ACTA Informatica Manuscript-Nr.
(will be inserted by hand later)

Efficiently building a parse tree from a regular expression*

Danny Dubé and Marc Feeley**

Université de Montréal C.P. 6128, succ. centre-ville, Montréal Canada H3C 3J7

Received: — / Accepted: —

Abstract. We show in this paper that parsing with regular expressions instead of context-free grammars, when it is possible, is desirable. We present efficient algorithms for performing different tasks that concern parsing: producing the external representation and the internal representation of parse trees; producing all possible parse trees or a single one. Each of our algorithms to produce a parse tree from an input string has an optimal time complexity, linear with the length of the string. Moreover, ambiguous regular expressions can be used.

Key words: regular expression – parse tree – parsing – context-free grammar – ambiguity – time complexity

1. Introduction

In language theory, regular expressions and context-free grammars are among the most important tools used to recognize languages. These are simple models and efficient algorithms exist to make them practical. Finite deterministic and non-deterministic automata, push-down automata, $LL(k)$ and $LR(k)$ parsers are all part of this technology.

In many application fields, such as compiling, the interest is not only in recognizing languages, but also in parsing them. That is, given a grammar and a string that it generates, constructing a parse tree for the string which indicates how the string can be derived from the grammar. The task of parsing is traditionally done using context-free grammars. Still there are cases where a kind of parsing with regular expressions is desired. For example, it is the case when one wants to recover a floating point number from a string like `-1.234e5`. The string as a whole matches the regular expression describing floating point numbers, but parts of it (sign, integer and fraction, exponent)

---

* Research supported in part by the Natural Sciences and Engineering Research Council of Canada.

** E-mail: `{dube,feeley}@IRO.UMontreal.CA`.

must be extracted in order to obtain useful information. For single regular expressions, such extraction is usually done in an *ad hoc* way.

Parsing is often done using algorithms with linear complexity based on $LR(k)$ or $LL(k)$ grammars. Unfortunately, the set of regular expressions does not fall in the class of $LR(k)$ grammars[1]; ambiguous regular expressions are not $LR(k)$.

To be able to parse with any grammar, even an ambiguous one, we must use general tools such as Earley's parser ([E70]) or the dynamic programming method (see [HU79], CYK algorithm). These algorithms have cubic complexity in the worst case.

There are some tools which allow to parse specifically with regular expressions. In the REGEXP package (see [RE87]) and in the regular expression tools included in the EMACS text editor (see [Emacs]), one can mark a sub-expression of a regular expression to indicate that a substring matching the marked sub-expression is desired. The result is either the substring itself or a range (pair of integers) indicating where the characters can be found in the original string. If the sub-expression is matched many times, there is a substring for each match (or for only one of them). This facility is not completely satisfactory. It can pose problems when more than one sub-expression is marked (using '`\(`' and '`\)`'), like in:

```
(a+\(c\)?b+\(c\)?)*
```

It is not possible to unambiguously determine where the 'c's appear in the repetitions, even if all the occurrences of 'c' are noted. For example, the strings '*aacbcab*' and '*abcacbb*' both cause the same informations to be saved: one 'c' string per mark. So, extraction by marks (and by *ad hoc* techniques) is not powerful enough to provide a clean and general solution.

What we show in this paper is that complete parsing with general regular expressions can be done in linear time. We present a set of techniques that provide complete parse trees and that do not impose restrictions on the kind of regular expressions one can use. The techniques are efficient in the sense that they have a linear time complexity in the length of the string to parse. Our approach is interesting since there are many cases where regular expressions are the most natural description one has.

Indeed, although the set of regular languages is smaller than the set of $LR(k)$ languages, many useful languages are regular. Configuration files for many systems can be described by regular expressions. In some cases, assembly source files can be regular. If the assembler does not support advanced features like macros, sources files basically contain lines with an optional label, an operator and its operands, plus some assembler directives and sectioning commands. All this gives a regular language. Finally, in the processing of natural languages, regular expressions sometimes are the most appropriate tools. They can be used in various lexicon-related tasks, in robust parsing, and in automatic language identification. They can be used as an approximate definition of the language to recognize, allowing fast processing due to their regularity. In some situations, like in the processing of Finnish, where the words don't have a specific order, they are a good solution to an otherwise difficult problem (see [LJN85]). One can find more complete descriptions of the use of regular expressions in natural language processing in [MN97] and [GM89].

We present our techniques in the following order. Section 2 presents definitions and conventions that we use. Among which is the description of the parse trees themselves.

---

[1] Strictly speaking, the regular expressions are not context-free grammars. But we can give to a regular expression a natural equivalent grammar. For example, $r = r'|r''$ can be transformed into the substitution rules $T_r \to T_{r'} \mid T_{r''}$, and $r = r'^*$, into $T_r \to \epsilon \mid T_{r'} T_r$.

We give them an *external* and an *internal* representation. Section 3 presents a non-deterministic automaton used to generate the external representation of a parse tree and Section 4 presents another one for the internal representation. The second is the most useful but the first is simpler. Section 5 presents an improvement of the preceding techniques using deterministic automata. Finally, Section 6 presents a technique to obtain an implicit representation of the set of all parse trees. This technique is needed when a regular expression is ambiguous and there are many parse trees corresponding to different parses of the same string.

We present many observations and properties throughout the text. For brevity, we do not include their proof. The proofs are usually simple but long and tedious. We believe the reader should easily be able to convince himself of the truth of the statements.

2. Definitions, conventions and tools

2.1. Notation

Let Σ be the set of symbols used to write the strings that we parse. We denote the empty string as ϵ. The length of a string w is $|w|$. Strings are concatenated by writing them one after the other.

The techniques described here are based primarily on automata. The automata are simply represented by graphs. The start state is indicated by a “>” sign. The accepting states are depicted as double circles. Every edge has a string attached to it, indicating what prefix of the input string should be consumed when a transition is made through the edge. A w-transition is a transition that consumes w. Most of the automata that we introduce are non-deterministic. Their nodes are usually denoted using the letters p, q or s. For a deterministic automaton, we use a hat on the node labels (as in $\hat{p}$).

$L(r)$ is the language of the regular expression r. $L(G)$ is the language of the context-free grammar G. $L(A(r))$ is the language accepted by the automaton $A(r)$.

Paths in a graph are denoted by writing the sequence of node labels. We separate labels by dashes if there can be confusion between them. Concatenated paths are written one after the other and separated by dots ($\cdot$). In a concatenation, the last node of a path must be the same as the first node of the following path.

In general, we try to use distinct letters to denote different things; such as v and w for strings and r for regular expressions.

2.2. Regular expressions

We define the set $\mathcal{R}$ of regular expressions as the smallest set such that the following hold:

$$\begin{array}{rcl}
\mathcal{R}_{\mathrm{B}} & \supseteq & \{\epsilon\} \cup \Sigma \cup \{(r) \mid r \in \mathcal{R}_{\mathrm{E}}\} \\
\mathcal{R}_{\mathrm{F}} & \supseteq & \mathcal{R}_{\mathrm{B}} \cup \{r^* \mid r \in \mathcal{R}_{\mathrm{B}}\} \\
\mathcal{R}_{\mathrm{T}} & \supseteq & \mathcal{R}_{\mathrm{F}} \cup \{r_0 r_1 \dots r_{n-1} \mid n \geq 2 \wedge r_i \in \mathcal{R}_{\mathrm{F}}, \forall\, 0 \leq i < n\} \\
\mathcal{R}_{\mathrm{E}} & \supseteq & \mathcal{R}_{\mathrm{T}} \cup \{r_0 | r_1 | \dots | r_{n-1} \mid n \geq 2 \wedge r_i \in \mathcal{R}_{\mathrm{T}}, \forall\, 0 \leq i < n\} \\
\mathcal{R} & \supseteq & \mathcal{R}_{\mathrm{E}}
\end{array}$$

We prefer to define the set of regular expressions with set inequalities rather than with a context-free grammar because it simplifies the remaining of the presentation.

The set of regular expressions that we consider is almost the standard one. We omit the positive closure (r^+) and the optional operator ($r^?$) which are simple modifications of the Kleene closure (r^*). Note also that we did not introduce the empty regular expression ($\emptyset$), which corresponds to the empty language. First, the problem of finding a parse tree for a string w matching $\emptyset$ never occurs. Second, a complex expression containing the expression $\emptyset$ can easily be reduced to the expression $\emptyset$ itself or to an expression that does not contain $\emptyset$.

2.3. Parse trees

We first describe the kind of parse trees that our automata should create from a string. An important issue is that a regular expression may represent an ambiguous context-free grammar. The expression $(a|b|ab)*$ is ambiguous because some strings such as aab have more than one possible parse tree.

Let $\mathcal{T}$ denote the set of all possible trees and $T : \mathcal{R} \times \Sigma^* \to 2^{\mathcal{T}}$ the function giving the set of valid parse trees from a regular expression and a string. That is, $T(r, w)$ is the set of all valid parse trees coming from the decomposition of w according to the grammar represented by the regular expression r. For our purpose, parse trees are built with symbols of Σ, with lists, and with selectors. More formally, we can define $\mathcal{T}$ as the smallest set such that:

$$\begin{array}{rl} c \in \mathcal{T}, & \forall c \in \Sigma \\ \#i : t \in \mathcal{T}, & \forall i \in \mathbb{N}, \quad \forall t \in \mathcal{T} \\ [t_0, t_1, \ldots, t_{n-1}] \in \mathcal{T}, & \forall n \geq 0, \quad \forall t_i \in \mathcal{T}, \quad 0 \leq i < n \end{array}$$

Let us describe formally the parse trees in $T(r, w)$. Note that $T(r, w) \neq \emptyset$ if and only if $w \in L(r)$.

$$\begin{array}{rcl}
T(\epsilon, w) & = & \left\{ \begin{array}{ll} \{[\,]\}, & \text{if } w = \epsilon \\ \emptyset, & \text{otherwise} \end{array} \right. \\
T(c, w) & = & \left\{ \begin{array}{ll} \{c\}, & \text{if } w = c \text{ (where } c \in \Sigma) \\ \emptyset, & \text{otherwise} \end{array} \right. \\
T((r'), w) & = & T(r', w) \\
T(r'^*, w) & = & \left\{ [t_0, \ldots, t_{n-1}] \;\middle|\; \begin{array}{l} n \geq 0 \wedge \\ \forall 0 \leq i < n, \exists w_i \in \Sigma^*, \text{ s.t.} \\ w = w_0 \ldots w_{n-1} \wedge \\ t_i \in T(r', w_i), \forall 0 \leq i < n \end{array} \right\} \\
T(r_0 \ldots r_{n-1}, w) & = & \left\{ [t_0, \ldots, t_{n-1}] \;\middle|\; \begin{array}{l} \forall 0 \leq i < n, \exists w_i \in \Sigma^*, \text{ s.t.} \\ w = w_0 \ldots w_{n-1} \wedge \\ t_i \in T(r_i, w_i), \forall 0 \leq i < n \end{array} \right\} \\
T(r_0 | \ldots | r_{n-1}, w) & = & \{\#i : t_i \mid 0 \leq i < n \wedge t_i \in T(r_i, w)\}
\end{array}$$

This is the meaning of each case:

- Case $r = \epsilon$. The corresponding parse tree is the empty list: [ ].

– Case $r = c \in \Sigma$. The parse tree returned in this case is the symbol c itself.
– Case $r = (r')$. The parentheses are used only to override the priority of the operators. They do not affect the shape of the parse trees.
– Case $r = r'^*$. The parse tree returned in this case is a list containing sub-trees. That is, if r matches w, then w is a concatenation of substrings w_i all matching r', and there is one sub-tree per sub-match.
– Case $r = r_0 \ldots r_{n-1}$. A parse tree is a list of length n. Each sub-tree of that list is a parse tree of a substring w_i of w according to r_i.
– Case $r = r_0 | \ldots | r_{n-1}$. Each parse tree is a selector indicating which subexpression has permitted a match and in which way.

It is obvious that a parse tree gives a complete description of a particular match. Given the regular expression used in the match, we can interpret the parse tree and find a derivation for the string. Note that with our parse tree representation two different regular expressions both matching the same string can have the same set of parse trees. For example, $T(a^*, aaa) = T(aaa, aaa) = \{[a, a, a]\}$; in the first case, the list comes from a repetition; in the second case, the list comes from a concatenation. On the other hand, parse trees coming from two different strings must be different since a parse tree contains all the symbols of the string in order.

Here are a few examples. The last two show ambiguous regular expressions. The last one shows a regular expression for which all the strings in its language have an infinite number of parse trees.

$$\begin{array}{rcl} T(a|b|c, b) & = & \{\#1 : b\} \\ T(a^*b^*c^*, aabbbcc) & = & \{[[a, a], [b, b, b], [c, c]]\} \\ T((a|aa)^*, aa) & = & \{[\#0 : a, \#0 : a], [\#1 : [a, a]]\} \\ T((a^*)^*, \epsilon) & = & \{[\,], [[\,]], [[\,], [\,]], [[\,], [\,], [\,]], \ldots\} \end{array}$$

2.4. External representation of parse trees

Note that the parse trees in $\mathcal{T}$ are only mathematically defined and we have not given them an actual representation. They first have an external representation. That representation consists in writing them down as strings.

We choose that their textual representation is identical to the one we used in the text before. The alphabet that we use to write them is $\Sigma \cup \{\text{‘[’}, \text{‘,’}, \text{‘]’}, \text{‘\#’}, \text{‘:’}\} \cup \mathcal{D}$, where $\mathcal{D}$ is the set of digits in a certain base. For example, $\mathcal{D} = \{0, 1, 2, 3\}$ in base 4.

Lists are written as a finite sequence of elements between square brackets ([and]) and separated by commas, such as $[a, b, c]$. Selectors are written as a natural number and a parse tree, each of these being preceded by # and :, respectively, as in $\#3 : [c, a]$.

2.5. Internal representation of parse trees

The parse trees in $\mathcal{T}$ also have an internal representation. That representation consists in building trees using data structures. This section describes those data structures.

The data structures needed are: the empty list, pairs, symbols and selector-pairs. Pairs are simply the traditional two-field data structures used to build lists. However, we will use a non-traditional representation of lists: last-to-first lists. That is, the last element

of a list is immediately accessible while an access to the first requires the traversal of all the pairs of the list. This representation is unusual but it is as easy to implement as the traditional one. We use it for technical reasons explained in Section 4.2. In our representation of lists, the first field of a pair references an element of the list (the last element) and the second field references the rest of the list (the first elements). When we need to build a pair, we use the following function: *make_pair*(t_1, t_2), where t_1 is an element added at the end of the list t_2.

A selector-pair has two fields. The first contains an integer which represents the number of the alternative chosen in a match. To build a selector-pair, we use: *make_selector*(i, t).

To evaluate the complexity of our algorithms, we assume that the basic operations on these data structures can be done in constant time.

3. External representation non-deterministic automaton

This section describes how to build a non-deterministic automaton and how to use it to build the external representation of the parse trees, and explains some of its properties. The automaton is very similar to the finite non-deterministic automaton that we use in language theory to recognize a regular language. The main difference is the addition of output strings to some edges of the automaton. When the automaton makes a transition through such an edge, it outputs the string that is indicated.

We start by explaining the process of producing a parse tree using the automaton. Then we present the construction rules and an example. Next, we mention some properties related to the automaton. Finally, we discuss performance issues.

3.1. Obtaining a parse tree

Let w be generated by r and let $A_E(r)$ be the external representation automaton corresponding to r. In order to describe precisely the computations happening with our automaton, we use *configurations*. A configuration is a tuple (p, w, v) that indicates that the automaton is currently in state p, that there is still the string w to be read from the input, and that v is the string that has been output until now.

The initial configuration of the automaton is (q_0, w, ϵ), where q_0 is the start state. As with a finite state automaton, a transition can be made using an edge if: the edge starts from the current state; and, the (input) string associated with the edge is a prefix of the remaining input. So, let us assume that our automaton is in configuration (p, w_p, v_p) and there exists an edge e from p to q that consumes w' and outputs v' (if there is no output string, ϵ is the default). If w' is a prefix of w_p, that is, if there exist w_q such that $w_p = w'w_q$, then the edge can be taken. By doing so, the automaton goes into the configuration (q, w_q, v_q), where $v_q = v_p v'$.

The automaton has successfully output a parse tree if we can make transitions up to a configuration (q_f, ϵ, v), where q_f is the accepting state. In such a case, v is the desired parse tree. In cases where r is ambiguous and w has many derivations, v is only one of the possible parse trees.

3.2. Construction

Figure 1 presents the construction rules for the external representation automaton $A_E(r)$ corresponding to a regular expression r. The construction rules are described in a

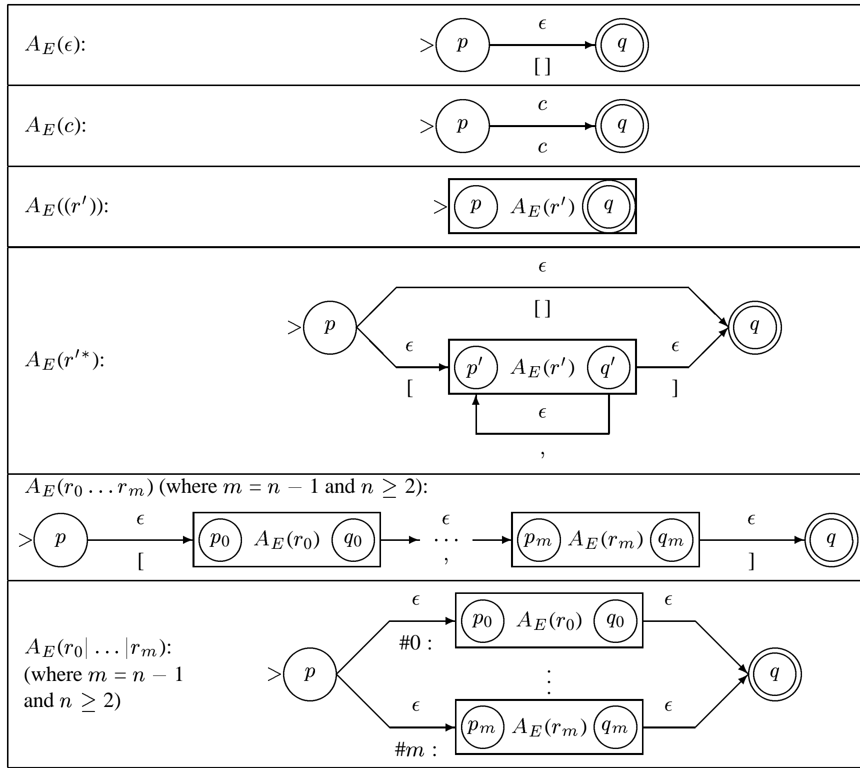


Fig. 1. Construction rules for the external representation automaton.

recursive manner. These rules are similar to those presented in [ASU86] and [HU79] with the exception that we add output strings on the edges. Note that the start and accepting states of an automaton included in a bigger one lose their special meaning.

3.3. Example

Let us see an example to help understand the way the automaton works. We will consider the regular expression $r = (a|b|ab)^*$ and the string $w = ab$. Clearly, r is ambiguous and w has more than one derivation. In fact, there are two different parse trees for w according to r:

$$T((a|b|ab)^*, ab) = \{[\#2 : [a, b]], [\#0 : a, \#1 : b]\}$$

The first corresponds to the derivation where the Kleene closure causes one repetition of the internal expression and ab is chosen among the three choices. The second corresponds to the derivation where two repetitions occur and first a, then b are chosen.

Figure 2 shows the automaton $A_E(r)$. Each path from state 0 to state 1 that consumes w will cause a parse tree to be output. One can see that there are exactly two such paths. Those are 0 – 2 – 8 – 10 – 11 – 12 – 13 – 9 – 3 – 1 and 0 – 2 – 4 – 5 – 3 – 2 – 6 – 7 – 3 – 1.

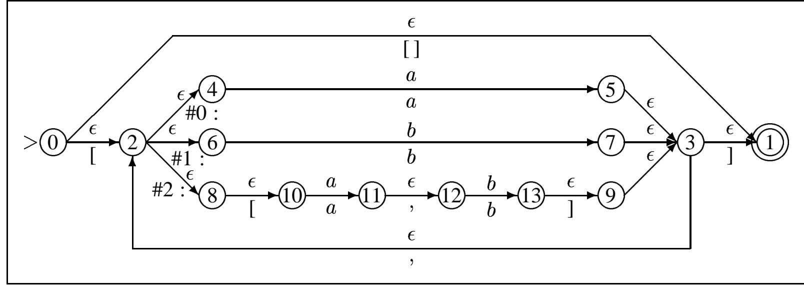


Fig. 2. The external non-deterministic automaton associated to $(a|b|ab)^*$.

First path	Second path
$(0,\ ab,\ \epsilon)$	$(0,\ ab,\ \epsilon)$
$(2,\ ab,\ [\,)$	$(2,\ ab,\ [\,)$
$(8,\ ab,\ [\#2:)$	$(4,\ ab,\ [\#0:)$
$(10,\ ab,\ [\#2:[\,)$	$(5,\ b,\ [\#0:a)$
$(11,\ b,\ [\#2:[a)$	$(3,\ b,\ [\#0:a)$
$(12,\ b,\ [\#2:[a,)$	$(2,\ b,\ [\#0:a,)$
$(13,\ \epsilon,\ [\#2:[a,b)$	$(6,\ b,\ [\#0:a,\#1:)$
$(9,\ \epsilon,\ [\#2:[a,b])$	$(7,\ \epsilon,\ [\#0:a,\#1:b)$
$(3,\ \epsilon,\ [\#2:[a,b])$	$(3,\ \epsilon,\ [\#0:a,\#1:b)$
$(1,\ \epsilon,\ [\#2:[a,b]])$	$(1,\ \epsilon,\ [\#0:a,\#1:b])$
Result: $[\#2:[a,b]]$	Result: $[\#0:a,\#1:b]$

Fig. 3. Two traces of $A_E((a|b|ab)^*)$ on ab.

Figure 3 shows the sequence of configurations that the automaton goes through for each of these paths. It is important to note in each sequence the first configuration $(0, w, \epsilon)$ and the last $(1, \epsilon, v)$.

3.4. Observations

We can make many observations about the automaton $A_E(r)$ created from r:

Correct language: The automaton accepts the same language as the one generated by the regular expression, that is: $L(A_E(r)) = L(r)$. Indeed, our method is similar to that presented in many papers (see [ASU86, HU79]).

Linear size: The size of the automaton (the number of states and edges) grows linearly with the size of the regular expression. We measure the size of r in elementary symbols: 'ϵ', 'c' ($c \in \Sigma$), '(', ')', '*' and '|'.

Soundness: The parse trees for a string output using the automaton are correct parse trees for the string. More formally, if α is a path from the start state to the accepting state of $A_E(r)$ and α consumes w, then α causes a valid parse tree $t \in T(r, w)$ to be output.

Exhaustiveness: The automaton can output any of the valid parse trees. Let $t \in T(r, w)$, then there exists a path α going from the start state to the accepting state of $A_E(r)$ and consuming w that causes t to be output.

Uniqueness: There is only one way to produce a particular tree. Let $\alpha_1 \neq \alpha_2$, two different paths traversing $A_E(r)$. Then the trees t_1 and t_2 that are output during the two traversals are different.

Note that the last three properties concern $A_E(r)$ for r in general. That is, they are true even if r is a sub-expression of another expression. It implies that a path traversing $A_E(r)$ may cause the consumption of just a part of the input string that is fed into the global automaton. Similarly, it may cause the production of just a sub-tree of the whole tree. This fact is important if one wants to prove these properties.

3.5. Complexity

We have not yet given a totally complete method to produce a parse tree. The non-deterministic automaton creates a valid parse tree as long as we have found a path that fulfills all the conditions. We describe a method to find such a path in the next paragraphs and another one in Section 5.

In order to find an appropriate path, we simply forget momentarily that there are output strings on the edges and use an algorithm simulating a non-deterministic automaton such as the one described in [HU79]. Then, by following the path and outputting the strings, we get our parse tree.

We do not repeat the description of the simulation technique here. What is important to us is its complexity. The technique finds a path in time $O(|w| \times n)$, where w is the string that we want to parse and n is the number of states of the automaton. Since we know that the number of states of the automaton grows linearly with the size of the corresponding regular expression r, the technique takes a time in $O(|w| \times |r|)$.

When we have found an appropriate path, we simply have to follow it through $A_E(r)$ and output the strings indicated on the edges. It takes $O(|w| \times |r|)$ to follow the path and to output the parse tree.

So, the whole process of writing a parse tree by simulation takes $O(|w| \times |r|)$ in time. In situations where r is known, we can consider that the technique takes $O(|w|)$ in time. Still, the hidden constant heavily depends on r.

Even if the steps of finding a path and following it both have a $O(|w| \times |r|)$ complexity, it is reasonable to think that the search for a path has a greater hidden constant in the average case. It could be profitable to get a faster algorithm for finding a valid path. Section 5 presents a method to search for a path in time $O(|w|)$, where the hidden constant does not depend on r, although r must be known a priori.

4. Internal representation non-deterministic automaton

Section 3 describes an automaton that we can use to output the external representation of a parse tree from a string generated by a regular expression. What we want to do here is to build the internal representation of a parse tree using data structures. So, instead of adding output strings to some edges of the automaton, we add construction commands. When a path traverses the automaton and consumes the whole input string, the sequence of commands that is emitted forms a *recipe* to build a parse tree.

We start by describing the instrumentation that is required to build internal parse trees. Then we present the construction rules and an example. We omit to list the properties of this automaton because they are very similar to those of the previous automaton.

4.1. Instrumentation

In order to build the internal representation of a parse tree, we augment the automaton with a *stack* and some *stack operations*.

4.1.1. The stack

The function of the stack is to contain the different pieces of the parse tree under construction. This stack does not turn the automaton into a push-down automaton. The automaton can only send commands (stack operations) to it and cannot read from it.

On most edges of our automaton there are stack operations to perform. So, when a transition is made through an edge, a part of the input string may be consumed and a stack operation may be performed on the stack. The automaton is built in such a way that, if a path is followed from the start state to the accepting state and if that path has caused the consumption of the input string then the resulting effect on the stack is that a valid parse tree has been pushed on it.

Now, we define the stack functions. The three basic stack functions are *push*, *pop* and *top*.

push(t, s) returns the stack s with the new tree t added on top.
pop(s) returns s without its top element. The argument stack s is not altered by this operation.
top(s) returns the top element of the stack s.

4.1.2. The stack operations

The operations that can be found on the edges of the automata are `push`, `snoc`, and `sel`. `Push` adds a tree on top of the stack. The `push` operator always has a constant tree as first argument. For example, `push [ ]` is an operator that takes a stack and pushes the empty list on top of it. `Snoc`[2] takes the top two elements of the stack, groups them in a pair and places the result back on top of the stack. One can see that we can build arbitrarily long lists on the stack with successive `push` and `snoc` operators. `Sel` takes the top element of the stack and encapsulates it in a selector. The `sel` operator always has a constant integer as first argument.

$$
\begin{aligned}
(\texttt{push}\ t)(s) &= \mathit{push}(t, s) \\
\texttt{snoc}(s) &= \mathit{push}(\mathit{make_pair}(\mathit{top}(s), \mathit{top}(\mathit{pop}(s))), \mathit{pop}(\mathit{pop}(s))) \\
(\texttt{sel}\ i)(s) &= \mathit{push}(\mathit{make_selector}(i, \mathit{top}(s)), \mathit{pop}(s))
\end{aligned}
$$

4.2. Construction

Figure 4 presents the construction of the internal representation non-deterministic automaton. The function A_I returns the internal automaton associated to a regular expression.

[2] This name is the reverse of `cons`, which is a common name for the function that adds an element in front of a list. We use `snoc` in reference to our reversed implementation of lists.

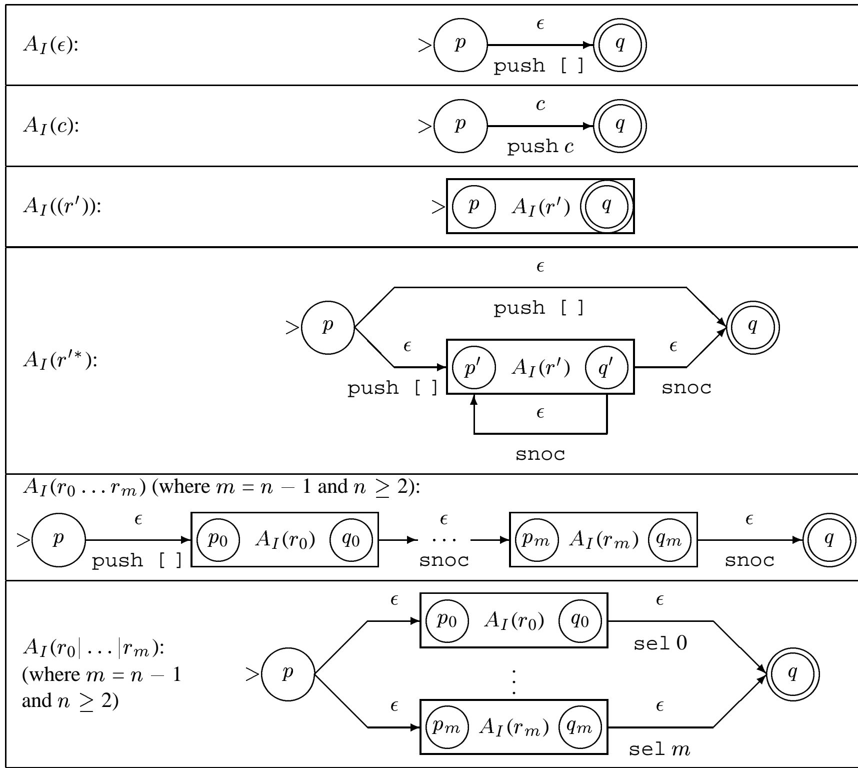


Fig. 4. Construction rules for the internal representation automaton.

The rules for $r = r'^*$ and $r = r_0 \ldots r_{n-1}$ are those involved in the creation of lists. It is because of these rules and the fact that $A_I(r)$ reads its input from left to right (i.e. normally) that we require a special implementation of lists (see Section 2.5). It would be possible to use only traditional lists if we changed the two rules mentioned above and made the automaton read its input backwards (from right to left).

4.3. Example

We illustrate the operation of the automaton with the same regular expression and string as in the example for the external representation automaton: $r = (a|b|ab)^*$ and $w = ab$. Figure 5 shows $A_I(r)$.

There are two paths that consume w: 0 – 2 – 8 – 10 – 11 – 12 – 13 – 9 – 3 – 1 and 0 – 2 – 4 – 5 – 3 – 2 – 6 – 7 – 3 – 1. Figure 6 shows the sequence of configurations that the automaton goes through along each path. The first path is particularly illustrative of the importance of using the reversed implementation of lists.

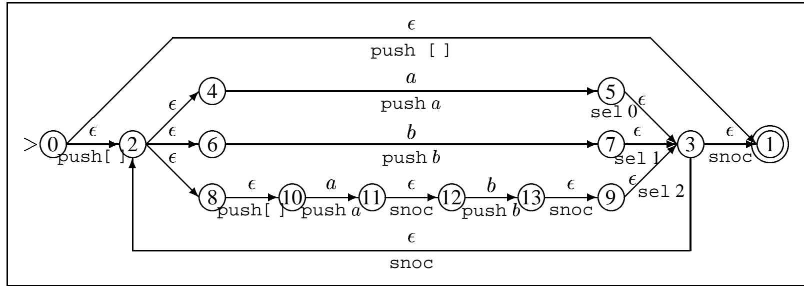


Fig. 5. The internal non-deterministic automaton associated to $(a|b|ab)^*$.

First path			Second path		
(0,	ab,	[ ])	(0,	ab,	[ ])
(2,	ab,	[[ ]])	(2,	ab,	[[ ]])
(8,	ab,	[[ ]])	(4,	ab,	[[ ]])
(10,	ab,	[[ ], [ ]])	(5,	b,	$[a, [\,]])$
(11,	b,	$[a, [\,], [\,]])$	(3,	b,	$[\#0 : a, [\,]])$
(12,	b,	$[[a], [\,]])$	(2,	b,	$[[\#0 : a]])$
(13,	ϵ,	$[b, [a], [\,]])$	(6,	b,	$[[\#0 : a]])$
(9,	ϵ,	$[[a, b], [\,]])$	(7,	ϵ,	$[b, [\#0 : a]])$
(3,	ϵ,	$[\#2 : [a, b], [\,]])$	(3,	ϵ,	$[\#1 : b, [\#0 : a]])$
(1,	ϵ,	$[[\#2 : [a, b]]])$	(1,	ϵ,	$[[\#0 : a, \#1 : b]])$
Result: $[\#2 : [a, b]]$			Result: $[\#0 : a, \#1 : b]$		

Fig. 6. Two traces of $A_I((a|b|ab)^*)$ on ab.

5. Deterministic automaton

As explained in previous sections, we expect the search for a valid path in a non-deterministic automaton to be quite expensive. So we develop here a deterministic equivalent to the non-deterministic automaton generated for a regular expression r. We arbitrarily choose to give the explanations using the internal automaton, but these also apply to the external automaton.

The algorithm using the deterministic automaton can find a valid path in time $O(|w|)$. Note the path itself has generally a length in $O(|r| \times |w|)$. The reason why it takes less time to find the path than it would take to *output* it is because the algorithm only creates a *skeleton* of the real path. It only returns the identity of numerous sub-paths. The concatenation of those sub-paths would give the desired path.

5.1. Construction

Basically, the construction of the deterministic automaton is almost identical to the usual construction used in language theory ([ASU86, HU79]), so we will not rewrite it here. The difference resides in the fact that we collect some information about the relation between the deterministic automaton and the non-deterministic one. We explain what is the nature of this information, but we do not formally describe the way it should be computed. We assume it is sufficiently straightforward.

We must keep two matrices of information, which we denote as functions here:

$f(\hat{p}, c, q)$ where $\hat{p}$ is a deterministic state (corresponding to a set of non-deterministic states), $c \in \Sigma$ and q a non-deterministic state. The function f returns a path α from some state p in $\hat{p}$ to the state q that goes through a c-transition first, and then, through zero or more ϵ-transitions. It is easy to identify p because it is the first state in α.

$g(q)$ where q is a non-deterministic state. The function g returns a path α from the non-deterministic start state to q that goes through ϵ-transitions only.

In order to describe the usage of the automaton, we first introduce some variables. We have a string $w = c_0 \ldots c_{n-1}$, where $c_i \in \Sigma$ for $0 \leq i \leq n-1$. p_{start} and q_{acc} are the start state and the accepting state of the non-deterministic automaton $A_I(r)$, respectively. The deterministic start state is $\hat{p}_{start}$. The deterministic states are distinguished by the hat they have on their name. Naturally, $\hat{p}_{start}$ is the ϵ-closure of p_{start}.

The first step in using the deterministic automaton consists in processing the input string w with it. One has to note each state in which the automaton goes into. Let us call $\hat{p}_0, \ldots, \hat{p}_n$ those states. Note that $p_{start} \in \hat{p}_{start} = \hat{p}_0$ and $q_{acc} \in \hat{p}_n$.

The second step consists in recovering the complete path traversing $A_I(r)$ and consuming w. We do this by finding $n+1$ sub-paths, α_0 to α_n, in reverse and concatenating them together. α_i consumes c_{i-1}, $1 \leq i \leq n$. α_0 starts at p_{start} and α_n ends at q_{acc}. Here is the technique:

- Initialize the tail of the path.

$$p_n = q_{acc}$$

- Find the sub-path α_i that consumes c_{i-1}.

$$\left.\begin{array}{rcl} \alpha_i & = & f(\hat{p}_{i-1}, c_{i-1}, p_i) \\ p_{i-1} & = & \text{the first state in } \alpha_i \end{array}\right\} \quad i = n, n-1, \ldots, 1$$

- Find the sub-path going to p_0.

$$\alpha_0 = g(p_0)$$

- Recover the whole path.

$$\alpha = \alpha_0 \cdot \ldots \cdot \alpha_n$$

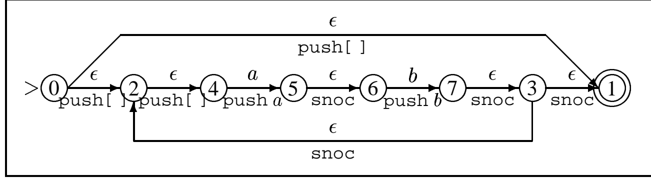
By taking care of just keeping a reference to each sub-path and not copying them entirely, we then have a skeleton of a valid path through the non-deterministic automaton in time $O(n)$ (where the hidden constant does not depend on r). Subsequently, the skeleton allows one to make a traversal of the real path in time $O(n \times |r|)$.

One might worry, with reason, about the size of the matrix for f. In the worst case, the argument $\hat{p}$ can take $2^{O(|r|)}$ values, the argument c, $|\Sigma|$ values and q, $O(|r|)$ values. Each answer can be of length $O(|r|)$. In practice, though, $\hat{p}$ takes much less than $2^{O(|r|)}$ values. Nevertheless, it is possible to re-express f in terms of three smaller matrices.[3] It is also possible to use table compression on f as it tends to be sparse and as many of its defined entries are similar. Such approaches are beyond the scope of this paper.

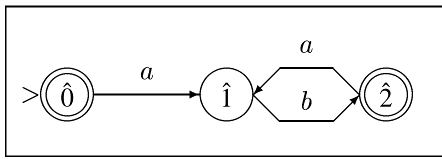
[3] Smaller in the sense of being only one- or two-dimensional, instead of three-dimensional as f.

5.2. Example

The following example illustrates the way the deterministic automaton works. Let $r = (ab)^*$ and $w = abab$. Then $A_I(r)$ is:



The deterministic automaton $\hat{A}_I(r)$ we can build from $A_I(r)$ is:



where
$$\hat{0} = \{0, 1, 2, 4\}$$
$$\hat{1} = \{5, 6\}$$
$$\hat{2} = \{1, 2, 3, 4, 7\}$$

The functions f and g associated to the deterministic automaton are:

$f =$

	$\hat{0}$		$\hat{1}$		$\hat{2}$	
	a	b	a	b	a	b
0	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	6731	⊥	⊥
2	⊥	⊥	⊥	6732	⊥	⊥
3	⊥	⊥	⊥	673	⊥	⊥
4	⊥	⊥	⊥	67324	⊥	⊥
5	45	⊥	⊥	⊥	45	⊥
6	456	⊥	⊥	⊥	456	⊥
7	⊥	⊥	⊥	67	⊥	⊥

$g =$

0	0
1	01
2	02
3	⊥
4	024
5	⊥
6	⊥
7	⊥

If we feed $\hat{A}_I(r)$ with the input string w, the automaton goes through this sequence of states:

$$\hat{p}_0 = \hat{0}, \hat{p}_1 = \hat{1}, \hat{p}_2 = \hat{2}, \hat{p}_3 = \hat{1}, \hat{p}_4 = \hat{2}$$

We recover the path α in $A_I(r)$ this way:

$$\begin{array}{lllll} \alpha_4 & = & f(\hat{p}_3, b, 1) & = & 6-7-3-1 & (p_3 = 6) \\ \alpha_3 & = & f(\hat{p}_2, a, p_3) & = & 4-5-6 & (p_2 = 4) \\ \alpha_2 & = & f(\hat{p}_1, b, p_2) & = & 6-7-3-2-4 & (p_1 = 6) \\ \alpha_1 & = & f(\hat{p}_0, a, p_1) & = & 4-5-6 & (p_0 = 4) \\ \alpha_0 & = & g(p_0) & = & 0-2-4 & \end{array}$$

$$\begin{array}{lll} \alpha & = & \alpha_0 \cdot \alpha_1 \cdot \alpha_2 \cdot \alpha_3 \cdot \alpha_4 \\ & = & 0-2-4-5-6-7-3-2-4-5-6-7-3-1 \end{array}$$

It is easy to verify that this path goes from the start state to the accepting state, consumes w and causes the push of the valid parse tree.

6. Representation of the set of parse trees

The previous sections describe ways to obtain a parse tree representing the decomposition of a string w according to a regular expression r. However, only one of the possible parse trees is built. In cases where there are more than one possible parse tree for a string, one might be interested in obtaining the complete set of trees. That is, obtaining $T(r, w)$ instead of a certain $t \in T(r, w)$.

6.1. Considerations

There are some important considerations we must take into account if we intend to get an algorithm able to return $T(r, w)$. First, it is unrealistic to try to return an *explicit* representation of $T(r, w)$. The set is, in general, too big. For example, the cardinality of $T((a^*)^*, \epsilon)$ is infinite, that of $T((a|a)^*, a^n)$ is exponential, and that of $T((a^*)^{k+1}, a^n)$ is polynomial of degree k. So, it is clear that we must return an *implicit* representation of $T(r, w)$.

Second, existing techniques for context-free grammars such as dynamic programming and the Earley parsing (see [HU79] and [E70]) produce an implicit representation of the set of parse trees. It is a tree-like representation: each "node" of this tree contains the set of all the different "sub-trees" that could be referenced by the node. That is, there is a node for the set of parse trees of each non-terminal and each substring of the input string. We say that the node compacts all the parse trees of a substring generated by a non-terminal (see [JM93]). Unfortunately, both algorithms have a time complexity of $O(|w|^3)$, where w is the string to parse. In particular, they still exhibit this time complexity even if we restrict the context-free grammars to be only translations of regular expressions.

Since our interest is in efficiency and since we restrict ourselves to regular expressions, we are able to present an algorithm producing an implicit representation of $T(r, w)$ in linear time. It takes the form of a context-free grammar which generates the set of parse trees. That is, the grammar $G_{r,w}$ produced is such that $L(G_{r,w}) = T(r, w)$. For the sake of simplicity, we present an algorithm producing a grammar that generates the external representation of the trees.

6.2. Construction

We describe how to produce the context-free grammar $G_{r,w}$ such that $L(G_{r,w}) = T(r, w)$. The idea behind the algorithm producing it is simple. Let $r \in \mathcal{R}$, $w \in L(r)$ and $A_E(r)$. The grammar is created in such a way that it *mimics* $A_E(r)$ consuming the input string w. That is, doing a substitution using a rule corresponds to making a transition in the automaton. Figure 7 gives the algorithm. The algorithm first produces rules simulating ϵ-transitions and c-transitions, respectively. The main non-terminal corresponds to the initial configuration. Finally, the last rule is the only one that can end a derivation, which is equivalent to recognizing the reaching of the accepting state after the consumption of the input string.

The indices of $P_{i,p}$ mean that i symbols of the input string have been consumed and that $A_E(r)$ is in state p. The set of strings that $P_{i,p}$ can generate is the set of strings that $A_E(r)$ can output if it is in state p and has already consumed the first i symbols

Make-Grammar($A_E(r), w$)
{ Suppose $w = c_0 \ldots c_{n-1}$, where $c_i \in \Sigma$ }
{ Suppose p_{start} and p_{acc} are the start and accepting states of $A_E(r)$ }
For i = 0 to n (inclusive)
  For each edge e in $A_E(r)$
    { $e = (p) \xrightarrow[w_{out}]{w_{in}} (q)$ }
    If $w_{in} = \epsilon$ Then
      Produce rule $P_{i,p} \to w_{out} P_{i,q}$
    Else If $i < n$ And $w_{in} = c_i$ Then
      Produce rule $P_{i,p} \to w_{out} P_{i+1,q}$
Mark $P_{0,p_{start}}$ as the main non-terminal
Produce rule $P_{n,p_{acc}} \to \epsilon$

Fig. 7. Algorithm producing a context-free grammar that generates an implicit representation for $T(r, w)$.

of w. Formally, here is the relation between the non-terminals of the grammar and the configurations in which $A_E(r)$ can be:

$$vP_{i,p} \text{ corresponds to } (p, w'', v) \text{ where } \exists w' \text{ s.t. } w = w'w'' \text{ and } |w'| = i$$

This relation allows us to easily obtain many properties of the grammar by translating properties of the external representation automaton.

The algorithm has a time complexity in $O(|r| \times |w|)$. The $|w|$ factor comes from the outer loop, which enumerates each position in the input string. The inner loop iterates on the edges of $A_E(r)$. Recall that the number of edges in $A_E(r)$ grows linearly with the size of r. This justifies the factor $|r|$. Since a production may be produced at each iteration of the inner loop, the algorithm generates a grammar which has $O(|r| \times |w|)$ productions.[4] Our algorithm is optimal in its time complexity in the following sense. Once the regular expression is known, the algorithm is able to generate $G_{r,w}$ in time $O(|w|)$, which is the best any algorithm can do.

The algorithm generally produces a grammar that is not clean.[5] That is, there are useless and unreachable non-terminals. We could have given an adapted algorithm that avoids this problem, but it would have been more complex and, as we point out in Section 6.5.1, it is not a serious problem.

6.3. Example

We illustrate our algorithm with a simple example. Let $r = w = abc$. Figure 8 shows $A_E(r)$. Figure 9 shows the grammar G = Make-Grammar($A_E(r), w$). In order to make things clear, we have separated the productions corresponding to the ϵ-transitions and the c_i-transitions, the final production and the main non-terminal.

[4] The fact that our algorithm creates a grammar with $O(|r| \times |w|)$ productions does not automatically imply that its time complexity is $O(|r| \times |w|)$. This is because the right hand side of the productions is not necessarily bounded in length. In particular, when w_{out} = #i :, it might be arbitrarily long. It takes $O(\log i)$ to denote i in, say, decimal digits. Nevertheless, this is not a big problem because an easy modification of the algorithm can eliminate this problem. That is, we can modify it so that it creates $O(|r| \times |w|)$ productions, each having a right hand side of at most four symbols. It is somehow technical and we won't present it here. We consider this problem solved for the remaining of the text.

[5] Some authors prefer to talk about *reduced* grammars.

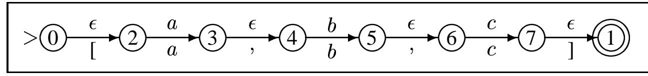


Fig. 8. The external representation automaton corresponding to the regular expression abc.

$P_{0,0}$	$\rightarrow$	$[P_{0,2}$	$P_{1,0}$	$\rightarrow$	$[P_{1,2}$	$P_{2,0}$	$\rightarrow$	$[P_{2,2}$	$P_{3,0}$	$\rightarrow$	$[P_{3,2}$
$P_{0,3}$	$\rightarrow$	$,P_{0,4}$	$P_{1,3}$	$\rightarrow$	$,P_{1,4}$	$P_{2,3}$	$\rightarrow$	$,P_{2,4}$	$P_{3,3}$	$\rightarrow$	$,P_{3,4}$
$P_{0,5}$	$\rightarrow$	$,P_{0,6}$	$P_{1,5}$	$\rightarrow$	$,P_{1,6}$	$P_{2,5}$	$\rightarrow$	$,P_{2,6}$	$P_{3,5}$	$\rightarrow$	$,P_{3,6}$
$P_{0,7}$	$\rightarrow$	$]P_{0,1}$	$P_{1,7}$	$\rightarrow$	$]P_{1,1}$	$P_{2,7}$	$\rightarrow$	$]P_{2,1}$	$P_{3,7}$	$\rightarrow$	$]P_{3,1}$
$P_{0,2}$	$\rightarrow$	$aP_{1,3}$	$P_{1,4}$	$\rightarrow$	$bP_{2,5}$	$P_{2,6}$	$\rightarrow$	$cP_{3,7}$			
Main non-terminal: $P_{0,0}$									$P_{3,1}$	$\rightarrow$	ϵ

Fig. 9. The grammar Make-Grammar($A_E(abc)$, abc).

Note that the productions corresponding to the ϵ-transitions are present for each position in the input. On the other hand, those corresponding to the d-transitions are present only for the position where the next input symbol is d. Note also the presence of useless and unreachable non-terminals. For example, $P_{0,3}$ is unreachable and $P_{3,6}$ is useless. There is only one derivation that we can make with this grammar:

$$\begin{array}{llllllllll} P_{0,0} & \rightarrow & [P_{0,2} & \rightarrow & [aP_{1,3} & \rightarrow & [a, P_{1,4} & \rightarrow & [a, bP_{2,5} \\ & \rightarrow & [a, b, P_{2,6} & \rightarrow & [a, b, cP_{3,7} & \rightarrow & [a, b, c]P_{3,1} & \rightarrow & [a, b, c] \end{array}$$

The string that is produced by the derivation is effectively the only parse tree in $T(r, w)$. Note that the path that we can extract from the derivation $(0-2-3-4-5-6-7-1)$ is a path traversing $A_E(r)$ and consuming w that outputs $[a, b, c]$.

6.4. *Observations*

The correspondence between the paths through an automaton and the derivations with the grammar allows us to obtain many interesting results quite easily.

Adequacy: Any string generated by $G_{r,w}$ is a valid parse tree of w according to r, and conversely. That is: $L(G_{r,w}) = T(r, w)$.

Unambiguousness: The grammars $G_{r,w}$ produced by our algorithm are unambiguous.

Notice that the grammars (and the languages they generate) produced by our algorithm are regular. So it should be possible to represent the same languages with regular expressions or finite automata. However, we prefer context-free grammars for two reasons. First, we have no guarantee that the smallest regular expression is as short as its corresponding context-free grammar because regular expressions do not have the sharing ability that context-free grammars have. Second, even if it is possible to produce a finite automaton as compact as its corresponding context-free grammar, we would lose in clarity due to the necessity to formally describe it.

6.5. Use of the grammar

6.5.1. Useless and unreachable non-terminals

The grammars generated by our algorithm have useless and unreachable non-terminals. The unreachable ones do not pose a real problem since they simply cause a grammar to have more productions than necessary.

The useless non-terminals are a more serious problem because they may make the generation of the set of parse trees more costly. For example, a generation of the trees by simple recursive descent would lose much time repetitively trying to generate trees and backtracking from a useless non-terminal.

Of course, one might think of an optimization consisting in noting the non-terminals that have not generated any sentence. This "optimization" could simply be replaced by a pre-processing phase of the grammar which consists in detecting and removing the useless non-terminals. An algorithm to remove those is described in [HU79]. The idea is to mark every non-terminal that is useful and then to remove the unmarked ones.

The complexity of this cleaning algorithm is in $O(L \times l)$, where L is the number of productions and l is the length of the longest right-hand side among the productions. We know that the number of productions in our grammars is in $O(|r| \times |w|)$ and that the length of the right-hand side of our productions is bounded by a constant. So the overall complexity of the algorithm is $O(|r| \times |w|)$.

So the cleaning of a grammar allows the generation of the set of parse trees using a naive recursive descent algorithm. Under the condition that there is only a finite number of trees, naturally.

6.5.2. Infinite sets of parse trees

For certain $r \in \mathcal{R}$ and $w \in L(r)$, there may be an infinite number of parse trees. In such a case, it is obviously not possible to generate all the parse trees.

Even if one is ready to enumerate as many trees as necessary to find a particular one, some care must be taken to make sure that the enumeration eventually reaches every possible tree. A grammar that has been cleaned before the search allows the search to proceed without loss of time because of useless substitutions.

Another option is to pay special attention in the design of the regular expressions in order to avoid situations where $|T(r, w)| = \infty$. This way, one can enumerate the parse trees in any order without ever risking to fall into an infinite computation.

6.5.3. Relation between the grammar and the regular expression

Unfortunately, the relation between the grammar generated by our algorithm and its regular expression is not obvious. The grammar is conceptually related to the automaton and not to the regular expression. There is no relation between a non-terminal and, say, a sub-expression of the regular expression.

This may pose a problem in applications where a sophisticated heuristic search through the set of parse trees is used to find a good one instead of brute force. In natural language parsing, for example, plausibility estimation can be done on the basis of the structure of the regular expression. But it would be difficult to adapt it to work on the basis of the paths in the automaton.

We believe that non-terminals like the following would be more intuitive. The non-terminal $Q_{r',w'}$ would generate the trees for the substring w' of w according to the sub-expression r' of r. That kind of non-terminal is closely related to the regular expression and, consequently, related to the intention inspiring the design of the regular expression. Unfortunately, a grammar based on these non-terminals would be much more expensive to generate because the number of non-terminals grows as a quadratic function of $|w|$ instead of as a linear one. Since we are interested in efficiency, we consider that an algorithm producing such a grammar is beyond the scope of this article.

6.6. Grammars for the internal representation

It may not be clear what grammars for the internal representation could be since the internal representation of a parse tree is not even a string. Nevertheless, the grammars we have described may be adapted to become an implicit representation of the internal trees.

We know that, in order to produce the internal representation of a parse tree, one must apply appropriate stack operations on a stack and take the top element. The appropriate sequences of stack operations are those prescribed by traversals of the internal representation automaton. So we can adapt the algorithm of Figure 7 to build grammars generating these sequences of stack operations. From each sequence that is generated by a grammar, one can build a parse tree.

As with the grammars for the external representation, those for the internal representation suffer from the same poor relationship with the original regular expression and the parse trees. A more intuitive representation would be one similar to the one used by the dynamic programming and Earley's parsers: one node for each sub-expression r' of r and each substring w' of w which contains the different parses of w' according to r'. Two mathematical models that could provide such representations are regular tree automata and regular tree grammars (see [B67] and [H68]). However, such representations necessarily have a quadratic number of nodes (states or non-terminals), one for each substring, and so, they cannot be produced efficiently. We consider that these approaches are beyond the scope of this article.

7. Conclusion

In this paper, we have shown that parsing with regular expressions, when it is possible, is desirable. There are efficient algorithms to do most of the tasks one might be interested in: external representation vs. internal representation; one parse tree vs. all of them. Moreover, there is no restriction on the particular regular expressions one might use.

Each of our algorithms to produce parse trees from an input string has a time complexity that is linear in the length of the string. So they are all optimal. Even the construction of non-deterministic automata is efficient (but not the construction of deterministic ones, though).

It is not possible to achieve such efficiency by simply considering a regular expression as a context-free grammar and using a parsing tool based on grammars. Indeed, some regular expressions are ambiguous and the fast algorithms based on $LL(k)$ and $LR(k)$ grammars cannot be used. Moreover, the general algorithms such as the Earley parser and the dynamic programming method show their worst case cubic time

complexity on some ambiguous grammars; some of which come directly from regular expressions.

Unfortunately, one of our results is not completely satisfactory. The grammar that we produce as the implicit representation of the set of parse trees (Section 6.5.3) is too artificial. The link between the original regular expression and the grammar representing the parse trees of a string is not natural. It might complicate the process of searching in the set for a parse tree corresponding to a certain criterion. This sacrifice seems unavoidable to have an algorithm with linear time complexity.

References

[ASU86] Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers Principles, Techniques and Tools.* Addison-Wesley. 1986

[B67] Brainerd, W.S.: *Tree generating systems and tree automata.* Ph.D. Thesis. Department of Mathematics, Purdue University. University microfilms, inc. Ann Arbor, Michigan. 1967

[E70] Earley, J.: An Efficient Context-Free Parsing Algorithm. In *Communications of the ACM.* February 1970

[Emacs] The *Emacs* text editor. Provided by Gnu: `gnu.org`

[GM89] Gazdar, G., Mellish, C.: *Natural Language Processing in PROLOG.* Addison-Wesley. 1989

[H68] Hossley, R.: *Finite-tree automata and ω-automata.* Technical report 102, Project MAC, Massachusetts Institute of Technology. Cambridge, Massachusetts. 1968

[HU79] Hopcroft, J.E., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley. 1979

[JM93] Jones, E.K., Miller, L.M.: *The L$*$ Parsing Algorithm.* Technical report CS-TR-93-9. Victoria University of Wellington. Department of Computer Science. December 1993

[LJN85] Lehtola, A., Jäppinen, H., Nelimarkka, E.: Language-based Environment for Natural Language Parsing. In *Proceedings of the 2nd Conference of the European Chapter of the Association for Computational Linguistics.* 98–106. 1985

[MN97] Mitkov, R., Nicolov, N.: *Recent Advances in Natural Language Processing.* Amsterdam, Philadelphia: John Benjamins. 1997

[RE87] Spencer, H.: *The* REGEXP *package.* A regular expression package written in C, for UNIX, available on the Internet

This article was processed by the author using the LaTeX style file *pljour1* from Springer-Verlag.