Two-Pass Greedy Regular Expression Parsing*

Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen,
and Ulrik Terp Rasmussen

Department of Computer Science, University of Copenhagen (DIKU)

Abstract. We present new algorithms for producing greedy parses for regular expressions (REs) in a semi-streaming fashion. Our lean-log algorithm executes in time $O(mn)$ for REs of size m and input strings of size n and outputs a compact bit-coded parse tree representation. It improves on previous algorithms by: operating in only 2 passes; using only $O(m)$ words of random-access memory (independent of n); requiring only kn bits of sequentially written and read log storage, where $k < \frac{1}{3}m$ is the number of alternatives and Kleene stars in the RE; processing the input string as a symbol stream and not requiring it to be stored at all. Previous RE parsing algorithms do not scale linearly with input size, or require substantially more log storage and employ 3 passes where the first consists of reversing the input, or do not or are not known to produce a greedy parse. The performance of our unoptimized C-based prototype indicates that our lean-log algorithm has also in practice superior performance and is surprisingly competitive with RE tools not performing full parsing, such as Grep.

1 Introduction

Regular expression (RE) parsing is the problem of producing a parse tree for an input string under a given RE. In contrast to most regular-expression based tools for programming such as Grep, RE2 and Perl, RE parsing returns not only whether the input is accepted, where a substring matching the RE and/or sub-REs are matched, but a full parse tree. In particular, for Kleene stars it returns *a list* of all matches, where each match again can contain such lists depending on the star depth of the RE.

An RE parser can be built using Perl-style backtracking or general context-free parsing techniques. What the backtracking parser produces is the *greedy* parse amongst potentially many parses. General context-free parsing and backtracking parsing are not scalable since they have cubic, respectively exponential worst-case running times. REs can be and often are grammatically ambiguous and can require arbitrary much look-ahead, making limited look-ahead context-free parsing techniques inapplicable. Kearns [1] describes the first linear-time algorithm for RE parsing. In a streaming context it consists of 3 passes: reverse

* This work has been partially supported by The Danish Council for Independent Research under Project 11-106278, "Kleene Meets Church: Regular Expressions and Types". The order of authors is insignificant.

S. Konstantinidis (Ed.): CIAA 2013, LNCS 7982, pp. 60–71, 2013.

the input, perform backward NFA-simulation, and construct parse tree. Frisch and Cardelli [2] formalize greedy parsing and use the same strategy to produce a greedy parse. Dubé and Feeley [3] and Nielsen and Henglein [4] produce parse trees in linear time for fixed RE, the former producing internal data structures and their serialized forms, the latter parse trees in bit-coded form; neither produces a greedy parse.

In this paper we make the following contributions:

1. Specification and construction of symmetric nondeterministic finite automata (NFA) with maximum in- and out-degree 2, whose paths from initial to final state are in one-to-one correspondence with the parse trees of the underlying RE; in particular, the greedy parse for a string corresponds to the lexicographically least path accepting the string.
2. NFA simulation with *ordered state sets*, which gives rise to a 2-pass greedy parse algorithm using $\lceil m \lg m \rceil$ bits per input symbol and the original input string, with m the size of the underlying RE. No input reversal is required.
3. NFA simulation optimized to require only $k \leq \lceil 1/3m \rceil$ bits per input symbol, where the input string need not be stored at all and the 2nd pass is simplified. Remarkably, this *lean-log algorithm* requires fewest log bits, and neither state set nor even the input string need to be stored.
4. An empirical evaluation, which indicates that our prototype implementation of the optimized 2-pass algorithm outperforms also in practice previous RE parsing tools and is sometimes even competitive with RE tools performing limited forms of RE matching.

In the remainder, we introduce REs as types to represent parse trees, define greedy parses and their bit-coding, introduce NFAs with bit-labeled transitions, describe NFA simulation with ordered sets for greedy parsing and finally the optimized algorithm, which only logs join state bits. We conclude with an empirical evaluation of a straightforward prototype to gauge the competitiveness of full greedy parsing with regular-expression based tools yielding less information for Kleene-stars.

2 Symmetric NFA Representation of Parse Trees

REs are finite terms of the form $0, 1, a, E_1 \times E_2, E_1 + E_2$ or E_1^*, where E_1, E_2 are REs.

Proviso. For simplicity and brevity we henceforth assume REs that do not contain sub-REs of the form E^*, where E is nullable (can generate the empty string). All results reported here can be and have been extended to such problematic REs in the style of Frisch and Cardelli [2]. In particular, our implementation BitC handles problematic REs.

REs can be interpreted as types built from singleton, product, sum, and list type constructors [2,5]; see Figure 1. Its structured values $\mathcal{T}[\![E]\!]$ represent the *parse trees* for E such that the regular language $\mathcal{L}[\![E]\!]$ coincides with the strings

$$
\begin{aligned}
\mathcal{T}[\![0]\!] &= \emptyset \\
\mathcal{T}[\![1]\!] &= \{()\}, \\
\mathcal{T}[\![a]\!] &= \{\mathsf{a}\}, \\
\mathcal{T}[\![E_1 \times E_2]\!] &= \{(V_1, V_2) \mid V_1 \in \mathcal{T}[\![E_1]\!], V_2 \in \mathcal{T}[\![E_2]\!]\}, \\
\mathcal{T}[\![E_1 + E_2]\!] &= \{\mathsf{inl}\ V_1 \mid V_1 \in \mathcal{T}[\![E_1]\!]\} \\
&\quad \cup \{\mathsf{inr}\ V_2 \mid V_2 \in \mathcal{T}[\![E_2]\!]\}, \\
\mathcal{T}[\![E_0^\star]\!] &= \{[V_1, \ldots, V_n] \mid n \geq 0 \wedge \\
&\qquad \forall 1 \leq i \leq n. V_i \in \mathcal{T}[\![E_0]\!]\}
\end{aligned}
$$

(a) Regular expressions as types.

$$
\begin{aligned}
\mathsf{flat}(()) &= \epsilon \\
\mathsf{flat}(a) &= \mathsf{a} \\
\mathsf{flat}((V_1, V_2)) &= \mathsf{flat}(V_1)\mathsf{flat}(V_2) \\
\mathsf{flat}(\mathsf{inl}\ V_1) &= \mathsf{flat}(V_1) \\
\mathsf{flat}(\mathsf{inr}\ V_2) &= \mathsf{flat}(V_2) \\
\mathsf{flat}([V_1, \ldots, V_n]) &= \mathsf{flat}(V_1) \ldots \mathsf{flat}(V_n)
\end{aligned}
$$

(b) Tree flattening.

$$
\begin{aligned}
\mathsf{code}(()) &= \epsilon \\
\mathsf{code}((V_1, V_2)) &= \mathsf{code}(V_1)\,\mathsf{code}(V_2) \\
\mathsf{code}(\mathsf{inl}\ V_1) &= 0\ \mathsf{code}(V_1)
\end{aligned}
\qquad\qquad
\begin{aligned}
\mathsf{code}(a) &= \epsilon \\
\mathsf{code}([V_1, \ldots, V_n]) &= 0\,\mathsf{code}(V_1) \ldots 0\,\mathsf{code}(V_n)\,1 \\
\mathsf{code}(\mathsf{inr}\ V_2) &= 1\ \mathsf{code}(V_2)
\end{aligned}
$$

(c) Bit-coding.

Fig. 1. The type interpretation of regular expressions and bit-coding of parses

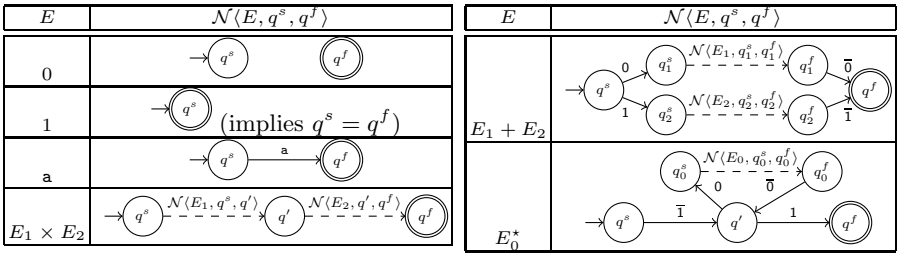


Fig. 2. aNFA construction schema

obtained by flattening the parse trees:

$$\mathcal{L}[\![E]\!] = \{\mathsf{flat}(V) \mid V \in \mathcal{T}[\![E]\!]\}.$$

We recall bit-coding from Nielsen and Henglein [4]. The bit code $\mathsf{code}(V)$ of a parse tree $V \in \mathcal{T}[\![E]\!]$ is a sequence of bits uniquely identifying V within $\mathcal{T}[\![E]\!]$; that is, there exists a function decode_E such that $\mathsf{decode}_E(\mathsf{code}(V)) = V$. See Figure 1 for the definition of code; the definition of decode_E is omitted for brevity, but is straightforward. We write $\mathcal{B}[\![\ldots]\!]$ instead of $\mathcal{T}[\![\ldots]\!]$ whenever we want to refer to the bit codings, rather than the parse trees. We use subscripts to discriminate parses with a specific flattening: $\mathcal{T}_s[\![E]\!] = \{V \in \mathcal{T}[\![E]\!] \mid \mathsf{flat}(V) = s\}$. We extend the notation $\mathcal{B}_s[\![\ldots]\!]$ similarly.

Note that a bit string by itself does not carry enough information to deduce which parse tree it represents. Indeed this is what makes bit strings a compact representation of strings where the underlying RE is statically known.

The set $\mathcal{B}[\![E]\!]$ for an RE E can be compactly represented by *augmented nondeterministic finite automaton (aNFA)*, a variant of enhanced NFAs [4] that has in- and outdegree at most 2 and carries a label on each transition.

Definition 1 (Augmented NFA) An *augmented NFA* (aNFA) is a 5-tuple $M = (Q, \Sigma, \Delta, q^s, q^f)$ where Q is the set of states, Σ is the input alphabet, and q^s, q^f are the starting states. The transition relation $\Delta \subseteq Q \times (\Sigma \cup \{\mathsf{0}, \mathsf{1}, \overline{\mathsf{0}}, \overline{\mathsf{1}}\}) \times Q$ contains directed, labeled transitions: $(q, \gamma, q') \in \Delta$ is a transition from q to q' with label γ, written $q \xrightarrow{\gamma} q'$.

We call transition labels in Σ *input labels*; labels in $\{\mathsf{0}, \mathsf{1}\}$ *output labels*; and labels in $\{\overline{\mathsf{0}}, \overline{\mathsf{1}}\}$ *log labels*.

We write $q \overset{p}{\rightsquigarrow} q'$ if there is a path labeled p from q to q'. The sequences $\mathsf{read}(p)$, $\mathsf{write}(p)$ and $\mathsf{log}(p)$ are the subsequences of input labels, output labels, and log labels of p, respectively.

We write: J_M for the *join states* $\{q \in Q \mid \exists q_1, q_2.\ (q_1, \overline{\mathsf{0}}, q), (q_2, \overline{\mathsf{1}}, q) \in \Delta\}$; S_M for the *symbol sources* $\{q \in Q \mid \exists q' \in Q, a \in \Sigma.\ (q, a, q')\}$; and C_M for the *choice states* $\{q \in Q \mid \exists q_1, q_2.\ (q, \mathsf{0}, q_1), (q, \mathsf{1}, q_2) \in \Delta\}$.

If M is an aNFA, then $\overline{M}$ is the aNFA obtained by *flipping* all transitions and exchanging the start and finishing states, that is reverse all transitions and interchange output labels with the corresponding log labels. □

Our algorithm for constructing an aNFA from an RE is a standard Thompson-style NFA generation algorithm modified to accomodate output and log labels:

Definition 2 (aNFA construction) We write $M = \mathcal{N}\langle E, q^s, q^f\rangle$ when M is an aNFA constructed according to the rules in Figure 2.

Augmented NFAs are dual under reversal; that is, flipping produces the augmented NFA for the reverse of the regular language.

Proposition 2.1. *Let $\overline{E}$ be canonically constructed from E to denote the reverse of $\mathcal{L}[\![E]\!]$. Let $M = \mathcal{N}\langle E, q^s, q^f\rangle$. Then $\overline{M} = \mathcal{N}\langle \overline{E}, q^f, q^s\rangle$.*

This is useful since we will be running aNFAs in both forward and backward (reverse) directions.

Well-formed aNFAs—and Thompson-style NFAs in general—are canonical representations of REs in the sense that they not only represent their language interpretation, but their type interpretation:

Theorem 2.1 (Representation). *Given an aNFA $M = \mathcal{N}\langle E, q^s, q^f\rangle$, M outputs the bit-codings of E:*

$$\mathcal{B}_s[\![E]\!] = \{\mathsf{write}(p) \mid q^s \overset{p}{\rightsquigarrow} q^f \wedge \mathsf{read}(p) = s\}.$$

3 Greedy Parsing

The *greedy parse* of a string s under an RE E is what a backtracking parser returns that tries the left operand of an alternative first and backtracks to try the right alternative only if the left alternative does not yield a successful parse. The name comes from treating the Kleene star $E^\star$ as $E \times E^\star + 1$, which "greedily" matches E against the input as many times as possible. A "lazy" matching

interpretation of $E^\star$ corresponds to treating $E^\star$ as $1 + E \times E^\star$. (In practice, multiple Kleene-star operators are allowed to make both interpretations available; e.g. $E\star$ and $E\star\star$ in PCRE.)

Greedy parsing can be formalized by an order $\lessdot$ on parse trees, where $V_1 \lessdot V_2$ means that V_1 is "more greedy" than V_2. The following is adapted from Frisch and Cardelli [2].

Definition 3 (Greedy order) The binary relation $\lessdot$ is defined inductively on the structure of values as follows:

$$\begin{array}{rll}
(V_1, V_2) \lessdot (V_1', V_2') & \text{if} & V_1 \lessdot V_1' \vee (V_1 = V_1' \wedge V_2 \lessdot V_2') \\
\mathsf{inl}\ V_0 \lessdot \mathsf{inl}\ V_0' & \text{if} & V_0 \lessdot V_0' \\
\mathsf{inr}\ V_0 \lessdot \mathsf{inr}\ V_0' & \text{if} & V_0 \lessdot V_0' \\
\mathsf{inl}\ V_0 \lessdot \mathsf{inr}\ V_0' & & \\
[] \lessdot [V_1, \ldots] & & \\
[V_1, \ldots] \lessdot [V_1', \ldots] & \text{if} & V_1 \lessdot V_1' \\
[V_1, V_2, \ldots] \lessdot [V_1, V_2', \ldots] & \text{if} & [V_2, \ldots] \lessdot [V_2', \ldots]
\end{array}$$

The relation $\lessdot$ is not a total order; consider for example the incomparable elements $(\mathsf{a}, \mathsf{inl}\ ())$ and $(\mathsf{b}, \mathsf{inr}\ ())$. The parse trees of any particular RE are totally ordered, however:

Proposition 3.1. *For each E, the order $\lessdot$ is a strict total order on $\mathcal{T}[\![E]\!]$.*

In the following, we will show that there is a correspondence between the structural order on values and the lexicographic order on their bit codings.

Definition 4 For bit sequences $d, d' \in \{0,1\}^\star$ we write $d \prec d'$ if d is lexicographically strictly less than d'; that is, $\prec$ is the least relation satisfying

1. $\epsilon \prec d$ if $d \neq \epsilon$
2. $b\ d \prec b'\ d'$ if $b < b'$ or $b = b'$ and $d \prec d'$.

Theorem 3.1. *For all REs E and values $V, V' \in \mathcal{T}[\![E]\!]$ we have $V \lessdot V'$ iff $\mathsf{code}(V) \prec \mathsf{code}(V')$.*

Corollary 3.1. *For any RE E with aNFA $M = \mathcal{N}\langle E, q^s, q^f\rangle$, and for any string s, $\min_\lessdot \mathcal{T}_s[\![E]\!]$ exists and*

$$\min_{\lessdot} \mathcal{T}_s[\![E]\!] = \mathsf{decode}_E(\min_{\prec}\{\mathsf{write}(p) \mid q^s \overset{p}{\rightsquigarrow} q^f \wedge \mathsf{read}(p) = s\}).$$

Proof. Follows from Theorems 2.1 and 3.1. □

We can now characterize greedy RE parsing as follows: Given an RE E and string s, find bit sequence b such that there exists a path p from start to finishing state in the aNFA for E such that:

1. $\mathsf{read}(p) = s$,
2. $\mathsf{write}(p) = b$,
3. b is lexicographically least among all paths satisfying 1 and 2.

This is easily done by a backtracking algorithm that tries 0-labeled transitions before 1-labeled ones. It is atrociously slow in the worst case, however: exponential time. How to do it faster?

4 NFA-Simulation with Ordered State Sets

Our first algorithm is basically an NFA-simulation. For reasons of space we only sketch its key idea, which is the basis for the more efficient algorithm in the following section.

A standard NFA-simulation consists of computing $\mathsf{Reach}^*(S, s)$ where

$$\begin{aligned}
\mathsf{Reach}^*(S, \epsilon) &= S \\
\mathsf{Reach}^*(S, a\,s') &= \mathsf{Reach}^*(\mathsf{Reach}(S, a), s') \\
\mathsf{Reach}(S, a) &= \mathsf{Close}(\mathsf{Step}(S, a)) \\
\mathsf{Step}(S, a) &= \{q' \mid q \in S, q \xrightarrow{a} q'\} \\
\mathsf{Close}(S') &= \{q'' \mid q' \in S', q' \overset{p}{\rightsquigarrow} q'', \mathsf{write}(p) = \epsilon\}
\end{aligned}$$

Checking $q^f \in \mathsf{Reach}^*(\{q^s\}, s)$ determines whether s is accepted or not. But how to construct an accepting *path* and in particular the one corresponding to the greedy parse?

We can *log* the set of states reached after each symbol during the NFA-simulation. After forward NFA-simulation, let S_i be the NFA-states reached after processing the first i symbols of input $s = a_1 \ldots a_n$. Given a list of logged state sets, the input string s and the final state q^f, the nondeterministic algorithm Path^* constructs a path from q^s to q^f through the state sets:

$$\begin{aligned}
\mathsf{Path}(S_i, q) &= (q', p) \text{ where } q' \in S_i, q' \overset{p}{\rightsquigarrow} q, \mathsf{read}(p) = a_i \\
\mathsf{Path}^*(S_0, q) &= p' \cdot p \text{ where } (q', p) = \mathsf{Path}(S_0, q), q^s \overset{p'}{\rightsquigarrow} q', \mathsf{read}(p') = \epsilon \\
\mathsf{Path}^*(S_i, q) &= p' \cdot p \text{ where } (q', p) = \mathsf{Path}(S_i, q), p' = \mathsf{Path}^*(S_{i-1}, q')
\end{aligned}$$

Calling $\mathsf{write}(\mathsf{Path}^*(S_n, q^f))$ gives a bit-coded parse tree, though not necessarily the lexicographically least.

We can adapt the NFA-simulation by keeping each state set S_i in a particular order: If $\mathsf{Reach}^*(\{q^s\}, a_1 \ldots a_i) = \{q_{i1}, \ldots q_{ij_i}\}$ then order the q_{ij} according to the lexicographic order of the paths reaching them. Intuitively, the highest ranked state in S_i is on the greedy path if the remaining input is accepted from this state; if not, the second-highest ranked is on the greedy path, if the remaining input is accepted; and so on.

The NFA-simulation can be refined to construct properly ordered state sequences instead of sets without asymptotic slow-down. The log, however, is adversely affected by this. We need $\lceil m \lg m \rceil$ bits per input symbol, for a total of $\lceil mn \lg m \rceil$ bits.

The key property for allowing us to list a state at most once in an order state squence is this:

Lemma 4.1. *Let s, t_1, t_2, and t be states in an aNFA M, and let p_1, p_2, q_1, q_2 be paths in M such that $s \overset{p_1}{\rightsquigarrow} t_1$, $s \overset{p_2}{\rightsquigarrow} t_2$, and $t_1 \overset{q_1}{\rightsquigarrow} t$, $t_2 \overset{q_2}{\rightsquigarrow} t$. If $\mathsf{write}(p_1) \prec \mathsf{write}(p_2)$ then $\mathsf{write}(p_1 q_1) \prec \mathsf{write}(p_2 q_2)$*

Proof. Application of the lexicographical ordering on paths. □

5 Lean-Log Algorithm

After the ordered forward NFA-simulation with logging, the algorithm Path above can be refined to always yield the greedy parse whend traversing the aNFA in backwards direction. Since the join states J_M of an aNFA M become the choice states $C_{\overline{M}}$ of the reverse aNFA $\overline{M}$ we only need to construct one "direction" bit for each join state at each input string position. It is not necessary to record any states in the log at all, and we do not even have to store the input string. This results in an algorithm that requires only k bits per input symbol for the log, where k is the number of Kleene-stars and alternatives occurring in the RE. It can be shown that $k \leq \frac{1}{3}m$; in practice we can observe $k \ll m$.

Our optimized algorithm is described in Figure 3 below. The forward pass keeps the aNFA and the current character in memory, requiring a $O(m)$ words of random access memory, writing nk bits to the log, and discarding the input string. Finally, the backward pass also requires $O(m)$ words of random access memory and reads from the log in reverse write order. The log is thus a 2-phase stack: In the first pass it is only pushed to, in the second pass popped from.

Both LClose and LStep run in time $O(m)$ per input symbol, hence the forward pass requires time $O(mn)$. Likewise, the backward pass requires time $O(mn)$.

LClose keeps track of visited states and returns the states reached ordered lexicographically according to the paths reaching them. Hence, the following theorem can be proved:

Theorem 5.1. *For any regular expression E and symbol sequence s, if $\mathcal{L}_l = \mathsf{LSim}(s)$, and $d = \mathsf{LTrace}(\mathcal{L}_l, q^f)$, then $\mathsf{decode}_E(d) = \min_{\lessdot} \mathcal{T}_s[\![E]\!]$.*

6 Evaluation

We have implemented the optimized algorithms in C and in Haskell, and we compare the performance of the C implementation with the following existing RE tools:

RE2: Google's RE implementation, available from [6].
Tcl: The scripting language Tcl [7].
Perl: The scripting language Perl [8].
Grep: The UNIX tool `grep`.
Rcp: The implementation of the algorithm "*DFASIM*" from [4]. It is based on Dubé and Feeley's method [3], but altered to produce a bit-coded parse tree.
FrCa: The implementation of the algorithm"*FrCa*" algorithm used in [4]. It is based on Frisch and Cardelli's method from [2].

In the subsequent plots, our implementation of the lean-log algorithm is referred to as *BitC*.

The tests have been performed on an Intel Xeon 2.5 GHz machine running GNU/Linux 2.6.

$$(Q, L) \oplus (Q', L') = (Q \cdot Q', L \cup L')$$

$$\mathsf{LClose}(q, L) = \begin{cases} ([q], L) & q \xrightarrow{a} q', a \in \Sigma \\ \mathsf{LClose}(q_0, L) \oplus \mathsf{LClose}(q_1, L) & q \xrightarrow{0} q_0, q \xrightarrow{1} q_1 \\ \mathsf{LClose}(q', L \cup \{q' \mapsto t\}) & q \xrightarrow{t} q', t \in \{\overline{0}, \overline{1}\}, q' \notin \mathrm{dom}(L) \\ ([], L) & \text{otherwise} \end{cases}$$

$$\mathsf{LStep}([], a, (Q, L)) = (Q, L)$$

$$\mathsf{LStep}(q \cdot qs, a, (Q, L)) = \begin{cases} \mathsf{LStep}(qs, a, (Q, L) \oplus \mathsf{LClose}(q, L)) & q \xrightarrow{a} q' \\ \mathsf{LStep}(qs, a, (Q, L)) & \text{otherwise} \end{cases}$$

$$\mathsf{LSim}'([], Q, \mathcal{L}) = \begin{cases} \mathcal{L} & \text{if } q^s \in Q \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{LSim}'(a \cdot s', Q, \mathcal{L}) = \begin{cases} \mathsf{LSim}(s', Q', L \cdot \mathcal{L}) & \text{if } (Q', L) = \mathsf{LStep}(Q, a, ([], \emptyset)), Q' \neq [] \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{LSim}(s) = \textbf{let } (Q_0, L_0) = \mathsf{LClose}(q^s, []) \textbf{ in } \mathsf{LSim}'(s, Q_0, [L])$$

(a) Forward pass.

$$\mathsf{LTrace}(L \cdot \mathcal{L}, q) = \begin{cases} [] & \text{if } q = q^s \\ \mathsf{LTrace}(L \cdot \mathcal{L}, q') \cdot \gamma & \text{if } q \xrightarrow{\gamma} q', \gamma \in \{0, 1\} \\ \mathsf{LTrace}(\mathcal{L}, q') & \text{if } q \xrightarrow{\gamma} q', \gamma \in \Sigma \\ \mathsf{LTrace}(L \cdot \mathcal{L}, q') & \text{if } q \xrightarrow{L[q]} q', L[q] \in \{\overline{0}, \overline{1}\} \end{cases}$$

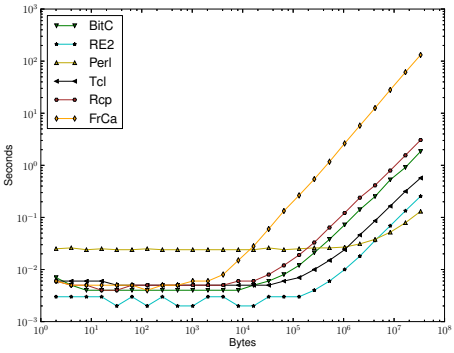
(b) Backward pass.

Fig. 3. Forward and backward pass algorithm

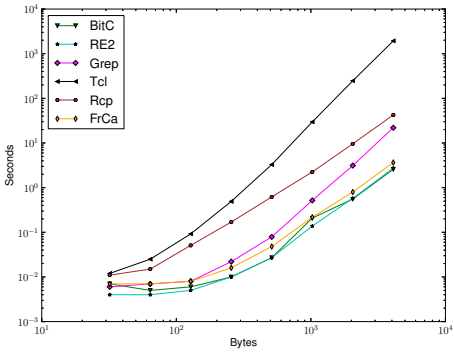
6.1 Pathological Expressions

To get an indication of the "raw" throughput for each tool, $\mathtt{a}^\star$ was run on sequences of as (Figure 4a). (Note that the plots use log scales on both axes, so as to accommodate the dramatically varying running times.) Perl outperforms the rest, likely due to a strategy where it falls back on a simple scan of the input instead. FrCa stores each position in the input string from which a match can be made, which in this case is every position. As a result, FrCa uses significantly more memory than the rest, causing a dramatic slowdown.

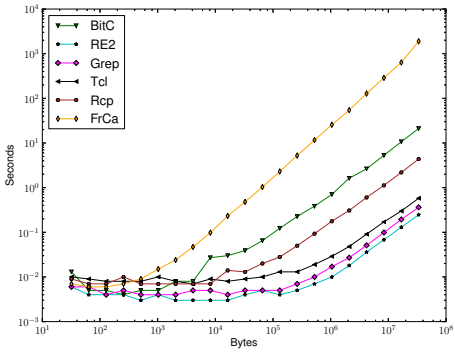
The expression $(\mathtt{a}|\mathtt{b})^\star\mathtt{a}(\mathtt{a}|\mathtt{b})^n$ with the input $(\mathtt{ab})^{n/2}$ is a worst-case for DFA-based methods, as it results in a number of states exponential in n. Perl has been omitted from the plots, as it was prohibitively slow. Tcl, Rcp, and Grep all perform orders of magnitude slower than FrCa, RE2, and BitC (Figure 4b), indicating that Tcl and Grep also use a DFA for this expression. If we fix n to 25, it becomes clear that FrCa is slower than the rest, likely due to high memory consumption as a result of its storing all positions in the input string (Figure 4c). The asymptotic running times of the others appear to be similar to each other, but with greatly varying constants.



(a) $\mathtt{a}^\star$, input $\mathtt{a}^n$.

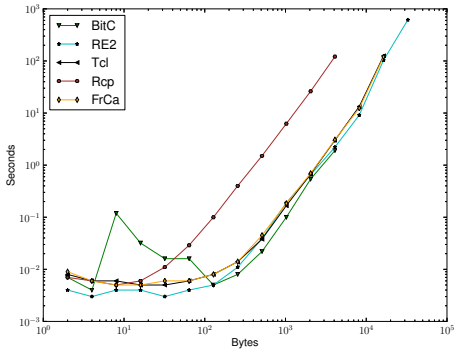


(b) $(\mathtt{a}|\mathtt{b})^\star\mathtt{a}(\mathtt{a}|\mathtt{b})^n$, input $(\mathtt{ab})^{n/2}$.

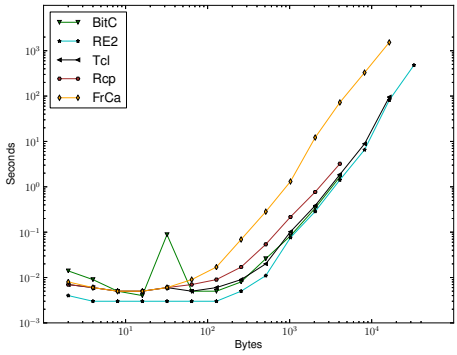


(c) $(\mathtt{a}|\mathtt{b})^\star\mathtt{a}(\mathtt{a}|\mathtt{b})^{25}$, input $(\mathtt{ab})^{n/2}$.

Fig. 4

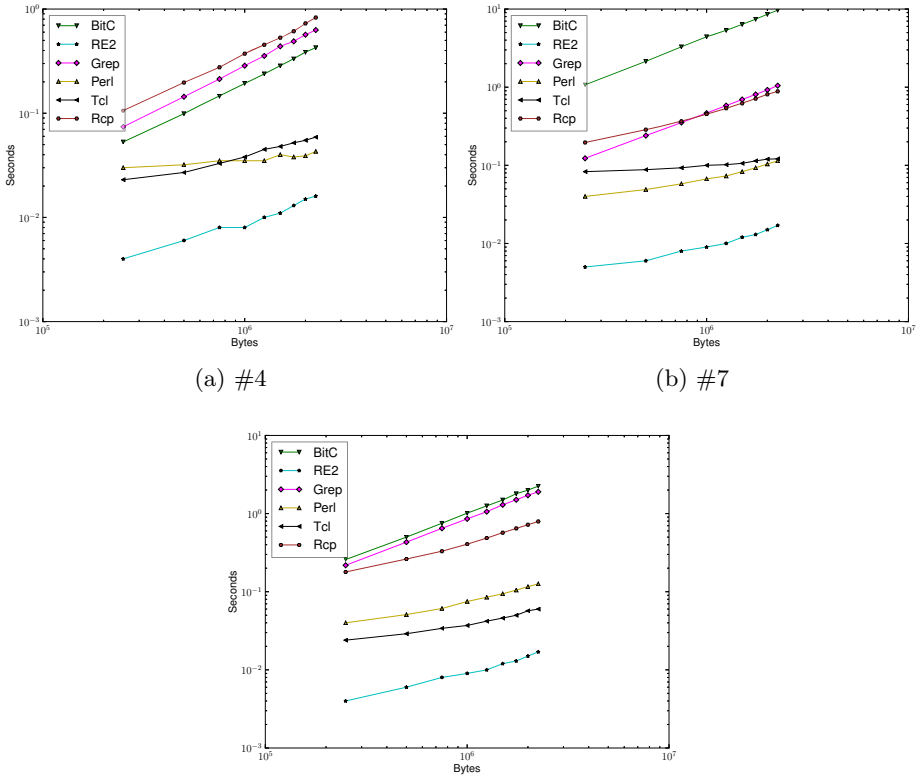


(a) $(\mathtt{a}?)^n\mathtt{a}^n$, input $\mathtt{a}^n$.



(b) $\mathtt{a}^n(\mathtt{a}?)^n$, input $\mathtt{a}^n$.

Fig. 5



(a) #4 (b) #7

(c) #8

Fig. 6. Comparison using various e-mail expressions. The numbers in the captions refer to the regexes in [9, Table 1].

For the backtracking worst-case expression $(\mathtt{a}?)^n\mathtt{a}^n$ in Figure 5a, BitC performs roughly like RE2.[1] In contrast to Rcp and FrCa, which are both highly sensitive to the *direction* of non-determinism, BitC has the same performance for both $(\mathtt{a}?)^n\mathtt{a}^n$ and $\mathtt{a}^n(\mathtt{a}?)^n$ (Figure 5b).

6.2 Practical Examples

We have run the comparisons with various "real-life" examples of REs taken from [9], all of which deal with expressions matching e-mail addresses. In Fig. 6b, BitC is significantly slower than in the other examples. This can likely be ascribed to heavy use of bounded repetitions in this expression, as they are currently just rewritten into concatenations and Kleene stars in our implementation.

[1] The expression parser in BitC failed for the largest expressions, which is why they are not on the plot.

In the other two cases, BitC's performance is roughly like that of Grep. This is promising for BitC since Grep performs only RE *matching*, not full *parsing*. RE2 is consistently ranked as the fastest program in our benchmarks, presumably due to its aggressive optimizations and ability to dynamically choose between several strategies. Recall that RE2 performs greedy leftmost subgroup matching, not full parsing. Our present prototype of BitC is coded in less than 1000 lines of C. It uses only standard libraries and performs no optimizations such as NFA-minimization, DFA-construction, cached or parallel NFA-simulation, etc. This is future work.

7 Related Work

The known RE parsing algorithms can be divided into four categories. The first category is Perl-style backtracking used in many tools and libraries for RE subgroup matching [10]; it has an exponential worst case running time, but always produces the greedy parse and enables some extensions to REs such as backreferences. Another category consists of context-free parsing methods, where the RE is first translated to a context-free grammar, before a general context-free parsing algorithm such as Earleys [11] using cubic time is applied. An interesting CFG method is derivatives-based parsing [12]. While efficient parsers exist for subsets of unambiguous context-free languages, this restriction propagates to REs, and thus these parsers can only be applied for subsets of unambiguous REs. The third category contains RE scalable parsing algorithms that do not always produce the greedy parse. This includes NFA and DFA based algorithms provided by Dubé and Feeley [3] and Nielsen and Henglein [4], where the RE is first converted to an NFA with additional information used to parse strings or to create a DFA preserving the additional information for parsing. This category also includes the algorithm by Fischer, Huch and Wilke [13]; it is left out of our tests since its Haskell-based implementation often was not competitive with the other tools. The last category consists of the algorithms that scale well and always produce greedy parse trees. Kearns [1] and Frisch and Cardelli [2] reverse the input; perform backwards NFA-simulation, building a log of NFA-states reached at each input position; and construct the greedy parse tree in a final forward pass over the input. They require storing the input symbol plus m bits per input symbol for the log. This can be optimized to storing bits proportional to the number of NFA-states reached at a given input position [4], although the worst case remains the same. Our lean log algorithm uses only 2 passes, does not require storing the input symbols and stores only $k < \frac{1}{3}m$ bits per input symbol in the string.

References

1. Kearns, S.M.: Extending Regular Expressions. PhD thesis, Columbia University (1990)
2. Frisch, A., Cardelli, L.: Greedy Regular Expression Matching. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 618–629. Springer, Heidelberg (2004)
3. Dubé, D., Feeley, M.: Efficiently Building a Parse Tree From a Regular Expression. Acta Informatica 37(2), 121–144 (2000)
4. Nielsen, L., Henglein, F.: Bit-coded Regular Expression Parsing. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 402–413. Springer, Heidelberg (2011)
5. Henglein, F., Nielsen, L.: Regular expression containment: Coinductive axiomatization and computational interpretation. In: Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL). SIGPLAN Notices, vol. 46, pp. 385–398. ACM Press (January 2011)
6. Cox, R.: RE2, `https://code.google.com/p/re2/`
7. Ousterhout, J.: Tcl: An Embeddable Command Language. In: Proc. USENIX Winter Conference, pp. 133–146 (January 1990)
8. Wall, L., Christiansen, T., Orwant, J.: Programming Perl. O'Reilly Media, Incorporated (2000)
9. Veanes, M.V.M., de Halleux, P., Tillmann, N.: Rex: Symbolic Regular Expression Explorer. In: Proc. 3d Int'l Conf. on Software Testing, Verification and Validation, Paris, France. IEEE Computer Society Press (April 6-10 2010)
10. Cox, R.: Regular Expression Matching can be Simple and Fast
11. Earley, J.: An Efficient Context-Free Parsing Algorithm. Communications of the ACM 13(2), 94–102 (1970)
12. Might, M., Darais, D., Spiewak, D.: Parsing with derivatives: a functional pearl. In: ACM SIGPLAN Notices, vol. 46, pp. 189–195. ACM (2011)
13. Fischer, S., Huch, F., Wilke, T.: A Play on Regular Expressions: Functional Pearl. In: Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, pp. 357–368. ACM, New York (2010)