ISSN 2007-9737

Double Parsing Approach for Building Finite State Automata from Regular Expressions

Fariza Bouhatem[1,2,*], Youcef Ait el hadj[3], Fatiha Souam[1], Ali Ait el hadj[1]

[1] Mouloud Mammeri University of Tizi-Ouzou,
Algeria

[2] LARI, Computer Science Department, UMMTO, Tizi-Ouzou,
Algeria

[3] Neverhack, Paris,
France

fariza.bouhatem@ummto.dz, itcefs@gmail.com, souam_fat@yahoo.fr, aaitelhadj@ymail.com

Abstract. The work described in this article is in the field of compilation. More precisely, it concerns the lexical analysis of a compiler or a translator in general. It can also be used in the search for patterns in texts or in any other system handling regular expressions. The goal is to build a finite state automaton from a regular expression. The construction of an automaton is accomplished in two distinct steps. The first step is a parser based on operator precedence. On receiving a regular expression as input, the parser outputs the corresponding postfix representation. The postfix expression obtained is transformed into the corresponding target finite state automaton in the second step. Experiments were carried out on several regular expressions in order to evaluate and validate our proposal. The results are conclusive and largely meet our expectations.

Keywords. Finite automaton, regular expression, postfix regular expression, operator precedence grammar, transition table, transducer.

1 Basic Concepts of Language Theory

The construction of a finite state automaton can be done in several ways. First of all, it should be noted that it is possible to carry out this construction manually or automatically, depending on the objectives. Currently, the trend is rather in favor of the automation of any construction, especially with an increasing demand from the software industry, where homemade construction no longer has its place. There are several methods of constructing finite state automata, especially those based on techniques using regular expressions. Theoretically, there are also procedures for transforming a regular grammar into the equivalent finite state automaton, and vice versa.

Still, apart from the formal proof of the equivalence of the two systems (regular grammar and finite automaton), no work has been found concerning the automation of the transformation procedures from one system to another (regular grammar $\rightarrow$ finite automaton, and vice versa). The following subsections will be devoted to the notions of finite state automata, with their equivalent representation system, namely regular expressions. The latter as well as finite state automata will be among the key elements around which our work in this article will be articulated.

The rest of this article is organized as follows: Section 2 is devoted to an overview of the state of the art on regular expressions and their relation to finite state automata. Section 3 is devoted to describe our different proposals with a special emphasis on their advantages. Section 4 is dedicated to the experiment. In section 5, we compare our proposal with some methods described in the state of the art. In section

6, we conclude by recalling the main lines that characterize our contribution and we give an idea about future work.

1.1 Regular Grammar

A regular grammar is defined by a 4-tuple $G = (V_N, V_T, S, P)$ where :

— V_N is a finite and non-empty set of non-terminal symbols.

— V_T is the vocabulary (or alphabet) of the language. It consists of a finite and non-empty set of terminal symbols.

— $S \in V_N$ is the initial or start non-terminal symbol, also called axiom of grammar.

— P is a finite set of production rules defined as follows:
If it is a right-regular grammar (also called right-linear grammar) all the production rules in P are of one of the following forms:

- $A \to a$, where A is a non-terminal in V_N and a is a terminal in V_T
- $A \to aB$, where A and B are non-terminals in V_N and a is in V_T
- $A \to \epsilon$, where ϵ denotes the empty string, i.e. the string of length 0.

If it is a left-regular grammar (also called left-linear grammar), all rules obey the forms

- $A \to a$, where A is a non-terminal in V_N and a is a terminal in V_T
- $A \to Ba$, where A and B are non-terminals in V_N and a is in V_T
- $A \to \epsilon$, where A is in V_N and ϵ denotes the empty string

1.2 Finite State Automaton

A finite state automaton is formally defined by the 5-tuple $A = (S, s_0, V_T, F, I)$, where

— S is a finite and non-empty ($\neq \varnothing$) set of states

— $s_0 \in S$ is the initial state of the automaton

— $F \subseteq S$ is the set of final states

— V_T is a finite and non-empty set ($\neq \varnothing$) of terminal symbols

— I is the transition function defined by $S \times V_T \longrightarrow P(S)$. $P(S)$ represents the set of parts of S. This means that at a given state s of the automaton A and for the same transition symbol a, the transition can lead to several states (when the automaton is not deterministic). When the automaton is deterministic $P(S)$ is equal to S. The deterministic automaton will be defined in the next paragraphs.

• About the automaton represented by the transition diagram in Figure 1:

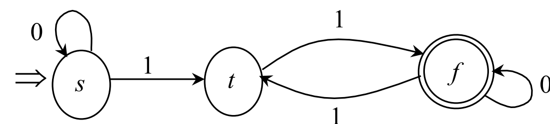


Fig. 1. Transition diagram of finite state automaton which recognizes L = $\{0^n(11)^m 0^p, n \geq 0, m \geq 1, p \geq 0\}$

— The automaton in question recognizes the language L = $\{0^n(11)^m 0^p, n \geq 0, m \geq 1, p \geq 0\}$, and it is deterministic. It is also simple because it has no ϵ-transition, i.e., $I(q, \epsilon) = \varnothing, \ \forall \ q \in S$. A finite automaton is said to be deterministic if its transition function $I(s, a)$ has no more than one member for any $s \in S$ and $a \in V_T$ [1].

— The annotated circles represent the states of the automaton, namely $S = \{s, t, f\}$.

— The initial state is prefixed with a double arrow (⇒).

— The final state f is represented by a concentric double circle.

— Words of the form 0 (11)^n 0 with $n \geq 1$, are accepted. Indeed, the automaton reaches the final state f when it has finished reading these words.

— Words 010 and 1000 are rejected by the automaton, since it blocks in the state t after reading 1.

— The words 0211 and 012 are also rejected, because 2 is not even part of the vocabulary. Indeed, on reading character 2 (of word 0211), the automaton stops operating at state s. It also stops in state t on reading 2 (of word 012).

A regular expression is a sequence of characters that define a search pattern. Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in theoretical computer science and formal language theory.

Regular expressions have been used for a long time in simple or advanced text editors (Word, TextPad, etc.). Extended regular expressions have been used extensively in systems like Lex [7] (known to be a lexical analyzer generator) and in query or navigation systems, like for example XSL-XSLT / XPath with XML[1]. They are also found in various programming systems like C, Java script, and many other computer systems. The extension of regular expressions is that new meta-characters are introduced to define new operations other than basic operations. The latter consist of Concatenation (.), Disjunction (|), and Kleene[2] closure (*).

Subsection 1.3 is devoted to the definition and description of regular expressions.

[1] XML, for Extensible Markup Language, refers to a computer language (or metalanguage to be more precise) used, among other things, in the design of websites and to facilitate the exchange of information on the Internet.

[2] In mathematical logic and computer science, the Kleene star (or Kleene operator or Kleene closure) is a unary operation, either on sets of strings or on sets of symbols or characters. It is widely used for regular expressions, which is the context in which it was introduced by Stephen Kleene to characterize certain automata, where it means "zero or more repetitions".

1.3 Regular Expressions

Before formally defining the regular expression, it is advisable to first define the regular set [1].

1.3.1 Definition 1

Let V_T a finite alphabet. A regular set over V_T is recursively defined as follows:

— $\varnothing$ (the empty set)

— $\{\epsilon\}$ is a regular set over V_T

— $\{a\}$ is a regular set over V_T for all a in V_T

— if P and Q are regular sets over V_T, then so are

- $P \cup Q$.
- PQ.
- P^*.

— Nothing else is a regular set

Thus, a subset of V_T^* is regular if and only if $\varnothing$, $\{\epsilon\}$ or $\{a\}$, for some a in V_T, or can be obtained from these by a finite number of applications of the operations union, concatenation, and closure.

1.3.2 Definition 2

Regular expressions [1] $over$ V_T and the regular sets they $denote$ are defined recursively, as follows:

— $\varnothing$ is a regular expression denoting the regular set $\varnothing$.

— ϵ is a regular set denoting the regular set $\{\epsilon\}$.

— a in V_T is a regular expression denoting the regular set $\{a\}$.

— if p and q are regular expressions denoting the regular sets P and P, respectively, then

- $(p|q)$ is a regular expression denoting $P \cup Q$.
- (pq) is a regular expression denoting PQ.
- $(p)^*$ is regular expression denoting P^*.

— Nothing else is a regular expression.

We shall use the shorthand [1] notation p^+ to denote the regular expression pp^*. Also, we shall remove redundant from regular whenever no ambiguity can arise. In this regard, we assume that * has the highest precedence, then concatenation, and then union (|). Thus, $0|10^*$ means $(0|(1(0^*)))$.

Some examples of regular expressions [1] are

1. 01, denoting 01
2. 0^*, denoting $\{0\}^*$
3. $(0|1)^*$, denoting $\{0, 1\}$
4. $(0|1)^*011$, denoting the set of all strings of 0's and 1's ending in 011.
5. $(a|b)(a|b|0|1)^*$, denoting the set of all strings in $\{0, 1, a, b\}^*$ beginning with a or b.
6. $(00|11)^*((01|10)(00|11)^*(01|10)(00|11)^*)^*$, denoting the set of all strings of 0's and 1's containing both an even number of 0's and an even number of 1's.

2 State of the Art Overview

There are several methods of constructing finite state automata. Some methods make it possible to obtain a finite automaton directly from the regular grammar [1, 5, 16]. These methods are based on formal proofs concerning the equivalence between finite automata and regular grammars.

Indeed, apart from the formal proof of the equivalence between regular grammar and finite state automaton, no work has been found concerning the automation of transformation procedures from one system to another (regular grammar finite automaton and vice versa).

It is obvious that the automaton is the most widely used and best suited tool for lexical entity recognition, especially for its ease of implementation. It is not forbidden to use a grammar, but it is not recommended. A grammar can be used, but with some disadvantages. Indeed, if the grammar is recursive, it implies that the procedure that simulates the recognition of entities by the grammar will be slow compared to the automaton.

Moreover, the method will not be general, and becomes invalid if the grammar is changed. This drawback does not affect the automaton, hence another reason for the preference of the automaton over the grammar.

Other methods instead use a regular expression to construct a deterministic finite state automaton (DFA) or a non-deterministic finite state automaton (NFA) [1].

The methods in question are practice-oriented. Indeed, they have been implemented in various computer systems, with conclusive results. As an example, the Lex (Flex[3]) [7] lexical analyzer generator has been used in the construction of many compilers.

The finite automaton remains the regular language recognition system that has been most widely used in practice. Regular expressions have the advantage of representing regular languages in a very concise way. They are also extensible or generalizable. But to be used, a regular expression must be transformed into the finite state automaton that recognizes the language it specifies.

These characteristics have been described in the section 1 concerning the fundamental notions of regular languages with their representation (regular expressions) and recognition systems (finite state automata).

Our problem lies in the transformation of regular expressions into finite state automata which is the main objective of this article. Thus, we will have to orient the state of knowledge on this objective in order to describe the way in which the construction of automata from regular expressions has been approached.

For a better conciseness, only the methods that transform a regular expression into a finite automaton will be described here. The methods that do the opposite, i.e. those that compute the regular expression from the automaton, will not be studied here, as they are not of interest for our work.

Below is given an overview of the different existing approaches to constructing an automaton from a regular expression.

[3]Flex is a free version of the Lex lexical analyzer. It is usually combined with the GNU Bison parser, the GNU version of Yacc [6].

ISSN 2007-9737

2.1 Derivative Method

The derivate method bears the name of Brzozowski derivative [15]. Janusz A. Brzozowski studied the properties of this derivative and demonstrated that the algorithm for constructing the finite automaton ends. The obtained derivatives which are distinct and in finite number correspond to the states of the constructed DFA (deterministic finite automaton).

Apart from the conciseness of its formalism and the rigor of the construction procedure, the method has remained formally manual.

Let us consider the regular expression $R = c(a^+|c)^*$ for which we want to build the automaton.

Thus, based on the derivative equations, we construct the deterministic finite automaton that recognizes the language specified by R. We will not dwell too much on this calculation which is relatively long. So, to be brief, we start the process, and we give directly the final result which is none other than the automaton described by the transition diagram in Figure 2.

The derivatives will be calculated with respect to the symbols a and c, respectively.

We set $s_0 = c(a^+|c)^*$.

– $s_0//a = c(a^+|c)^*//a = \varnothing$

– $s_0//a = c(a^+|c)^*//c = (a^+|c)^* = s_1$. Then $I(s_0, c) = s_1$, with $s_1 \in F$ because it can denote the empty string (ϵ).

We continue the calculations with s_1, then we proceed in the same way for each new derivative calculated:

– $s_1//a = (a^+|c)^*//a = (a^+|c)//a(a^+|c)^* = a^*(a^+|c)^* = s_2 \in F$, then $I(s_1, a) = s_2$

– $s_1//c = (a^+|c)^*//c = (a^+|c)//c(a^+|c)^* = (a^+|c)^* = s_3 \in F$, then $I(s_1, c) = s_3$

– A state is final if the derivative it represents can denote the empty string as is the case with s_1, s_2 and s_3.

By continuing the calculations we note that all the derivatives have been calculated. Indeed, the derivatives of s_2 by the symbols a and c give s_2 and s_3, respectively. Likewise, the derivatives of s_3 by the symbols a and c also give s_2 and s_3, respectively. The calculations are stopped because there are no new calculated derivative. By recapitulating all the calculations carried out, we obtain the automaton whose transition diagram is in Figure 2.

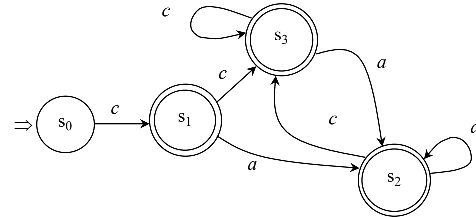


Fig. 2. Transition diagram of deterministic finite automaton built by using derivative method from the regular expression $R = c(a^+|c)^*$

This method directly results in a DFA automaton.

2.2 Method of Constructing an NFA from a Regular Expression

This method is based on Thompson's algorithm[4][9] which converts any regular expression into an NFA that recognizes the language specified by the regular expression in question.

The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression [3]. For each sub-expression the algorithm constructs an NFA with a single accepting state (final state). More precisely the automaton is built progressively using constructions for union (|), Kleene's star (*) and concatenation (.). These constructions make epsilon transitions (ϵ) appear which are then eliminated.

The following rules are depicted according to Aho et al. in [2, 3]. In what follows, N(s) and N(t) are the NFA of the sub-expressions s and t, respectively.

The empty expression ϵ and the expression a where a is a letter of the input alphabet $\{a\}$, are converted to the automata A_1 and A_2, respectively (Figures 3). The application of the other construction rules (union, concatenation

[4]Kenneth Lane Thompson is an American pioneer of computer science. Thompson worked at Bell Labs for most of his career where he designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor to the C programming language.

ISSN 2007-9737

and Klenne's star) is illustrated by the transition diagrams corresponding to automata A$_3$, A$_4$ and A$_5$, respectively (Figure 4).

Each A_j, with $j = 1$ to 5, has exactly one initial state i, which is not accessible from any other state. In other words, for any state q, and any symbol a, $I(q, a)$ does not contain i. Each A_j, with $j = 1$ to 5, has exactly one final state f which is not co-accessible from any other state. That is, for any letter a, $I(\text{f}, a) = \varnothing$.

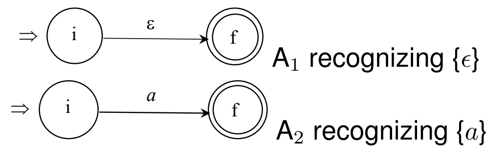


Fig. 3. Transition diagrams of automata A$_1$ and A$_2$ corresponding to the regular expressions ϵ and a, respectively

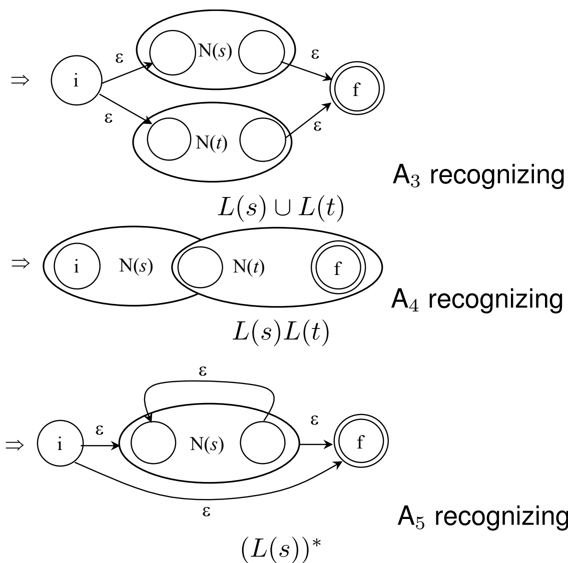


Fig. 4. Transition diagrams of automata A$_3$, A$_4$, and A$_5$ corresponding to union, concatenation, and Kleene's star, respectively

For example, let the regular expression $R = (a|b)^*c$. The syntax tree and the regular definitions corresponding to R are shown in Figure 5.

The sub-expressions r$_1$ and r$_2$ make it possible to construct the NFA in Figure 6. The intermediate

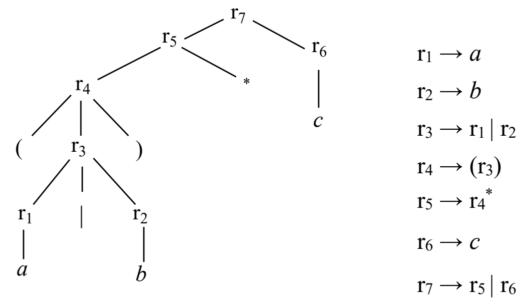


Fig. 5. The syntax tree and the regular definitions of the regular expression $R = (a|b)^*c$

constructions concerning the regular definitions r$_3$, ... r$_7$ are continued according to Thompson's algorithm until the end of the process. When all composition operations are complete, the resulting final transition diagram is shown in Figure 7.

The NFA obtained can be used by an algorithm which simulates its behavior without going through an intermediate DFA.

The NFA can also be transformed into a DFA and use the latter by an algorithm which simulates the behavior of a DFA.

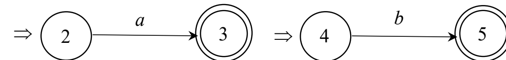


Fig. 6. Transition diagrams of NFA constructed using r$_1$ $\to a$ and r$_2$ $\to b$, respectively

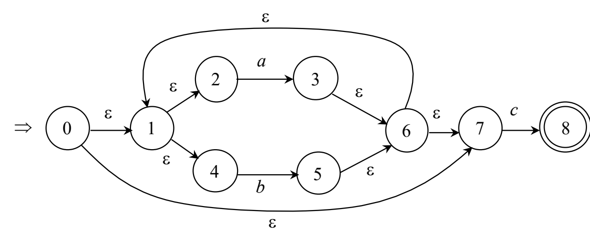


Fig. 7. NFA transition diagram which recognizes the language specified by the regular expression $R = (a|b)^*c$

Kleene's theorem [10] states that the set of regular languages over an alphabet V_T is exactly the set of languages over V_T recognizable by finite automaton. There are algorithms to go from one

to the other. Thompson's algorithm [9] allows to go from expression to automaton, as does Glushkov's construction [11].

Glushkov's algorithm allows to build a non-deterministic automaton, but with a smaller number of states compared to the one obtained by Thompson's algorithm. McNaughton and Yamada's algorithm [12] allows to go in the other direction. Kleene's algorithm transforms a given non-deterministic finite automaton (NFA) into a regular expression. Indeed, according to Gross and Yellen [14] the algorithm can be traced back to Kleene [10]. A presentation of the algorithm in the case of deterministic finite automata (DFAs) is given in Hopcroft et al. [13]. The presentation of the algorithm for NFAs below follows Gross and Yellen [14].

A complete, detailed and well structured taxonomy on the construction of finite state automata is published by Bruce W. Watson in [8]. J. Hopcroft et al. have devoted an entire book to the theory of automata [13]. There is a lot of interesting information on finite state automata, regular expressions and their properties.

2.3 Method of Constructing a DFA from a Regular Expression

It is possible to convert a regular expression to DFA by adopting an indirect method. More precisely, by first constructing an NFA by one of the methods previously mentioned in subsection 2.2. The NFA obtained is then used by the so-called subset algorithm reported in Aho et al. [2, 3]. The algorithm in question constructs the corresponding DFA.

Alternatively, it is possible to convert a regular expression to DFA directly by adopting another algorithm. The latter was reported by Aho et al. in the books [2, 3]. Hereafter are given the outlines to get an idea about the method.

To build a DFA directly from a regular expression, it is necessary to first build its abstract tree, then calculate four functions: $nullable$, $firstpos$, $lastpos$, and $followpos$, which will be defined after having first described some concepts. Using these four functions, the method's algorithm adopts almost the same approach as Glushkov's algorithm. The difference is that the latter builds a finite state automaton not necessarily deterministic and without ϵ-transitions.

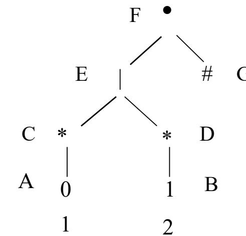


Fig. 8. Abstract tree of augmented regular expression $R = 1^*|0^*\#$

To better explain the method, let us consider the augmented regular expression
$R = 1^*|0^*\#$ whose abstract tree is shown in Figure 8. The decoration of the tree consists of the marking of its nodes as follows:
– The leaves are annotated with the capital letters A, B, ... G, and the numbers 1, 2 and 3 which represent the positions of the symbols 0, 1, and $\#$ in the extended regular expression.
– Internal nodes are labeled only by the capital letters C, D, E, and F.
– For clarity, the concatenation is represented by the symbol "•" as in Figure 8.
– The marker # is used to indicate the end of the regular expression.
The functions mentioned above, namely $followpos$, $nullable$, $firstpos$, and $lastpos$ are based on walks of the abstract tree of the augmented regular expression $R = 1^*|0^*\#$. Indeed, the functions $nullable$, $firstpos$, and $lastpos$ are defined on the nodes of the abstract tree and used to compute $followpos$ which is defined on all the positions of the expression $R = 1^*|0^*\#$. Defining $followpos$ consists in computing $followpos(i)$ by answering the question: what are the positions to be reached on a single symbol from position i?
Hereafter is shown how to calculate the values of the functions $nullable$, $firstpos$ and $lastpos$.
– $nullable(n)$ = true, if the string represented by n can generate an empty string ϵ as it is the case of $nullable$ (C) = true, $nullable$ (D) = true, and $nullable$ (E) = true.

ISSN 2007-9737

– $firstpos(n)$: the $firstpos$ value of a node n is the set of positions that correspond to those of the first symbol of a certain string derivable from the sub-expression rooted in n (of root n). For example, for nodes C, D and E, $firstpos$ (C) = {1}, $firstpos$ (D) = {2}, so $firstpos$ (E) ={1, 2}, because E = C | D.

– $lastpos(n)$: the $lastpos$ value of a node n is the set of positions that correspond to those of the last symbol of a certain string derivable from the sub-expression rooted in n. Thus, for nodes C and D, $lastpos$ (C) = {1}, and $lastpos$ (D) = {2}. Therefore $lastpos$ (E) = {1, 2}, because E = C | D.

It is advisable to specify the calculation rules used to be able to easily calculate the $followpos(i)$ sets, in particular the two fundamental rules of the Concatenation nodes, as well as the Star nodes. – Rule 1: concatenation $c_1 \bullet c_2$. If i is a position that belongs to $lastpos(c_1)$, then any position in $firstpos(c_2)$ is in $followpos(i)$.

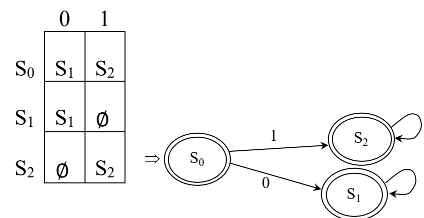
– Rule 2: star c^*. If i is a position in $lastpos$(c), then every position in $firstpos$(c) is in $followpos(i)$. Thus, the values of the $firstpos$, $lastpos$, and $followpos$ sets are given in Table 1. To conclude this state of the art it is appropriate to apply the algorithm for converting the regular expression $R = 1^*|0^*\#$ into DFA on the results summarized in Table 1. The algorithm in question can be found in the book known as the dragon book [2, 3].

Table 1. Results obtained for the $nullable$, $firstpos$, and $lastpos$ functions with the regular expression $R = 1^*|0^*\#$

node	$firstpos$	$lastpos$	$followpos$
A	{1}	{1}	{1,3}
B	{2}	{2}	{2,3}
C	{1}	{1}	–
D	{2}	{2}	–
E	{1,2}	{1,2}	–
F	{1,2,3}	{3}	–
G	{3}	{3}	–

To start the construction process, the initial state represented by $firstpos(root) = \{1, 2, 3\}$ is stacked. The stack is therefore not empty, so the state s_0 = {1, 2, 3} is unstacked. The latter contains the state 3 which is final, so it is both an initial and final state. In other words, the automaton recognizes the empty string epsilon (ϵ), which is consistent with $R = 0^*|1^*$. With symbol 0, $followpos(1)$ = {1, 3} = s_1. Therefore, $I(s_0, 0) = s_1$. The state s_1 is also a final state. $s_1 = \{1, 3\}$ is stacked because it is not yet marked with symbol 1, $followpos(2) = \{2, 3\} = s_2$. Therefore, $I(s_0, 1) = s_2$. The state s_2 is also a final state. $s_2 = \{2, 3\}$ is stacked because it is not yet marked, $followpos(3) = \varnothing$ The stack is still not empty, the stack top which is equal to {2, 3} is unstacked and it is marked as follows: with symbol 1, $followpos(2) = \{2, 3\} = s_2$. Therefore, $I(s_2, 1) = s_2$. The state s_2 is not stacked, because it is already marked. The stack is still not empty, so the stack top which is equal to {1, 3} is unstacked and it is marked as follows: With symbol 0, $followpos(1) = \{1, 3\} = s_2$. So, $I(s_1, 0) = s_1$. The state s_1 is not stacked because it is already marked, $followpos(3) = \varnothing$.

The stack is empty, so the computations are over. The deterministic finite automaton that recognizes the regular expression $R = 0^*|1^*$ is represented by the following table and/or associated transition diagram in Figure 9.



	0	1
S0	S1	S2
S1	S1	Ø
S2	Ø	S2

Fig. 9. Transition table and transition diagram of augmented regular expression $R = 1^*|0^*\#$

3 Construction of Finite Automata from Regular Expressions

As previously announced, our approach is two-step based. The first step is dedicated to parsing a regular expression and translating it into postfix representation. The next step is for

parsing the postfix form obtained and building the corresponding finite state automaton.

Our choice to approach the construction of automata in this way is motivated by several reasons. On the one hand, there is the practical aspect where time complexity is a determining factor. On the other hand, the analogy of regular expressions with arithmetic or logical expressions is persuasive in the use of a parser based on the precedence of operators. Note that the bottom-up parsing method based on operator precedence is known for its efficiency.

It is therefore well established that we must first define the operator precedence grammar that generates the regular expressions. Then, we will have to describe the parsing method, after having studied the precedence between different operators of the regular expressions.

3.1 Definition of the Grammar that Generates Regular Expressions

Knowing the similarity with arithmetic or logical expressions, and taking into account the precedence of operators, we can easily determine the rules of production of the grammar that generates regular expressions.

To know the order of execution of operations in a regular expression, it is necessary to define the precedence of the operators to properly structure the production rules of the grammar that generates the regular expressions. The conventional operator precedence is as follows:

— The unary operator * has highest precedence and is left associative.

— Concatenation has second highest precedence and is left associative.

— $|$ has lowest precedence and is left associative.

The unary operator representing the positive iteration + will not be added in the grammar, because $a^+ = aa^* = a^*a$; this is enough not to integrate it into the grammar. The concatenation will be represented by the bullet $\bullet$, otherwise the production rule concerning concatenation will not even have the form of an operator grammar rule. The symbol $\varnothing$ representing the void will not be part of the terminal alphabet of the grammar.

The grammar of regular expressions is represented by $G = (V_N, V_T, E, P)$ where:
– V_N is a finite and non-empty set of non-terminal symbols.
– V_T is the terminal alphabet.
– $E \in V_N$ is the initial or start non-terminal symbol of G.
– P is a finite set of production rules defined as follows:

$E \to E|T$; $E \to T$;
$T \to T{\bullet}F$; $T \to F$;
$F \to G*$; $F \to G$;
$G \to a$; $G \to \epsilon$; $G \to (E)$

Thus, $V_T = \{a, (,), \epsilon, *, \bullet, |\}$ and $V_N = \{E, T, F, G\}$. The symbol a belonging to V_T can be a letter or a digit. To simplify grammar, the symbol $\varnothing$ is not part of the alphabet V_T.

G is an operator grammar; it is even an operator precedence grammar. The precedence relations in Table 2 confirm that it is indeed an operator precedence grammar (OPG). Indeed, when there is a precedence relation between two operators, it must be unique. The uniqueness confirms that this grammar is deterministic in the sense of the bottom-up operator precedence parsing.

Table 2. Operator precedence table

	a, ϵ	$\mid$	$\bullet$	*	(	)	$
a, ϵ		$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$	$\gtrdot$
$\mid$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
$\bullet$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
*		$\gtrdot$	$\gtrdot$			$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	
)		$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$		

The relations of Table 2 also confirm that the grammar respects the precedence and the left associativity of the operators. These relations play a crucial role in triggering semantic translation actions. As an indication, the relation $\gtrdot$ suggests a reduction action, while the relations $\lessdot$ and $\doteq$ indicate a shift action. We will see the meaning of these relations in more detail in the next paragraphs.

ISSN 2007-9737

In accordance with the assigned objectives, the rest of our proposal will be split into two distinct parts: one is devoted to the compilation of regular expressions, the other is dedicated to the construction of corresponding finite automata.

3.2 Compiling a Regular Expression

The regular expression compilation pass involves both the parsing and the generation of the appropriate intermediate code. Regarding parsing, we adapt the operator precedence algorithm reported in the Dragon book [2]. The algorithm in question allows bottom-up parsing based on the shift-reduction principle.

The shift and reduction operations determine the semantic actions to be performed to translate in parallel the current regular expression into the corresponding intermediate representation.

An overview of the main intermediate representations usually used for the compilation of expressions will be given in the following subsections.
Below is a version of the pseudo code (Algorithm 3.1) of the parsing algorithm applied to the regular expression w\$. The end of the expression is followed by the symbol \$.

Algorithm 3.1: OPT PREC ALGO()

```
sp ← 1;
while (w[sp] ≠ $ or STACK[top] ≠ $)
do {
    if STACK[top] ⋖ w[sp] or
    STACK[top] ≐ w[sp]
    then { push(w[sp]);
           // * [S: Shift] * //
           sp ← sp + 1 }
    else { if STACK[top] ⋗ w[sp]
           // * [R: Reduction] * //
           then { repeat
                    pop(x);
                  until
                  STACK[top] ⋖ x }
           else error() }
}
```

Basically, this algorithm only performs parsing. But in order to complete the task of compiling a regular expression, some semantic actions will be added. These will be defined and described once we have given an idea of the different intermediate representations that exist, and chosen the one that suits us.

3.3 Choice of the Intermediate Representation

There are three intermediate forms commonly used in practice. First, $tree\ representation$, $three\ address\ code$ and the two forms of Polish [5] notation, namely, $prefix$ Polish and $postfix$ Polish (reverse Polish) representations.

We choose the reverse Polish representation for its relative simplicity and ease of handling.

3.3.1 Polish Notation

Let Θ be a set of binary (e.g., +, *), and let Σ be a set of operands. The two forms of Polish notation [1], $prefix$ Polish and $postfix$ Polish, are defined recursively as follows:

1. If an infix expression E is a single operand a, in Σ, then both the prefix Polish and postfix Polish representation of E is a.
2. If $E_1\theta E_2$ is an infix expression, where θ is an operator, and E_1 and E_2 are infix expressions, the operands of θ, then
   (a) $\theta E_1' E_2'$ is the prefix Polish representation of $E_1\theta E_2$, where E_1' and E_2' are the prefix Polish representation of E_1 and E_2, respectively, and
   (b) $E_1'' E_2 \theta''$ is the postfix Polish representation of $E_1\theta E_2$, where E_1'' and E_2'' are the postfix Polish representation of E_1 and E_2, respectively.
3. If (E) is an infix expression, then
   (a) The prefix Polish representation of (E) is the prefix Polish representation of E, and

[5] The term Polish is used, as this notation was first described by the Polish Mathematican Lukasiewicz [1].

ISSN 2007-9737

(b) The postfix Polish representation of (E) is the postfix Polish representation of E.

For example, consider the infix expression $(a|b){\bullet}c$. This expression is of the form $E_1{\bullet}E_2$, where $E_1 = (a|b)$ and $E_2 = c$. Thus the prefix and postfix Polish expressions for E_2 are both c. The prefix expression for E_1 is the same as that for $a|b$, which is $|ab$. Thus the prefix expression for $(a|b){\bullet}c$ is ${\bullet}|abc$.

Similarly, the postfix expression for $a|b$ is $ab|$, so the postfix expression for $(a|b){\bullet}c$ is $ab|c{\bullet}$.

This notation is also valid for unary operators. Indeed, it suffices to eliminate one of the operands, E_1 or E_2.

As previously announced, and in accordance with the actions mentioned by comments ($\boxed{R\text{: Reduction}}$ and $\boxed{S\text{: Shift}}$) in the pseudo-code of the Algorithm 3.1 the semantic actions are described as follows.

3.3.2 Description of Semantic Translation Actions

The semantic actions identified are the reduction R and the shift S defined as follows.

The reduction R which consists of the semantic routines $R_a, R_\epsilon, R_|, R_\bullet$, and R^*. The abbreviation S consists of the semantic action shift.

— R_a is used to generate the code for operand a.

— R_ϵ is used to generate the code for operand ϵ.

— $R_|$ is applied to generate the code for the union of two operands.

— $R_\bullet$ is applied to generate the code for the concatenation of two operands.

— R^* is the action applied to generate the code corresponding to the Kleene iteration (*). Recall that * is a left associative unary operator with the highest precedence.

— The shift action S indicates to stack the current symbol of the input expression and move to the next symbol of the current sub-expression.

Below are some explanations concerning the parsing algorithm (Algorithm 3.1) and the semantic actions listed above.

— First of all, it should be noted that the algorithm in question follows the shift-reduce principle. The shift-reduce logic is dictated by the different relations ($\lessdot$, $\doteq$, and $\gtrdot$).

— As previously indicated, the string w\$ is the input regular expression to be parsed and translate. Initially sp is positioned on the first symbol of w \$.

— The semantic action S allows the following two operations:

  - shifts the symbol pointed to by sp on the stack;
  - advance sp on the next symbol as input.

Thus, after the application of action S, the symbol $w[ps]$ is stacked and becomes the new top of the stack. This stack is called STACK in the Algorithm 3.1.

Semantic actions R ($R_a, R_\epsilon, R_|, R_\bullet$, R^*) perform the required reduction as explained below.

— $\boxed{S}$ is the shift action, which consists of

1. Stacking the current symbol in the stack named STACK in the algorithm; and
2. Reading the next input symbol.

— $\boxed{R_a}$ is the abbreviation of the reduction of symbol a.

1. The symbol a is then stacked in an output semantic stack (not to be confused the latter with the stack named STACK in the algorithm).
2. The semantic action R_a returns as a result a pointer to the top of the stack where the operand a is stacked.

— $\boxed{R_\epsilon}$ is the abbreviation of the reduction of symbol ϵ.

ISSN 2007-9737

1. The symbol ϵ is then stacked in an output semantic stack (not to be confused the latter with the stack named STACK in the algorithm).
2. The semantic action R_ϵ returns as a result a pointer to the top of the stack where the operand ϵ is stacked.

— $\boxed{R_\bullet}$ corresponds to the semantic routine that triggers actions: primitive stack operation Pop, and concatenation $\bullet$. As this is the postfix notation, the operations are executed as follows:

1. Pop(opd_2); Pop(opd_1);
2. Concat('opd_1', 'opd_2', '$\bullet$'); Push the result of the concatenation; this gives $\boxed{opd_1 opd_2 \bullet}$ in the stack. This result can be sent as output.

— $\boxed{R_|}$ corresponds to the semantic routine that triggers actions: primitive stack operation Pop, and union $|$. As this is the postfix notation, the operations are executed as follows:

1. Pop(opd_2); Pop(opd_1);
2. Concat('opd_1', 'opd_2', '|'); Push the result of the union; this gives $\boxed{opd_1 opd_2 |}$ in the stack. This result can be sent as output.

— $\boxed{R^*}$

1. Pop(opd); we unstack only once because * is a unary operation.
2. Concat('opd', '*'); Push the result of the star operation; this gives $\boxed{opd*}$ in the stack. This result can be sent as output.

These actions show how the postfix intermediate representation of a regular expression $w\$$ presented as input to the algorithm is generated. These semantic actions can be easily modified if we want to generate other intermediate representations such as the *tree representation* or the *three-address code*.

Concerning the handling of errors, the operator precedence table does not cover certain errors. In effect, apart from the errors listed in the table (see blanks entries in Table 2), the parsing algorithm based on the precedence of operators is known not to detect other errors. The latter are due to the precedence relations between certain operators, as shown by the expressions $\boxed{a|\bullet a}$, $\boxed{a||a*}$, $\boxed{a\bullet\bullet a}$, $\boxed{a\bullet * a}$, etc. Indeed, these expressions are obviously incorrect, but there is a precedence between some of their operators which allows them to be accepted temporarily.

But this obviously presents a drawback for the rest of the work. To remedy this defect, a simple preprocessor based on a finite state automaton (Table 3) makes it possible to detect this type of error. The automaton in Table 3 indicates how the symbols of a well-formed regular expression should follow each other.

States 0, 1, and 2 indicate how the symbols of an expression follow each other. State 3 indicates the final state to stop parsing. Thus, the error case mentioned in the algorithm becomes unnecessary in the implementation.

Table 3. Input states in regular expressions

	|	•	a, ϵ	*	(	)	$
0			1		0		
1	0	0		2		1	3
2	0	0				1	3
3							

This automaton (Table 3) is considered as a filter that prevents an erroneous expression from being processed. Thus, on the basis of this automaton and the relations of the operator precedence table (Table 2), the intermediate code (postfix representation) will be returned as output. It is the latter that will be used afterwards to build the finite automaton that recognizes the language specified by the regular expression presented as input.

Before starting the next step, it is necessary to show how the algorithm works on a simple example with the three operators $|$, *, and $\bullet$, as well as the parentheses (,).

Let $r = (a^*|a)^* \bullet a$ be the regular expression, to convert. But to optimize the parser, the ambiguous grammar (Table 4) is used because it has fewer production rules. The ambiguity of the grammar does not generate any problem since the grammar

which generates the regular expressions defined above is an operator precedence grammar which always allows a deterministic bottom-up parsing.

Table 4. Syntax-directed translation scheme with ambiguous grammar which generates regular expressions

Production rules	Semantic rules
$E \to E \vert E$	EE '|'
$E \to E \bullet E$	EE '•'
$E \to E*$	E '*'
$E \to (E)$	E
$E \to a$	a
$E \to \epsilon$	ϵ

Hereinafter the syntactic analysis sequences of the regular expression $(a^*|a)^* \bullet a$ in Table 5.

Table 5. Parsing of $r = (a^*|a)^* \bullet a$ using operator precedence method, and parallel generation of the corresponding postfix representation.

Stack	In	Rel	Action	Out
$	(	$\$ \lessdot ($	shift	–
$(	a	$(\lessdot a$	shift	–
$(a$	*	$a \gtrdot *$	reduce (5)	a
$(E$	*	$(\lessdot *$	shift	–
$(E*$	|	$* \gtrdot \vert$	reduce (3)	a^*
$(E$	|	$(\lessdot \vert$	shift	–
$(E \vert$	a	$\vert \lessdot a$	shift	–
$(E \vert a$	)	$a \gtrdot)$	reduce (5)	a^*a
$(E \vert E$	)	$\vert \gtrdot)$	reduce (1)	$a^*a \vert$
$(E$	)	$(\doteq)$	shift	–
(E)	*	$) \gtrdot *$	reduce (4)	–
E	*	$\$ \lessdot *$	shift	–
$E*$	•	$* \gtrdot \bullet$	reduce (3)	$a^*a \vert ^*$
E	•	$\$ \lessdot \bullet$	shift	–
$E•$	a	$\bullet \lessdot a$	shift	–
$E•a$	$	$a \gtrdot \$$	reduce (5)	$a^*a \vert ^*a$
$E•E$	$	$\bullet \gtrdot \$$	reduce (2)	$a^*a \vert ^*a \bullet$
E	$	–	Accept	$a^*a \vert ^*a \bullet$

$a^*a \mid {}^*a \bullet$ is the postfix representation of the regular expression $r = (a^* \mid a)^* \bullet a$ presented as input.

3.4 Automaton Construction Step from the Postfix Representation of the Regular Expression

After the first step, it is now necessary to start the construction phase of the automaton from the obtained postfix form. In this context, we first present a general outline of the algorithm performing this construction.

Let w\$ then be the postfix regular expression presented as input (Algorithm 3.2). The end of the expression is followed by the symbol $.

Algorithm 3.2: POST REG-EXP TO NFA()

```
sp ← 1;
while (w[sp] ≠ $)
do { case w[sp] of
       { 'a', 'ε':  Push(w[sp]);
              '|':  Union;
              '•':  Concatenation;
              '*':  Star
     sp ← sp + 1
```

Before providing details concerning the Push, Union, Concatenation, and Star primitives, it is first necessary to describe some objects as well as the data structure used for the representation and the construction of the automaton. The adequate data structure representing the automaton is obviously its transition table. Our problem in this second part of our work is to find the way to build this table. To do this, the postfix regular expression is split according to the entity encountered (an operand or an operator) as illustrated by the algorithm (Algorithm 3.2).

Indeed, the treatment of an operand (symbolically represented by a or ϵ) differs from the treatment reserved for an operator. The operators themselves differ in the nature of the operations they are supposed to perform. Three fundamental operators (union, concatenation and Kleene's star) participate in order to build step by step the target automaton. So to build the NFA corresponding to a or to ϵ, we create an initial state i and a final state f such that $I(i, a) = f$ or $I(i, \epsilon) = f$ depending on whether the symbol a or the symbol ϵ. Step-by-step construction

ISSN 2007-9737

practically follows Thompson's approach [9]. The rest of the construction is based practically on the application of the operations (|, •, and *) to the components (sub-automata) already built as illustrated by Thompson's construction algorithm in Section 3.2.

As an indication if NFA$_1$ and NFA$_2$ are two already built sub-automata having i_1 and i_2 as initial states, the application of the uninon (|) operation, produces a new NFA such that its initial state i is connected respectively to i_1 or i_2 on the symbol ϵ. The final states f_1 and f_2 respectively of NFA$_1$ and NFA$_2$ will be connected on the symbol ϵ to the final state (f) of NFA. Thus the construction ends when the postfix regular expression is completely scanned. The transition table obtained represents the targeted NFA. More information about its implementation will be given below.

About the primitives mentioned (Push, Union, Concatenation, and Star) in the above algorithm (Algorithm 3.2):

– Push($w[sp]$) consists first of creating the initial and final states i and f such that the final state f is stored in the entry $I(i, x)$, ($x = a$ or $x = \epsilon$). Then i and f are saved (stacked) in a stack for later use. I is the name given to the transition table. This name is also the name of the transition function of the NFA.

– Union: When the operator $w[sp]$ = '|', a new initial state i and a new final state f will be created. Then, from i will come out two arcs (transitions on the symbol ϵ) which will go to i_1 or i_2. Finally, from the states f_1 and f_2 will come out two arcs (transitions on ϵ) which will go to the newly created final state f. Recall that i_1 and f_1 as well as i_2 and f_2 were previously saved in a stack. The state i and f will be saved in a stack for later use.

– Concatenation: If the operator $w[sp]$ = '•', the initial state (i_1) of NFA$_1$ becomes the initial state (i) of the whole NFA. The final state (f_1) of NFA$_1$ is linked with the initial state (i_2) of NFA$_2$ by an ϵ-transition ($I(f_1, \epsilon) = i_2$). The final state (f_2) of NFA$_2$ becomes the final state (f) of the whole NFA. At the end of the procedure, states i and f are saved in a stack.

– Star: If $w[sp]$ = '*', a new initial state i and a new final state f will be created. Then the initial state i is connected via an ϵ-transition to the initial state (i_0) of the sub-automaton NFA$_0$, or the final state f of the NFA. Recall that NFA$_0$ is the automaton whose states i_0 and f_0 were previously saved in a stack. Another transition from f_0 to i_0 via ϵ is added in order to perform the Kleene repetition on NFA$_0$. Finally, another ϵ-transition from f_0 to f is added. The procedure ends by saving i and f in a stack for later use.

4 Implementation of the Method for Converting a Regular Expression into NFA

The implementation is, as expected, divided into two steps:

– The first step is to generate the postfix representation of a given regular expression. The algorithm (Algorithm 3.1) has been implemented and allows to generate any regular postfix expression. Concrete examples to support the reliability of the implementation will be given below.

– The second step, by means of the algorithm (Algorithm 3.2), uses any expression obtained in the first step and converts it into the corresponding NFA.

4.1 Test Results Obtained with Some Regular Expressions in the First Step

In Table 6 some regular expressions with their respective postfix representations.

Regular expressions compiled previously cannot easily be validated manually. But, in order to perform this task easily, we have replaced in our program (Algorithm 3.1) the postfix translation with a prefix translation. This was achieved by replacing the semantic routines which generate the postfix representation with those which generate the prefix representation.

The test was carried out for all the expressions of the previous tests. Below are the results of this new test.

– $w = a\bullet(a*|a)*\bullet((a)*)*$, prefix($w$) $= \bullet*\bullet a|*aa**a$

– $w = ((a|a\bullet(a*\bullet a)|a*))*\bullet a$, prefix($w$) $= \bullet*||a\bullet a\bullet*aa*aa$

– $w = ((a|a\bullet(a*\bullet a)|a*))*\bullet a|((a*|a\bullet(a*)))$: prefix($w$) $= |\bullet*||a\bullet a\bullet*aa*aa|*a\bullet a*a$

doi: 10.13053/CyS-30-2-3966

Table 6. Converting regular expressions to corresponding postfix expressions

w : Regular expression as input P : Postfix regular expression obtained as output
$w = a\bullet(a * \mid a) * \bullet((a)*)*$ $P = aa * a\mid\bullet * a * *\bullet$
$w = ((a\mid a\bullet(a * \bullet a)\mid a*)) * \bullet a$ $P = aaa * a\bullet\bullet\mid a * \mid * a\bullet$
$w = ((a\mid a\bullet(a * \bullet a)\mid a*)) * \bullet a\mid((a * \mid a\bullet(a*)))$ $P = aaa * a\bullet\bullet\mid a * \mid * a\bullet a * aa * \bullet\mid\mid$
$w = (((a*) * \bullet a\mid a) * \bullet a)\mid a$ $P = a * *a\bullet a\mid * a\bullet a\mid$
$w = a\bullet a\mid a\mid a\bullet a * \mid a * \bullet a\bullet((a*)*)$ $P = aa\bullet a\mid aa * \bullet\mid a * a\bullet a * *\bullet\mid$
$w = (a*) * \mid a\bullet a * \mid a * \bullet((a)*) * \mid a\bullet a\mid a*$ $P = a * *aa * \bullet\mid a * a * *\bullet\mid aa\bullet\mid a * \mid$

– $w = (((a*) * \bullet a|a) * \bullet a)|a$, prefix($w$) $= |\bullet * |\bullet * *aaaaa$
– $w = a\bullet a|a|a\bullet a * |a * \bullet a\bullet((a*)*)$, prefix($w$) $= |||\bullet aaa\bullet a * a\bullet\bullet * aa * *a$
– $w = (a*) * |a\bullet a * |a * \bullet((a)*) * |a\bullet a|a*$, prefix($w$) $= |||| * *a\bullet a * a\bullet * a * *a\bullet aa * a$

For the validation, we used a pushdown transducer that translates a prefix regular expression into its corresponding postfix. The transducer in question is an adaptation of the transducer found in [1], page 229, except that the latter relates to arithmetic expressions.

For this we wrote a program that simulates the behavior of the pushdown transducer (prefix $\longrightarrow$ postfix). The latter is defined by the following 8-tuple:
$(Q, V_{T1}, \Gamma, V_{T2}, I, q, \#, F) = (\{q\}, \{a, |, \bullet, *\}, \{a, |, \bullet, *, \#\}, \{a, |, \bullet, *\}, I, q, \#, \{q\})$,
where the elements Q,$V_{T1}, \Gamma, V_{T2}, I, q, \#$, and F are respectively, in this order, the set of states, the input alphabet, the stack alphabet, the output alphabet, the transition function, the start state, the initial stack symbol, and the set of the final states.

The transducer transition function I is defined by the following instructions:
- $I(q, a, \#) = q, -, a$; $I(q, |, \#) = q, \#\#|, -$; $I(q, \bullet, \#) = q, \#\#\bullet, -$
- $I(q, *, \#) = q, \#*, -$; $I(q, |, -) = q, -, |$; $I(q, \bullet, -) = q, -, \bullet$; $I(q, *, -) = q, -, *$.

For example, $I(q, a, \#) = q, -, a$ means that at the state q, on the symbol a, the transducer pops the symbol $\#$, and returns the recognized symbol a as output; this action is indicated by the triplet $q, -, a$ which is the value of $I(q, a, \#)$.

The action $I(q, |, \#) = q, \#\#|, -$ means that at the state q, on the symbol $|$, the transducer push $\#\#|$, but does not return any symbol on output as indicated by the triplet $q, \#\#|, -$.

But, to explain the translation (prefix $\longrightarrow$ postfix), we manually performed the validation on the prefix expression $w = |\bullet*|\bullet**aaaaa$ corresponding to the infix expression $w = (((a*) * \bullet a|a) * \bullet a)|a$.

The terminal symbol ϵ is part of the alphabet in the same way as the symbol a, but to simplify the explanations, we have knowingly omitted it in these examples.

Here are the sequences of analysis, and the translation of the prefix expression into the corresponding postfix expression.
$(q, |, \#, -) \Rightarrow (q, \bullet, \#\#|, -) \Rightarrow (q, *, \#\#\bullet\#|, -) \Rightarrow (q, |, \# * \#\bullet\#|, -) \Rightarrow$
$(q, \bullet, \#\#| * \#\bullet\#|, -) \Rightarrow (q, *, \#\#\bullet\#| * \#\bullet\#|, -) \Rightarrow (q, *, \# * \#\bullet\#| * \#\bullet\#|, -) \Rightarrow$
$(q, a, \# * *\#\bullet\#| * \#\bullet\#|, -) \Rightarrow (q, a, * * \#\bullet\#| * \#\bullet\#|, a) \Rightarrow (q, a, *\#\bullet\#| * \#\bullet\#|, a*)$
$\Rightarrow (q, a, \#\bullet\#| * \#\bullet\#|, a * *) \Rightarrow (q, a, \bullet\#| * \#\bullet\#|, a * *a) \Rightarrow (q, a, \#| * \#\bullet\#|, a * *a\bullet) \Rightarrow$
$(q, a, | * \#\bullet\#|, a * *a\bullet a) \Rightarrow (q, a, *\#\bullet\#|, a * *a\bullet a|) \Rightarrow (q, a, \#\bullet\#|, a * *a\bullet a|*) \Rightarrow$
$(q, a, \bullet\#|, a * *a\bullet a| * a) \Rightarrow (q, a, \#|, a * *a\bullet a| * a\bullet) \Rightarrow (q, a, |, a * *a\bullet a| * a\bullet a) \Rightarrow$
$(q, -, -, \boxed{a * *a\bullet a| * a\bullet a|})$.
Thus, we obtained the postfix representation of $w = (((a*) * \bullet a|a) * \bullet a)|a$, which is exactly the one obtained with the test in Table 6 for the infix expression of the fourth row of the table. There are other ways of validation. Indeed, it is possible to replace the semantic actions allowing to generate the postfix representation by those which make it possible to generate the abstract binary tree of any regular expression. The binary tree obtained will be traversed with a simple algorithm in postorder. Thus we have verified the veracity of the first step of our method.

4.2 Implementation of the NFA Generation Algorithm

The generation of the automaton implies the construction of the corresponding transition table.

The role of the different functions and procedures of the algorithm (Algorithm 3.2) is explained above, based on the fundamental operators (union (|), concatenation (•), and Kleene's iteration (*)) applied to regular expressions.

However, another important issue concerns the construction of the transition table. Indeed, for this construction to be successful, it is necessary to have an estimate on the size of this table.

Knowing the size of the latter in advance allows us to reserve the necessary and sufficient memory space for its storage. In this context, the size of the NFA can be calculated based on the size of the regular expression.

The size $length(r)$ of a regular expression r is measured by the number of its symbols, excluding parentheses. The postfix regular expression is therefore ideal for this estimation.
Thus, $length(\epsilon) = length(a) = 1$.
Let r and s be two regular expressions. Then, $length(r|s) = length(r{\bullet}s) = length(r)+length(s)+1$; $length(r*) = length(r) + 1$.
Let N(r) be the number of states of the NFA(r) specified by the regular expression r. Then, in accordance with the approach followed [9], the number of states of the NFA is estimated as follows:
N(ϵ) = N(a) $= 2$, N($r|s$) = N(r) + N(s) $+ 2$, N($r{\bullet}s$) = N(r) + N(s) $- 1$, N($r*$)=N(r) $+ 2$. Therefore, N(r) $\leq 2 \times length(r)$. Thus, from each state at most two arrows come out. In other words, the number of transitions is at most double the number of states. These estimates make it possible not only to measure the sufficient memory space, but also to choose an adequate memory allocation (dynamic) to create the automaton (states and transitions) step by step.

Taking into account these data, we implemented the transition table construction algorithm (Algorithm 3.2), then we performed some tests to verify that the mentioned procedures do well the union, the concatenation as well as the Kleene star.

4.3 Test Results Obtained with Some Postfix Regular Expressions

Here are regular expressions in postfix form for which we will describe the respective transition tables below.

– $w = aa*{\bullet}$ The corresponding infix expression is = $a{\bullet}a*$
– $w = aa|a|$ The corresponding infix expression is = $a|a|a$
– $w = aa*{\bullet}a*|$ The corresponding infix expression is = $a{\bullet}a*|a*$
– $w = aa{\bullet}a{\bullet}a{\bullet}$ The corresponding infix expression is = $a{\bullet}a{\bullet}a{\bullet}a$
– $w = aa{\bullet}aba{\bullet}||$ The corresponding infix expression is = $a{\bullet}a|a|b{\bullet}a$
– $w = ab{\bullet}a*|$ The corresponding infix expression is = $a{\bullet}b|a*$

As expected, the automata (NFA) corresponding to these regular expressions are represented by their respective transition tables (I, ...VI, in Table 7). The abbreviations S and A (1st row, 1st column) in these tables represent respectively the states and the alphabet of automata.

The initial state and the final state are marked by the symbols i and f respectively in last column of the transition tables (Table 7). The numbers I, II, ..., VI, are on the entry (last column, first row) of these tables.

For example, let us test the words $\boxed{aa}$, $\boxed{a}$, $\boxed{\epsilon}$, $\boxed{aaaa}$, $\boxed{ba}$, and $\boxed{ab}$ with the transition tables (Table 7) I, II, III, IV, V, and VI respectively.

Transitions are represented by arrows from left to right. The obtained analysis sequences are the following:

— For the word $\boxed{aa}$ with table (I), the analysis starts at initial state 1 and ends at final state 6. Thus, we have $1 \xrightarrow{a} 2 \xrightarrow{\epsilon} 5 \xrightarrow{\epsilon} \{3,6\}$. To continue we choose state 3, so $3 \xrightarrow{a} 4 \xrightarrow{\epsilon} \{3,6\}$. To continue, we also choose state 3 (the multiple choice is due to the fact that we have a non-deterministic automaton: NFA). We end with $3 \xrightarrow{a} 4 \xrightarrow{\epsilon} \{3,6\}$. We then choose state 6 which is obviously the final state. The analysis stops successfully at state 6.

ISSN 2007-9737

— For the word $\boxed{a}$ with table (II), the analysis starts at initial state 9 and ends at final state 10. Indeed, $9 \xrightarrow{\epsilon} \{5, 7\}$. We choose state 7. Thus, we have $7 \xrightarrow{a} 8 \xrightarrow{\epsilon} 10$. The state 10 being the final state, where the analysis ends successfully

— For the word $\boxed{\epsilon}$ with table (III), the analysis starts at initial state 11 and ends at final state 12. Thus, we obtain the sequence $11 \xrightarrow{\epsilon} \{1, 9\}$. To continue, we select state 9, so $9 \xrightarrow{\epsilon} 10$. We end with $10 \xrightarrow{\epsilon} 12$. The state 12 being the final state where the analysis ends.

— For the word $\boxed{aaaa}$ with table (IV), the analysis starts at initial state 1 and ends at final state 8. Indeed, the analysis sequence is $1 \xrightarrow{a} 2 \xrightarrow{\epsilon} 3 \xrightarrow{a} 4 \xrightarrow{\epsilon} 5 \xrightarrow{a} 6 \xrightarrow{\epsilon} 7 \xrightarrow{a} 8$ which ends successfully at final state 8.

— For the word $\boxed{ba}$ with table (V), the analysis starts at initial state 13 and ends at final state 14. The analysis sequence is $13 \xrightarrow{\epsilon} \{1, 11\}$.

Table 7. Transition tables of the automata constructed from the postfix regular expressions given above

A / S	a	b	...	z	ϵ	(I)
1	2					i
2					5	
3	4					
4					$\{3, 6\}$	
5					$\{3, 6\}$	
6						f

A / S	a	b	...	z	ϵ	(II)
1	2					
2					6	
3	4					
4					6	
5					$\{1, 3\}$	
6					10	
7	8					
8					10	
9					$\{5, 7\}$	i
10						f

A / S	a	b	...	z	ϵ	(III)
1	2					
2					5	
3	4					
4					$\{3, 6\}$	
5					$\{3, 6\}$	
6					12	
7	8					
8					$\{7, 10\}$	
9					$\{7, 10\}$	
10					12	
11					$\{1, 9\}$	i
12						f

A / S	a	b	...	z	ϵ	(IV)
1	2					i
2					3	
3	4					
4					5	
5	6					
6					7	
7	8					
8						f

We choose state 11 to continue, so $11 \xrightarrow{\epsilon} \{5, 7\}$. We select the state 7 to continue, so $7 \xrightarrow{b} 8 \xrightarrow{\epsilon} 9 \xrightarrow{a} 10 \xrightarrow{\epsilon} 12 \xrightarrow{\epsilon} 14$ which ends successfully at final state 14.

— For the word $\boxed{ab}$ with table (VI), the analysis starts at initial state 9 and ends at final state 10. The analysis sequence is $13 \xrightarrow{\epsilon} \{1, 11\}$. We choose state 11 to continue, so $9 \xrightarrow{\epsilon} \{1, 7\}$. We select the state 1 to continue, so $1 \xrightarrow{a} 2 \xrightarrow{\epsilon} 3 \xrightarrow{b} 4 \xrightarrow{\epsilon} 10$ which ends successfully at final state 10.

– To be able to analyze a number of words without limitation, it is advisable to use a simulator of an NFA. Such a simulator can be implemented by relying, for example, on the algorithm (Simulation of an NFA) described in Aho et al. [3] page 156.

– Each NFA can be transformed into an equivalent

doi: 10.13053/CyS-30-2-3966

ISSN 2007-9737

S \ A	a	b	...	z	ϵ	(V)
1	2					
2					3	
3	4					
4					14	
5	6					
6					12	
7		8				
8					9	
9	10					
10					12	
11					$\{5,7\}$	
12					14	
13					$\{1,11\}$	i
14						f

S \ A	a	b	...	z	ϵ	(VI)
1	2					
2					3	
3		4				
4					10	
5	6					
6					$\{5,8\}$	
7					$\{5,8\}$	
8					10	
9					$\{1,7\}$	i
10						f

DFA (deterministic finite automaton). The subset algorithm [3], pages 153–155, makes it possible to carry out this transformation. There is also the algorithm [3], page 151 which allows to simulate the behavior of a DFA for the recognition of words like those which have been tested previously with our method.

To end with the implementation and experimentation part, we give below, the transition table (Table 8) obtained with our method for the regular expression $(a|b)^{*}\bullet a$ whose postfix form is $ab|*a\bullet$.

We also draw the corresponding transition diagram in Figure 10.

Table 8. Transition table of the automata constructed from the postfix regular expression $ab|*a\bullet$

S \ A	a	b	...	z	ϵ	
1	2					
2					6	
3		4				
4						
5					$\{1,3\}$	
6					$\{5,8\}$	
7					$\{5,8\}$	i
8					9	
9	10					
10						f

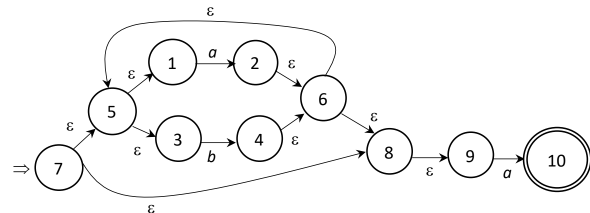


Fig. 10. Transition diagram corresponding to the NFA represented in Table 8

5 Proposed Method versus Some Related-Work Methods

Our method differs from most of the methods reported in the state of the art. However, our method has certain characteristics in common with Thompson's construction.

To remove any ambiguity with Thompson's approach, our method effectively follows it, but only a little in the second step. The latter consists of the effective construction of the automaton from a postfix regular expression.

Even at this level, we can notice some differences. Indeed, with Thompson's approach (Subsection 2.2, Figure 4), the concatenation of two automata consists in directly merging the final state of the first automaton with the initial state of the second automate. In the case of our method, the concatenation (Subsection 3.4) consists to link

the final state of the first automaton to the initial state of the second automaton ($I[f_1, \epsilon] = i_2$).

The first stage of our work differs literally from Thomson's method approach. Indeed, in our method an operator precedence grammar (Subsection 3.1) which generates regular expressions is proposed. The precedence of operators allowed the use of bottom-up parsing. The semantic actions of translating a regular expression into postfix form are clearly defined. This characteristic is very important, since the semantic routines can be changed at any time to generate any desired intermediate form (postfix, prefix, three-address code, and tree representation). This allows some portability to our algorithm. Thomson's approach works in a different way. Indeed, to generate the postfix representation of a regular expression, Thompson's algorithm follows a top-down strategy similar to a recursive descent [9].

We can also note some differences. Indeed, with Thompson's approach [9], the concatenation symbol ($\bullet$) is added by the preprocessor that precedes the parsing of a regular expression. In our case, the concatenation symbol ($\bullet$) is part of the grammar that generates the regular expressions (Subsection 3.1). Therefore, the preprocessor in our case is only supposed to check whether a regular expression is a well-formed expression before starting the parsing and translation process (Subsection 3.3.2, Table 3).

6 Conclusions

The expected objective has been practically achieved. Indeed, we have proposed a method for constructing a finite state automaton from a regular expression, and we have obtained conclusive results. The proposed method is divided into two parts.

The first part relates to the compilation of a regular expression. For this purpose, an operator precedence grammar which generates regular expressions is proposed. Such a grammar allows the construction of a bottom-up parser known for its speed and efficiency.

This type of parser makes it easy to add the appropriate semantic actions for the generation of the desired intermediate representation. We have chosen the postfix representation for its simplicity, and its efficiency of use in the construction of the finite automaton.

The postfix notation as well as the other representations (abstract tree and three-address code) are very important in computer science, especially in compilers, interpreters and calculators. Moreover, it is possible to switch from one representation to another.

The second part is dedicated to the actual construction of the finite automaton starting from the postfix regular expression obtained in the previous part.

There are several algorithms to simulate the behavior of the resulting automaton. As this automaton is an NFA, it can be simulated either directly or first transformed into a DFA (Deterministic Finite Automaton) using the subset method [2, 3], or even minimized by the minimization algorithm [2, 3, 5].

This approach can be improved by eliminating the epsilon transitions. This will significantly reduce the number of states and the number of transitions.

References

1. **Aho, A.V., Ullman, J.D. (1972).** The theory of parsing, translation, and compiling, Volume I: Parsing. Prentice-Hall, series in automatic computations.

2. **Aho, A.V., Sethi, R., Ullman, J.D. (1986).** Dragon Book (Compilers: Principles, techniques, and tool). Addison-Wesley.

3. **Aho, A.V., Sethi, R., Lam, M.S., Ullman, J.D. (2006).** Dragon Book (Compilers: Principles, techniques, and tool). Pearson Education Edition.

4. **Aït el hadj, A. (2015).** Analyse syntaxique et traduction. Technosup, Ellipse Editions, Paris.

5. **Aït el hadj, A. (2018).** Théorie des langages et compilation. Technosup, Ellipse Editions, Paris.

6. **Johnson, S.C. (1975).** Yacc–yet another compiler compiler. Comp. Sci. Tech.Rep. 32, AT& T Bell Labs., Murray Hill, NJ.

7. **Lesk, M.E.** Lex a lexical analyser generator, Comp. Sci. Tech. Rep. 39, AT& T Bell Labs, Murray Hill, NJ, 1975.

8. **Watson, B.W. (1993).** A taxonomy of finite automata construction algorithms (Technical report). Eindhoven University of Technology, Computing Science Report 93/43.

9. **Thompson, K. (1968).** Programming Technique, Regular expression search algorithm. Communications of the ACM. 11 (6): 419–422. doi: 10.1145/363347.363387.

10. **Kleene, S.C. (1956).** Representation of Events in Nerve Nets and Finite Automate. Automata Studies, Annals of Math. Studies. Princeton Univ. Press. 34. Here: sect.9, p.37-40.

11. **Glushkov, V.M. (1961).** The abstract theory of automata. Uspekhi Mat. Nauk, 16:5(101), 3–62; Russian Math. Surveys, 16:5 (1961), 1–53.

12. **McNaughton, R. (1960).** Regular expressions and state graphs for automata, IRE Trans. Electronic Computers, vol. EC–9, no 1, p. 39–47. doi: 10.1109/TEC.1960.5221603.

13. **Hopcroft, J.E., Motwani, R., Ullman, J.D. (2007).** Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 3e ed.

14. **Gross, J. L., Yellen, J. (2004).** Handbook of Graph Theory. Discrete Mathematics and it Applications. CRC Press. Here: sect.2.1, remark R13 on p. 65.

15. **Brzozowski, J.A. (1964).** Derivatives of Regular Expressions. J ACM. 11 (4): 481–494. doi: 10.1145/321239.321249.1964.

16. **Falcone, Y., Fernandez, J.C. (2020).** Automates à états finis et langages réguliers, Dunod Edt.

Article received on 12/05/2021; accepted on 24/04/2026.
*Corresponding author is Fariza Bouhatem.