# Computing $\epsilon$-Free NFA from Regular Expressions in $O(n\log^2(n))$ Time*

Christian Hagenah and Anca Muscholl

Institut für Informatik, Universität Stuttgart,
Breitwiesenstr. 20-22, 70565 Stuttgart, Germany

**Abstract.** The standard procedure to transform a regular expression of size $n$ to an $\epsilon$-free nondeterministic finite automaton yields automata with $O(n)$ states and $O(n^2)$ transitions. For a long time this was supposed to be also the lower bound, but a result by Hromkovič et al. [8] showed how to build an $\epsilon$-free NFA with only $O(n\log^2(n))$ transitions. The current lower bound is $\Omega(n\log(n))$, [8]. A rough running time estimation for the construction in [8] yields a cubic algorithm. In this paper we present a sequential algorithm for the construction of Hromkovič et al. which works in time $O(n\log(n) + \text{size of the output})$. On a CREW PRAM their construction can be performed in time $O(\log(n))$ using $O(n + (\text{size of the output})/\log(n))$ processors. We also present a simpler proof of the lower bound for the size.

## 1 Introduction

Among various descriptions of regular languages regular expressions are especially interesting because of their succinctness. On the other hand, the high degree of expressiveness leads to algorithmically hard problems, for example testing equivalence is PSPACE-complete. Finite automata are easier to design and analyze than other equivalent models. Given a regular expression we are often interested in computing an equivalent nondeterministic finite automaton *without $\epsilon$-transitions* (*$\epsilon$-free NFA* for short). This conversion is of interest due to some operations which can be easily performed on $\epsilon$-free NFA, as for example intersection or membership test.

In this paper we present efficient sequential and parallel algorithms for converting regular expressions into small $\epsilon$-free NFA. For a regular expression $E$ we take the number of letters as the size of $E$. The size of an NFA is measured as the number of *transitions*, which is a realistic storage measure for automata, especially when we have a sequence of NFA operations to perform (a similar example is the use of binary decision diagrams in automatic verification). It is known that the translation from NFA to regular expressions can yield an exponential blow-up, [4]. The other direction however can be achieved in polynomial

* Research was partly supported by the French-German project PROCOPE. Current affiliation of the second author: LIAFA, Université Paris VII. Address: 2, pl. Jussieu, case 7014, F-75251 Paris Cedex 05.

time. A well-known method for constructing $\epsilon$-free NFA from regular expressions is based on position automata (Glushkov automata). This classical construction yields NFA of quadratic size (see [5, 10] or [1, 3, 2] for more recent expositions). A substantial improvement on this construction was achieved in [8], where a non-deterministic version of the position automata construction was shown to yield $\epsilon$-free NFA with $O(n \log^2(n))$ transitions. This is optimal up to a possible $\log(n)$ factor, as shown also in [8] by proving an $\Omega(n \log(n))$ lower bound. However, the precise complexity of the conversion proposed in [8] was not investigated. A trivial estimation of the construction of [8] leads to a cubic time algorithm.

Performing the conversion from regular expressions to NFA efficiently is important from a practical viewpoint. The best one can hope for is to have the construction in time proportional to the output size. In the present paper we propose efficient sequential and parallel algorithms for converting regular expressions to $\epsilon$-free NFA. Our approach is based on the idea of common follow-sets which was proposed in [8], but we use a slightly different presentation. This allows us to obtain an algorithm which works in time $O(n \log(n) + \text{size of the output})$. Therefore, our algorithm has worst-case complexity of $O(n \log^2(n))$.

In the parallel setting we are able to perform the construction on a CREW PRAM in $O(\log(n))$ time. The parallel version uses $O(n)$ processors for computing the description of the states of the NFA, resp. $O(n \log(n))$ processors in the worst-case for outputting the NFA. Our parallel algorithm can be compared with an $O(\log(n))$ time algorithm which computes an NFA *with* $\epsilon$-transitions using $O(n/\log(n))$ processors, [6]. More recently, an $O(\log(n))$ PRAM algorithm using $O(n^2/\log(n))$ processors was proposed for the construction of the Glushkov automaton, [12].

The paper is organized as follows. The sequential algorithm is presented in Section 4. Basic notions on position automata are recalled in Section 2, whereas Section 3 deals with the common follow sets construction of [8]. In Section 5 we present the parallel algorithm. Finally, in Section 6 we present a simple proof for the $\Omega(n \log n)$ lower bound.

A preliminary version of this paper was presented at MFCS'98, [7].

## 2 Preliminaries

Let $A$ denote a finite alphabet. We consider non-empty regular expressions $E$ over $A$, that is, $E$ is built from the empty word $\epsilon$ and the letters in $A$ using concatenation $\cdot$, union $+$ and Kleene star $*$. The regular language defined by a regular expression $E$ is denoted $\mathcal{L}(E)$. Nondeterministic finite automata are denoted by $\mathcal{A} = (Q, A, \delta, I, F)$, with $Q$ as set of states, $\delta \subseteq Q \times A \times Q$ as transition relation, $I$ as set of initial states and $F$ as set of final states. The language recognized by $\mathcal{A}$ is denoted by $L(\mathcal{A})$. The size of $\mathcal{A}$, denoted $|\mathcal{A}|$, is the number of transitions of $\mathcal{A}$.

For algorithmic purposes a regular expression $E$ over $A$ is given by a syntax tree $t_E$, which corresponds to an arbitrary, fixed bracketing of the expression. The tree $t_E$ has leaves labeled by $\epsilon$ or symbols from $A$. The inner nodes are either binary and labeled by $+$ or $\cdot$ or they are unary and labeled by $*$. The inner nodes of a syntax tree will be named $F, G, \ldots$ and we follow the notations of [8] by identifying them with (occurrences of) subexpressions of $E$. Thus, a subexpression $F$ of $E$ really means the occurrence of a subexpression. For subexpressions $F, G$ of $E$ we write $F \leq G$ (resp. $F < G$) if the node $F$ is an ancestor (resp. a proper ancestor) of the node $G$ in the tree $t_E$. For a subexpression $F$ let firststar$(F)$ denote the maximal node $G$ (maximal with respect to $\leq$), which satisfies $G \leq F$ and such the parent node of $G$ is $G^*$. Thus, firststar$(F)$ represents the subexpression $G$ of smallest size such that $G^*$ strictly contains $F$ as a subexpression.

A *subtree* $t$ of $t_E$ is meant as a connected subgraph (i.e. a tree) of $t_E$. A subtree $t$ is called *full subtree* if it contains all descendants of its root. This means that a full subtree of $t_E$ corresponds to a subexpression of $E$.

We suppose without loss of generality that the leaves of the expression tree $t_E$ are labeled with pairwise distinct letters. This allows to identify the leaves of $t_E$ labeled by $A$ uniquely by their labeling. For example, for $E = (a^* + b)^*(a+\epsilon)b^*(ab+\epsilon)$ we replace $A$ by $\{a_1, a_2, a_3, b_1, b_2, b_3\}$ and $E$ by $E' = (a_1^* + b_1)^*(a_2+\epsilon)b_2^*(a_3b_3+\epsilon)$. The expression $E'$ is usually called the linearization of $E$. Note that once we construct an NFA for $E'$, we obtain an NFA for $E$ by the homomorphism mapping $a_i$ to $a$ for each $a \in A$. For the rest of the paper we will assume without further mentioning that $E$ is linearized and thus, $\mathrm{pos}(E) \subseteq A$.

## 3 Position Automata

In this section we recall some basic notions related to the construction of position (Glushkov) automata from regular expressions. We follow the notations of [3, 8]. Then we recall the nondeterministic construction of position automata using common follow-sets, which was proposed in [8].

### 3.1 Positions and sets of positions

Given a regular expression $E$, the set $\mathrm{pos}(E)$ comprises all positions of $E$ which are labeled by letters from $A$. According to our convention, $\mathrm{pos}(E) \subseteq A$. Positions of $E$ will be named $x, y, \ldots$. In general, the size of the syntactic tree $t_E$ for $E$ might be much larger than the number of positions $|\mathrm{pos}(E)|$. However, the next lemma shows that we can easily compute an equivalent expression of size linear in $|\mathrm{pos}(E)|$:

**Lemma 1.** *Let $E$ be a regular expression with $n = |pos(E)|$. Then we can compute in linear time an expression $E'$ such that $\mathcal{L}(E) = \mathcal{L}(E')$ and the length of $E'$ is in $O(n)$.*

**Proof:** We first build a syntax tree $t_E$ for $E$ and determine all nodes $F$ of $t_E$ with $\mathcal{L}(F) = \{\epsilon\}$, replacing in this case the full subtree rooted at $F$ by $\epsilon$. If $\mathcal{L}(F) \neq \{\epsilon\}$ we further simplify $t_E$ by replacing

1. every subtree $F + \epsilon$ (resp. $\epsilon + F$) by $F$, if $\epsilon \in \mathcal{L}(F)$,
2. every subtree $F \cdot \epsilon$ (resp. $\epsilon \cdot F$) by $F$,
3. every subtree $F^{**}$ by $F^*$.

Let $t'_E$ be the syntax tree thus obtained. Note that $n = |\text{pos}(t_E)| = |\text{pos}(t'_E)|$. We claim that $t'_E$ has at most $2n$ leaves, which gives the result together with the third condition above. The claim is easily seen by showing inductively that every full subtree with root $F \neq \epsilon$ has at most $2|\text{pos}(F)| - 1$ (resp. $2|\text{pos}(F)|$) leaves if $\epsilon \notin \mathcal{L}(F)$ (resp., if $\epsilon \in \mathcal{L}(F)$). $\square$

Throughout the paper we denote by $n$ the number of positions $|\text{pos}(E)|$ of $E$. The lemma above says that we may assume that the size of the given syntax tree $t_E$ satisfies $|t_E| \le 6n \in O(|\text{pos}(E)|)$.

Let $t$ be a subtree of $t_E$. Then $\text{pos}(t)$ denotes the set of positions occurring in $t$ and $|t|$ is the number of nodes in the subtree of $t_E$ corresponding to $t$. Note that $|\text{pos}(t)| \le |t|$. For an arbitrary subtree $t$ of $t_E$, we do not have $|t| \in O(|\text{pos}(t)|)$, in general.

The following distinguished subsets of positions – *first-*, *last-* and *follow-sets* – form the basis of the classical construction of position automata and of the improved common follow sets construction. For a (linearized) regular expression $E$ we define $\text{first}(E)$ and $\text{last}(E)$ as follows. The set $\text{first}(E) \subseteq \text{pos}(E)$ contains all positions which can occur as first letter in some word in $\mathcal{L}(E)$. Similarly, $\text{last}(E)$ contains all positions which can occur as last letter in some word in $\mathcal{L}(E)$. Formally:

$$\begin{aligned}\text{first}(E) &= \{x \in \text{pos}(E) \mid xA^* \cap \mathcal{L}(E) \neq \emptyset\}\,,\\ \text{last}(E) &= \{x \in \text{pos}(E) \mid A^*x \cap \mathcal{L}(E) \neq \emptyset\}\,.\end{aligned}$$

The sets $\text{first}(E), \text{last}(E)$ can be computed inductively by noting that $\text{first}(F + G) = \text{first}(F) \cup \text{first}(G)$, $\text{first}(F^*) = \text{first}(F)$ and $\text{first}(F \cdot G) = \text{first}(F)$ if $\epsilon \notin \mathcal{L}(F)$, resp. $\text{first}(F \cdot G) = \text{first}(F) \cup \text{first}(G)$ if $\epsilon \in \mathcal{L}(F)$ (symmetrically for $\text{last}(F)$). For any position $x \in \text{pos}(E)$ let $\text{follow}(x) \subseteq \text{pos}(E)$ contain all positions $y$ which are immediate successors of $x$ in some word of $\mathcal{L}(E)$:

$$\text{follow}(x) = \{y \in \text{pos}(E) \mid A^*xyA^* \cap \mathcal{L}(E) \neq \emptyset\}\,. \tag{1}$$

The set $\text{follow}(x)$ can be defined inductively as well. Let $\text{follow}(x) = \text{follow}(E, x)$, where $\text{follow}(F, x)$ is defined as follows for positions $x \in \text{pos}(F)$:

$$\text{follow}(a, x) = \emptyset$$

$$\text{follow}(F + G, x) = \begin{cases} \text{follow}(F, x) & \text{if } x \in \text{pos}(F) \\ \text{follow}(G, x) & \text{if } x \in \text{pos}(G) \end{cases}$$

$$\text{follow}(F \cdot G, x) = \begin{cases} \text{follow}(F, x) & \text{if } x \in \text{pos}(F) \setminus \text{last}(F) \\ \text{follow}(F, x) \cup \text{first}(G) & \text{if } x \in \text{last}(F) \\ \text{follow}(G, x) & \text{if } x \in \text{pos}(G) \end{cases}$$

$$\text{follow}(F^*, x) = \begin{cases} \text{follow}(F, x) & \text{if } x \in \text{pos}(F) \setminus \text{last}(F) \\ \text{follow}(F, x) \cup \text{first}(F) & \text{if } x \in \text{last}(F) \end{cases}$$

For the rest of the paper we use the global definition of follow-sets (1). Our algorithm is based on restrictions of follow-sets, which we denote by $\text{follow}_F(x)$ and $\text{follow}_t(x)$. If $F$ denotes a subexpression of $E$ and $t$ is a subtree of $t$ then let $\text{follow}_F(x) = \text{follow}(x) \cap \text{pos}(F)$ and $\text{follow}_t(x) = \text{follow}(x) \cap \text{pos}(t)$. Note that if $\text{follow}(F, x)$ in the recursive definition is defined, then it is a subset of $\text{follow}_F(x)$.

*Example 2.* Consider the (linearized) expression $E = ((a+b\cdot c)\cdot d)^* \cdot e$. Then $A = \text{pos}(E) = \{a, b, c, d, e\}$, $\text{first}(E) = \{a, b, e\}$, $\text{follow}(a) = \{d\}$, $\text{follow}(d) = \{a, b, e\}$. Let $F = a + b \cdot c$, then $\text{follow}_F(d) = \{a, b\}$ but $\text{follow}(F, d)$ is not defined.

### 3.2 Automata

First-, last- and follow-sets are the basic components of an $\epsilon$-free NFA $\mathcal{A}_E$ recognizing $\mathcal{L}(E)$, called *position automaton* in [8]. Let $\mathcal{A}_E = (Q, A, \delta, \{q_0\}, F)$ be defined by

$$\begin{aligned} Q &= \text{pos}(E)\,\dot\cup\{q_0\}\,, \\ \delta &= \{(q_0, x, x) \mid x \in \text{first}(E)\} \cup \{(x, y, y) \mid y \in \text{follow}(x)\}\,, \\ F &= \begin{cases} \text{last}(E) & \text{if } \epsilon \notin \mathcal{L}(E) \\ \text{last}(E) \cup \{q_0\} & \text{otherwise.} \end{cases} \end{aligned}$$

Recall for the above definition that $\text{pos}(E) \subseteq A$, the expression $E$ being a linearized expression.

**Proposition 3 ([5, 10]).** *For every regular expression $E$ we have $\mathcal{L}(\mathcal{A}_E) = \mathcal{L}(E)$.*

The construction above yields $\epsilon$-free automata with $n+1$ states and $O(n^2)$ transitions. The improvement proposed in [8] decreases the number of transitions by combining subsets of follow-sets and introducing a nondeterministic choice among several subsets of the follow-sets. The heart of the construction is the notion of a *system of common follow sets (CFS system)*, which is defined as follows:

**Definition 4 ([8]).** *Let $E$ be a regular expression. A CFS system $S$ for $E$ is given as $S = (dec(x))_{x \in pos(E)}$, where each $dec(x) \subseteq \mathcal{P}(pos(E))$ is a decomposition of $follow(x)$, that is:*

$$follow(x) = \bigcup_{C \in dec(x)} C\,.$$

*Let $\mathcal{C}_S = \{\mathit{first}(E)\} \cup \bigcup_{x \in pos(E)} dec(x)$. The CFS automaton $\mathcal{A}_S$ associated with $S$ is defined by $\mathcal{A}_S = (Q, A, \delta, \{q_0\}, F)$ where*

$$Q = \mathcal{C}_S \times \{0, 1\}\,,$$
$$q_0 = \begin{cases} (\mathit{first}(E), 1) & \mathit{if}\ \epsilon \in \mathcal{L}(E) \\ (\mathit{first}(E), 0) & \mathit{otherwise} \end{cases},$$
$$\delta = \{((C, f), x, (C', f')) \mid x \in C,\ C' \in dec(x)\ \mathit{and}\ f' = 1 \Leftrightarrow x \in last(E)\}\,,$$
$$F = \mathcal{C}_S \times \{1\}\,.$$

**Lemma 5 ([8]).** *Let $E$ be a regular expression and let $S$ be a CFS system for $E$. Then the CFS automaton $\mathcal{A}_S$ recognizes $\mathcal{L}(E)$.*

It is shown in [8] how to obtain a CFS system $S$ for a given regular expression $E$ such that the following conditions are satisfied:

1. $|\mathcal{C}_S| \in O(n)$,
2. $\sum_{C \in \mathcal{C}_S} |C| \in O(n \log n)$,
3. $|\mathrm{dec}(x)| \in O(\log n)$ for each $x \in \mathrm{pos}(E)$.

This yields an $\epsilon$-free nondeterministic automaton (denoted in the following *CFS automaton*) with $O(n)$ states and $O(n \log^2(n))$ transitions. We first present in Section 4 an optimal sequential algorithm which computes a CFS system $S$ with the above properties. The computation of the CFS system is a simplified version of the procedure proposed in [8].

*Remark 6.* If one is mainly interested in the construction of an $\epsilon$-free NFA for the membership test, then the test can be done just using the CFS system. Another possible way is to use the ZPC structure introduced by Champarnaud et al. for constructing $\epsilon$-free NFA of quadratic size, [11, 13]. A test based on the CFS system needs time proportional to $n \log n$, whereas the ZPC-structure needs time proportional to $n$. The larger size of the CFS system stems from the fact that we are interested in reducing the number of transitions. This is done by adding redundancy in the states.

# 4 Computing a Common Follow Sets System

## 4.1 Properties of follow-sets

The running time of our algorithm relies heavily on some structural properties of follow-sets which are presented in the following. The next lemma will be used in connection with a representation of follow-sets by first-sets.

**Lemma 7.** *Let $E$ be a regular expression and let $F, G$ be subexpressions of $E$ with $F \leq G$. Then we have:*

1. *$\mathit{first}(F) \cap \mathit{first}(G) \neq \emptyset$ implies $\mathit{first}(G) \subseteq \mathit{first}(F)$.*

2. $F \leq H \leq G$ *and* $\emptyset \neq \mathit{first}(G) \subseteq \mathit{first}(F)$ *implies* $\mathit{first}(G) \subseteq \mathit{first}(H) \subseteq \mathit{first}(F)$.
3. $x \in \mathit{pos}(G) \setminus \mathit{first}(G)$ *implies* $x \notin \mathit{first}(F)$.

The proof of the lemma is a straightforward application of the inductive definition. An analogous lemma can be stated for last-sets.

The next lemma deals with the relation between follow-sets and a decomposition of the syntax tree, which will be used recursively in the definition of the CFS system. Recall that we denote for $x \in \mathrm{pos}(E)$, a subexpression $E \leq F$ and a subtree $t$ of $E$, by $\mathrm{follow}_F(x)$ ($\mathrm{follow}_t(x)$, respectively) the restriction $\mathrm{follow}(x) \cap \mathrm{pos}(F)$ ($\mathrm{follow}(x) \cap \mathrm{pos}(t)$, respectively). The proof of the lemma below follows directly from the definitions (see also Fig. 1).

**Lemma 8.** *Given a regular expression $E$ and subexpressions $F, F_1$ of $E$ with $F < F_1$. Let $t_1, t_2$ be subtrees of $t_E$ such that $\mathit{pos}(t_2) \subseteq \mathit{pos}(F) \setminus \mathit{pos}(F_1)$ and $\mathit{pos}(t_1) \subseteq \mathit{pos}(F_1)$. Then we have for positions $x, x', y \in \mathit{pos}(E)$:*

1. $\mathit{follow}_{t_2}(x) = \emptyset$ *for all* $x \in \mathit{pos}(t_1) \setminus \mathit{last}(F_1)$.
2. $\mathit{follow}_{t_2}(x) = \mathit{follow}_{t_2}(x')$ *for all* $x, x' \in \mathit{pos}(t_1) \cap \mathit{last}(F_1)$.
3. $\mathit{follow}_{t_1}(y) = \mathit{first}(F_1) \cap \mathit{pos}(t_1)$ *for all* $y \in \mathit{pos}(t_2)$ *with* $\mathit{follow}_{t_1}(y) \neq \emptyset$.

### 4.2 Recursive definition of CFS systems

The CFS system defined in [8] is based on a divide-and-conquer construction. Consider a subtree $t$ of $t_E$ and let $F$ denote the root of $t$. Let $x \in \mathrm{pos}(t)$. If $|\mathrm{pos}(t)| = 1$ then we define

$$C_0 = \mathrm{follow}_t(x) = \mathrm{follow}(x) \cap \{x\}, \quad \mathrm{dec}(x,t) = \{C_0\}\,.$$

Suppose now that $|\mathrm{pos}(t)| > 1$. Then let $t_1$ be a subtree of $t$ such that

$$\frac{1}{3}|\mathrm{pos}(t)| \leq |\mathrm{pos}(t_1)| \leq \frac{2}{3}|\mathrm{pos}(t)|$$

and let $t_2 = t \setminus t_1$ (see also Fig. 1). Let $F_1$ denote the root of $t_1$. Clearly, for every position $x \in \mathrm{pos}(t)$ we have $\mathrm{follow}_t(x) = \mathrm{follow}_{t_1}(x) \cup \mathrm{follow}_{t_2}(x)$ and $\mathrm{follow}_{t_1}(x) \cap \mathrm{follow}_{t_2}(x) = \emptyset$. We distinguish two cases, depending on $x \in \mathrm{pos}(t_1)$ or $x \in \mathrm{pos}(t_2)$.

i) Let $x \in \mathrm{pos}(t_1)$. If $x \notin \mathrm{last}(F_1)$ then by Lemma 8 we have $\mathrm{follow}_{t_2}(x) = \emptyset$. Otherwise, for $x \in \mathrm{last}(F_1)$ then again by Lemma 8 we have $\mathrm{follow}_{t_2}(x) = \mathrm{follow}_{t_2}(x')$ for all $x' \in \mathrm{last}(F_1) \cap \mathrm{pos}(t_1)$.
Let $C_1 = \mathrm{follow}_{t_2}(x')$ for some $x' \in \mathrm{pos}(t_1) \cap \mathrm{last}(F_1)$ and define $\mathrm{dec}(x,t)$ as

$$\mathrm{dec}(x,t) = \begin{cases} \mathrm{dec}(x,t_1) & \text{if } x \notin \mathrm{last}(F_1) \\ \mathrm{dec}(x,t_1) \cup \{C_1\} & \text{otherwise} \end{cases}$$

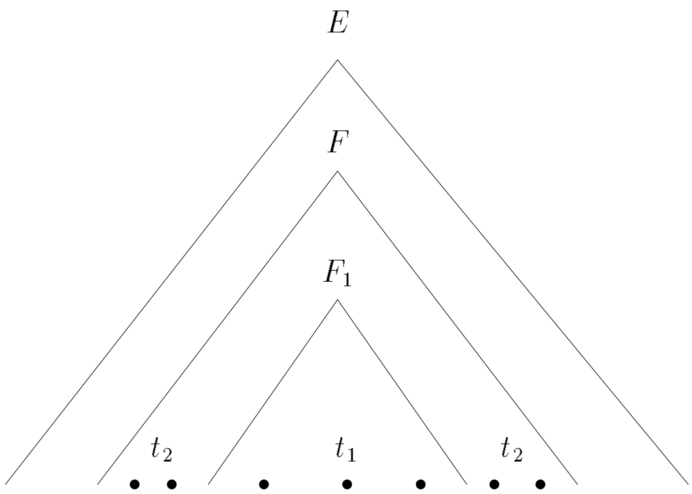


**Fig. 1.** Recursion step.

ii) Let $x \in \mathrm{pos}(t_2)$. If $\mathrm{follow}_{t_1}(x) \neq \emptyset$ then we have $\mathrm{follow}_{t_1}(x) = \mathrm{first}(F_1) \cap \mathrm{pos}(t_1)$ by Lemma 8.
Let $C_2 = \mathrm{first}(F_1) \cap \mathrm{pos}(t_1)$ and define $\mathrm{dec}(x, t)$ as

$$\mathrm{dec}(x,t) = \begin{cases} \mathrm{dec}(x,t_2) & \text{if } \mathrm{follow}_{t_1}(x) = \emptyset \\ \mathrm{dec}(x,t_2) \cup \{C_2\} & \text{otherwise} \end{cases}$$

It can be easily verified that $\mathrm{dec}(x,t)$ is a decomposition of $\mathrm{follow}_t(x)$, i.e. we have $\mathrm{follow}_t(x) = \bigcup_{C \in \mathrm{dec}(x,t)} C$. Hence, we obtain a CFS system $\mathcal{C}(t)$ restricted to $t$, where

$$\mathcal{C}(t) = \bigcup \{\mathrm{dec}(x,t) \mid x \in \mathrm{pos}(t)\} = \{C \mid C \in \mathrm{dec}(x,t) \text{ for some } x \in \mathrm{pos}(t)\}\,.$$

Note that $|\mathcal{C}(t)| \leq |\mathcal{C}(t_1)| + |\mathcal{C}(t_2)| + 2$. This yields $|\mathcal{C}(t)| \leq 3|\mathrm{pos}(t)| - 2$. Similarly, the following estimations can be easily verified (see also Lemma 4 of [8]):

- $\sum_{C \in \mathcal{C}(t)} |C| \leq 3|\mathrm{pos}(t)| \log(|\mathrm{pos}(t)|)$,
- $|\mathrm{dec}(x,t)| \leq 2\log(|\mathrm{pos}(t)|) + 1$, for all $x \in \mathrm{pos}(t)$.

We conclude this section with an example for the CFS automaton.

*Example 9.* Let $E = ((a + b \cdot c) \cdot d)^* \cdot e$ be the expression from Example 2. For the first recursion step, we decompose $t_E$ into $t_1$, $t_2$, where $t_1$ is the subexpression $F_1 = a + b \cdot c$ and $t_2 = t \setminus t_1$. We have $C_1 = \mathrm{follow}_{t_2}(a) = \{d\}$ and $C_2 = \mathrm{first}(F_1) = \{a, b\}$. The remaining recursive calls add the sets $\{e\}$ and $\{c\}$ to the CFS system. The resulting CFS automaton is shown in Figure 2. Note how $\mathrm{follow}(d) = \{a, b, e\}$ is splitted into $\{a, b\}$ and $\{e\}$.

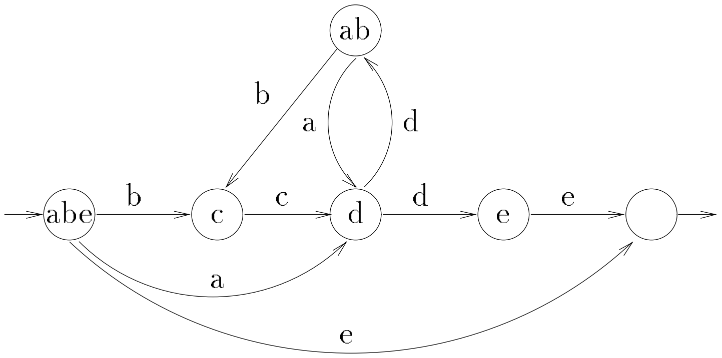


**Fig. 2.** CFS automaton for $E = ((a + b \cdot c) \cdot d)^* \cdot e$.

## 5 A Sequential $O(n \log(n))$ Algorithm for Computing a Common Follow Sets System

We describe now an efficient way to compute the sets defined in the previous section. For $C_0 = \text{follow}(x) \cap \{x\}$ we have to determine whether $x \in \text{follow}(x)$. The sets $C_1, C_2$ are given as a follow-set (resp. first-set) intersected with the set of positions of a subtree. First note that computing explicitly all follow-sets is too expensive, since the sum of all sizes of follow-sets is in the worst case quadratic in $n$. As an example for the worst-case consider the following expression (see also Section 7):

$$E_n = (a_1 + \epsilon)(a_2 + \epsilon) \cdots (a_n + \epsilon)\,.$$

Checking whether $x \in \text{follow}(x)$ is equivalent to the condition $x \in \text{last}(S) \cap \text{first}(S)$, where $S = \text{firststar}(x)$. For the recursion step we have to determine $C_1, C_2$ with

$$C_1 = \text{follow}_{t_2}(x) = \text{follow}(x) \cap \text{pos}(t_2) \quad \text{and} \quad C_2 = \text{first}(F_1) \cap \text{pos}(t_1)\,.$$

We want to compute the sets $C_1, C_2$ for all positions $x \in \text{pos}(t)$ in time $O(|t|)$. As shown below, the computation of $C_1$ reduces to computing a union of first-sets restricted to $\text{pos}(t_2)$. This yields two problems: First we need an efficient way to compute intersections of first-sets with a given set of positions. Second, performing the union must be done efficiently, which actually means that the union has to be carried over pairwise disjoint sets. This is one main reason why we have chosen to work with follow-sets relativized to subtrees instead of using the inductive definition of follow-sets, since the inductive definition leads to overlapping of first-sets. The solution to both problems will rely on a suitable data structure for first-sets. Before discussing the data structure let us consider the set $C_1$ in more details.

**Definition 10.** *Let $E \leq F$ be regular expressions. We define $fnext(F) \subseteq pos(E)$ as*

$$fnext(F) = \begin{cases} first(G) & \text{if } F \cdot G \text{ is the parent node of } F \\ first(F) & \text{if } F^* \text{ is the parent node of } F \\ \emptyset & \text{otherwise} \end{cases}$$

*Analogously, $lprev(F)$ is defined by replacing first by last and by requiring that $G \cdot F$ (instead of $F \cdot G$) is the parent node of $F$.*

Using the fnext operator we can express follow-sets as unions of first-sets. Lemma 3 in [8] states that $\mathrm{follow}(x)$ can be written as a union of sets $\mathrm{fnext}(G)$, where the union is taken over all nodes $E \leq G$ with $x \in \mathrm{last}(G)$. Using relativized follow-sets we are able to identify exactly the sets of positions which contribute to the intersection of $\mathrm{follow}(x)$ with $\mathrm{pos}(t_2)$. Note that in the proposition below at most *one* node, $\mathrm{firststar}(F)$, which is outside of $F$ contributes to $\mathrm{follow}_E(x) \cap \mathrm{pos}(t_2)$. This property is needed in order to be able to determine the set $C_1$ in time $O(|t|)$.

**Proposition 11.** *Let $E$ be a regular expression and $F < F_1$ two subexpressions of $E$. Let $t_2$ be a subtree of $t_E$ with root $F$ and $pos(t_2) \cap pos(F_1) = \emptyset$. Assume that $x \in last(F_1)$, then we have*

$$follow_{t_2}(x) = \bigcup_{G \in \mathcal{G}} (fnext(G) \cap pos(t_2))$$

*where the union is taken over the set*

$$\mathcal{G} = \{G \mid last(G) \supseteq last(F_1) \text{ and } (F < G \leq F_1 \text{ or } G = firststar(F))\}.$$

**Proof:** Note that for every $G \leq F_1$ with $\mathrm{last}(F_1) \subseteq \mathrm{last}(G)$ we have $\mathrm{fnext}(G) \cap \mathrm{pos}(t_2) \subseteq \mathrm{follow}_{t_2}(x)$. Conversely, consider a position $y \in \mathrm{follow}_{t_2}(x)$ with $y \notin \mathrm{fnext}(G)$, for all $F < G \leq F_1$ with $\mathrm{last}(F_1) \subseteq \mathrm{last}(G)$. Hence, there exists some node $G$, $E \leq G \leq F$, with $y \in \mathrm{fnext}(G)$ and $\mathrm{last}(F_1) \subseteq \mathrm{last}(G)$. Clearly, the parent node of $G$ is $G^*$ (otherwise, $\mathrm{fnext}(G) \cap \mathrm{pos}(t_2) = \emptyset$), thus $y \in \mathrm{first}(G) \cap \mathrm{pos}(t_2)$. If $G = \mathrm{firststar}(F)$ then we are done. Otherwise $G < H = \mathrm{firststar}(F)$. In this case it is not difficult to verify using Lemma 7 that for all $G < H$ with $\mathrm{first}(G) \cap \mathrm{pos}(t_2) \neq \emptyset$ we also have $\mathrm{first}(G) \cap \mathrm{pos}(t_2) = \mathrm{first}(H) \cap \mathrm{pos}(t_2)$. Therefore, $y \in \mathrm{first}(H) \cap \mathrm{pos}(t_2)$. $\square$

Our sequential algorithm is based on a suitable ordering of positions of $E$, which allows manipulating first-sets efficiently. We use an array called firstdata such that for each subexpression $F$ of $E$ the set $\mathrm{first}(F)$ is a subarray of firstdata. The crucial point for the sequential algorithm is the order of positions within firstdata. This particular ordering will allow us to perform the intersection of a follow-set and the set of positions $\mathrm{pos}(t_2)$ in time linear in $|t_2|$. Since the follow-set is represented as a (disjoint) union of first-sets, the ordering of these first-sets in the union must be compatible with the order of positions in $t_2$. Recall that $t_E$

is a fixed syntax tree of $E$. We first define a forest $\mathcal{F}$ by deleting all edges in $t_E$ which go from nodes labeled $F \cdot G$ to the child labeled $G$, whenever $\epsilon \notin \mathcal{L}(F)$. Let $\mathcal{F}$ consist of trees $T_1, \ldots, T_k$. We denote in the following the trees $T_i$ as *first-trees*.[1] Note that each $\text{first}(F)$ consists of all leaves $x$ with $F < x$ where $F, x$ belong to the same first-tree.

We define a total order on $\mathcal{F}$ as follows. For $i \neq j$ let $T_i \prec T_j$ whenever the roots $F_i, F_j$ of $T_i$, resp. $T_j$ satisfy

- either $F_j < F_i$, i.e. $F_j$ is an ancestor of $F_i$ in $t_E$,
- or $F_i$ and $F_j$ are incomparable with respect to $<$ and $F_i$ is situated to the right of $F_j$.

The order $\prec$ corresponds thus to a reversed preorder traversal of $t_E$, i.e. right child—left child—parent node.

Suppose that after renaming we have $\mathcal{F} = \{T_1, \ldots, T_k\}$ with $T_1 \prec \cdots \prec T_k$. The array $\mathsf{firstdata}$ is defined by the concatenation of $\mathsf{fdata}(T_1), \ldots, \mathsf{fdata}(T_k)$, with $\mathsf{fdata}(T_i)$ being the list of positions corresponding to the leaves of $T_i$ ordered from left to right. By a preorder traversal of each first-tree $T_i$ we can determine for each subexpression $F$ within $T_i$ the subinterval of $\mathsf{fdata}(T_i)$ corresponding to $\text{first}(F)$. The set $\text{first}(F)$ is described by its starting position $\mathsf{fstart}(F)$ within $\mathsf{firstdata}$ (i.e., $\mathsf{fdata}(T_i)$) and its length $\mathsf{flength}(F) = |\text{first}(F)|$.

*Remark 12.* $(i)$ Let $F, G$ be subexpressions of $E$. Then we have $\emptyset \neq \text{first}(F) \subseteq \text{first}(G)$ if and only if $\mathsf{fstart}(G) \leq \mathsf{fstart}(F)$ and $\mathsf{fstart}(F) + \mathsf{flength}(F) \leq \mathsf{fstart}(G) + \mathsf{flength}(G)$, i.e. if the subinterval corresponding to $\text{first}(G)$ covers the subinterval corresponding to $\text{first}(F)$. Moreover, $\mathsf{firstdata}$ allows to determine the intersection $\text{first}(F) \cap \text{pos}(t)$ in $O(|t|)$ time. Hereby is $F$ a subexpression (given by a node in some $T_i$) and $t$ is subtree of $t_E$ (given by a set of positions in increasing order).

$(ii)$ A similar data structure $\mathsf{lastdata}$ can be defined for the last-sets.

Before describing the main procedure which computes the sets $C_1, C_2$ let us describe the data structures on the expression from Example 2.

*Example 13.* Let $E = ((a + b \cdot c) \cdot d)^* \cdot e$. Figure 3 contains the first-trees obtained from the expression tree $t_E$. Note the reversed preorder of first-trees, $T_1, T_2, T_3$.

The table in Figure 4 shows the entries of $\mathsf{firstdata}$ in the first row. The remaining rows denote the subexpressions of $E$ and their first-sets as subintervals of $\mathsf{firstdata}$. For example, $\text{first}(a + b \cdot c) = \text{first}((a + b \cdot c) \cdot d)$ is the subinterval $a, b$.

We are now ready to describe an algorithm UnionFirstSets for the following problem. Let $F, F_1$ be subexpressions of $E$ with $F < F_1$ and let $t, t_2$ be subtrees of $F$, resp. $t_1$ a subtree of $F_1$, where $t = t_1 \dot\cup t_2$, $\text{pos}(F_1) \cap \text{pos}(t_2) = \emptyset$. Moreover, let

[1] Let us mention the similar data structure introduced in [11, 13].

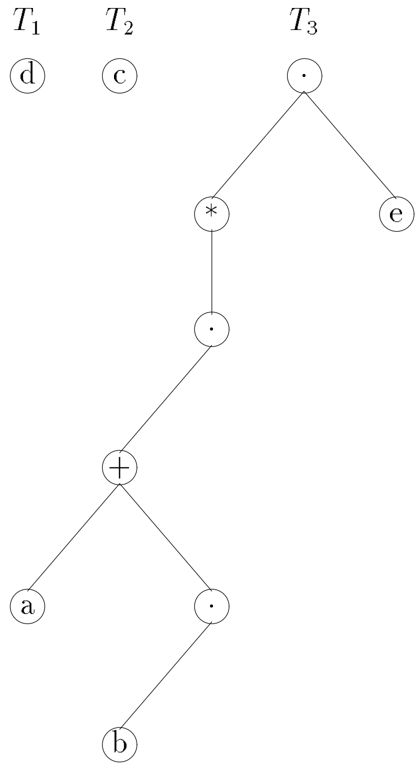


**Fig. 3.** First-trees of $E$.

<table>
<tr><td>$d$</td><td>$c$</td><td>$a$</td><td>$b$</td><td>$e$</td></tr>
</table>

<table>
<tr><td>$d$</td><td>$c$</td><td>$a$</td><td>$b$<br>$b \cdot c$</td><td>$e$</td></tr>
<tr><td colspan="2"></td><td colspan="2">$a + b \cdot c$<br>$(a + b \cdot c) \cdot d$<br>$((a + b \cdot c) \cdot d)^*$</td><td></td></tr>
<tr><td colspan="2"></td><td colspan="3">$((a + b \cdot c) \cdot d)^* \cdot e$</td></tr>
</table>

**Fig. 4.** first-sets in firstdata.

$x \in \text{last}(t_1)$ be a position for which we want to compute the set $C = \text{follow}_{t_2}(x)$. Recall from Proposition 11 that

$$C = \bigcup_{G \in \mathcal{G}} (\text{fnext}(G) \cap \text{pos}(t_2)),$$

with $G \in \mathcal{G}$ if and only if $\text{last}(F_1) \subseteq \text{last}(G)$, and either $F < G \leq F_1$ or $G = \text{firststar}(F)$.

The function UnionFirstSets computes a list of nodes rootlist such that $C$ is the union of $\text{first}(H)$, with $H$ either in rootlist or $H = \text{firststar}(F)$. Recall that we want to obtain $C$ as a disjoint union of first-sets. We obtain the disjointness as follows. Every node $H$ satisfying $\text{last}(F_1) \subseteq \text{last}(H)$ and $F < H \leq F_1$ is (temporarily) added to rootlist and included in the list tocheck. Concretely, tocheck is a list of pointers to elements of rootlist, which might be deleted later from rootlist. Everytime when a node $G$ is considered for inclusion into rootlist, we have to check whether there is some $H$ in rootlist and tocheck with $\text{first}(H) \subseteq \text{first}(G)$, and delete $H$ from rootlist, if this is the case. Note that only nodes $G$ with $G^*$ as parent node can yield this situation. After performing all deletions, the only node from rootlist which might be deleted later is node $G$ itself, hence we let tocheck consist of $G$, only.

The data structures tree, node, nodelist used below correspond to subtrees of $t_E$, nodes of $t_E$, resp. lists of nodes in $t_E$. Concatenation of lists is denoted by $\circ$.

```
function UnionFirstSets (node F_1, tree t_2) : nodelist;
var rootlist: nodelist;
    tocheck: list of references to node;
    G: node;
begin
    rootlist := nil;
    tocheck := nil;
    G := F_1;
    while (G ≠ root(t_2) and last(F_1) ⊆ last(G)) do
        A := parent node of G;
        if A = G* then
           rootlist := rootlist \ {H | H ∈ tocheck and first(H) ⊆ first(G)};
           rootlist := rootlist ∘ G;
           tocheck := (G);
        else if A = G · H then
              if ε ∈ L(G) then rootlist := rootlist ∘H;
              else rootlist := H∘ rootlist endif;
              tocheck := tocheck ∘ H;
        endif;
        G := A;
    endwhile;
    return(rootlist);
end
```

The next proposition states the basic properties of the algorithm UnionFirstSets. First, we show that UnionFirstSets computes the union of fnext$(G)$ over all nodes $G$ with $F < G \leq F_1$ and last$(F_1) \subseteq$ last$(G)$ as a *disjoint* union of first-sets. Second, we show that the ordering of the list of nodes computed by UnionFirstSets is compatible with the reversed preorder $\prec$. This makes it possible to perform an intersection with pos$(t_2)$ in time linear in $|t_2|$, assuming that pos$(t_2)$ is sorted also with respect to $\prec$.

**Proposition 14.** *Let $F < F_1$ be subexpressions of $E$ and let $t_1, t_2$ be subtrees of $t_E$ such that $F_1$, $F$ is the root of $t_1$, resp. $t_2$. Assume that $pos(F_1) \cap pos(t_2) = \emptyset$. Let $x \in last(t_1)$ be a position in $t_1$ and let*

$$\mathcal{G} = \{G \mid last(G) \supseteq last(F_1) \text{ and } (F < G \leq F_1 \text{ or } G = firststar(F))\} .$$

*be defined as above. Then UnionFirstSets$(F_1, t_2)$ yields a list* $\mathsf{rootlist} = (H_1, \ldots, H_l)$ *of (roots of) subexpressions of $E$ satisfying the following:*

1. *The sets $first(H_i)$ are pairwise disjoint and we have:*

$$\bigcup_{i=1}^{l} first(H_i) = \bigcup_{G \in \mathcal{G}, G \neq firststar(F)} fnext(G) .$$

2. *Let $T(H_i)$ denote the first-tree in the forest $\mathcal{F}$ containing the subexpression $H_i$. Then $T(H_i) \preceq T(H_j)$ for all $i < j$. Moreover, if $T(H_i) = T(H_j)$ then $H_i$ precedes $H_j$ with respect to preorder in $t_E$.*
3. *UnionFirstSets$(F_1, t_2)$ runs in time $O(|t_2|)$.*

*Remark 15.* Before giving the proof of Proposition 14 we explain how to compute the set $\bigcup_{G \in \mathcal{G}}(\text{fnext}(G) \cap \text{pos}(t_2))$ in time $O(|t_2|)$ using $\mathsf{rootlist}$. The first step is to compute in time $O(|t_2|)$ the set of positions of $t_2$ with respect to the reversed preorder $\prec$ (i.e., right child – left child – parent node). This is done by using at each recursive call the restriction of $\mathsf{firstdata}$ to pos$(t)$, where $t$ is the current subtree. Let us call the resulting list $\mathsf{fdata}(t_2)$. Thus, if $x$ precedes $y$ in $\mathsf{fdata}(t_2)$ and $T, T'$ denote the first-trees containing $x, y$, respectively, then either $T \prec T'$, or $T = T'$ and $x$ precedes $y$ in $T$ with respect to preorder. For example, $\mathsf{fdata}(F_1)$ with $F_1 = a + b \cdot c$ from Example 9 is $(c, a, b)$.

Therefore, $\mathsf{rootlist}$ and $\mathsf{fdata}(t_2)$ can be read sequentially in parallel, building their intersection in time linear in $|t_2|$. Note by Remark 12 that we can determine in constant time whether a position belongs to a set first$(G)$. Note also that the output of UnionFirstSets, $\mathsf{rootlist}$, has at most $|t_2|$ elements, since the while-loop in UnionFirstSets is executed at most $|t_2|$ times. Finally, if $S =$ firststar$(F)$ is defined then we can check whether last$(F_1) \subseteq$ last$(S)$ in constant time and compute first$(S) \cap$ pos$(t_2)$ in time $O(|t_2|)$.

**Proof** of Proposition 14:

1,2. First, note that UnionFirstSets considers all nodes $G$ satisfying $F < G \leq F_1$ and last$(F_1) \subseteq$ last$(G)$ and adds fnext$(G)$ to $\mathsf{rootlist}$. Later, fnext$(G)$ might get deleted, but only when it is included in some other set $\mathrm{fnext}(G')$ which is currently added to rootlist. This shows that the two unions of first-sets are equal.

Let now $H, H'$ be nodes contained in rootlist at the end of the procedure. Let us suppose that $G, G'$ are the nodes considered in the while-loop of UnionFirstSets which caused the insertion of $H$, resp. $H'$ into rootlist. We want to show that $\mathrm{first}(H)$ and $\mathrm{first}(H')$ are disjoint and that $H, H'$ are ordered as claimed in the proposition. Suppose that we have $\mathrm{fnext}(G) = \mathrm{first}(H)$ and $\mathrm{fnext}(G') = \mathrm{first}(H')$. Moreover, suppose that $H$ was inserted after $H'$, thus $G$ is an ancestor of $G'$, $G < G'$. Finally, let $T, T'$ be the first-trees containing $H, H'$ and define $X$, resp. $X'$, as the root of $T$, resp. $T'$. We distinguish two cases:

i) Suppose that $G^*$ is the parent node of $G$, hence $H = G$. Thus, the root $X$ of the first-tree containing $H$ is an ancestor of the root $X'$ for $H'$. With $X \leq X'$ we obtain by definition $T' \preceq T$. Moreover, $H = G$ is an ancestor of $H'$ and we have $T' \neq T$ due to tocheck (since $H'$ still belongs to rootlist after $G$ is inserted).

ii) Suppose that $G \cdot H$ is the parent node of $G, H$. If $\epsilon \in \mathcal{L}(G)$ then $T$ contains $H, G$ and $G \cdot H$. Thus, $X \leq G \cdot H$ and therefore we have again $X \leq X'$ and $T' \preceq T$. Furthermore, $H'$ precedes $H$ with respect to preorder ($H'$ is to the left of $H$). Otherwise, if $\epsilon \notin \mathcal{L}(G)$, then $X = H$ and $T$ contains neither $G$ nor $G \cdot H$. Thus, we have either $X' < X$ or $X', X$ are incomparable and $X'$ is to the left of $X$. In both cases we obtain $T \prec T'$.

Note that the only case considered above where we might get $\mathrm{first}(H) \cap \mathrm{first}(H') \neq \emptyset$ is the case where the parent node of $G$ is $G^*$ (conversely, if the parent of $G$ is $G \cdot H$, then $\mathrm{first}(H')$ and $\mathrm{first}(H)$ are necessarily disjoint). But in this case we have $\mathrm{first}(H') \subseteq \mathrm{first}(H)$ and UnionFirstSets would detect the inclusion, deleting $H'$ from rootlist (if $H'$ still belongs to rootlist).

3. The main loop in UnionFirstSets is executed at most $|t_2|$ times. Moreover, every node $G$ with $F < G \leq F_1$ is considered at most once with respect to the list tocheck. Finally, the test $\mathrm{first}(H) \subseteq \mathrm{first}(G)$ is also performed in constant time, see Remark 12.

The running time of our sequential algorithm is optimal with respect to the worst case size of the CFS system, resp. CFS automaton:

**Theorem 16.** *Let $E$ be a regular expression given by a syntax tree $t_E$ of size $O(|E|) = O(n)$. We can compute a CFS system $S$ for $E$ in time $O(n \log(n))$. Therefore, we can compute an $\epsilon$-free NFA $\mathcal{A}_S$ for $E$ in time $O(n \log(n) + |\mathcal{A}_S|)$. The worst case complexity of the algorithm is $O(n \log^2(n))$.*

**Proof:** The algorithm essentially matches the recursive definition of $\mathrm{dec}(x, t)$ given in Section 4.2. Suppose that $t$ is the current subtree of $t_E$, for which we want to compute $\mathrm{dec}(x, t)$ for all positions $x$ in $t$. First, we decompose $t$ in linear time into subtrees $t_1$ and $t_2$ with

$$\frac{1}{3}|\mathrm{pos}(t)| \leq |\mathrm{pos}(t_1)| \leq \frac{2}{3}|\mathrm{pos}(t)|$$

We first perform the recursive calls on $t_1$, $t_2$ and compute $\mathrm{dec}(x_1, t_1)$, $\mathrm{dec}(x_2, t_2)$ for all positions $x_1$ of $t_1$ and $x_2$ of $t_2$. Note that the subtrees $t_1$, $t_2$ are disjoint and they can be copied for the recursive call (as an alternative, we can use a flag for marking which nodes in $t_E$ are splitting nodes of the recursive calls).

Next we consider the sets which have to be added to $\mathrm{dec}(x, t)$. For a position $x \in \mathrm{pos}(t_1)$ we test whether $x \in \mathrm{last}(F_1)$ in constant time (using lastdata), whereas $C_1 = \mathrm{follow}_{t_2}(x)$ can be computed in time $O(|t_2|)$ by Proposition 14. The case where $x \in \mathrm{pos}(t_2)$ is dual. Here, we can test for any position $x \in \mathrm{pos}(t_1)$ whether it belongs to $\mathrm{first}(F_1)$ in constant time, and the set $C_2 = \mathrm{first}(F_1) \cap \mathrm{pos}(t_1)$ can be computed in time $O(|t_1|)$. Determining which $x \in \mathrm{pos}(t_2)$ satisfies $\mathrm{follow}_{t_1}(x) \neq \emptyset$ also requires $O(|t_1|)$ steps. To see this, note that if $C_2 \neq \emptyset$, then for any position $y \in \mathrm{first}(F_1)$ we have

$$\{x \in \mathrm{pos}(t_2) \mid \mathrm{follow}_{t_1}(x) \neq \emptyset\} = \mathrm{precede}(y) \cap \mathrm{pos}(t_2)\,,$$

where $\mathrm{precede}(y)$ is defined as the dual of $\mathrm{follow}(y)$, i.e.,

$$\mathrm{precede}(y) = \{x \mid A^* x y A^* \cap \mathcal{L}(E) \neq \emptyset\}\,.$$

Moreover, by duality we have

$$\mathrm{precede}(y) \cap \mathrm{pos}(t_2) = \bigcup_{G \in \mathcal{G}} (\mathrm{lprev}(G) \cap \mathrm{pos}(t_2))\,,$$

with $G \in \mathcal{G}$ if $\mathrm{first}(F_1) \subseteq \mathrm{first}(G)$, and either $F < G \leq F_1$ or $G = \mathrm{firststar}(F)$.

Therefore, we can compute in time $O(|t|)$ the sets $\mathrm{dec}(x, t)$ from $\mathrm{dec}(x, t_1)$ and $\mathrm{dec}(x, t_2)$ for all positions $x$ of $t$. The recursion depth of the algorithm is in $O(\log(n))$ and the recursive calls are performed on disjoint subtrees. Hence, our algorithm computes a CFS system $S$ and the states of $\mathcal{A}_S$ in time $O(n \log(n))$. Recall that a state of $\mathcal{A}_S$ has the form $(C, f)$, where either $C = \mathrm{first}(E)$ or $C$ belongs to some $\mathrm{dec}(x)$, and $f \in \{0, 1\}$. A set $C \subseteq \mathrm{dec}(x)$ which is computed by our algorithm is just a list of positions. Finally, transitions of $\mathcal{A}_S$ have the form $((C, f), x, (C', f'))$, where $x \in C$ and $C' \in \mathrm{dec}(x)$ (and $f' = 1$ iff $x \in \mathrm{last}(E)$). Thus, computing the transitions of $\mathcal{A}_S$ can be done in time $O(|\mathcal{A}_S|)$. □

# 6 A Parallel $\log(n)$ Algorithm for Computing a Common Follow Sets System

We describe in this section a parallel version of our algorithm which computes a CFS system on a PRAM in $O(\log(n))$ time using $O(n)$ processors. All computations performed in the sequel assume the concurrent-read-exclusive-write PRAM model. First let us summarize the parallel complexity of some basic precomputation steps:

**Lemma 17 ([6]).** *Let $E$ be a regular expression of length $m$. Then the problems listed below can be solved in parallel in time $O(\log(m))$ using $O(m)$ processors:*

1. *Determine all subexpressions $F$ with $\mathcal{L}(F) = \{\epsilon\}$ (resp. $\epsilon \in \mathcal{L}(F)$).*
2. *Compute a syntax tree $t_E$ with $\mathcal{L}(t_E) = \mathcal{L}(E)$ of size $O(|pos(E)|)$.*

For the rest of the section we denote by $n = |\text{pos}(E)|$ the size of the expression $|E|$, resp. of its syntax tree $t_E$.

**Lemma 18.** *Let $t_E$ be a syntax tree for $E$ of size $O(n)$. Then the problems listed below can be solved in parallel in time $O(\log(n))$ using $O(n)$ processors:*

1. *Compute firststar$(F)$ for all subexpressions $F$.*
2. *Determine $|pos(F)|$ for all subexpressions $F$.*

**Proof:** Computing firststar$(F)$ for all subexpressions $F$ can be done by a standard pointer doubling algorithm. For $|\text{pos}(F)|$ we can apply the usual technique for expression evaluation, see e.g. [6]. □

For any pair of nodes $F, G$ in $t_E$ we denote by parent$(F, G)$ the lowest common ancestor of $F, G$, that is, the expression $H$ of smallest size with $H \leq F$ and $H \leq G$.

**Lemma 19.** *Let $t_E$ be a syntax tree for $E$ of size $O(n)$. Then $t_E$ can be preprocessed in time $O(\log(n))$ using $O(n/\log(n))$ processors such that*

1. *parent$(F_1, F_2)$ can be computed in $O(1)$ steps sequentially for any subexpressions $F_1, F_2$.*
2. *For each subtree $t$ of $t_E$ a decomposition $t = t_1 \dot\cup t_2$ satisfying $1/3|pos(t)| \leq |pos(t_1)| \leq 2/3|pos(t)|$ can be computed in $O(1)$ steps using $O(|t|)$ processors.*

**Proof:** The first part of the statement can be found in [9], p. 128. The proof idea is to use the enumeration of the vertices of the tree as visited by an Euler tour. The result of the preprocessing is a constant number of arrays of size $O(n)$. We omit the details of the construction. For the second assertion consider a subtree $t$ of $t_E$ and assume that each node of $t$ contains the number of positions which are its descendants in $t$. Let $F$ be the root of $t$. Let $G$ be a descendant of $F$ of maximal depth such that the subtree $t'$ rooted at $G$ satisfies $|\text{pos}(t')| \geq 2/3|\text{pos}(t)|$. Note that $G$ is uniquely determined. Moreover, at least one of the children of $G$ contains at least $1/3|\text{pos}(t)|$ positions. Thus, we can fix a node $F_1$ for the $1/3 - 2/3$ decomposition of $t$. We can compute the size of the decomposition subtrees $t_1, t_2$ in constant time using the subroutine for parent$(F_1, F_2)$. □

For the data structure firstdata used for first-sets in the previous section we will not assume a particular ordering among the first-trees of the forest $\mathcal{F}$ in the parallel case. This stems from the fact that intersecting two sets in parallel can be done in constant time, independently of any ordering. We can choose for example to order first-trees $T_1, \ldots, T_k$ by their root nodes in preorder. The array firstdata is defined as in Section 5 by the concatenation of fdata$(T_1), \ldots,$ fdata$(T_k)$.

**Lemma 20.** *Let $t_E$ be a syntax tree for $E$ of size $O(n)$. Then we can compute* firstdata *in $O(\log(n))$ steps using $O(n)$ processors.*

**Proof:** We can sort in $O(\log(n))$ steps using $O(n)$ processors, see e.g. [9]. Each comparison between two positions for determining the order between their first-trees means that we compute a lowest common ancestor. By Lemma 19 this can be done in constant time (after preprocessing). □

Since each set follow$(x)$ is the union of first-sets, we are interested in an efficient parallel test for the relationship first$(F) \subseteq$ follow$(x)$, with $x \in$ pos$(E)$ and $F$ a subexpression of $E$.

**Proposition 21.** *Let $F$ be a subexpression of $E$ with $x \in pos(E) \setminus pos(F)$. Let $P = parent(F, x)$ be the lowest common ancestor of $F, x$. Then we have $first(F) \subseteq follow(x)$ if and only if one of the following conditions is satisfied:*

- *$first(F) = \emptyset$*
- *$P = G \cdot H$, $x \in last(G)$ and $first(F) \subseteq first(H)$*
- *$S = firststar(P)$ is defined, $x \in last(S)$ and $first(F) \subseteq first(S)$.*

**Proof:** We show the necessary condition, only. Assume that $\emptyset \neq$ first$(F) \subseteq$ follow$(x)$. If $P = G \cdot H$ and $x \in$ last$(G)$ then first$(H) \subseteq$ follow$(x)$ follows by definition. Moreover, by Lemma 7 and our assumption we have first$(F) \subseteq$ first$(H)$. So suppose that either $P = G \cdot H$ and $x \notin$ last$(G)$, or else $P = G + H$. If $S =$ firststar$(P)$ is undefined then clearly first$(F) \cap$ follow$(x) = \emptyset$ and we obtain a contradiction. Hence, $S$ exists and for each $y \in$ first$(F)$ we have $y \in$ follow$(x)$ if and only if $x \in$ last$(S)$ and $y \in$ first$(S)$. □

**Proposition 22.** *Let $t_E$ be a syntax tree for $E$ and let $t$ be a subtree with root $F$. Assume that $t = t_1 \dot\cup t_2$. Then given* firstdata*, $dec(x_1, t_1)$ and $dec(x_2, t_2)$ for all $x_1 \in pos(t_1), x_2 \in pos(t_2)$ we can compute $dec(x, t)$ for all $x \in pos(t)$ in time $O(1)$ using $O(|t|)$ processors.*

**Proof:** Recall that $C_1 =$ follow$_{t_2}(x)$ for some $x \in$ last$(F_1)$. Using $O(|t_2|)$ processors we can compute in $O(1)$ steps for all positions $y \in$ pos$(t_2)$ the node parent$(x, y)$. Then we use Proposition 21 and firstdata, lastdata in order to determine in constant time whether $y \in$ follow$(x)$. Symmetrically, for $x \in$ pos$(t_2)$ we can determine whether follow$_{t_1}(x) = \emptyset$ in $O(1)$ steps using $O(|t_1|)$ processors. □

The parallel algorithm below computes a CFS system for a subtree $t$ of $t_E$:

**procedure** SetDecomposition ($t$ : tree) ;
  **if** $|$pos$(t)| = 1$ **then**
    $x :=$ position of $t$;
    **if** first$(x) \subseteq$ follow$(x)$ **then** $C_0 := \{x\}$ **else** $C_0 := \emptyset$;
    dec$(x) :=$ dec$(x) \cup \{C_0\}$;
  **else**
    decompose $t = t_1 \dot\cup t_2$; (* 1/3–2/3 *)
    $F_1 :=$ root node of $t_1$;
    let $x_1 \in$ last$(F_1)$;

$C_1 := \{x_2 \in \text{pos}(t_2) \mid \text{first}(x_2) \subseteq \text{follow}(x_1)\}$;
$C_2 := \text{first}(F_1) \cap \text{pos}(t_1)$;
**for all** $x_1 \in \text{pos}(t_1)$ **pardo**
  **if** $x_1 \in \text{last}(F_1)$ and $C_1 \neq \emptyset$ **then**
    $\text{dec}(x_1) := \text{dec}(x_1) \cup \{C_1\}$;
**for all** $x_2 \in \text{pos}(t_2)$ **pardo**
  **if** $\text{first}(F_1) \subseteq \text{follow}(x_2)$ and $C_2 \neq \emptyset$ **then**
    $\text{dec}(x_2) := \text{dec}(x_2) \cup \{C_2\}$;
**pardo**
  SetDecomposition($t_1$);
  SetDecomposition($t_2$);
**end pardo**
**end if**

By Lemma 20 and Propositions 21, 22 we obtain:

**Theorem 23.** *Given a regular expression $E$ and a syntax tree $t_E$ for $E$ of size $O(|E|) = O(n)$. We can compute a CFS system $S$ for $E$ on a CREW PRAM in time $O(\log(n))$ using $O(n)$ processors. Therefore, we can compute an $\epsilon$-free NFA $\mathcal{A}_S$ for $E$ of size $|\mathcal{A}_S|$ in time $O(\log(n))$ using $O(n + |\mathcal{A}_S|/\log(n))$ processors (resp., $O(n\log(n))$ processors in the worst case).*

## 7 The Lower Bound – A Simple Proof

The current lower bound $\Omega(n \log n)$ on the number of transitions of $\epsilon$-free NFA build from regular expressions of size $n$ was shown in [8] on the following example:

$$E_n = (a_1 + \epsilon)(a_2 + \epsilon) \cdots (a_n + \epsilon)\,.$$

The aim of this section is to give an simpler proof of the lower bound on the example above.

**Lemma 24.** *Let $\mathcal{A} = (Q, \{a_1, \ldots, a_n\}, \delta, I, F)$ be an $\epsilon$-free NFA with minimal number of transitions such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(E_n)$. Then we can assume that*

1. *$Q = \{1, 2, \ldots, n+1\} = I = F$.*
2. *For all $1 \leq i \leq n$ we have $i = \min\{j \mid (i, a_j, k) \in \delta, k \in Q\}$. Moreover, the state $n+1$ has no outgoing transitions.*

**Proof:** Let $\mathcal{A} = (Q, \{a_1, \ldots, a_n\}, \delta, I, F)$ be an $\epsilon$-free NFA recognizing $\mathcal{L}(E_n)$ and consider two states $p, p' \in Q$ such that $\min\{i \mid (p, a_i, q) \in \delta, q \in Q\} = \min\{i \mid (p', a_i, q') \in \delta, q' \in Q\}$. It is obvious that we can merge the states $p, p'$ into one state, without changing neither the language accepted nor the number of transitions. Together with $a_1 \cdots a_n \in \mathcal{L}(E_n)$ we obtain both claims. □

**Proposition 25.** *Any $\epsilon$-free NFA $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(E_n)$ has $\Omega(n \log n)$ transitions.*

**Proof:** Consider without loss of generality an $\epsilon$-free NFA $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(E_n)$, satisfying the conditions of Lemma 24. Thus, we let $Q = \{1, 2, \ldots, n+1\}$ denote the set of states of $\mathcal{A}$. Assume for convenience that $n$ is even and let $Q_1 = \{1, \ldots, n/2\}$, $Q_2 = Q \setminus Q_1$. By $\mathcal{A}_1$ we denote the restriction of $\mathcal{A}$ to the states from $Q_1$, where all transitions labeled by $a_j$, $j \geq n/2$, have been deleted. Analogously, $\mathcal{A}_2$ is the restriction of $\mathcal{A}$ to $Q_2$, where all transitions labeled by $a_j$, $j \leq n/2$, have been deleted.

Consider $E_1 = (a_1 + \epsilon) \cdots (a_{n/2-1} + \epsilon)$ and $E_2 = (a_{n/2+1} + \epsilon) \cdots (a_n + \epsilon)$. Note that every word $u \in \mathcal{L}(E_1) a_{n/2}$ is accepted by $\mathcal{A}$ on a path within $\mathcal{A}_1$, followed by a transition labeled by $a_{n/2}$ which leaves $Q_1$. Symmetrically, every word $u \in a_{n/2}\mathcal{L}(E_2)$ is accepted by $\mathcal{A}$ on a path within $\mathcal{A}_2$, preceded by a transition labeled by $a_{n/2}$ which enters $Q_2$. Thus, we have $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(E_1)$ and $\mathcal{L}(\mathcal{A}_2) = \mathcal{L}(E_2)$.

We show finally that there are $\Omega(n/2)$ many transitions $(i, a_j, k) \in \delta$, with $i \in Q_1$ and $k \in Q_2$. Consider words of the form $a_\ell a_m \in \mathcal{L}(\mathcal{A})$ with $\ell \leq n/2 \leq m$. For each path labeled by $a_\ell a_m$ one of the transitions must go from $Q_1$ to $Q_2$. Suppose by symmetry that some $\ell \leq n/2$ exists such that there is no transition labeled by $a_\ell$ from $Q_1$ into $Q_2$. By the previous remark there exist transitions labeled by $a_m$ for every $m > n/2$, hence the claim. $\square$

*Remark 26.* It remains an open problem whether a minimal $\epsilon$-free NFA for $E_n$ has $o(n \log^2(n))$ transitions or not. Moreover, we don't have any example for the lower bound that uses an alphabet of constant size.

**Acknowledgment:** We wish to thank Volker Diekert for many contributions and especially for initiating this work. We are also indebted to the referees of *RAIRO*, for their effort and valuable comments that led to an improved paper.

## References

1. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.
2. J. Berstel and J.-É. Pin. Local languages and the Berry-Sethi algorithm. *Theoretical Computer Science*, 155:439–446, 1996.
3. A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:197–213, 1993.
4. A. Ehrenfeucht and P. Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12:134–146, 1976.
5. V. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16:1–53, 1961.
6. A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1989.
7. C. Hagenah and A. Muscholl. Computing $\epsilon$-free NFA from regular expressions in $O(n \log^2(n))$ time. In L. Brim, editor, *Proceedings of the 23rd Symposium on Mathematical Foundations of Computer Science (MFCS'98, Brno, Czech Rep.)*, number 1450 in Lecture Notes in Computer Science, pages 277–285, Berlin-Heidelberg-New York, 1998. Springer.

8. J. Hromkovič, S. Seibert, and T. Wilke. Translating regular expressions into small $\epsilon$-free nondeterministic finite automata. In R. Reischuk, editor, *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97, Lübeck, Germany)*, number 1200 in Lecture Notes in Computer Science, pages 55–66, Berlin-Heidelberg-New York, 1997. Springer.
9. J. JáJá. *An introduction to parallel algorithms.* Addison-Wesley, Reading, MA, 1992.
10. R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on electronic computers*, EC-9(1):39–47, 1960.
11. J.-L. Ponty, D. Ziadi, and J.-M. Champarnaud. A new quadratic algorithm to convert a regular expression into an automaton. In D. Raymond et al., editor, Automata implementation: First International Workshop on Implementing Automata, WIA'96, number 1260 in Lecture Notes in Computer Science, pages 109–119, Berlin-Heidelberg-New York, 1997. Springer.
12. D. Ziadi and J.-M. Champarnaud. An optimal parallel algorithm to convert a regular expression into its Glushkov automaton. *Theoretical Computer Science*, 215(1-2):69–87, 1999.
13. 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.