Wood, Theory of computation, *Wiley*, New York, 1987.
- [34] D. Ziadi, J.-L. Ponty and J.-M. Champarnaud, Passage d'une expression rationnelle à un automate fini non-déterministe, *Bull. Belg. Math. Soc.*, 4:177-203, 1997.
- [35] D. Ziadi. *Algorithmique parallèle et séquentielle des automates*. PhD thesis, LIR, Université de Rouen, France, 1996. Rapport LIR TH96.01.
- [36] D. Ziadi and J.-M. Champarnaud, From Regular Expressions to Small NFAs, *Unpublished manuscript*, May 1998.

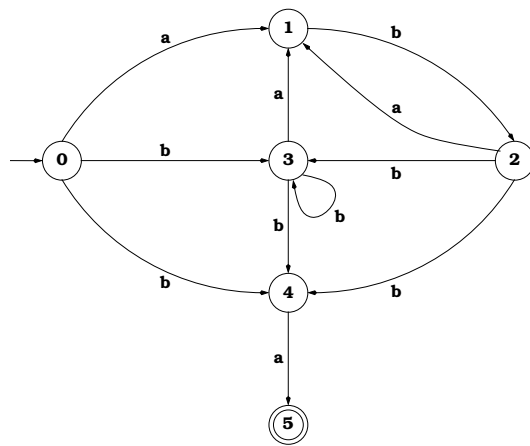


Figure 2: Glushkov automaton of $e = (ab + b)^*ba$.