A deterministic parsing algorithm for ambiguous regular expressions

Angelo Borsotti · Luca Breveglieri · Stefano Crespi Reghizzi · Angelo Morzenti

Keywords Berry-Sethi recognizer, regular expression deterministic parsing, ambiguous regular expression, regular expression parsing tool

Abstract We introduce a new parser generator, called Berry-Sethi Parser (BSP), for ambiguous regular expressions (RE). The generator constructs a deterministic finite-state trans-ducer that recognizes an input string, as the classical Berry-Sethi algorithm does, and ad-ditionally outputs a linear representation of all the syntax trees of the string; for infinitely ambiguous strings, a policy for selecting representative sets of trees is chosen. To construct the transducer, the RE symbols, including letters, parentheses and other metasymbols, are distinctly numbered, so that the corresponding language becomes locally testable. In this way a deterministic position automaton can be constructed, which recognizes and translates the input into a compact DAG representation of the syntax trees. The correctness of the construction is proved. The transducer operates in a linear time on the input. Its descriptive complexity is analyzed as a function of established RE parameters: the alphabetic width, the number of null string symbols and the height of the RE tree. A condition for checking RE ambiguity on the transducer graph is stated. Experimental results of running the parser gen-erator and the parser on a large RE collection are presented. The POSIX RE disambiguation criterion has also been applied to the parser.

1 Introduction

The popularity of regular expressions (REs) as a notation for specifying text patterns comes from their expressiveness and also from the availability of efficient algorithms for string recognition. Actually, the term “regular expression” has different meanings: the classical for-mal language notation introduced by S. Kleene (also known as rational expression), and var-ious technical notations that are supported by certain libraries, such as RE2, or are available within programming languages such as *Perl*. In this paper we exclusively refer to the clas-sical notation introduced by S. Kleene but, since most technical notations include Kleene’s REs as a core, our work may be of some value also for people interested in technical REs.

Most applications transform an RE into a finite automaton, deterministic (DFA) or not (NFA). Such an automaton simply checks that the input text is correct, i.e., it acts as a language *recognizer*. But this is insufficient for the applications that require also a syntax tree of the recognized text; in that case a *parser*, rather than a simple recognizer, is needed. Moreover, if the RE is ambiguous, a multiple matching of the same text is possible, each matching corresponding to a distinct syntax tree. In such cases, the ability to select one or a few syntax trees out of the many (even infinitely many) possible is sometimes placed as a further requirement on the parser. Our approach is to generate a representation of all

A. Borsotti
E-mail: angelo.borsotti@mail.polimi.it

L. Breveglieri
DEIB - Politecnico di Milano, E-mail: luca.breveglieri@polimi.it

S. Crespi Reghizzi
DEIB - Politecnico di Milano and CNR-IEIIT, E-mail: stefano.crespireghizzi@polimi.it

A. Morzenti
DEIB - Politecnico di Milano, E-mail: angelo.morzenti@polimi.it

the syntax trees for an ambiguous text, in such a way that a subsequent disambiguation, if required, is possible; for instance, we later mention the way to incorporate the standard POSIX disambiguation criterion into the parser. It should be obvious that such an approach is preferable to a parser that incorporates a fixed immutable disambiguation criterion.

To illustrate, malware detection in programs is a security application where RE parsing may be useful. In [1] a method is proposed, based on obtaining a few suspicious program execution traces (or executable files) and modeling them as strings over a (finite) alphabet $\Sigma = \{ b_k \mid \ 1 \leq k \leq n \}$ of $n \geq 1$ basic blocks (each block b_k also has some attributes such as an address, parameters, etc.). Such strings are collected in a library and are scanned for malicious basic block patterns P, which are described by REs; e.g., $P = (b_1 \ b_i^*)^* \ b_2$ with $i \neq 2$. Notice that if it holds $b_1 = b_i$ for some i, the pattern itself is described ambiguously. Clearly, this amounts to parse the library with the possibly ambiguous RE $\Sigma^* \ P \ \Sigma^*$. Furthermore, to assess the security risk, it is important to determine the occurrence place of the malicious pattern in the trace (or file); e.g., whether the pattern occurs in the header, trailer, etc. This amounts to model the trace (or file) structure, i.e., to compute the syntax tree(s) of the strings in the library. Software tools that support the description, detection and classification of malicious RE patterns would benefit from the inclusion of an efficient RE parser. Similar situations occur in querying semi-structured data bases, where the paths that connect a pair of objects in the data base are specified by typically ambiguous REs.

Some methods (discussed in Sect. 5) for obtaining parsers for ambiguous REs are known, which differ both in how they construct the pure recognizer, and in how they construct and represent the syntax trees. Here we present a new practical parser generator, which is based on rigorous concepts from the theory of finite automata and languages, so that its correctness can be formally proved. We call our parser / parser-generator *Berry-Sethi Parser* (BSP), as it adds parsing capabilities to the classical recognizer – known as *BS algorithm* – of the same name [3]. The latter belongs to the class of *position automata*, because its states are keyed to the positions of the input letters within the RE.

We construct the syntax trees of input strings by the positional approach (inspired by [20]), but now we include also the positions of the *metasymbols*, i.e., parentheses, null string symbols and operators (star and concatenation). All the trees of an ambiguous input string are compactly encoded in the parser output, which is abstractly represented by a directed acyclic graph (DAG) to avoid the duplication of common subtrees. Additionally, the REs that have an infinite ambiguity degree – also known as *problematic REs* – raise the issue of parsing termination, thus they need a criterion for stopping the syntax tree computation after producing a sufficient sample of trees. Problematic REs are rarely permitted by existing parsers, but they are safely handled by BSP.

The abstract model of our parser is a *deterministic finite-state transducer*: at each transition it reads an input character and outputs a finite string, which represents a finite slice of the syntactic DAG. Therefore, the transducer operates deterministically and in real-time. Moreover, given such a transducer, it is straightforward to check whether the RE is ambiguous, by inspecting the transducer graph, a useful feature.

The state and transition complexities of the transducer and of the underlying BS recognizer are identical; it is known that the size of the classic BS DFA is related to the number of input characters (alphabetic positions) occurring in the RE. To estimate the size of the transducer, we compute an upper bound on the size of the output function, thus obtaining a relation to some parameters of the RE, such as the height of the syntax tree of the RE and the number of null string symbols present in the RE.

We have implemented the BSP generator and parsing algorithm with attention to performance. Then we have measured the parser generation time, parser size and parsing speed, on inputs of increasing length, obtaining encouraging results for a large collection of REs.

To sum up, the main contributions of this paper are the following:

- a novel rigorous and efficient deterministic algorithm for parsing any ambiguous RE and for returning a representation of *all* the syntax trees
- a correctness proof of the algorithm and an analysis of the parser descriptive complexity
- the suitability to the applications that require a selection of the trees, since the tree representation permits filtering by a disambiguation criterion, e.g., the POSIX standard one
- a new ambiguity test for a RE
- an open implementation of the parser, coded in Java, which has been extensively experimented by using unbiased benchmarks and which compares favorably with a widespread RE library such as RE2

The paper is organized as follows. Sect. 2 lists the basic definitions and the representations used for REs and linearized syntax trees. It introduces the sets that characterize the *local* regular languages, and it ends by recalling the construction of the classic Berry-Sethi recognizer. Sect. 3 presents first the BSP parser generator algorithm, intuitively and formally, then the construction of the DAG and the linearized syntax trees. The analysis of the parser size, the correctness proof, and the statement of the RE ambiguity condition end the section. Sect. 4 describes our implementation, reports experimental measures, and sketches the BSP extension that incorporates the POSIX disambiguation criterion. Sect. 5 lists and concisely compares some existing RE parsers, and Sect. 6 concludes.

2 Basic concepts

For the basic notions needed about REs and finite automata / transducers, it suffices to list our terminology and notation.

2.1 Regular expressions and trees

The *terminal alphabet* is denoted by Σ. The *input alphabet* is the union of Σ with two special marks: the *start-of-text* "$\vdash$" and the *end-of-text* "$\dashv$". The *empty* (or *null*) string is denoted by ε. For a string $x \in \Sigma^*$, the length is $|\,x\,| \geq 0$, the j-th character is $x\,[j]$ with $1 \leq j \leq |\,x\,|$, and $|\,x\,|_a$ is the number of occurrences of character $a \in \Sigma$ in the string x.

An RE is a formula over the alphabet $\Sigma \cup M$, where M is the set (disjoint from Σ) of *metasymbols* and is listed in Tab. 1. Notice that in an RE the empty string is denoted by 1 instead of ε. As usual, union and concatenation are associative operations, and the operator priority is in descending order: iteration, concatenation and union. We assume that the argument of an iteration operator is always parenthesized.

Example 1 (running example) The following two RE are equivalent because of the well known identity $(\,e\,)^* = (\,e\,)^+ \mid 1$, where e is any RE:

$$\Big((\,a\,)^+ \mid b\,a \mid a\,b\,a \Big)^*\, b \tag{1}$$

Table 1 Set M of the metasymbols that may occur in an RE.

metasymbol	*meaning*
1	empty string ε
$\mid$	union operator
$\cdot$	concatenation operator – optional
$*$ $+$	iteration operators – Kleene star and cross
$($ $)$	delimiters of (non-empty) subexpressions – square brackets are also used for better readability

$$\Big(\big((\,a\,)^+ \mid b\,a \mid a\,b\,a \big)^+ \mid 1 \Big)\; b \tag{2}$$

For brevity, the parser construction algorithm will do without the star. □

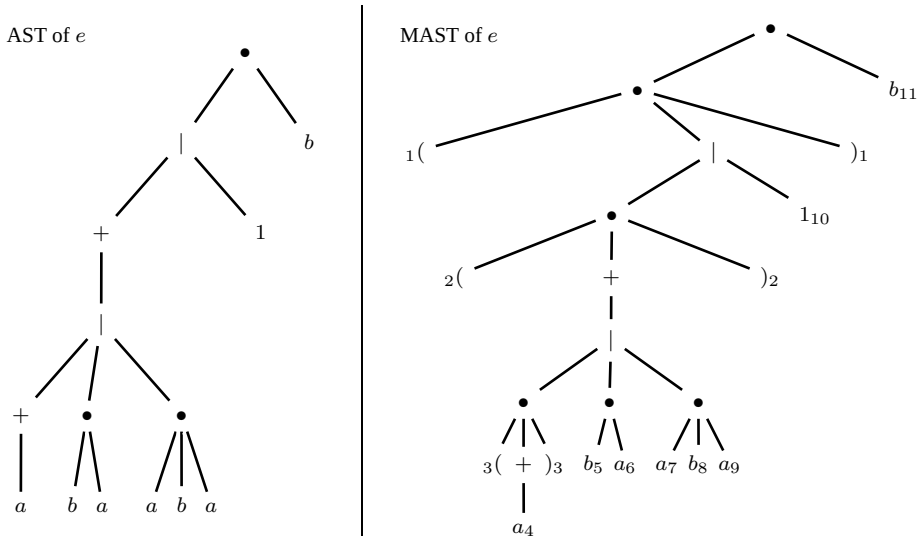
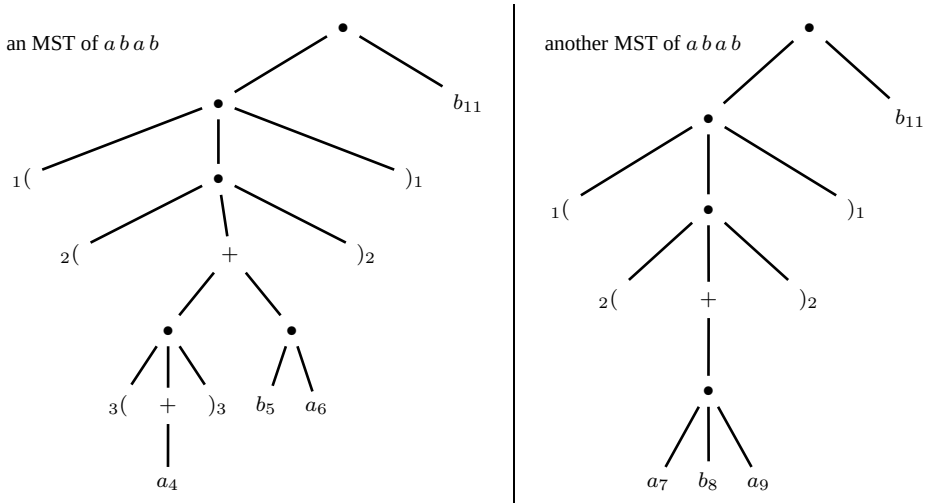


Fig. 1 Abstract syntax trees of an RE. Left: AST of the RE $e = \big(\big((\,a\,)^+ \mid b\,a \mid a\,b\,a \big)^+ \mid 1 \big)\; b$ in Eq. (2). Right: Marked AST (MAST) of the same RE – see also the marked RE $\check{e}$ in Eq. (3).

The *language generated* by an RE e with terminal alphabet Σ is $L\,(e) \subseteq \Sigma^*$, and each string in language $L\,(e)$ is called *legal*. If $\varepsilon \in L\,(e)$, both the language and the RE are *nullable*.

Trees of RE and of legal strings To prevent confusion, we call *abstract* the trees representing the syntax structure of an RE. A self-explanatory *abstract syntax tree* (AST) is shown in Fig. 1 (left). We also need a richer representation, where the tree structure shows up in the tree frontier in the form of parentheses, and all the leaf symbols are numbered, say, from left



LST: $\omega_1 = {}_1(\,{}_2(\,{}_3(\,a_4\,)_3\,b_5\,a_6\,)_2\,)_1\,b_{11}$

LST: $\omega_2 = {}_1(\,{}_2(\,a_7\,b_8\,a_9\,)_2\,)_1\,b_{11}$

Fig. 2 Two marked syntax trees (MST) for the ambiguous string $a\,b\,a\,b$ generated by RE $e = \big(\,((\,a\,)^+ \mid b\,a \mid a\,b\,a\,)^+ \mid 1\,\big)\ b$ and their string representations ω_1 and ω_2 as linearized syntax trees (LST) – see Fig. 1.

to right, to make them distinct. This representation is called a *marked abstract syntax tree* (MAST), and it is exemplified in Fig. 1 (right) and formalized in the Def. 1 below.

Every string in language $L\,(e)$ has at least one syntax tree. If a legal string has two or more trees, it is *ambiguous*, with *ambiguity degree* equal to the number of trees; also the RE e is called ambiguous. Notice that the ambiguity degree of a string is not necessarily finite.

We want to represent a syntax tree of a string in language $L\,(e)$ by means of the same marking used in the MAST of e, and we call it a *marked syntax tree* (MST). To illustrate, consider the ambiguous string $a\,b\,a\,b$, which is generated by the RE of Eq. (2) in two different ways, represented in Fig. 2 by the left and right MSTs.

More explicitly, an MST of a string $x \in L\,(e)$ is a tree such that the internal nodes are of the types "$\bullet$" or "$+$". Notice that the nodes of type "$\mid$" are unnecessary. The leaf nodes are of the same types as the nodes in the MAST of e. In an MST, each node of type "$+$" has a number of child nodes equal to the number of iterations of the subexpression under "$+$". For instance, in Fig. 2 (left) the iteration node identified by $_2(+)_2$ has two child nodes, while in Fig. 2 (right) the same iteration node has one.

It is important to observe that, as an AST represents the structure of an RE e, a MAST represents the structure of a different RE, which we call a *marked RE* (MRE), denoted $\breve{e}$. For instance, we list the MRE having as MAST the tree in Fig. 1 (right); for clarity we use square brackets instead of parentheses to group subexpressions:

$$\breve{e} = {}_1(\,\Big[\,{}_2(\,\Big[\,{}_3(\,[\,a_4\,]^+\,)_3 \;\mid\; b_5\,a_6 \;\mid\; a_7\,b_8\,a_9\,\Big]^+\,)_2 \;\mid\; 1_{10}\,\Big]\,)_1 b_{11} \tag{3}$$

The set of the terminals of the MRE in Eq. (3) (different from those of e) is denoted by Ω:

$$\Omega = \underbrace{\{\,a_4,\ b_5,\ a_6,\ a_7,\ b_8,\ a_9,\ b_{11}\,\}}_{\text{marked input alphabet } \breve{\Sigma}} \cup \underbrace{\{\,{}_1(,\ )_1,\ {}_2(,\ )_2,\ {}_3(,\ )_3,\ 1_{10}\,\}}_{\text{marked metasymbols } \breve{M}}$$

where the first set is the *marked input alphabet*, denoted by $\breve{\Sigma}$, and the second set consists of the *marked metasymbols*, denoted by $\breve{M}$. Two strings ω_1 and ω_2 in $L(\breve{e})$ are:

$$\omega_1 = {}_1(\,{}_2(\,{}_3(\,a_4\,)_3\,b_5\,a_6\,)_2\,)_1\,b_{11} \qquad \text{and} \qquad \omega_2 = {}_1(\,{}_2(\,a_7\,b_8\,a_9\,)_2\,)_1\,b_{11}$$

Looking at Fig. 2, the meaning of such strings is to linearly represent the marked syntax trees of the strings generated by the original RE e. Such strings are called *linearized syntax trees* (LST) and will be the output of the BSP parser.

It is time to collect and formalize the relevant technical terms into a definition.

Definition 1 (marked regular expression and linearized syntax tree) The *marked regular expression* MRE $\breve{e}$ associated to an RE e over Σ is defined by the following procedure:

1. Apply to e the translation $T[\![\,\cdot\,]\!]$ inductively defined as follows, where e_1 and e_2 are REs, and $a \in \Sigma$:

$$T[\![\,(e)^+\,]\!] = \big(\,[\,T[\![e]\!]\,]^+\,\big) \qquad T[\![\,(e)\,]\!] = \big(\,[\,T[\![e]\!]\,]\,\big)$$
$$T[\![e_1 \cdot e_2]\!] = T[\![e_1]\!] \cdot T[\![e_2]\!] \qquad T[\![e_1 \mid e_2]\!] = T[\![e_1]\!] \mid T[\![e_2]\!]$$
$$T[\![a]\!] = a \qquad T[\![1]\!] = 1$$

   Let $T[\![e]\!]$ be the result of step 1.

2. In $T[\![e]\!]$, assign a distinct number, e.g., in increasing left to right order, to the following symbols: open parentheses, symbols in Σ and empty string symbols " 1 ". Assign to each closed parenthesis the same number as the one of the matching open parenthesis (notice that square brackets and operator symbols are not numbered).
   Let $\breve{\Sigma}$, called *marked input alphabet*, denote the set of the numbered terminals, and let $\breve{M} = \{\,{}_h(,\,)_h,\,\ldots,\,1_i,\,\ldots\,\}$ denote the set of *marked metasymbols*.

The *terminal alphabet* of the MRE is the union $\Omega = \breve{\Sigma} \cup \breve{M}$. For any symbol in the alphabet Ω, define the letter-to-letter non-erasing homomorphism *unmark*: $\Omega \to \Sigma \cup \{\,\text{`}(\text{'}, \text{`})\text{'}, 1\,\}$ that deletes the subscript. For a numbered terminal $b_h \in \breve{\Sigma}$, we say that b_h belongs to the *class* identified by the "plain" symbol $b = \textit{unmark}(b_h)$.

For any symbol in the alphabet Ω, define the letter-to-letter erasing homomorphism *flatten*: $\Omega \to \Sigma$ as $\textit{flatten}(\alpha) = \textit{unmark}(\alpha)$ if $\alpha \in \breve{\Sigma}$ and as $\textit{flatten}(\alpha) = \varepsilon$ if $\alpha \in \breve{M}$. In other words, *flatten* unmarks all the subscripted terminals and deletes all the subscripted metasymbols. For instance, $\textit{flatten}\,\big({}_1(\,{}_2(\,a_7\,b_8\,a_9\,)_2\,)_1\,b_{11}\,\big) = a\,b\,a\,b$.

Each string in language $L(\breve{e})$ is called a *linearized syntax tree* (LST). More precisely, we define two sets of LSTs:

$$\text{for any } x \in L(e) \qquad \textit{LST}(x) = \{\,\omega \in L(\breve{e}) \mid \ \textit{unmark}(\omega) = x\,\}$$
$$\textit{LST}(e) = \bigcup_{x \in L(e)} \textit{LST}(x)$$

Notice that if it holds $\omega \in \textit{LST}(x)$, then string ω is the frontier of an MST of string x and we say that ω is the *linearized representation* of such a tree.

A string $x \in L(e)$ is *ambiguous* if, and only if, it holds $|\,\textit{LST}(x)\,| > 1$. □

The next property immediately follows from the above definitions.

Proposition 1 (parenthesis run) *For any RE e, any linearized syntax tree $\omega \in \textit{LST}(e)$ does not contain more than $2h$ consecutive parentheses, where the integer $h \geq 0$ is the maximum nesting depth of the parentheses in the RE e (if $h = 0$, in e there are no parentheses).* □

Examples of MREs are in Eq. (3) and in the Ex. 2 below. Notice that an MRE never uses the metasymbol 1 (empty string), though it may contain a numbered copy such as 1_3. Fig. 2 shows two LSTs of the same string.

Infinite ambiguity If an iterated subexpression is nullable, then for one or more legal strings the ambiguity degree is infinite. This situation is singled out as "problematic" in [11] and is illustrated in the next example. In practice, it is useless to enumerate all the trees of such strings, and just one tree or a few ones can be chosen (to be better explained in Sect. 3).

Example 2 (infinite ambiguity) The degree of ambiguity of every string in $L(e_1)$:

$$e_1 = (\, a \mid 1 \,)^+ \qquad\qquad \breve{e}_1 = {}_1(\,[\, a_2 \mid 1_3 \,]^+\,)_1 \tag{4}$$

is infinite because the iterated subexpression $(\, a \mid 1 \,)$ is nullable. Consider the associated MRE $\breve{e}_1$ in Eq. (4). The set $LST(a)$ comprises infinitely many strings, such as the following:

$$ {}_1(\, a_2 \,)_1 \qquad {}_1(\, a_2\, 1_3 \,)_1 \qquad {}_1(\, 1_3\, a_2 \,)_1 \qquad {}_1(\, 1_3\, a_2\, 1_3 \,)_1 \qquad {}_1(\, a_2\, 1_3\, 1_3 \,)_1 \qquad \ldots \tag{5}$$

Each such string represents a different syntax tree. In practice, it is hard to imagine any reason for the parser to return all such insignificantly different trees, and the simplest ones suffice, e.g., the first four, which do not contain two adjacent 1_3 symbols. □

Local languages The well known family of *local* languages, strictly included within the regular language family, is characterized by a very simple type of finite automaton, which serves as baseline for the Berry-Sethi construction.

For any characters $a,\ b \in \Sigma$, any strings $x,\ y \in \Sigma^*$ and any language $L \subseteq \Sigma^*$, we define the sets of *initials Ini* $\subseteq \Sigma$, *finals Fin* $\subseteq \Sigma$, *digrams Dig* $\subseteq \Sigma^2$ and *followers Fol* $\subseteq \Sigma$:

$$\begin{aligned} \mathit{Ini}\,(L) &= \{\, a \mid \quad a\,x \in L \,\} & \mathit{Fin}\,(L) &= \{\, b \mid \quad x\,b \in L \,\} \\ \mathit{Dig}\,(L) &= \{\, a\,b \mid \quad x\,a\,b\,y \in L \,\} & \mathit{Fol}\,(L,\, a) &= \{\, b \mid \quad a\,b \in \mathit{Dig}\,(L) \,\} \end{aligned}$$

If $L = L(e)$, we write $\mathit{Ini}\,(e)$ for $\mathit{Ini}\,\big(L(e)\big)$ and similarly for the other sets. We omit the well known (e.g., in [9]) simple algorithms for computing such sets.

The above sets characterize the family of *local* languages [4, 18], also known as 2-strictly locally testable or sliding-window recognizable languages.

Definition 2 (local language) A language $L \subseteq \Sigma^*$ is *local* if there exist finite sets *Ini*, *Fin* and *Dig* such that:

$$\forall x \neq \varepsilon \quad x \in L \iff \big(\, x \in \mathit{Ini}\,(L)\,\Sigma^* \ \wedge\ x \in \Sigma^*\,\mathit{Fin}\,(L) \ \wedge\ \mathit{Dig}\,(\{\, x \,\}) \subseteq \mathit{Dig}\,(L) \,\big) \tag{6}$$

Notice that an equivalent definition is possible using the follower set instead of the digrams.

The DFA recognizing the local language defined by Eq. (6) is straightforward: given a string, it checks that the initial letter is in the set *Ini*, the final one is in *Fin*, and that, if any two letters $a,\ b$ are read in a row, the digram $a\,b \in \mathit{Dig}$. Such a recognizer is called a *local automaton*.

The following well-known sufficient condition is later needed.

Proposition 2 (RE and local language) *If every symbol of alphabet Σ occurs at most once in an RE e, then the language $L(e)$ is local.* □

2.2 Classical Berry-Sethi recognizer

It is well-known (e.g., see [24]) that the BS method [3] for constructing a DFA that recognizes language $L(e)$ is related to the so-called position automaton methods, in the first place those of McNaughton-Yamada and Glushkov. The idea of BS is to transform a given RE e by distinctly numbering each letter occurring in it, thus obtaining a new RE denoted by $\bar{e}$, such that by Prop. 2 a local DFA accepts language $L(\bar{e}) \subseteq \breve{\Sigma}^*$, where $\breve{\Sigma}$ is the marked alphabet of Def. 1. Then, by erasing the numbers from the arc labels of the DFA, a recognizer for the original language $L(e)$ is obtained, which can be directly constructed to be deterministic by means of the subset construction. The original construction and correctness proof in [3] are based on the Brzozowski derivatives, but we prefer to follow the simpler approach in [4] (also in [9]), which relies on local languages.

Definition 3 (BS DFA) Let e be an RE over $\Sigma \cup M$. The *input marked* RE $\bar{e}$ over $\breve{\Sigma} \cup M$ is obtained from e by marking each symbol in Σ with a distinct integer (notice that the metasymbols are not marked). The initial and digram sets of language $L(\bar{e}\ \dashv)$ are resp. denoted *Ini* $(\bar{e}\ \dashv)$ and *Dig* $(\bar{e}\ \dashv)$. Define the DFA $A_{BS} = \big(\Sigma,\, Q_{BS},\, q_0,\, \delta_{BS},\, F\big)$, where:

- Each state $q \in Q_{BS}$ is uniquely identified by a non-empty set, called the *contents* of q and denoted by $I(q) \subseteq \breve{\Sigma} \cup \{\dashv\}$, i.e., a set comprising marked symbols and possibly also the end-of-text.
- The state q_0 is such that $I(q_0) = \mathit{Ini}\,(\bar{e}\ \dashv)$. Each final state $q \in F$ is such that $\dashv \in I(q)$.
- For every state $q \in Q_{BS}$ and character $a \in \Sigma$, let:

$$I_a(q) = \big\{\ a_h \in \breve{\Sigma} \mid \quad a_h \in I(q)\ \wedge\ \mathit{unmark}\,(a_h) = a\ \big\}$$

- The function or graph $\delta_{BS} \colon Q_{BS} \times \Sigma \to Q_{BS}$ is defined as follows:

$$\delta_{BS}(q,\, a) = q' \iff \begin{cases} \exists\, a_h \in I_a(q) \quad \wedge \\ I(q') = \big\{\ b_k \in \breve{\Sigma} \cup \{\dashv\} \mid \quad a_h\, b_k \in \mathit{Dig}\,(\bar{e}\ \dashv)\ \big\} \end{cases}$$

Example 3 (Ex. 1 continued) The input marked RE (choosing for comparability the same numbering as in Fig. 1) is:

$$\begin{aligned} \breve{\Sigma} &= \{\ a_4,\ b_5,\ a_6,\ a_7,\ b_8,\ a_9,\ b_{11}\ \} \\ \bar{e}\ \dashv &= \Big(\ (\ a_4\ )^+ \ \mid\ b_5\ a_6 \ \mid\ a_7\ b_8\ a_9\ \Big)^*\ b_{11}\ \dashv \end{aligned} \tag{7}$$

The initial set of $L(\bar{e}\ \dashv)$ is $\mathit{Ini}\,(\bar{e}\ \dashv) = \{\ a_4,\ b_5,\ a_7,\ b_{11}\ \}$ and the digram set is:

$$\mathit{Dig}\,(\bar{e}\ \dashv) = \left\{ \begin{array}{l} a_4a_4,\ a_4b_5,\ a_4a_7,\ a_4b_{11},\ b_5a_6,\ a_7b_8, \\ b_8a_9,\ a_9a_4,\ a_9b_5,\ a_9a_7,\ a_9b_{11},\ b_{11}\ \dashv \end{array} \right\}$$

The BS recognizer is shown in Fig. 3.

3 BS Parser

We extend the BS method to generate, instead of a recognizer, a parser called *Berry-Sethi parser* (BSP) that returns the linearized syntax trees of the input string. The parser is a *deterministic finite-state transducer* (DFT), which has the BS recognizer as underlying DFA.

First, we introduce the main ideas informally and by means of examples, then we list the generation algorithm of the parser and we analyze the parser size. We finish with the algorithm that computes the linearized syntax trees and its correctness proof.

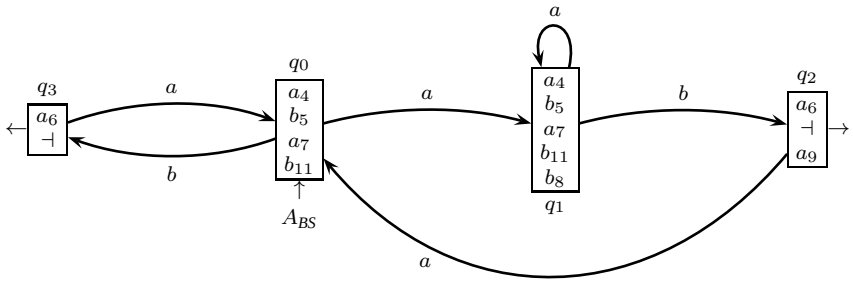


Fig. 3 The BS recognizer A_{BS} for the input marked RE $\bar{e} = \big((\, a_4 \,)^+ \mid b_5\, a_6 \mid a_7\, b_8\, a_9 \big)^*\ b_{11}$ of Ex. 3.

3.1 Intuitive presentation

Recognizer of linearized syntax trees The first conceptual step builds the recognizer of the set of linearized syntax trees $LST\,(\breve{e}\ \dashv) \subseteq \big(\breve{\Sigma} \cup \breve{M}\big)^*\ \{\dashv\}$, see Def. 1. Since all the terminal symbols occurring in $\breve{e}\ \dashv$ are distinct by definition, by Prop. 2 the language $LST\,(\breve{e}\ \dashv)$ is local and its local automaton is obvious.

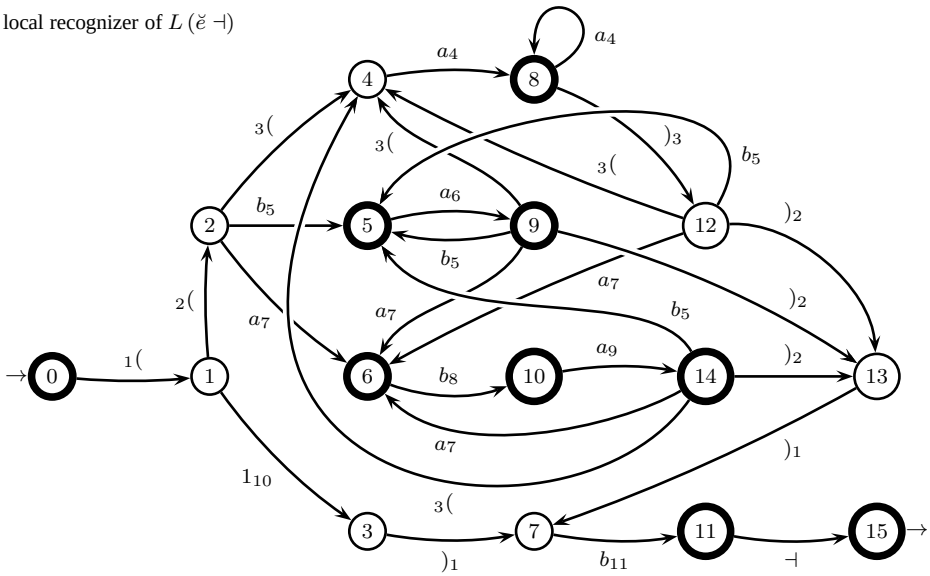
We start from non-infinitely ambiguous REs, by considering the RE e of Eq. (2) and the associated MRE $\breve{e}$ in Fig. 4 (top). The top graph of Fig. 4 shows the recognizer of $LST\,(\breve{e}\ \dashv)$, with nodes drawn differently: the initial state and the states entered by labeled input symbols are thicker, while the nodes entered by metasymbols are thin. The bottom of Fig. 4 shows a language-equivalent finite-state machine, the arcs of which are labeled by a finite language. We call such a machine "state-trimmed", because all the thin nodes have been eliminated. It is important to notice that, for all the arcs, every string in the labelling finite language has a fixed format: it starts with zero or more metasymbols and ends with an input symbol, e.g., the string "$\,{}_1(\,{}_2(\,b_5\,$" on the arc $0 \to 5$. Such strings are called (*LST*) *segments*, and the input symbol at their end is called *end symbol*.

We show how the trimmed graph in Fig. 4 reflects the fact that the RE has a non-infinite ambiguity degree. In the top graph, every path between thick nodes that passes only through thin nodes is acyclic, e.g., the path from 0 to 5. Therefore, in the trimmed graph, the label of arc $0 \to 5$, obtained by collapsing finitely many paths from 0 to 5, is a finite language.

LST recognizer of an infinitely ambiguous RE When the ambiguity degree is unbounded, the arc labels in the state-trimmed graph may become infinite languages. For the RE e_1 and MRE $\breve{e}_1$ in Eq. (4) (see Ex. 2), the recognizer of $L\,(\breve{e}_1\ \dashv)$ is shown in Fig. 5 (top left). Now, some paths from thick node to thick node, traversing only thin nodes, are cyclic, e.g., path $0 \xrightarrow{{}_1(} 1 \xrightarrow{1_3} 3 \xrightarrow{1_3} 3\ \ldots\ 3 \xrightarrow{a_2} 2$. In such cases, when the thin nodes traversed are eliminated, to preserve equivalence, the new arc in the state-trimmed graph must have an infinite language for label. Thus, the arc $0 \to 2$ of the trimmed graph in Fig. 5 (top right) is labeled by language "$\,{}_1(\,a_2 \mid {}_1(\,1_3\,[\,1_3\,]^*\,a_2\,$". Similarly, the other arcs are labeled by infinite languages. We observe that all the LSTs of string $a_2\ \dashv$ (enumerated by Eq. (5) in Ex. 2) are included in the label of the recognizing path $0 \to 2 \to 5$. It is important to observe that, by Prop. 1, every cyclic path in the local recognizer of the LSTs in $L\,(\breve{e})$ contains at least one numbered letter or one numbered symbol 1; otherwise a string sufficiently long to violate Prop. 1 could be obtained by iterating the cycle.

$$\check{e} = {}_1(\left[{}_2(\left[{}_3([\,a_4\,]^+)_3 \;\mid\; b_5\,a_6 \;\mid\; a_7\,b_8\,a_9\right]^+)_2 \;\mid\; 1_{10}\right])_1\,b_{11}$$

local recognizer of $L\,(\check{e}\,\dashv)$



state-trimmed recognizer of $L\,(\check{e}\,\dashv)$

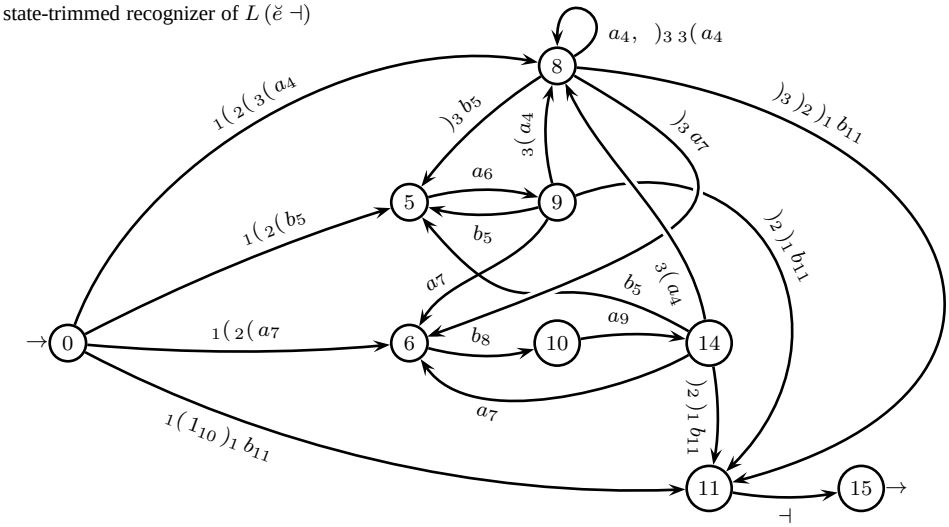


Fig. 4 Top graph: the local recognizer of LSTs for the running example Ex. 1. Bottom "state-trimmed" graph: the nodes are exactly the thick ones of the top graph.

Since we want to limit the number of LSTs returned for each string of infinite ambiguity degree, we have to choose a criterion for discarding an infinite number of LSTs. The following criterion is reasonable and easy to implement, but any other would fit into the

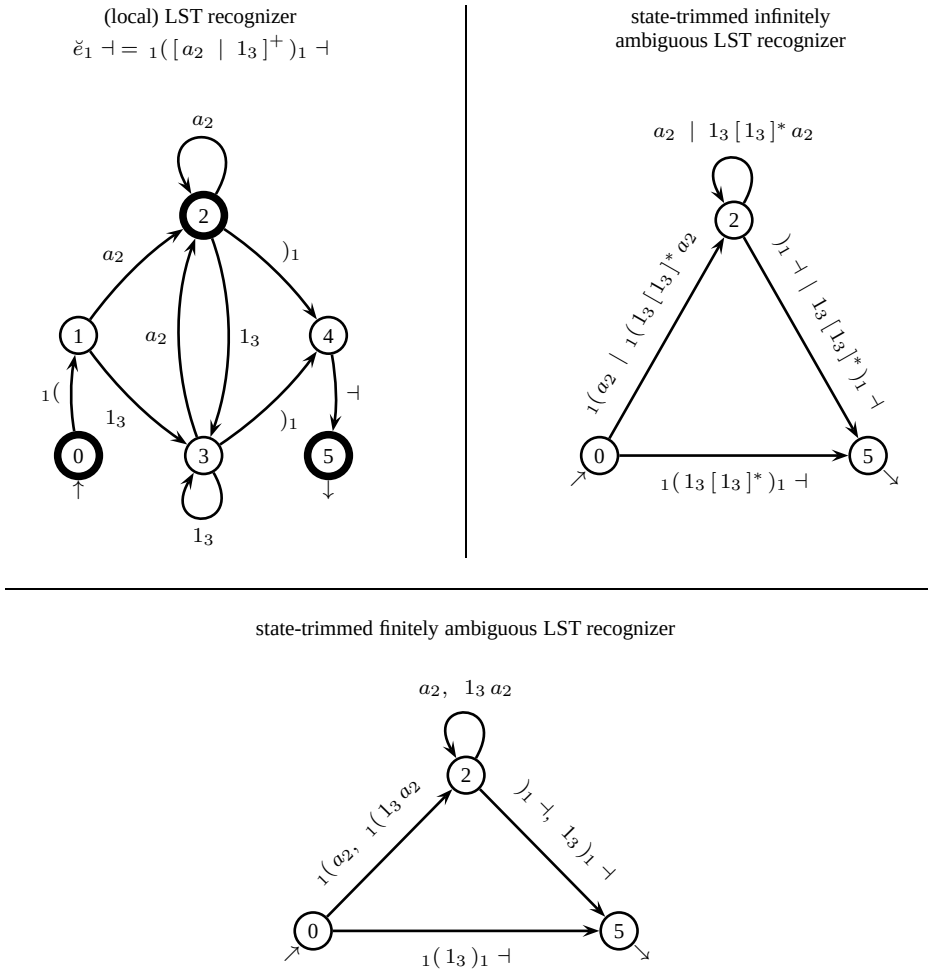


Fig. 5 Top left: the (local) recognizer of the LSTs for the infinitely ambiguous (i.e., problematic) RE $e_1 = (\,a \mid 1\,)^+$ of Ex. 2. Top right: the equivalent state-trimmed machine. Bottom: the same state-trimmed machine restricted to a finite ambiguity degree.

BSP parser, provided it bounds the number of LSTs. We state the criterion as a rule for transforming the local LST recognizer into a state-trimmed finitely ambiguous recognizer.

How to bound ambiguity For each pair of thick nodes p and q (possibly coincident) in the local recognizer of $L\,(\check{e}\ \dashv)$, take every path that connects p to q without traversing another thick node, and that does not traverse twice an arc labeled by the same empty string symbol, say by 1_3. The labels of the taken paths are called *acyclic segments* (AS) of an LST. To construct the state-trimmed ambiguity bounding machine, shown in Fig. 5 (bottom), we collapse such paths into an arc and we label it with the union of the ASs of the paths.

To illustrate, the label of the path $2 \xrightarrow{1_3} 3 \xrightarrow{1_3} 3 \xrightarrow{)_1} 4 \xrightarrow{\dashv} 5$ of Fig. 5 (top left) is not an AS according to the ambiguity bounding criterion, therefore it is not attached to the arc from 2 to 5 of the state-trimmed graph at the bottom of the same figure.

From recognizer to parser From the machine that recognizes the LSTs, we move ahead towards the construction of the parser that recognizes the input strings and computes their LSTs. The parser is a deterministic finite-state transducer, which extends the classic A_{BS} recognizer with an output function denoted by ρ. At each transition, the function ρ emits a piece of information essentially consisting of metasymbols, which represents a part of one or more LSTs. For convenience, the input string is marked on the left by the start-of-text symbol $\vdash$. The initial state of the parser is called *start*. The initial transition from state *start* reads the start-of-text and emits the first piece of LSTs. Upon termination, the concatenation of all such pieces encodes all the LSTs of the given input string.

We recall that a transition $\left(q \xrightarrow{a} q' \right) \in \delta_{BS}$ is associated to a set of digrams $b\,c \in \mathit{Dig}\,(\bar{e}\ \dashv)$, i.e., the digrams of the input marked RE $\bar{e}$. More precisely, the marked input symbols b and c are in the sets $I_a\,(q)$ and $I\,(q')$, respectively (see Sect. 2.2). When taking the transition $q \xrightarrow{a} q'$, the transducer emits an output consisting of a generalization of the digram $b\,c$, namely a set of segments of a linearized syntax tree; more precisely, the value $\rho\,(q,\ a)$ of the output function includes all the marked metasymbol strings that may be enclosed between b and c in any LST. To illustrate, for the running example of RE $\breve{e}$ in Fig. 4 and the digram $a_4\,a_4$ in the input string, the output function ρ includes, among others, the two marked metasymbol strings ε and " $)_3\,3($ ". We represent each such string within a 3-tuple, which has the digram symbols as first and third component, as follows:

digram	*metasymbolic string*	*representation as a* 3*-tuple*
$a_4\,a_4$	ε	$\langle\, a_4,\ \varepsilon,\ a_4 \,\rangle$
$a_4\,a_4$	$)_3\,3($	$\langle\, a_4,\ )_3\,3(\ ,\ a_4 \,\rangle$

In the top graph of Fig. 4, the strings ε and " $)_3\,3($ " respectively correspond to the arc $8 \xrightarrow{a_4} 8$ and to the path:

$$8 \xrightarrow{)_3} 12 \xrightarrow{{}^3(} 4 \xrightarrow{a_4} 8$$

The preceding ideas are presented precisely in the following section.

3.2 The BSP algorithm

As the BSP algorithm is based on the entities intuitively presented above, we list more precisely their definitions. Let $x \in L\,(e\ \dashv)$, with $x = a_1\ \ldots\ a_n\ \dashv$ (for $n \geq 1$) or $x = \dashv$, and let $\omega \in \mathit{LST}\,(x)$, therefore $\omega \in \left(\breve{\Sigma} \cup \breve{M}\right)^*\ \{\dashv\}$ and, by Def. 1, it is $x = \mathit{flatten}\,(\omega)$.

Definition 4 (factorization) The *factorization into segments* of a string $\omega \in \mathit{LST}\,(x)\ \dashv$ is:

$$\omega = \zeta_1 \cdot \ldots \cdot \zeta_j \cdot \ldots \cdot \zeta_n \cdot \zeta_{n+1} \qquad \text{with } n \geq 0, \text{ where:} \tag{8}$$

$$\zeta_j = \mu_j\,a_j \qquad \mu_j,\ a_j \in \breve{M}^*,\ \breve{\Sigma} \quad \text{for } 1 \leq j \leq n \tag{9}$$

$$\zeta_{n+1} = \mu_{n+1}\ \dashv \qquad \mu_{n+1} \in \breve{M}^* \tag{10}$$

Each term ζ_k, with $1 \leq k \leq n+1$, is called a *segment*. Each term μ_j is a (possibly empty) string of marked metasymbols. The symbols a_j (marked input letter) and $\dashv$ are called the *end symbols* of the segments ζ_j and ζ_{n+1}, respectively. □

For every linearized syntax tree $\omega \in \mathit{LST}(x)\ \dashv$, the factorization into segments is clearly unique. Later, we sometimes omit the end-of-text $\dashv$ from the last segment.

To illustrate, we list the factorization into segments of two strings $\omega_1,\ \omega_2 \in \mathit{LST}(a\,a\,a)$, where $a\,a\,a \in L(e_1)$ and $e_1 = (\ a\ \mid\ 1\ )^+$ (see Ex. 2). The MRE is $\breve{e}_1 = {}_1(\ [\,a_2\ \mid\ 1_3\,]^+\ )_1$:

$$\zeta_1 \cdot \zeta_2 \cdot \zeta_3 \cdot \zeta_4 = \underbrace{{}_1(\ 1_3}_{\mu_1}\ a_2 \cdot \underbrace{\varepsilon}_{\mu_2}\ a_2 \cdot \underbrace{1_3}_{\mu_3}\ a_2 \cdot \underbrace{1_3\ )_1}_{\mu_4}$$

$$\zeta_1 \cdot \zeta_2 \cdot \zeta_3 \cdot \zeta_4 = \underbrace{{}_1(\ 1_3}_{\mu_1}\ a_2 \cdot \underbrace{\varepsilon}_{\mu_2}\ a_2 \cdot \underbrace{1_3\,1_3}_{\mu_3}\ a_2 \cdot \underbrace{1_3\ )_1}_{\mu_4}$$

Actually, the set $\mathit{LST}(a\,a\,a)$ contains infinitely many other strings that differ only in the number of occurrences of the marked empty string symbol 1_3. We believe that listing such cases would be wasteful and in the next definition we formalize an idea for binding the ambiguity exposed by the parser. Let all the symbols be defined as in Def. 4.

Definition 5 (acyclic segment – AS) A *segment* $\zeta = \mu\, a_h$ or $\zeta = \mu \dashv$ is *acyclic*, shortened as AS, if and only if $|\,\mu\,|_{1_j} \leq 1$ for all j, i.e., all the metasymbols of type 1 that occur in μ (if any) have distinct marks. The *acyclic marked language* $L_{\mathit{acyclic}}(e)$ defined by RE e is:

$$L_{\mathit{acyclic}}(e) = \{\ \omega \in L(\breve{e}) \mid\quad \omega = \zeta_1\ \ldots\ \zeta_j\ \ldots\ \zeta_n \text{ and every segment } \zeta_j \text{ is acyclic}\ \}$$

The *set AS* (e) *of the acyclic segments* of RE e is:

$$\mathit{AS}(e) = \{\ \zeta_j \mid\quad \zeta_1\ \ldots\ \zeta_j\ \ldots\ \zeta_n \in L_{\mathit{acyclic}}(e)\ \}$$

The set of the *acyclic linearized syntax trees* of a string $x \in L(e)$ is:

$$\mathit{LST}_{\mathit{acyclic}}(x)\ =\ \mathit{LST}(x)\ \cap\ L_{\mathit{acyclic}}(e)$$

As a consequence of Prop. 1, the set $AS(e)$ is always finite. Therefore, it is possible for the parser to compute online the acyclic LSTs.

Notice that language $L_{\mathit{acyclic}}(e)$ is obtained from the language $L(\breve{e}) \equiv \mathit{LST}(e)$ of Def. 1 by deleting all the strings that contain two or more instances of the same marked metasymbol 1 without a marked input symbol $\breve{\Sigma}$ in between. This does not exclude that two identically subscripted parentheses may occur in an acyclic segment.

Our approach to construct the parser, similarly to the BS recognizer, relies on the sets of initials and followers, the elements of which are the acyclic segments instead of the marked alphabet symbols. In the next definition all the symbols are as in Def. 4 and 5.

Definition 6 (initial / follower segment) The *set of initial acyclic segments* and the *set of the acyclic segments that follow* a_h are:

$$\mathit{Ini}_{\mathit{AS}}\ (e\ \dashv) = \{\ \zeta_1 \mid\quad \zeta_1\ \ldots\ \zeta_{n+1} \in L_{\mathit{acyclic}}(e\ \dashv)\ \}\qquad \text{with } n \geq 0$$

$$\mathit{Fol}_{\mathit{AS}}(e\ \dashv,\ a_h) = \{\ \zeta_{j+1} \mid\quad \zeta_1\ \ldots\ \zeta_j\,\zeta_{j+1}\ \ldots\ \zeta_{n+1} \in L_{\mathit{acyclic}}(e\ \dashv)\ \text{ and }\ \zeta_j = \mu_j\, a_h\ \ \}$$

We say that ζ_{j+1} is a *follower* of a_h and that $\mathit{Fol}_{\mathit{AS}}(e\ \dashv,\ a_h)$ is the *follower set* of a_h.

Clearly, the set $\mathit{Ini}_{\mathit{AS}}(e\ \dashv)$ and all the sets $\mathit{Fol}_{\mathit{AS}}(e\ \dashv,\ a_h)$ of acyclic segments are finite and computable.

For the RE of the running example, the set $\mathit{Ini}_{\mathit{AS}}$ and all the sets $\mathit{Fol}_{\mathit{AS}}$ are listed in Tab. 2. Such sets and their contents are orderly listed by scanning the MRE from left to right. By erasing the metasymbols $\breve{M}$ in Tab. 2, the initial and follower symbols (or equivalently the digrams) coincide with those of the classical BS method for the same RE e; see Sect. 2.2.

Table 2 The set of *initial* AS and all the sets of *follower* AS for the MRE of the running example.

$$\breve{e} = {}_1(\left[{}_2(\left[{}_3([\,a_4\,]^+)_3 \mid b_5\,a_6 \mid a_7\,b_8\,a_9 \right]^+)_2 \mid 1_{10} \right])_1\, b_{11}$$

$$\mathit{Ini}_{AS}(e\,\dashv) = \{\ {}_1(\,{}_2(\,{}_3(\,a_4,\ {}_1(\,{}_2(\,b_5,\ {}_1(\,{}_2(\,a_7,\ {}_1(\,1_{10}\,)_1\,b_{11}\ \}$$

marked symbol a_h	*set of followers of* a_h – $\mathit{Fol}_{AS}(e\,\dashv, a_h)$				
a_4	a_4	$)_3\ 3(\ a_4$	$)_3\ b_5$	$)_3\ a_7$	$)_3\ )_2\ )_1\ b_{11}$
b_5	a_6				
a_6	${}_3(\ a_4$	b_5	a_7	$)_2\ )_1\ b_{11}$	
a_7	b_8				
b_8	a_9				
a_9	${}_3(\ a_4$	b_5	a_7	$)_2\ )_1\ b_{11}$	
b_{11}	$\dashv$				

3.3 Parser Generation

We put together the preceding intuitions and definitions into Alg. 1, which constructs the BSP parser as a deterministic finite-state transducer (DFT). It is convenient, though slightly redundant, to formalize the transducer before its construction; a formal definition is also necessary for proving the correctness of the BSP parser. We define the transducer by extending the BS recognizer with a new initial state and adding the output function; as before, each state is identified by its contents.

Definition 7 (finite-state transducer) Let e be an RE over alphabet Σ, let $\breve{e}$ be the corresponding MRE, and let $A_{BS} = \left(\Sigma,\ Q_{BS},\ q_0,\ \delta_{BS},\ F\right)$ be the DFA of Def. 3. The transducer A is the 7-tuple $A = \left(\Sigma \cup \{\vdash\},\ Q,\ \mathit{start},\ \delta,\ \rho,\ F,\ O\right)$, where:

- $Q = Q_{BS} \cup \{\ \mathit{start}\ \}$, where *start* is the initial state and its *contents* are $I\,(\mathit{start}) = \{\vdash\}$
- the set of final states is F as in A_{BS}
- the state-transition graph is $\delta = \delta_{BS} \cup \left\{\ \mathit{start} \xrightarrow{\vdash} q_0\ \right\}$, where q_0 is the initial state of A_{BS}
- the output alphabet is $O = \wp\left(\left(\breve{\Sigma} \cup \{\vdash\}\right) \times \breve{M}^* \times \left(\breve{\Sigma} \cup \{\dashv\}\right)\right)$
- the *output function* $\rho\colon Q \times \left(\Sigma \cup \{\vdash\}\right) \to O$ is defined as follows:

$$\rho\,(\mathit{start}, \vdash) = \left\{\ \langle \vdash, \mu, c\rangle \mid \quad c \in \Sigma\ \wedge\ \mu\,c \in \mathit{Ini}_{AS}\,(e\,\dashv)\ \right\}$$

$$\forall\, q,\, q' \in Q_{BS} \quad \forall\, a \in \Sigma \qquad \text{such that } \left(q \xrightarrow{a} q'\right) \in \delta_{BS}$$
$$\rho\,(q,\, a) = \left\{\ \langle b,\, \mu,\, c\rangle \mid \quad b \in I_a\,(q)\ \wedge\ \mu\,c \in \mathit{Fol}_{AS}\,(e\,\dashv,\, b)\ \right\}$$

We recall that symbol a is an input letter (unmarked), and that symbols b and c are marked; in particular, it holds $b \in \breve{\Sigma}$ and $c \in \breve{\Sigma} \cup \{\dashv\}$. Moreover, symbol b is of class a, i.e., $a = \mathit{unmark}\,(b)$. $\sqcap$

Algorithm 1: Construction of the Berry-Sethi Parser (BSP).

Input: the sets Ini_{AS} and Fol_{AS} of an RE e
Output: the transducer $A = \left(\Sigma \cup \{ \vdash \},\ Q,\ start,\ \delta,\ \rho,\ F,\ O \right)$

$I\,(start) := \{\ \vdash\ \}$ // **create initial state *start***
$Q := \{\ start\ \}$ // initialize state set Q

tag state *start* // **process initial state *start***
$I\,(q_0) := \{\ c \mid \quad \mu\, c \in Ini_{AS}\ (e\ \dashv)\ \}$ // create new state q_0
untag state q_0 // new state q_0 is still unprocessed
$TS := \{\ \langle \vdash, \mu, c \rangle \mid \quad \mu\, c \in Ini_{AS}\ (e\ \dashv)\ \}$ // assign set *TS* of output 3-tuples
$Q := Q \cup \{\ q_0\ \}$ // update state set Q
$\delta := \{\ start \xrightarrow{\vdash} q_0\ \}$ // initialize transition function δ
$\rho := \{\ start \xrightarrow{TS} q_0\ \}$ // initialize output function ρ

while $\exists$ state $q \in Q$ that is untagged **do** // **create and process other states**
  tag state q // process state q
  foreach input symbol $a \in \Sigma$ **do** // scan each input symbol a
    $I\,(q') := \emptyset$ // create new state q' (initially empty)
    untag state q' // new state q' is still unprocessed
    $TS := \emptyset$ // initialize 3-tuple set *TS*
    foreach $b \in I_a\,(q)$ **do** // scan each marked $b \in \breve{\Sigma}$ of class a in q
      $I\,(q') := I\,(q') \cup \{\ c \mid \quad \mu\, c \in Fol_{AS}\ (e\ \dashv, b)\ \}$ // update state q
      $TS := TS \cup \{\ \langle b, \mu, c \rangle \mid \quad \mu\, c \in Fol_{AS}\ (e\ \dashv, b)\ \}$ // update set *TS*
    if $I\,(q') \neq \emptyset$ **then** // if new state q' is not empty, then
      if $q' \notin Q$ **then** // if q' is not in the state set Q, then
        $Q := Q \cup \{\ q'\ \}$ // update state set Q
      $\delta := \delta \cup \{\ q \xrightarrow{a} q'\ \}$ // update transition function δ
      $\rho := \rho \cup \{\ q \xrightarrow{TS} q'\ \}$ // update output function ρ

$F := \{\ q \in Q \mid \quad \dashv \in I\,(q)\ \}$ // **create the set F of final states**

Notice that the value of the output function ρ is a finite set of 3-tuples included in the domain O. As customary, in the examples we represent the state-transition function and the output function as arc labels.

Explanation of Alg. 1 In the first steps, the algorithm creates the initial state *start*, the state q_0 of the DFA A_{BS}, and the arc $start \xrightarrow{\vdash} q_0$ with the output ρ defined by the initial segments $Ini_{AS}\,(e)$. Then, the `while` loop examines each state $q \in Q$ in turn and tags it to avoid reexamining. For the current state q, the outermost `for` loop creates a new state q' as the target of an arc $q \xrightarrow{a} q'$ with input $a \in \Sigma$; the innermost `for` loop examines the contents of state q and for each marked symbol b of class a therein, it inserts into the set $I\,(q')$ all the *end symbols* c of the acyclic segments $Fol_{AS}\,(e\ \dashv, b)$. The value of the output function ρ for the arc $q \xrightarrow{a} q'$ is built by using the metasymbolic part μ of the same acyclic segments. At last, the algorithm identifies the final states, which contain the end-of-text $\dashv$.

Example 4 (transducer construction) The result of the application of Alg. 1 to RE e, given the sets Ini_{AS} and Fol_{AS} in Tab. 2, is shown in Fig. 6. By construction, every state of transducer A is accessible from the initial state and is connected to a final state.

We compare the transducer A in Fig. 6 and the DFA A_{BS} in Fig. 3. If we disregard the output function ρ and the initial state *start*, the state-transition graphs of A and A_{BS} are identical. Moreover, for each pair of corresponding states in A and A_{BS}, the state contents $I(\cdot)$ are identical sets.

Thus the following relation holds between the languages recognized: $L(A) = \vdash L(A_{BS})$. This proves that Alg. 1 is correct with respect to the language recognition property, and it remains to be proved (in Sect. 3.6) that the output of transducer A is correct.

In fact, the novelty of Alg. 1 is the output function that encodes all the LSTs of the input, and we illustrate with the computation of $\rho(\vdash b)$. For completeness, we list all the 3-tuples in the output, including some that are useless for the construction of the LSTs (to be explained in Sect. 3.5):

$$\rho(\vdash b) = \left\{ \begin{array}{c} \langle \vdash, {}_1(\, {}_2(\, {}_3(\, , a_4 \rangle \\ \langle \vdash, {}_1(\, {}_2(\, , b_5 \rangle \\ \langle \vdash, {}_1(\, {}_2(\, , a_7 \rangle \\ \langle \vdash, {}_1(\, 1_{10} \,)_1 , b_{11} \rangle \end{array} \right\} \cdot \left\{ \begin{array}{c} \langle \, b_5, \varepsilon, a_6 \, \rangle \\ \\ \langle \, b_{11}, \varepsilon, \dashv \rangle \end{array} \right\}$$

In Sect. 3.5 we explain how such an output represents the LSTs of the input string. In fact, string b has just the LST encoded by $\langle \vdash, {}_1(\, 1_{10} \,)_1 , b_{11} \rangle \cdot \langle b_{11}, \varepsilon, \dashv \rangle$, namely: "${}_1(\, 1_{10} \,)_1 \, b_{11}$". More examples are in Fig. 8.

The next example illustrates the case of an infinitely ambiguous RE.

Example 5 (infinitely ambiguous case) For the RE of Ex. 2 reproduced below:

$$e_1 = (\, a \;\mid\; 1 \,)^+ \qquad\qquad \breve{e}_1 = {}_1(\, [\, a_2 \;\mid\; 1_3 \,]^+ \,)_1$$

with the local recognizer seen in Fig. 5 (top left) we list the initial and follower sets needed by Alg. 1:

$$\begin{aligned} Ini_{AS}(e_1 \dashv) &= \{ \, {}_1(\, a_2, \; {}_1(\, 1_3 \, a_2, \; {}_1(\, 1_3 \,)_1 \dashv \, \} \\ Fol_{AS}(e_1 \dashv, a_2) &= \{ \, a_2, \; 1_3 \, a_2, \;)_1 \dashv, \; 1_3 \,)_1 \dashv \, \} \end{aligned}$$

The graph of the transducer is shown in Fig. 7. We list the output emitted for two input strings, namely ε and a, and for brevity we show only the 3-tuples that occur in some LST:

input	*output*
$\vdash \varepsilon$	$\{ \langle \vdash, {}_1(\, 1_3 \,)_1, \dashv \rangle \}$
$\vdash a$	$\left\{ \begin{array}{c} \langle \vdash, {}_1(\, , a_2 \rangle \\ \langle \vdash, {}_1(\, 1_3, a_2 \rangle \end{array} \right\} \cdot \left\{ \begin{array}{c} \langle \, a_2, \,)_1, \dashv \rangle \\ \langle \, a_2, \, 1_3 \,)_1, \dashv \rangle \end{array} \right\}$

The syntax trees of strings ε and a are depicted in Fig. 9 in Sect. 3.5.

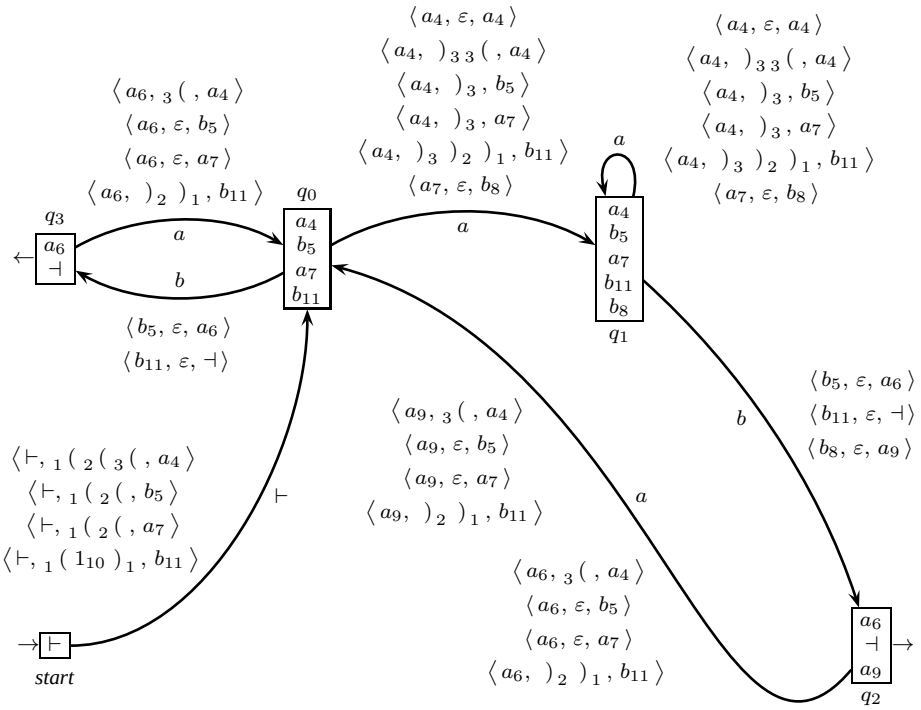


Fig. 6 The graph of transducer A constructed by Alg. 1 for the RE e of Eq. (3). The sets Ini_{AS} and Fol_{AS} used by the algorithm are taken from Tab. 2. Each arc carries an input symbol and the output produced, which is a set of 3-tuples, omitting for brevity the brackets " { " and " } ".

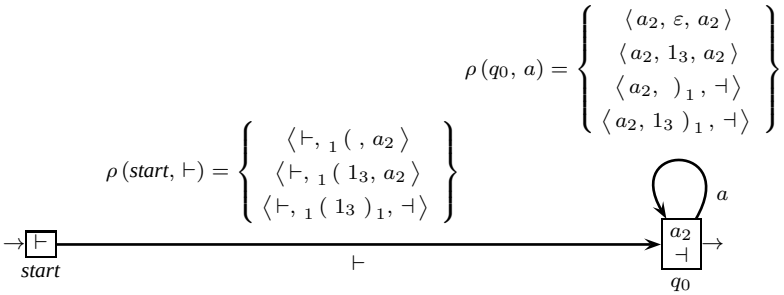


Fig. 7 The graph of transducer A constructed by Alg. 1 for the RE $e_1 = (a \mid 1)^+$ of Ex. 5.

3.4 Transducer size

To analyze the size or descriptive complexity of the transducer A constructed by Alg. 1, we first define some complexity measures for REs. Let e, $\bar{e}$ and $\check{e}$ be an RE over alphabet Σ, the input marked RE and the marked RE, respectively.

Definition 8 (complexity measures of RE**)** The *alphabetic size* of e is the cardinality of the marked input alphabet $\breve{\Sigma}$. The *ε-size* of e, denoted by n_ε, is the number of marked symbols 1 in $\breve{e}$ ($n_\varepsilon \geq 0$). The maximum nesting depth of the operator "$+$" in e is denoted as h ($h \geq 1$). □

It is straightforward to figure out how the number of states and transitions of transducer A depend on the above complexity measures. From Def. 7, the graph of A has the same nodes and arcs as the DFA A_{BS} has, plus the state *start* and the arc *start* $\xrightarrow{\vdash} q_0$. In the worst case, the size of A_{BS} is:

$$\begin{aligned} &|\, Q_{BS} \,| = 2^{|\,\breve{\Sigma}\,|+1} \\ &|\, \delta_{BS} \,| = |\,\Sigma\,| \times |\, Q_{BS} \,| = |\,\Sigma\,| \times 2^{|\,\breve{\Sigma}\,|+1} \end{aligned} \qquad \text{where term } +1 \text{ is for } \dashv$$

Thus the worst-case size of transducer A in terms of nodes and arcs is:

$$\begin{aligned} &|\, Q \,| = |\, Q_{BS} \,| + 1 = 2^{|\,\breve{\Sigma}\,|+1} + 1 \in \mathcal{O}\big(\, 2^{|\,\breve{\Sigma}\,|} \,\big) \\ &|\, \delta \,| = |\, \delta_{BS} \,| + 1 \;\; = |\,\Sigma\,| \times 2^{|\,\breve{\Sigma}\,|+1} + 1 \in \mathcal{O}\big(\, |\,\Sigma\,| \times 2^{|\,\breve{\Sigma}\,|} \,\big) = \overbrace{\mathcal{O}\big(\, 2^{|\,\breve{\Sigma}\,|} \,\big)}^{\text{if } |\,\Sigma\,| \text{ constant}} \end{aligned} \tag{11}$$

where $|\, \delta \,|$ reduces to $\mathcal{O}\big(\, 2^{|\,\breve{\Sigma}\,|} \,\big)$ if we consider the alphabet size $|\,\Sigma\,|$ as a constant.

Every arc of transducer A supports the output function ρ. It takes some effort to figure out how the size of ρ depends on the measures of RE e. For every arc $\big(\, q \xrightarrow{a} q' \,\big) \in \delta$, the value of $\rho\,(q,\, a)$ is a set of 3-tuples $\langle\, b,\, \mu,\, c \,\rangle$, where $b\,c$ is a digram of $\vdash \bar{e} \dashv$, i.e., the input marked RE encompassed by the start and end marks. Therefore, it holds $b \in \breve{\Sigma} \cup \{ \vdash \}$ and $c \in \breve{\Sigma} \cup \{ \dashv \}$. By construction we have: $b \in I\,(q)$, $c \in I\,(q')$, $\mu\, c \in \mathit{Fol}_{AS}\,(\vdash e \dashv,\, b)$ and $\mathit{Fol}_{AS}\,(\vdash e \dashv,\, \vdash) = \mathit{Ini}_{AS}\,(e)$. We want to count how many 3-tuples may label an arc of A.

For any digram $b\,c \in \mathit{Dig}\,(\vdash \bar{e} \dashv)$, let $C\,(b,\, c)$ be the number of substrings of type $b\,\zeta$, where $\zeta = \mu\, c$ is an acyclic segment (Def. 5) of some string $\omega \in L\,(\breve{e} \dashv)$ that does not include any two copies of a marked symbol 1_j without any input letter in between. We denote as $C'\,(b,\, c)$ and $C''\,(b,\, c)$ the numbers of the segments ζ that do not include any symbol of type 1_j and that include one or more symbols of type 1_j, respectively. Clearly, it holds $C\,(b,\, c) = C'\,(b,\, c) + C''\,(b,\, c)$.

In the worst case we have $C'\,(b,\, c) = h$, where h is the nesting depth of the operator "$+$" in the RE e, because a minimum of one and a maximum of h such operators may be applied to generate $b\,\zeta = b\,\mu\, c$. For instance, the following RE e_2 and MRE $\breve{e}_2$ have nesting depth $h = 3$ and the value C listed below:

$$e_2 = \Big(\big(\, (\, a \,)^+ \,\big)^+ \Big)^+ \qquad\qquad \breve{e}_2 = \; {}_1(\, \Big[\, {}_2(\, \Big[\, {}_3(\, [\, a_4 \,]^+ \,)_3 \,\Big]^+ \,)_2 \,\Big]^+ \,)_1$$

$$C(a_4, a_4) = C'(a_4, a_4) = \Big|\, \big\{\; \langle a_4,\, \varepsilon,\, a_4 \rangle\,,\; \langle a_4,\,)_3\; {}_3(,\, a_4 \rangle\,,\; \langle a_4,\,)_3\;)_2\; {}_2(\; {}_3(,\, a_4 \rangle \;\big\} \,\Big| = 3$$

Concerning the term $C''(b,\, c)$, still in the worst case, we have the following (with $n_\varepsilon \geq 1$):

$$C''\,(b,\, c) = \sum_{i=1}^{n_\varepsilon} h^i \times D_{n_\varepsilon,\, i} \; = \sum_{i=1}^{n_\varepsilon} h^i \times n_\varepsilon \times (n_\varepsilon - 1) \;\times\; \ldots \;\times\; (n_\varepsilon - i + 1) \tag{12}$$

where the term $D_{n_\varepsilon,\, i}$ is the number of *ordered selections without repetitions* of $n_\varepsilon \geq 1$ marked symbols 1 in groups of i symbols. The count for the number C'' corresponds to all the possible ways of including any number of marked symbols 1 in any order, by applying the nested operators "$+$", from a minimum of one up to a maximum of h, to generate a

(sub)segment of type $1_j\,\mu\,1_k$, where the string $\mu \subset \Omega^*$ does not include any marked symbol 1. To illustrate how number C'' is computed, consider the RE e_3 and the MRE $\breve{e}_3$:

$$e_3 = a\Big(\big((\,1 \mid 1 \mid 1 \mid 1\,)^+\big)^+\Big)^+ b$$

$$\breve{e}_3 = a_1\,{}_2(\,\Big[\,{}_3(\,\Big[\,{}_4(\,[\,1_5 \mid 1_6 \mid 1_7 \mid 1_8\,]^+)_4\Big]^+\,)_3\Big]^+\,)_2\,b_9$$

Then, it is $h = 3$ and $n_\varepsilon = 4$, and $C''\,(a_1,\, b_9)$ counts many elements, such as the following:

$$\langle\, a_1,\, {}_2(\,{}_3(\,{}_4(\,1_6\,)_4\,)_3\,{}_3(\,{}_4(\,1_5\,)_4\,)_3\,)_2,\, b_9\,\rangle$$

$$\langle\, a_1,\, {}_2(\,{}_3(\,{}_4(\,1_8\,)_4\,{}_4(\,1_5\,)_4\,)_3\,{}_3(\,{}_4(\,1_7\,)_4\,)_3\,)_2,\, b_9\rangle$$

$$\ldots$$

We remark that large values for the term C'' computed through Eq. (12) arise in the contrived case of the REs that have a high number n_ε of symbols of type 1_j that are deeply nested within the iteration operators. In Sect. 4 we show that in the practical cases, where the number of marked symbols 1 is limited or null, the transducer size grows quite slowly with the RE size, because the term C'' is small or vanishes.

To sum up, in the frequent situation when the RE contains no or few symbols 1, it holds $C\,(b,\, c) \approx h$. Then, considering the input alphabet size to be a constant, in the worst-case the transducer size $|\,A\,|$ is bounded by the sum of the nodes and arcs (each arc is weighted with the size of its output label), as follows:

$$|\,A\,| \leq |\,Q\,| + |\,\delta\,| \times \sum_{\substack{b \in \breve{\Sigma} \cup \{\vdash\} \\ c \in \breve{\Sigma} \cup \{\dashv\}}} C\,(b,\, c) = |\,Q\,| + |\,\delta\,| \times \big(\,|\,\breve{\Sigma}\,| + 1\big)^2 \times\, h \in \mathcal{O}\big(\,2^{|\,\breve{\Sigma}\,|} \times |\,\breve{\Sigma}\,|^2 \times h\,\big)$$

At last, we notice that in practice the exponential factor $2^{|\,\breve{\Sigma}\,|} = \mathcal{O}\,(\,|\,Q_{BS}\,|\,)$ from Eq. (11) is marginal, since the number of nodes of the DFA A_{BS} that are underlying transducer A is often limited.

3.5 Construction of linearized syntax tree

First, we show that the output computed by transducer A for an input string x can be interpreted as a *Directed Acyclic Graph* (DAG), denoted by $DAG\,(x)$. The DAG nodes carry as label a string of marked symbols and metasymbols. Then, we show that the labels of certain DAG paths represent the language $LST_{acyclic}\,(x)$ of Def. 5.

The transducer A accepts a string $\vdash x \in L\,(\vdash e)$ with these computations:

$$\begin{aligned} &\mathit{start} \xrightarrow{\vdash} q_0 \xrightarrow{x[1]} q_1\ \ldots\ q_{j-1} \xrightarrow{x[j]} q_j\ \ldots\ q_{n-1} \xrightarrow{x[n]} q_n && \text{if } |\,x\,| = n > 0 \\ \text{or } &\mathit{start} \xrightarrow{\vdash} q_0 && \text{if } x = \varepsilon \end{aligned} \tag{13}$$

and computes the non-empty output sequence ρ, as follows:

$$\begin{aligned} \rho\ (\mathit{start}, \vdash x) &= \rho\,(\mathit{start}, \vdash)\ \rho\,(q_0,\, x[1])\ \ldots\ \rho\,(q_{j-1},\, x[j])\ \ldots\ \rho\,(q_{n-1},\, x[n]) \\ &= \rho_1\ \ldots\ \rho_n\ \rho_{n+1} \end{aligned} \tag{14}$$

In accordance with Def. 7 and with Alg. 1, each factor ρ_j with $1 \leq j \leq n+1$, to be called a DAG *slice*, is a finite set of 3-tuples t of the form:

$$t = \langle\, b,\, \mu,\, c \,\rangle \in \left(\breve{\Sigma} \,\cup\, \{ \vdash \} \right) \times \breve{M}^* \times \left(\breve{\Sigma} \,\cup\, \{ \dashv \} \right)$$

For each slice ρ_j with $1 \leq j \leq n+1$ and for each tuple $t = \langle\, b,\, \mu,\, c \,\rangle \in \rho_j$, we define the *indexed tuple* $t_j = \langle\, b,\, \mu,\, c \,\rangle_j$ obtained by appending to t the index j as a subscript. In this way we distinguish any two identical tuples that occur in different slices. The DAG graph is next defined.

Definition 9 (DAG of a string and recognizing labeled path) The DAG of a string $x \in L\,(e)$ is a pair $\mathit{DAG}\,(x) = (V,\, E)$, where V and E are the sets of vertices and edges.

– The sets V of the *vertices* and E of the *edges* of the DAG are defined as follows:

$$\begin{aligned} V &= \left\{\, t_j \mid \quad 1 \leq j \leq n+1 \;\wedge\; t \in \rho_j \,\right\} \\ E &= \left\{\, \left(\langle\, b,\, \mu,\, c \,\rangle_j \,,\, \langle\, b',\, \mu',\, c' \,\rangle_{j+1} \right) \,\middle|\, \quad 1 \leq j \leq n \;\wedge\; c = b' \,\right\} \end{aligned} \tag{15}$$

  Set V contains all the indexed tuples t_j. A vertex $\langle \vdash,\, \mu,\, c \rangle_1$ is *initial* and a vertex $\langle\, b,\, \mu,\, \dashv \rangle_{n+1}$ is *final*. Notice that this classification is not mutually exclusive and that at least one initial and one final vertex exists. Each edge is defined by a pair of vertices.

– The label λ of a vertex $\langle\, b,\, \mu,\, c \,\rangle$ is the segment resulting from the concatenation of its second and third component: $\lambda\,(\,\langle\, b,\, \mu,\, c \,\rangle\,) = \mu\, c$.

– A *labeled path* is a sequence of one or more labeled vertices connected by edges. A labeled path from an initial vertex to a final vertex is called *recognizing*.

– The *label* of a recognizing path $t_1 \,\ldots\, t_{n+1}$ is the concatenation of the labels of its vertices: $\lambda\,(t_1 \,\ldots\, t_{n+1}) = \lambda\,(t_1) \,\ldots\, \lambda\,(t_{n+1})$. □

The labels of the DAG nodes are strings of the same type as the *segments* ζ_j of an LST, see Eq. (8). Since in Eq. (15) an edge connects two 3-tuples such that the third component of the first 3-tuple agrees with the first component of the second, every recognizing path corresponds to the computation by which the transducer A recognizes the input string x. Yet a DAG may also contain edges that do not belong to any recognizing path.

To sum up, the structure of a recognizing path is the following, with the node labels shown above the horizontal brackets:

$$\overbrace{\langle \vdash, \mu_1, a_1 \rangle_1}^{\mu_1\, a_1} \longrightarrow \overbrace{\langle\, a_1, \mu_2, a_2 \,\rangle_2}^{\mu_2\, a_2} \;\ldots\; \overbrace{\langle\, a_{n-1}, \mu_n, a_n \,\rangle_n}^{\mu_n\, a_n} \longrightarrow \overbrace{\langle\, a_n, \mu_{n+1}, \dashv \rangle_{n+1}}^{\mu_{n+1}\, \dashv} \tag{16}$$

initial node 1 — arc 1 — node 2 — nodes & arcs — node n — arc n — final node $n+1$

To illustrate, we show the output for two preceding examples.

Example 6 (DAG and recognizing paths – I) For the RE e of Eq. (3), we show in Fig. 8 (top) a computation by the transducer of Fig. 6. The input string is $a\,a\,b$. The graph $\mathit{DAG}\,(a\,a\,b)$ is partially shown in Fig. 8 (middle). All the edges in the recognizing paths are drawn as solid arrows, and only a few others as dashed arrows. Since string $a\,a\,b$ is 2-ambiguous, there are two (acyclic) LSTs, ω_1 and ω_2, and the DAG has two recognizing paths with the path labels:

$$\underbrace{1(_2(_3(a_4}_{\text{slice 1}}\;\underbrace{a_4}_{\text{slice 2}}\;\underbrace{)_3)_2)_1\, b_{11}}_{\text{slice 3}}\;\underbrace{\dashv}_{\text{slice 4}} = \omega_1 \quad \text{and} \quad \underbrace{1(_2(_3(a_4}_{\text{slice 1}}\;\underbrace{)_3\,3(_4 a_4}_{\text{slice 2}}\;\underbrace{)_3)_2)_1\, b_{11}}_{\text{slice 3}}\;\underbrace{\dashv}_{\text{slice 4}} = \omega_2$$

where the labels of the individual path nodes in the slices are highlighted. On the other hand, the path:

$$\big\langle \vdash,\ {}_1(\ {}_2(\ ,\ a_7\ \big\rangle_1 \rightarrow \langle\ a_7,\ \varepsilon,\ b_8\ \rangle_2$$

stops prematurely in the slice ρ_2 and fails to recognize the input. The non-recognizing path:

$$\big\langle \vdash,\ {}_1(\ {}_2(\ {}_3(\ ,\ a_4\ \big\rangle_1 \rightarrow \langle\ a_4,\ \varepsilon,\ a_4\ \rangle_2 \rightarrow \langle\ a_4,\ )_3,\ b_5\ \rangle_3 \rightarrow \langle\ b_5,\ \varepsilon,\ a_6\ \rangle_4$$

ends in a non-final node in the slice ρ_4. The useless nodes (dashed) can be eliminated. □

Example 7 (DAG and recognizing paths – II) Returning to the RE e_1 of Ex. 5, we depict in Fig. 9 two computations by the transducer of Fig. 7, for the (ambiguous) strings ε and a. One DAG suffices for both strings, since string ε is a prefix of string a. □

3.6 Correctness of the LST construction

We have already justified the correctness of the construction of the LST in our presentation and on the examples. Indeed, the following proof does no more than orderly collecting and formalizing previous remarks and hints. Its directness and simplicity strengthen our claim that the BSP approach is a very natural one for parsing REs.

Theorem 1 (correctness of linearized syntax tree construction) *For every string $x \in L(e)$, the set of the labels of the recognizing paths of graph DAG(x) coincides with the set LST$_{acyclic}(x)\ \dashv$ of the acyclic linearized syntax trees (see Def. 5).* □

Proof We discuss only the case $x \neq \varepsilon$, the other case being simpler. There are two parts:

part 1 – *set of the recognizing path labels of DAG$(x) \subseteq$ LST$_{acyclic}(x)\ \dashv$*

Let λ be the label of a recognizing path (Def. 9). We prove that $\lambda \in$ *LST$_{acyclic}(x)\ \dashv$*. String λ has the form shown in Eq. (16), i.e., $\lambda = \mu_1\, a_1\, \mu_2\, a_2\ \ldots\ \mu_n\, a_n\, \mu_{n+1}\ \dashv$, where each segment $\mu_i\, c$, with $1 \leq i \leq n+1$ and $c \in \breve{\Sigma} \cup \{\dashv\}$, is the label of a DAG vertex $t_i \in \rho_i$ of type $\langle \vdash, \mu_1, a_1 \rangle$ for $i = 1$, or of type $\langle a_{i-1}, \mu_i, a_i \rangle$ for $2 \leq i \leq n$, or of type $\langle a_n, \mu_{n+1}, \dashv \rangle$ for $i = n+1$. From Eq. (13), the BSP computation is:

$$\mathit{start} \xrightarrow{\vdash} q_0 \xrightarrow{x\,[1]} q_1\ \ldots\ q_{j-1} \xrightarrow{x\,[j]} q_j\ \ldots\ q_{n-1} \xrightarrow{x\,[n]} q_n$$

while from Eq. (14) the output sequence ρ is:

$$\begin{aligned}\rho\ (\mathit{start}, \vdash x) &= \rho\ (\mathit{start}, \vdash)\ \rho\ (q_0, x\,[1])\ \ldots\ \rho\ (q_{j-1}, x\,[j])\ \ldots\ \rho\ (q_{n-1}, x\,[n]) \\ &= \rho_1\ \ldots\ \rho_n\ \rho_{n+1}\end{aligned}$$

From the Def. 7 of transducer A, the output function takes the values:

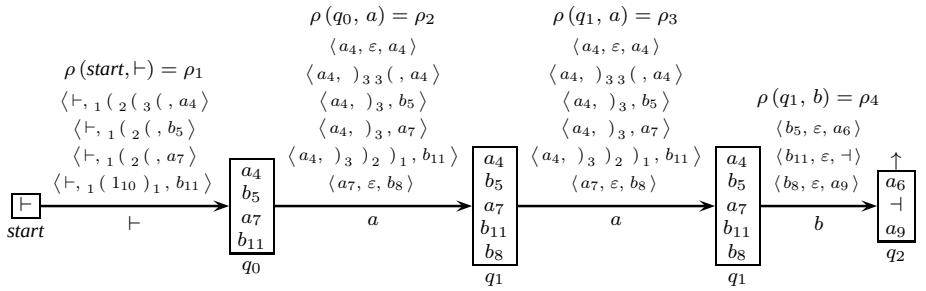
$$\begin{aligned}\rho\,(\mathit{start}, \vdash) &= \big\{\ \langle \vdash, \mu, c \rangle\ \mid\ \ c \in \Sigma\ \wedge\ \mu\, c \in \mathit{Ini}_{AS}\,(e\ \dashv)\ \big\} \\ \rho\,(q, a) &= \big\{\ \langle b, \mu, c \rangle\ \mid\ \ b \in I_a\,(q)\ \wedge\ \mu\, c \in \mathit{Fol}_{AS}\,(e\ \dashv, b)\ \big\} \text{ for all } q,\ a \in Q_{BS},\ \Sigma\end{aligned}$$

By combining the above equations, we get:

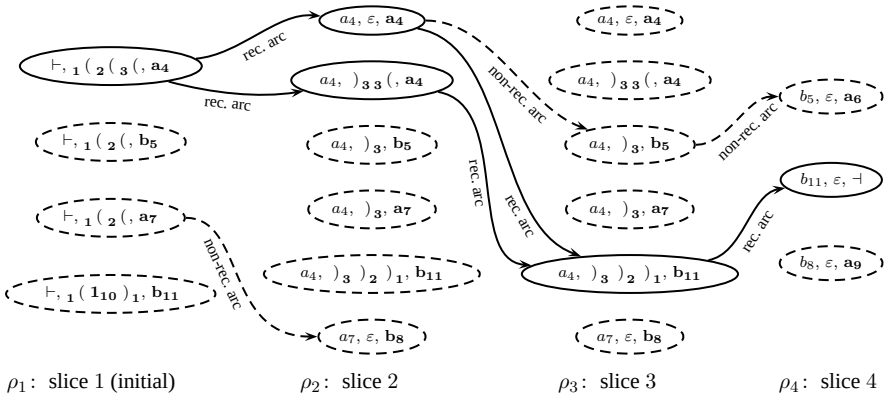
$$\begin{aligned}\mu_1\, a_1\ &=\ \mu_1\, x[1]\ \in\ \mathit{Ini}_{AS}\,(e\ \dashv) \\ \mu_i\, a_i\ &=\ \mu_i\ x[i]\ \in\ \mathit{Fol}_{AS}\,(e\ \dashv, x[i-1]) \qquad\qquad \text{for all } 2 \leq i \leq n \\ \mu_{n+1}\ \dashv\ &\in\ \mathit{Fol}_{AS}\,(e\ \dashv, x[n])\end{aligned}$$

therefore, from Def. 6, we obtain $\lambda \in$ *LST$_{acyclic}(x\ \dashv)$*.

BSP (transducer A) accepting path for the string $a\,a\,b$ with output function ρ – see Fig. 6



DAG $(a\,a\,b)$ — all the arcs in the recognizing paths (solid) and a few other arcs (dashed)



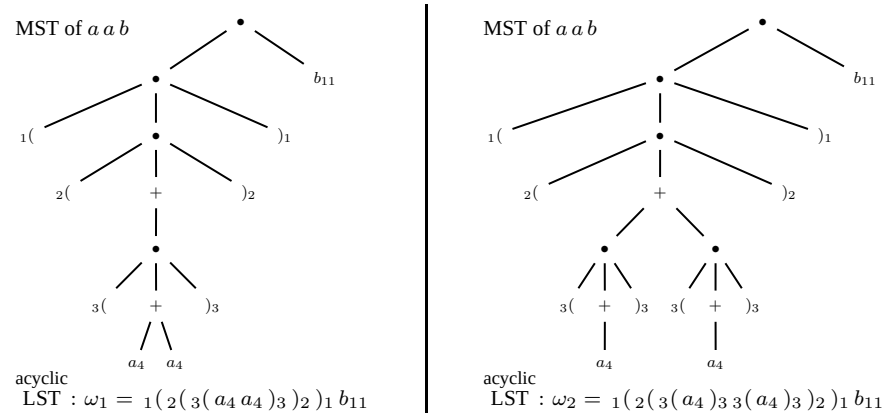


Fig. 8 LST construction of string $a\,a\,b$ for Ex. 6. Top: accepting path of transducer A. Middle: DAG of string $a\,a\,b$ with two recognizing paths – the brackets and indices of the 3-tuples are omitted and the node labels are highlighted in bold. Bottom: marked and linearized syntax trees.

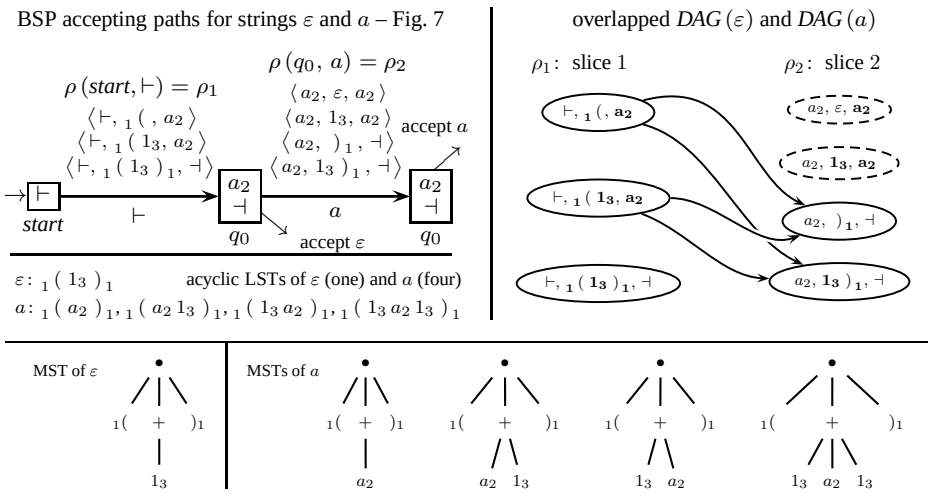


Fig. 9 LST construction for the ambiguous strings ε and a of Ex. 7 – the ambiguity of ε is not preserved.

part 2 *– LST*$_{acyclic}(x)\ \dashv \subseteq$ *set of the recognizing path labels of DAG* (x)

We prove that if the linearized syntax tree $\omega \in$ *LST*$_{acyclic}(x)\ \dashv$, then there exists a recognizing path in the graph *DAG* (x) with label λ equal to ω. From Def. 1, there is a string $x \dashv \in L(e\ \dashv)$ with *flatten* $(\omega) = x\ \dashv$. From Eq. (8), and from Def.s 5 and 6, the decomposition of ω into ASs is $\omega = \zeta_1\ \ldots\ \zeta_n\,\zeta_{n+1}$, where $\zeta_1 = \mu_1\,a_1,\ \ldots,\ \zeta_n = \mu_n\,a_n$, $\zeta_{n+1} = \mu_{n+1}\ \dashv$, with *unmark* $(a_j) = x\,[j]$ $(1 \le j \le n)$. From Def. 7, there is a transducer A for e. From Eq.s (13) and (14), the transducer A accepts string $\vdash x$ and outputs $\rho = \rho_1\ \ldots\ \rho_n\,\rho_{n+1}$. From Eq.s (15) in Def. 9, slice ρ_1 contains a 3-tuple $\langle \vdash,\ \mu_1,\ a_1 \rangle$, each slice ρ_i with $2 \le i \le n$ contains a 3-tuple $\langle a_{i-1},\ \mu_i,\ a_i \rangle$, and slice ρ_{n+1} contains a 3-tuple $\langle a_n,\ \mu_{n+1},\ \dashv \rangle$. From Eq. (16), the graph *DAG* (x) contains a recognizing path and the label λ is equal to ω. □

3.7 Complexity of parsing

It is well known that, for any RE e, the classical Berry-Sethi recognizer of language $L(e)$ (Sect. 2.2) has a linear time complexity in the length $|\,x\,|$ of the accepted string x. The same property holds for the BSP parser, including the construction of the transducer output, because the output alphabet is fixed, determined by the RE e. Since the number of DAG nodes in every slice is limited, the DAG construction is also linear in $|\,x\,|$. All the acyclic LSTs of string x can be produced by a (say, depth-first) visit of the DAG, and the building cost of each LST is linear in $|\,x\,|$, because the length of every segment is limited.

Proposition 3 (parsing complexity) *For any RE* e *and for any string* $x \in L(e)$*, the complexity of computing* $\rho\,(\mathit{start}, \vdash x)$ *is* $\mathcal{O}\left(|\,x\,|\right)$*. The complexity of computing the set of acyclic linearized syntax trees of* x *is* $\mathcal{O}\left(|\,x\,| \times |\,\mathit{LST}_{acyclic}(x)\,|\right)$.

3.8 RE ambiguity condition

In practical applications, it is sometimes necessary to know how to decide whether a given RE is ambiguous, which is a problem known to be decidable since long [6]. For the users faced with such a requirement, we show how to detect RE ambiguity on the transducer graph.

To proceed without losing generality, we need to revise the definition of acyclic linearized syntax tree, i.e., of the set $L_{acyclic}(e)$ in Def. 5. In fact, if a string $x \in L(e)$ is finitely ambiguous, the cardinality of the set $LST_{acyclic}(x)$ is greater than one, which means that the ambiguity of x remains visible when the cyclic segments are eliminated. The situation may differ in the case of a problematic RE, i.e., when the degree of ambiguity is unlimited. Although the ambiguity is usually preserved in $LST_{acyclic}(x)$ for an infinitely ambiguous string x, yet, for certain particular problematic REs, some string x, though ambiguous, has only one acyclic LST. See this RE:

$$(\,1\,)^{+} \qquad \text{marked as} \qquad {}_1(\,[\,1_2\,]^{+}\,)_1$$

where the empty string ε is ambiguous, though it is left with only one representative in the set $LST_{acyclic}(\varepsilon)$, namely "$\,{}_1(\,1_2\,)_1\,$".

If preserving ambiguity in such cases is mandatory, it suffices to extend the notions of acyclic segment AS and acyclic segment set $AS(e)$ (given in Def. 5), so as to keep the two segments "$\,{}_1(\,1_2\,)_1\,$" and "$\,{}_1(\,1_2\,1_2\,)_1\,$" in $L_{acyclic}(\varepsilon)$. But care must be taken not to jeopardize the finiteness of the set of acyclic segments. Next, we extend the definition of acyclic segment.

Definition 10 (extended acyclic segment – EAS) A *segment* $\zeta = \mu\, a_h$ or $\zeta = \mu\ \dashv$ is *extended acyclic* if and only if it holds $|\,\mu\,|_{1_j} \leq 2$ for all j, i.e., all the metasymbols of type 1 occur in μ no more than twice. Extended acyclic segments are denoted *EAS* and their set is called $EAS(e)$. □

Clearly, a comparison of Def. 5 and Def. 10 shows that it holds $AS(e) \subseteq EAS(e)$ for any RE e, and the containment is strict only for a problematic RE. The subsequent definitions of the sets $L_{acyclic}(e)$ and $LST(x)$ in Def. 5 remain unchanged, though now they use EAS instead of AS.

To prevent confusion, we qualify as *ambiguity preserving* the transducer A constructed by Alg. 1 using the EAS of Def. 10, and we denote it by A_{AP} instead of A. Notice that the state-transition graphs of A (based on $AS(e)$) and A_{AP} are structurally identical, and they only differ in the output function: that of A_{AP} contains more 3-tuples than that of A does, if (and only if) the RE is problematic. Fig. 10 shows both graphs for the problematic RE $e = (\,1\,)^{+}$, where it holds $\rho(start, \vdash) \subset \rho_{AP}(start, \vdash)$.

Therefore, the correctness proof in Th. 1 holds true unchanged for the graph $DAG(x)$ and the set $LST(x)$ of a string x accepted by transducer A_{AP}, as the proof is based on the state-transition graph. Thus the transducer A_{AP} can build all the acyclic syntax trees that the transducer A can, and a few more if (and only if) the RE is problematic.

The next ambiguity condition (Prop. 4) captures RE ambiguity without exception. This ambiguity condition is stated on transducer A_{AP} (instead of A), first for a string x generated by an RE e, then for a whole RE e (the first statement is functional to the second).

In the discussion below, for any string x, we call *clean* the graph $DAG(x)$ of x deprived of all the nodes and arcs that are not part of a recognizing path.

Proposition 4 (string and RE ambiguity) *Assume it holds* $x \in \Sigma^*$, $|\,x\,| = n \geq 0$ *and* $1 \leq i \leq n+1$. *These two statements hold, for a string and an RE, respectively:*

string *A string* $x \in L\,(e)$ *is ambiguous if, and only if, the DAG* (x)*, computed by the transducer* A_{AP} *of* e *and cleaned, has a slice* ρ_i *that contains two* 3*-tuples* $t_i = \langle\, a,\, \mu,\, b \,\rangle_i$ *and* $t'_i = \langle\, a',\, \mu',\, b' \,\rangle_i$ *with equal third symbols* $b = b'$ *(where* $b,\, b' \in \breve{\Sigma} \cup \{\dashv\}$*).*

RE *An RE* e *is ambiguous if, and only if, the state-transition graph of the transducer* A_{AP} *of* e *has an arc* γ *such that the output function* ρ *on* γ *contains two* 3*-tuples* t *and* t' *(as above but without subscript* i*) with equal third symbols* $b = b'$ *(as above).* □

Proof By Th. 1 (restated for A_{AP}), a string x has two or more LSTs in the set *LST* (x) if, and only if, the graph *DAG* (x) has as many recognizing paths. The two proofs follow:

string ***if part*** – Suppose string x is ambiguous, then it has two LSTs and its graph *DAG* (x) comprises two recognizing paths. If such paths merge, there is a slice ρ_i, not the last ($i \neq n+1$), that contains two 3-tuples t_i, t'_i that have the same successor 3-tuple t''_{i+1} in the next slice ρ_{i+1}. Therefore t_i, t'_i have equal third symbols $b = b' \in \Sigma$, since by the DAG construction such a symbol is the first of t''_{i+1}. Else, if such paths end separately, since both are recognizing, the last slice ρ_{n+1} contains two final 3-tuples t_{n+1}, t'_{n+1}, which therefore have equal third symbols $b = b' = \dashv$.
only-if part – Suppose a slice ρ_i of the cleaned *DAG* (x) of string x contains two 3-tuples t_i, t'_i. By the DAG cleanliness, the 3-tuples t_i, t'_i belong to two recognizing paths in *DAG* (x), thus string x has two LSTs and is ambiguous.
comment – The DAG cleanliness is unnecessary for the "if" part, and the equality $b = b'$ is unnecessary for the "only if" part. However, the equality is necessary for both sides of the next RE ambiguity statement.

RE ***if part*** – Suppose RE e is ambiguous, then its transducer A_{AP} recognizes an ambiguous string $\vdash x$. From the string statement (if part), the graph *DAG* (x) has a slice ρ_i that contains two 3-tuples t_i, t'_i that have equal third symbols $b = b'$. By the DAG construction, the state-transition graph of A_{AP} has an arc γ such that the output function ρ on γ is ρ_i. Thus ρ on γ contains two 3-tuples t, t', obtained from t_i, t'_i by canceling their subscript i, which therefore have equal third symbols $b = b'$.
only-if part – Suppose that the transducer A_{AP} of RE e has an arc γ, where:

$$\gamma = \left(\, q \xrightarrow{c} r \,\right) \qquad q,\, r \text{ are states of } A_{AP} \text{ and } c \in \Sigma \cup \{\vdash\}$$

and that the set $\rho\,(q,\, c)$, i.e., the output function ρ on γ, contains two 3-tuples t, t' that have equal third symbols $b = b'$. By Alg. 1, the state-transition graph of A_{AP} is trim, thus there is a string $\vdash x = \vdash x\,[1] \ldots x\,[n] \in \vdash \Sigma^*$ that labels a recognizing path of A_{AP} traversing γ. Then, there is a graph *DAG* (x) such that its slice $\rho_i = \rho\,(q,\, x\,[i-1])$ with $x\,[i-1] \in \Sigma$ for $2 \leq i \leq n+1$, or its slice $\rho_1 = \rho\,(\mathit{start}, \vdash)$ for $i = 1$ – see also Eq. (14) – contains two 3-tuples t_i, t'_i, obtained from t, t' by applying the subscript i, that have equal third symbols $b = b'$. String x can always be taken in such a way that the 3-tuples t_i, t'_i belong to recognizing paths of *DAG* (x), thus the DAG can be assumed to be *clean*, as follows:

1. By the DAG construction, the 3-tuples t_i, t'_i are necessarily *reachable* from some initial node(s) through two DAG paths, or they are themselves initial nodes.
2. If $n \geq 1$, i.e., string x is not empty, the DAG has two or more slices, therefore:

– If slice ρ_i is not the last, i.e., $i \neq n+1$, by Alg. 1 transducer A_{AP} has an arc β:

$$\beta = \left(r \xrightarrow{x\,[i]} s \right) \quad s \text{ is a state of } A_{AP} \text{ and } x\,[i] = \mathit{unmark}\,(b) = \mathit{unmark}\,(b')$$

where $\rho\,(r,\, x\,[i])$, i.e., the output function ρ on β, contains a 3-tuple t'' that has first symbol $a'' = b = b'$. Slice ρ_{i+1} is equal to $\rho\,(r,\, x\,[i])$, thus it contains a 3-tuple t''_{i+1}, into which the two DAG paths merge. If t''_{i+1} is final ($b'' = \dashv$), the two paths have *reached* – and merged into – one final node. Else, by repeating the argument from t''_{i+1} onwards, the two paths – unified from now on – will *reach* together one final node through a series of 3-tuples, i.e., DAG nodes, driven by the string suffix $x\,[i] \ldots x\,[n]$, the symbols $x\,[\cdot]$ of which are taken so as to match the classes of the third symbols b in the series of 3-tuples.

– If slice ρ_i is the last, namely ρ_{n+1}, i.e., $i = n+1$, the 3-tuples t_{n+1}, t'_{n+1} are final ($b = b' = \dashv$), thus the two DAG paths separately *reach* two final nodes.

3. If $n = 0$, i.e., string x is empty and the only slice is ρ_1, each DAG path separately *reduces* to a single node (without edges), which is both initial and final.

In all cases, the two DAG paths through the 3-tuples t_i, t'_i go from an initial node to a final node, thus they are *recognizing*. Hence the 3-tuples t_i, t'_i, which have equal third symbols $b = b'$, belong also to the *cleaned* graph $\mathit{DAG}\,(x)$. From the string statement (only-if part), string x is ambiguous, therefore RE e is ambiguous.

comment – As said, equality $b = b'$ is necessary and sufficient for RE ambiguity. □

Example 8 (RE ambiguity condition – Prop. 4) We show two cases. For the first one, consider the RE in Eq. (2). Since it is not problematic, we do not need the transducer A_{AP}, because in this case the definition of acyclic segment AS in Def. 5 preserves ambiguity. In Fig. 6, the transducer A has the arc $q_0 \xrightarrow{a} q_1$, the output of which comprises the two following 3-tuples t, t' with equal third symbols a_4 that meet Prop. 4 (RE statement):

$$\rho\,(q_0,\, a) \supset \{\ \underbrace{\langle\, a_4,\, \varepsilon,\, a_4 \,\rangle}_{\text{3-tuple } t},\ \underbrace{\langle\, a_4,\,)_3\, 3(,\, a_4 \,\rangle}_{\text{3-tuple } t'}\ \}$$

Therefore, the RE in Eq. (2) is ambiguous. In Fig. 8, string $a\,a\,b$ is a witness of such an ambiguity, with the two 3-tuples belonging to slice ρ_2 of the (clean) DAG.

The second case is such that the ambiguity of certain strings is not observable on the standard transducer A and is instead visible on the ambiguity preserving transducer A_{AP}. The RE $e_1 = (\ a\ \mid\ 1\ )^+$ in Ex. 5 ambiguously generates the empty string ε. The state-transition graph of the transducer A of e_1, constructed by Alg. 1, is in Fig. 7. The transducer A recognizes string ε by taking the arc $\mathit{start} \xrightarrow{\vdash} q_0$ with output $\rho\,(\mathit{start}, \vdash) = \{\ \langle \vdash,\ {}_1(\ 1_3\ )_1,\ \dashv \rangle\ \}$. Since such an output contains only one 3-tuple, the ambiguity of ε is not observable on A. In fact, the DAG of ε shown in Fig. 9 (top right) reduces to a single initial and final node (in slice ρ_1), which corresponds to a single syntax tree (bottom left of Fig. 9).

The ambiguity of ε is detected by the ambiguity preserving transducer A_{AP} of e_1, yet for brevity we do not construct the whole state-transition graph of A_{AP}, and we focus instead on its relevant arc and output function label. Thus we recalculate the set of initials, according to the notion of *extended acyclic segment* (EAS in Def. 10), and we denote it Ini_{EAS}:

$$\mathit{Ini}_{EAS}\,(e_1\ \dashv) = \{\ {}_1(\ a_2,\ \ {}_1(\ 1_3\, a_2,\ \ \underbrace{{}_1(\ 1_3\ )_1\ \dashv}_{\text{initial for } \varepsilon},\ \ \underbrace{{}_1(\ 1_3\, 1_3\ )_1\ \dashv}_{\text{initial for } \varepsilon}\ \}$$

where symbol 1_3 may occur up to twice (but no more) in a segment. Therefore, the ambiguity of string ε becomes visible on transducer A_{AP}, as the initial arc has to hold two 3-tuples with equal third symbol $\dashv$, coming from the last two initials (outlined). Notice that RE e_1 contains an iterated nullable subexpression, or differently said it is problematic.

On the other hand, the ambiguity of string a is already visible on the standard transducer A of Fig. 7, where Prop. 4 (RE statement) is verified on the arcs of path $\mathit{start} \xrightarrow{\vdash} q_0 \xrightarrow{a} q_0$. In fact, four recognizing paths of string a are visible in the DAG of a shown in Fig. 9, and the corresponding four syntax trees are drawn at the bottom (right) of the same figure. □

Example 9 (ambiguity preservation and detection) Fig. 10 shows both parsers A and A_{AP} for the problematic RE $e = (\,1\,)^+$, with MRE $\breve{e} = \left[\,{}_1(\,1_2\,)_1\,\right]^+$, which generates the (infinitely ambiguous) empty string ε; the DAGs of ε are also depicted for both parsers. Transducer A does not satisfy Prop. 4, whereas A_{AP} does so, as the two 3-tuples in $\rho_{AP}(\mathit{start}, \vdash)$ have equal third symbols $\dashv$ (case 3 in Prop. 4). □

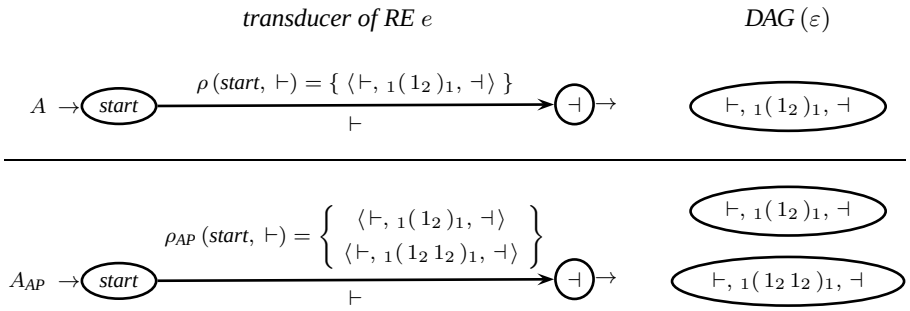


Fig. 10 Transducers A (BS Parser) and A_{AP} (ambiguity preserving) for the problematic RE $e = (\,1\,)^+$ (left), and both DAGs of ε (right) – the DAG computed by A_{AP} has two recognizing paths (each of one node).

Observation The examples above should have made clear that, to decide whether an RE is ambiguous, it suffices to inspect the transducer graph and look for any arc holding any two 3-tuples with equal third symbols. Deciding whether a single string is ambiguous requires to compute its DAG by means of the transducer, then to clean it and eventually check if a slice still contains two 3-tuples; cleaning is easily done by means of well-known algorithms.

4 Implementation and experimentation

We realized a parser generator tool that reads an RE and constructs the BSP parser, and we measured the generation process speed, the size of the generated transducer, and the BSP parsing speed. Both the generator and the produced parser are coded in Java.[1] In detail:

1. We describe the main phases composing the generator and we present the timing measurement for the generator, obtained using as input a large random collection of REs.

[1] The code is available at `https://github.com/FLC-project/BSP` together with the input data used for the experiments.

2. We present the experimental results pertaining to the BSP parser, which include the size of the transducers and the parsing speed of the generated parsers.
3. We outline an implementation option for generating a parsing algorithm that incorporates the POSIX disambiguation criterion and returns the syntax tree selected in agreement with such a criterion.

Phases of the parser generator The *generator* takes as input an RE e and returns the BSP parser. The parser consists of a fixed Java program plus some data structures, called *parsing tables*, that are specific for e and ensure a faster access by the parser code. In the first two phases the generator prepares the input data needed by Alg. 1:

1. Compute the marked abstract syntax tree MAST of e, and the set of the digrams that occur in the marked RE $\check{e}$.
2. Compute the acyclic segments $AS\,(e)$ (see Def. 5).

Then the parser construction properly starts, applies Alg. 1 and comprises two more phases:

3. Construct the transducer graph and, in particular, its output function.
4. Reformat the transducer graph and the output function into the parsing tables.

At parsing time, the parsing tables efficiently drive the parser moves. Separate time measurements for the execution of the phases from (1) to (4) are later reported.

Experimental measurements We report some experimental results about the generator and the generated parser: parser building speed, parser size and parsing speed.

A practical problem we had to face is what collection of REs to choose for our experimentation, since – to the best of our knowledge – no benchmark for measuring and comparing the performance of algorithms related to REs is publicly available. Thus we created a large collection of REs, randomly generated by another tool that we have developed in a related project [8].[2] The current collection comprises $1,000$ REs of increasing length up to 100 characters, counting the terminals and metasymbols, and it is subdivided in ten classes of length $[0-10]$, $[10-20]$, $\ldots$, $[90-100]$.

First, we report how the size of the generated BSP parser, actually the size of the transducer, depends on the length of the RE. For the transducer, we measured the number of states, arcs and 3-tuples of the output function ρ, and we computed the ratios of such numbers to the RE length, i.e., the total number of RE symbols. Then we averaged the result for each of the ten classes of REs. The ratios *num. of states / RE length* and *num. of arcs / RE length* are almost constant as the length of the RE increases (therefore the values are not reported), meaning that the DFA recognizer underlying the transducer has a linearly increasing size. On the other hand, for the output function ρ, the average number of 3-tuples per arc takes the following values:

length of the RE	10	20	30	40	50	60	70	80	90	100
num. 3-tuples / RE length of the output function ρ	0.78	0.81	0.96	1.28	1.35	2.24	4.07	5.69	6.47	8.10

From the discussion of the transducer size in Sect. 3.4, it is expected that the number of 3-tuples should grow more than linearly with respect to the RE length: in our benchmark

[2] The benchmark and generator codes are available at `https://github.com/FLC-project/BSP`.

we found the dependence to be approximately quadratic. Consequently, the parser generation speed decreases as the RE length increases. Yet, the above experimental figures support our remark in Sect. 3.4 that the combinatorial growth – expressed by Eq. (12) – for the number of 3-tuples labelling the transducer transitions, does not occur in practice. The mea-

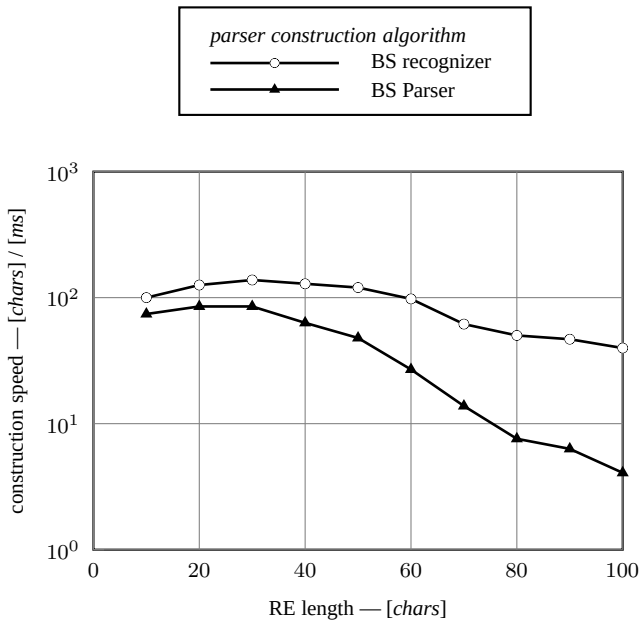


Fig. 11 Parser construction speed vs RE length – for comparison the generation speed for the pure BS recognizer is also shown (upper plot).

surements[3] in Fig. 11 show the generator speed. For comparison, the plot displays also the generation speed of the BS recognizer; the generation is faster since it does not pay the price of computing the acyclic segments AS and the derived information needed by the parser. The plots in Fig. 12 split the total parser generation time into the time spent in each phase of the generator. We found that the weight of phases (1) and (2) decreases for longer REs. The variance of all the measured distributions increases with the RE length, a fact to be expected from the presence of more heterogeneous REs within the classes of higher length.

Next, we give some figures for the parsing time, which we have obtained by running the parsers on many input texts. For each RE e in the benchmark, we randomly generated a collection of input texts, in such a way as to ensure that the RE operators in e are uniformly chosen in the generation of such texts. The parsing speed is about 43 [*chars*]/[*ms*] and it remains essentially constant for all the texts, in agreement with Prop. 3. For comparison, the recognition speed of the pure BS recognizer is about 89 [*chars*]/[*ms*], slightly less than twice the parser speed.

Disambiguation by POSIX *or other criteria* Since the BSP parser delivers all the linearized parse trees of an ambiguous text, it is not difficult to extend it in order to select the one tree

[3] On a computer AMD Athlon 64 X2 4200+ with clock 2.2 GHz and operating system Windows 10.

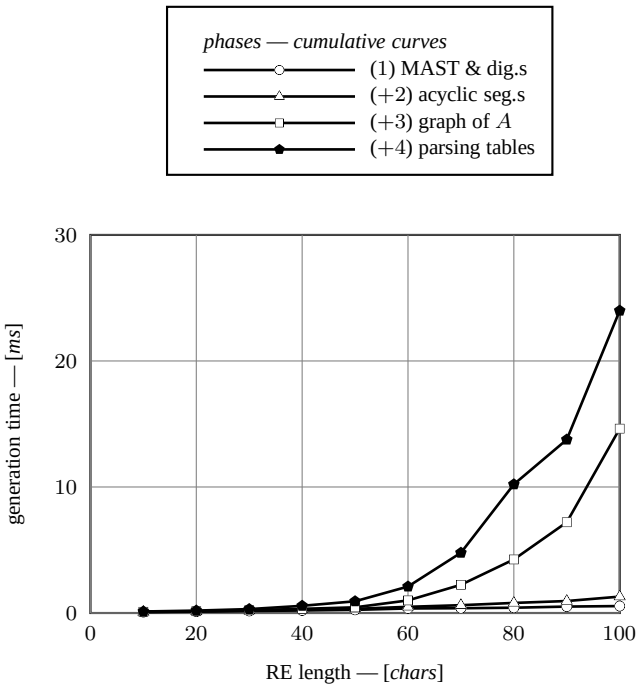


Fig. 12 Parser generation time vs RE length – averaged over each RE length class.

that meets specific disambiguation criteria, such as the POSIX criterion [15] or the *greedy* one, e.g., [11]. We implemented both, and we briefly illustrate the POSIX case.

The disambiguation algorithm we used is essentially the one described by Okui and Suzuki [20], within their parser generator. In particular, they present a method for stepwise comparing the linearized syntax trees while they are constructed, and for choosing the tree to be returned according to POSIX. We briefly explain how we applied a similar method on top of BSP. The method consists of two phases:

1. For an accepted string x, with $|\,x\,| = n$, the graph $\mathit{DAG}\,(x)$ (see Def. 9) is traversed in reverse. The nodes of $\mathit{DAG}\,(x)$ are identified and tagged, starting from the final nodes in the slice ρ_{n+1} and following backwards the paths according to the conditions defined by Eq. (15). The DAG obtained by discarding the untagged nodes, i.e., the clean DAG (see Sect. 3.8), is used in the next phase.
2. The DAG is visited in the opposite direction, by touching the slices from ρ_1 to ρ_{n+1} following the arrows depicted in Fig. 8 (middle). For every slice, a bounded amount of information is locally used to choose the node having POSIX priority; this allows the method to identify the LST of the prior tree with a time complexity linear in n.

The BSP parser that performs POSIX disambiguation is necessarily slower, because it pays the extra cost of computing the prior LST. It may be interesting to compare the speed of about 43 [*chars*]/[*ms*] without disambiguation, which – as said – is essentially independent of the RE length, against the speed with POSIX disambiguation, plotted in Fig. 13. For large REs the POSIX parser runs on average at about 20% of the BSP speed.

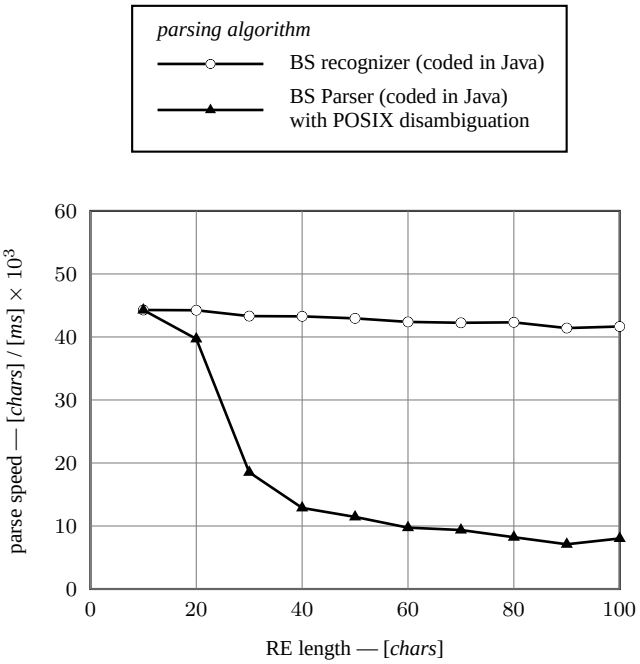


Fig. 13 Parse speed with POSIX disambiguation vs RE length – texts of $[0-1]$ MB.

Comparison with an existing library To provide a rough comparison with the performance of the existing software that supports RE parsing, we mention that the speed of BSP compares favorably with that of the popular program library RE2, as shown in Fig. 14.[4]

5 Related work

The study of ambiguous REs started very early – the original proof of the decidability of ambiguity is in [6] – but developments of practical parser generators for ambiguous REs came much later and many new algorithms have been proposed in recent years. We are going to discuss some of them, which are closer to ours.

For clarity, we recall the distinction between the algorithms for RE *matching* and RE *parsing*, which respectively mean just a recognizer of the input string or a recognizer that also outputs one (or more) syntax tree(s). Within the class of RE parsers, the algorithms differ with respect to the coverage of the syntax trees they produce: *total* versus *partial*. The algorithms that perform a partial coverage also differ from one another and produce an

[4] Since RE2 outputs one tree and is coded in C++, to offset the difference due to the programming language we implemented a version of BSP that uses POSIX disambiguation for selecting one tree and is coded in C++ as well; some experimental results are available at `https://github.com/FLC-project/BSP`. A systematic experimental comparison between existing RE parsing algorithms would be interesting, but it requires more research and presents practical difficulties. Only a few published algorithms come with well-engineered and available programs, and such programs may be coded in different languages. Moreover, the parsing process may return incomparable information on the syntax trees. Lastly, such a research has to face the problem of choosing an unbiased collection of REs as a benchmark.

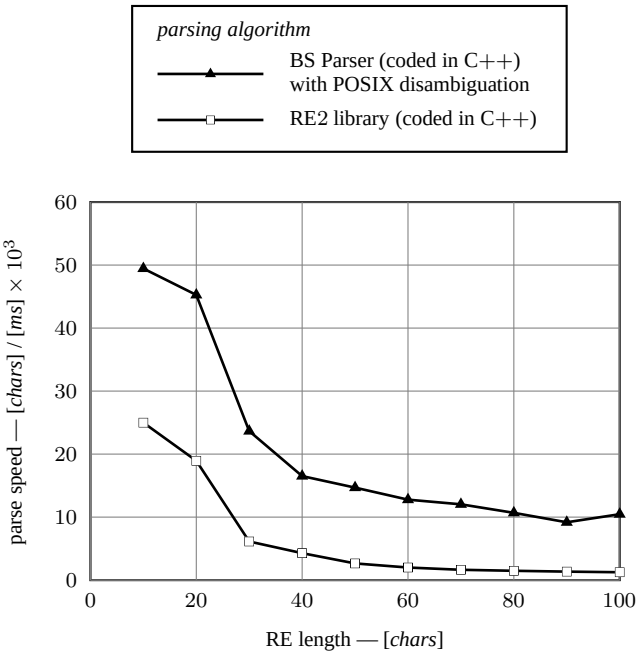


Fig. 14 Performance comparison of BSP with the RE2 library – texts of $[0-1]$ MB.

arbitrary or randomly chosen syntax tree, or the one tree that meets a precise criterion for disambiguation, chiefly the POSIX and the greedy criteria, already mentioned.

The well-known classic RE matching methods, namely McNaughton-Yamada, Gluskhov, Brzozowski, Antimirov and Thompson, translate an RE into a finite automaton. They have been later improved in many ways; in particular, the Berry-Sethi positional method, which we use as a baseline, stems from the McNaughton-Yamada and Brzozowski derivative approaches. Since the classic methods and their successors have been thoroughly analyzed and compared in the literature – see, e.g., [2, 13, 24] where a large bibliography is included – we do not discuss them, and we start by motivating our choice of Berry-Sethi as the base of our development.

Our BSP parser, like most but not all the others, uses an FA to recognize the input string, and additionally computes and outputs the syntax trees. This task is easier and faster if the internal states traversed by the FA have a direct relation to the RE syntactic structure, i.e., to its abstract syntax tree. For this goal, the so-called *positional* methods score better than the others, because each state of the FA is keyed to a set of RE positions, which correspond to positions in the RE abstract syntax tree. Moreover, such a correspondence determines at parsing time a relation between the FA states visited and the slices of the syntax tree(s) being constructed. An inspiring example of performing parsing on top of a positional recognizer occurs in the POSIX parser [20], which uses the McNaughton-Yamada method. We choose instead the Berry-Sethi method because it directly produces a DFA that is typically quite close to the minimal one; furthermore, the generation algorithm and the DFA produced have a simple formalization, if one follows the approach [4] based on local languages.

The vast literature on RE matching and parsing can be roughly categorized into:

i computational theoretic studies on the space and time needed for matching and parsing

ii parsing algorithms with different coverage of syntax trees (total vs partial)
iii RE software libraries

Since our focus is on practical and provably correct algorithms, for brevity we only discuss category (*ii*), with one exception in category (*i*), i.e., [5], but we recall that we have experimentally found that the BSP parsing speed compares favorably with the popular RE2 library. A representative list of parsing algorithms is in Tab. 3, where each one is accompanied by a

Table 3 A representative chronological list of algorithms for parsing ambiguous REs.

authors and reference	*algorithm description and comments*
S. Kearns [16]	It is based on a recursive procedure, which delivers the sequence of the states that are traversed by an NFA; then the sequence is converted into a parse tree; it does not state the tree selection criterion.
D. Dubé and M. Feeley [10]	It finds all the parse trees that match a DFA computation; it uses the sequence of DFA states to find a path in an NFA graph, then it emits the parse tree as transduction; it does not state the tree selection criterion.
V. Laurikari [17]	It relies on an NFA, which performs a quasi-POSIX matching in linear-time; it hints at a method to transform such an NFA into a DFA.
A. Frisch and L. Cardelli [11]	It gives a rigorous definition of the *greedy* criterion; it makes a first backward pass, by using an NFA and saving the list of traversed states; then it makes a forward pass, in which it builds the prior tree.
S. Okui and T. Suzuki [20]	It is based on an NFA and performs a POSIX matching; its efficient linear-time construction of the parse tree has inspired our BSP.
L. Nielsen and F. Henglein [19]	It is linear-time and produces a compact representation of the parse tree; it is a variant of *greedy* matching; it is superseded by Grathwohl paper [12].
S. Haber *et al.* [14]	It performs a *greedy* matching in a way similar to the *Java.regex* library. It scans the strings backwards and then it processes in linear-time the state list obtained.
N. Grathwohl *et al.* [12]	It performs a two-pass *greedy* parsing, based on an NFA simulation.
N. Schwarz, A. Karper, and O. Nierstrasz [21]	It performs a *greedy* matching in linear-time, based on an FA with memorization; it combines parser generation and parsing phases.
M. Sulzmann *et al.* [22, 23]	These two algorithms, based on Brzozowski derivatives, the first setting the ground for the second, generate a DFA delivering the POSIX tree in linear-time. A development for checking RE ambiguity is in [23].
P. Bille and I. Li Gørtz [5]	This theoretical study, not based on automata theory, strives to minimize the asymptotic time and space complexities; it returns *some* tree that achieves to minimize time and space.

short description, to which we add a few comments.

For the non-positional methods used for POSIX parsing, we mention the utilization of a Thompson recognizer in the parser [10]. A drawback of Thompson method is that the recognizer is non-deterministic and performs many spontaneous moves. Truly, the Thompson construction went through several optimization stages – see [13] for an account – which reduce the number of spontaneous transitions, therefore it may be promising for parser genera-

tion. Yet, to our knowledge, such upgrades have not been considered for ambiguous parsing until now.

Actually, the proposed or existing parsers also differ with respect to the output they produce, i.e., the syntax trees, and to whether the tree construction is carried out on-line or in a subsequent pass after string recognition. In contrast to the BSP parser, most existing parsers known to us produce only one syntax tree, which is either casual or the prior one according to the POSIX / greedy criterion. An example of the former case is the work by P. Bille and I. Li Gørtz [5], which focuses on the analysis of time and space complexity. One of the Sulzmann parsers [23] is able to enumerate the syntax trees of the input string. The list in Tab. 3 includes a majority of cases where the methods used for performing recognition, and especially for computing and selecting the trees, are not completely formalized.

To our knowledge, none of the existing parsers includes all the key features of our method, namely: the construction of a deterministic finite-state transducer able to output on-line a compact representation of all the syntax trees; the ability to handle *problematic* REs; the controlled exclusion of the irrelevant trees that arise in the infinitely ambiguous strings; the correctness proof of the parser; the compact representation of all the relevant syntax trees as a DAG; the predisposition of the parser to perform disambiguation by the POSIX or *greedy* criterion; the possibility of checking, directly on the transducer graph, whether a string or an RE is ambiguous; and the support by a software tool, validated experimentally.

6 Conclusion

We hope that the algorithm we have proposed and the supporting software tools and benchmark will be useful, not only for RE-based search applications, but also for other RE applications that may require adjustment or specialization. Examples of such applications are: malware detection [1], and parsing for *Extended* BNF grammars, i.e., for context-free grammars that use REs in their production rules. BSP could be possibly used as a component for such applications since its tailoring to new requirements should require less effort than other parsers would need, thanks to the simple formal structure of our parser generator.

Further developments of BSP may be considered in the future. In our time complexity analysis, we have considered the non-uniform membership problem, that is, we have not counted the RE size. Yet in some application scenarios an RE is used just for one or a few texts. In such cases, it would be interesting to minimize the complexity of the uniform membership problem by developing a parser similar to BSP, but based on a non-deterministic position automaton. Another direction is to extend the RE metalanguage, for instance to accept REs that include counting operators.

More experimental work is also planned to compare the performance of BSP and other parsers, as well as RE program libraries.

Acknowledgement To the anonymous reviewers for their valuable suggestions and references.

References

1. Aaraj, N., Raghunathan, A., Jha, N.K.: Dynamic binary instrumentation-based framework for malware defense. In: D. Zamboni (ed.) DIMVA, *LNCS*, vol. 5137, pp. 64–87. Springer (2008)
2. Allauzen, C., Mohri, M.: A unified construction of the Glushkov, Follow, and Antimirov automata. In: R. Kralovic, P. Urzyczyn (eds.) MFCS, *LNCS*, vol. 4162, pp. 110–121. Springer (2006)

3. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theor. Comput. Sci. **48**(1), 117–126 (1986)
4. Berstel, J., Pin, J.E.: Local languages and the Berry-Sethi algorithm. Theor. Comput. Sci. **155**(2), 439–446 (1996)
5. Bille, P., Gørtz, I.L.: From regular expression matching to parsing. In: P. Rossmanith, P. Heggernes, J. Katoen (eds.) MFCS, *LIPIcs*, vol. 138, pp. 71:1–71:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
6. Book, R., Even, S., Greibach, S., Ott, G.: Ambiguity in graphs and expressions. IEEE Trans. on Comp. **C-20**(2), 149–153 (1971)
7. Borsotti, A., Breveglieri, L., Crespi Reghizzi, S., Morzenti, A.: From ambiguous regular expressions to deterministic parsing automata. In: F. Drewes (ed.) CIAA, *LNCS*, vol. 9223, pp. 35–48. Springer (2015)
8. Borsotti, A., Breveglieri, L., Crespi Reghizzi, S., Morzenti, A.: A benchmark production tool for regular expressions. In: M. Hospodár, G. Jirásková (eds.) CIAA, *LNCS*, vol. 11601, pp. 95–107. Springer (2019)
9. Crespi Reghizzi, S., Breveglieri, L., Morzenti, A.: Formal Languages and Compilation, Third Edition. Texts in Computer Science. Springer (2019)
10. Dubè, D., Feeley, M.: Efficiently building a parse tree from a regular expression. Acta Inf. **37**(2), 121–144 (2000)
11. Frisch, A., Cardelli, L.: Greedy regular expression matching. In: J. Díaz, J. Karhumäki, A.Lepistö, D. Sannella (eds.) ICALP, *LNCS*, vol. 3142, pp. 618–629. Springer (2004)
12. Grathwohl, N., Henglein, F., Nielsen, L., Rasmussen, U.: Two-pass greedy regular expression parsing. In: S. Konstantinidis (ed.) CIAA, *LNCS*, vol. 7982, pp. 60–71. Springer (2013)
13. Gruber, H., Holzer, M.: From finite automata to regular expressions and back - A summary on descriptional complexity. Int. J. Found. Comput. Sci. **26**(8), 1009–1040 (2015)
14. Haber, S., Horne, W., Manadhata, P., Mowbray, M., Rao, P.: Efficient submatch extraction for practical regular expressions. In: A.H. Dediu, C. Martin Vide, B. Truthe (eds.) LATA, *LNCS*, vol. 7810, pp. 323–334. Springer (2013)
15. IEEE: std. 1003.2, *POSIX*, regular expression notation, section 2.8 (1992)
16. Kearns, S.: Extending regular expressions with context operators and parse extraction. Softw. Pract. Exper. **21**(8), 787–804 (1991)
17. Laurikari, V.: NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In: P. de la Fuente (ed.) SPIRE, pp. 181–187. IEEE Computer Society (2000)
18. McNaughton, R., Papert, S.: Counter-free automata. The MIT Press, Cambridge, MA (1971)
19. Nielsen, L., Henglein, F.: Bit-coded regular expression parsing. In: A.H. Dediu, S. Inenaga, C.M. and (eds.) LATA, *LNCS*, vol. 6638, pp. 402–413. Springer (2011)
20. Okui, S., Suzuki, T.: Disambiguation in regular expression matching via position automata with augmented transitions. In: M. Domaratzki, K. Salomaa (eds.) CIAA, *LNCS*, vol. 6482, pp. 231–240. Springer (2010)
21. Schwarz, N., Karper, A., Nierstrasz, O.: Efficiently extracting full parse trees using regular expressions with capture groups. PeerJ PrePrints **3**, e1248 (2015)
22. Sulzmann, M., Lu, K.Z.M.: *POSIX* regular expression parsing with derivatives. In: M. Codish, E. Sumii (eds.) FLOPS, *LNCS*, vol. 8475, pp. 203–220. Springer (2014)
23. Sulzmann, M., Lu, K.Z.M.: Derivative-based diagnosis of regular expression ambiguity. Int. J. Found. Comput. Sci. **28**(5), 543–562 (2017)
24. Watson, B.: A taxonomy of finite automata minimization algorithms. Report (Computing Science Notes, vol. 9343), Department of Mathematics and Computing Science - Eindhoven University of Technology, Eindhoven, The Netherlands (1993)