UNIVERSITATEA "ALEXANDRU IOAN CUZA"
IAŞI
**FACULTATEA DE INFORMATICĂ**

LUCRARE DE LICENȚĂ

# Algoritmul lui Rytter - Vizualizare

propusă de
**DANA-IOANA ZAHARIA**

**Sesiunea:** *februarie 2018*

Coordonator științific:
**Conf. Dr. Ştefan Ciobâcă**

UNIVERSITATEA "ALEXANDRU IOAN CUZA"
IAŞI
**FACULTATEA DE INFORMATICĂ**

# Algoritmul lui Rytter - Vizualizare

**DANA-IOANA ZAHARIA**

**Sesiunea:** februarie 2018

Coordonator științific:
**Conf. Dr. Ştefan Ciobâcă**

# DECLARAŢIE PRIVIND ORIGINALITATEA ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul „Algoritmul lui Rytter - Vizualizare" este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imaginile etc. preluate din proiecte open-sources sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi, Data

_____

Dana-Ioana Zaharia

_____

# DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „Algoritmul lui Rytter - Vizualizare”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” Iaşi să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi, Data

_____

Dana-Ioana Zaharia

_____

# CUPRINS

Introducere viii
Animaţiile şi efectul lor asupra învăţării viii
Expresii regulate şi automate finite viii
Utilitatea transformării expresiilor regulate în automate finite ix
Algoritmi paraleli optimali ix
Alte vizualizări ale transformării expresiilor regulate în automate finite x
Descrierea proiectului prezentat în această lucrare x
Descrierea sumară a lucrării x
1 PREZENTAREA ALGORITMULUI LUI RYTTER 1
1.1 Descriere 2
1.2 Exemplificare a algoritmului lui Rytter 4
1.2.1 Expresia regulată elementară "a" 4
1.2.2 Expresia regulată "a|b" sau "a+b" 4
1.2.3 Expresia regulată "ab" sau "a.b" 5
1.2.4 Expresia regulată "a*" 5
1.3 Comparaţie cu alţi algoritmi paraleli 6
1.3.1 Rytter - Sedgewick 6
1.3.2 Ziadi şi Champarnaud 8
1.4 Concluzii la algoritmul lui Rytter 9
2 IMPLEMENTAREA VIZUALIZĂRII 10
2.1 Interfaţa 10
2.2 Utilizare 11
2.3 Intern 12
2.3.1 PEG.js 12
2.3.2 JavaScript 14
2.3.3 D3.js 15
2.4 Versiuni 17
Concluzii 18
BIBLIOGRAFIE 19

# LISTĂ DE FIGURI


Figura 1 Regulile de construire a automatelor complexe conform Aho & Ullman. 1
Figura 2 Automatele corespunzatoare expresiilor regulate elementare (exceptând mulţimea vidă), conform Aho & Ullman. 2
Figura 3 Expresia elementară "a" - screenshot Refiner. 4
Figura 4 Expresia regulată "a|b" - screenshot Refiner. 5
Figura 5 Expresia regulată "ab" - screenshot Refiner. 5
Figura 6 Expresia regulată "a*" - screenshot Refiner. 6
Figura 7 Genealogia primilor 3 algoritmi paraleli optimali pentru problema transformării unei expresii regulate în automat finit. 7
Figura 8 Parcursul uzual pentru construirea unui program care să recunoască un pattern descris de o expresie regulată. 8
Figura 9 Arborele rezultat în urma parsării expresiei regulate $(ac(a|b)^*)^*(ab^*(a+b)^*)$ 10


# LISTĂ DE TABELE


Tabela 1 Operatorii folosiţi în aritmetica expresiilor regulate 2


# LISTINGS


Listing 1 Algoritmul lui Rytter [9] 3
Listing 2 Varianta paralelă optimală a algoritmului lui Sedgewick [9] 7
Listing 3 Algoritmul semnat de Ziadi şi Champarnaud [24] 9

Listing 4 Aritmetica expresiilor regulate în format PEG 13
Listing 5 Obiectul REFINER 14
Listing 6 Obiectul REFINER.rytter 15
Listing 7 Funcţia drawRytter - eliminarea noduri-lor şi a muchiilor din automat 16

# INTRODUCERE

Rolul proiectului de faţă este acela de a propune, în scop didactic, o vizualizare a primului algoritm paralel optimal de transformare a expresiilor regulate în automate finite echivalente. Acest algoritm este predat în Facultatea de Informatică, la disciplina Construcţia Compilatoarelor reprezentând o etapă importantă în intenţia disciplinei de a obţine un compilator la finalul cursului.

Algoritmul este dezvoltat de către Wojciech Rytter [9] în 1987 şi porneşte de la transformarea clasică a lui Thompson [20], adaptată de către Hopcroft şi Ullman [12]; acesta urmăreşte obţinerea unui automat finit nedeterminist cu $\epsilon$-tranziţii.

În general un algoritm presupune o suită de paşi, iar vizualizarea acestora poate determina existenţa unei mişcări. În acest caz putem vorbi despre o animaţie.

## ANIMAŢIILE ŞI EFECTUL LOR ASUPRA ÎNVĂŢĂRII

Atât în literatura din domeniul animaţiilor cu scop didactic [22] cât şi în aceea specifică, a animaţiilor folosite pentru învăţarea algoritmilor [5], [4] a fost evidenţiat faptul că acestea, deşi sunt atractive pentru utilizatori, trebuie să îndeplinească anumite condiţii pentru a influenţa pozitiv procesul de învăţare. Cu alte cuvinte, animaţiile sunt utile dacă permit interacţiunea cu utilizatorul, dacă acesta poate controla evoluţia în timp a animaţiei şi dacă păstrează accentul pe informaţia principală de transmis. Acestea sunt şi obiectivele vizualizării prezentate în această lucrare.

## EXPRESII REGULATE ŞI AUTOMATE FINITE

Expresiile regulate şi automatele finite sunt modalităţi diferite de descriere a aceloraşi limbaje. Expresiile regulate sunt notaţii algebrice, uşor de folosit în interfeţele de utilizator bazate pe text. Pe de altă parte, automatele sunt reprezentări grafice, preferate în programare, întrucât ele pot fi translate facil în programe care să recunoască un pattern descris de o expresie regulată [3], [1].

### UTILITATEA TRANSFORMĂRII EXPRESIILOR REGULATE ÎN AUTOMATE FINITE

Identificarea unui şablon este necesară atât în editoarele de text şi în construcţia compilatoarelor şi a circuitelor electronice [1], cât şi în biologia computaţională unde "multe seturi importante de şiruri aflate în biosecvenţe, în special în proteine, pot fi specificate ca expresii regulate" [11].

Recunoaşterea patternurilor se face cu programe specializate care pot fi descrise ca automate finite. Automatele sunt însă mai greu de schiţat, decât expresiile regulate [1]; de aici şi necesitatea transformării expresiilor regulate în automate echivalente - care descriu acelaşi limbaj ca şi expresia regulată.

### ALGORITMI PARALELI OPTIMALI

Un algoritm paralel specifică mai multe operaţii la fiecare pas – în contrast cu un algoritm secvenţial, care atribuie unui pas o singură operaţie [2].

Algoritmii paraleli au rolul de a îmbunătăţi performanţa rezolvării problemelor pe care le adresează, pe mai multe feluri de computere, fie ele multiprocesor sau nu.

Aceşti algoritmi sunt dezvoltaţi pe modele abstracte, multiprocesor în cazul de faţă, dintre care putem menţiona modelul *PRAM* - parallel random access machine. "Un model PRAM cu n procesoare este format dintr-un set de n procesoare, toate conectate la o memorie comună, partajată."[2].

Un alt model abstract este *CREW-PRAM*. Acesta permite acces nelimitat la o resursă partajată, pentru operaţii de citire, iar pentru scriere permite unui singur procesor să acceseze, la un moment dat, o resursă [2].

Calitatea de a fi optimal a unui algoritm paralel constă în egalitatea dintre produsul numărului de procesoare - n - cu timpul paralel - p - şi timpul de execuţie al celui mai rapid algoritm secvenţial care rezolvă aceeaşi problemă ca şi cel paralel [9] [24].

#### *Particularităţi ale algoritmului lui Rytter care ţin de proprietatea de a fi paralel*

În cazul algoritmului lui Rytter, subrutinele de calcul a automatelor pentru arborii cu rădăcina în fiecare nod sunt proiectate pentru a fi calculate independent unele de altele şi, în acelaşi timp, pe unităţi de procesare diferite.

### Alte vizualizări ale transformării expresiilor regulate în automate finite

Există şi alte vizualizări, disponibile online, pentru problema transformării expresiilor regulate în automate finite, precum JFLAP[18] sau AutomataTutor[7]. Acestea sunt proiecte complexe, care pot însoţi un întreg curs de teoria automatelor adresând mai multe alte noţiuni specifice acestui domeniu. Pentru algoritmul lui Rytter, însă, nu a fost găsită o vizualizare specifică.

### Descrierea proiectului prezentat în această lucrare

*Contribuţia acestui proiect*

- constă în construirea unei vizualizări simple, concise şi specifice, pentru algoritmul lui Rytter. Vizualizarea este disponibilă online, ca aplicaţie web, la adresa: `https://sunbrush.github.io/bach/`.

*Tehnologii folosite*

Aplicaţia este de tip single-page, iar marcarea şi stilizarea acesteia sunt realizate cu HTML şi CSS. Funcţionalitatea este implementată într-un subset al limbajului Javascript, evidenţiat de către dezvoltatorul web Douglas Crockford. Subsetul este denumit "the Good Parts" şi este descris în cartea intitulată „Javascript: The Good Parts” [6].

Animarea interfeţei grafice este realizată prin intermediul bibliotecii D3.js. Aceasta este o bibliotecă nucleu dezvoltată în JavaScript, care oferă o modalitate de redare grafică a datelor.

Aplicaţia foloseşte, pentru parsarea expresiei regulate introduse de utilizator, un parser obţinut cu biblioteca PEG.js. Această bibliotecă generează parsere în JavaScript, pornind de la un anumit tip de gramatică.

### Descrierea sumară a lucrării

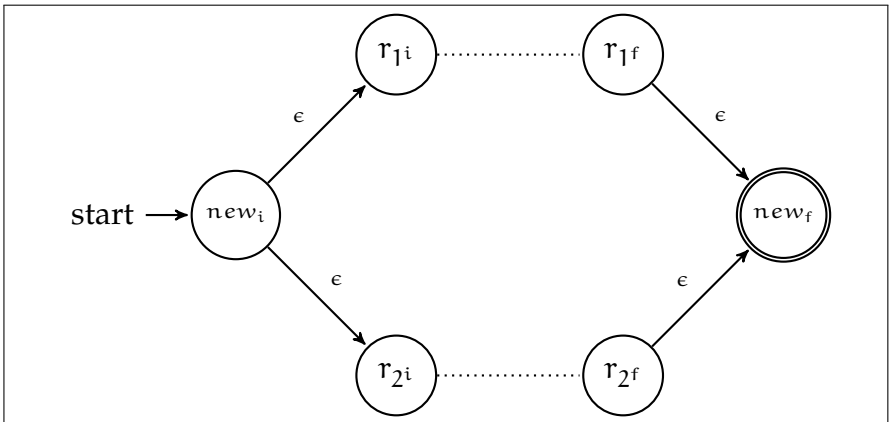
În continuare, lucrarea este organizată în două capitole. Primul capitol este axat pe background-ul teoretic al algoritmului lui Rytter, cuprinzând şi exemplificări ale algoritmului folosind vizualizarea şi o comparaţie cu alţi doi algoritmi paraleli care rezolvă acelaşi gen de problemă. Al doilea capitol cuprinde detalii despre implementarea aplicaţiei web; descrie interfaţa, un scenariu de utilizare, parserul aplicaţiei, implementarea în JavaScript şi D3.js şi versiunile aplicaţiei.

# 1 PREZENTAREA ALGORITMULUI LUI RYTTER

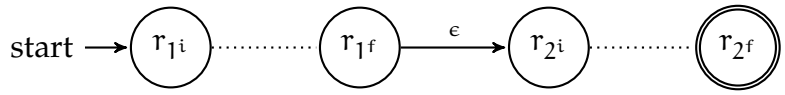
Wojciech Rytter propune prima paralelizare a transformării unei expresii regulate într-un automat finit nedeterminist, pornind de la algoritmul clasic secvenţial – documentat de către Hopcroft şi Ullman [12].

Hopcroft şi Ullman descriu un algoritm recursiv care, pentru o expresie regulată dată, construieşte automatele echivalente cu subexpresiile acesteia. Apoi compune din acestea automatul final, în funcţie de operatorii prezenţi în expresia regulată iniţială - şi prezentaţi în Tabela 1 - şi respectând regulile din Figura 1.

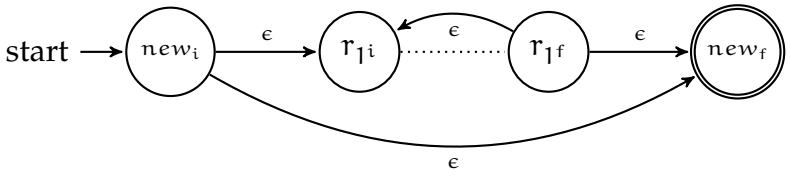
Rytter foloseşte aceleaşi reguli de construire a automatelor complexe ca şi Hopcroft şi Ullman, dar permite obţinerea simultană a automatelor pentru toate subexpresiile, pe unităţi de procesare diferite care alcătuiesc alături de o memorie partajată - un model CREW-PRAM.




(a) Automat pentru $r_1|r_2$, uniunea a două expresii regulate $r_1$,$r_2$.




(b) Automat pentru $r_1.r_2$, concatenarea a două expresii regulate $r_1$,$r_2$.




(c) Automat pentru $r_1{}^*$, închiderea unei expresii regulate $r_1$.

Figura 1: Regulile de construire a automatelor complexe conform Aho & Ullman. [1]

| SIMBOL | EXPRESIE | NUME OPERATOR | PRECEDENŢĂ |
|---|---|---|---|
| * | a* | închidere | 2 |
| . | a.b | concatenare | 1 |
| \| | a\|b | uniune | 0 |
| + | a + b | uniune | 0 |

Tabela 1: Operatorii folosiţi în aritmetica expresiilor regulate

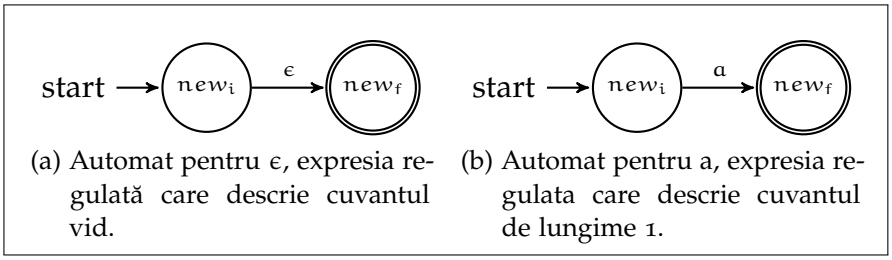
## 1.1 DESCRIERE

Algoritmul lui Rytter foloseşte arborele rezultat în urma parsării expresiei regulate. Acest arbore este format din noduri interne care reprezintă operatorii expresiei şi noduri frunză care reprezintă operanzii - expresii regulate elementare de tip caracter. Operatorii folosiţi in aritmetica expresiilor regulate se pot vedea în Tabela 1.

Fiecărui nod din arbore îi este asociată perechea (stareIniţială, stareFinală). Aceste două stări aparţin automatului echivalent cu subexpresia a cărui arbore de parsare are rădăcina în nodul respectiv.

Prelucrarea perechilor (stareIniţială, stareFinală) in funcţie de regulile de obţinere a automatelor din Figura 1 şi Figura 2 duc la completarea informaţiilor despre automatul final în 3 vectori: symbol, next1 şi next2. Pentru fiecare stare **s** a automatului, tablourile conţin, după cum urmează:

- symbol[s] - simbolul care etichetează muchia automatului ce pleacă din starea s - caracter sau cuvântul vid $\epsilon$
- next1[s] - prima stare aflată la distanţă de o muchie de starea s
- next2[s] - a doua stare aflată la distanţă de o muchie din starea s, dacă această muchie există.




(a) Automat pentru $\epsilon$, expresia regulată care descrie cuvantul vid.

(b) Automat pentru a, expresia regulata care descrie cuvantul de lungime 1.

Figura 2: Automatele corespunzatoare expresiilor regulate elementare (exceptând mulţimea vidă), conform Aho & Ullman. [1]

```
    Algoritmul optim paralel al lui Rytter, adaptat dupa
       Algoritmul secvential al lui Hopcroft & Ullman

Pasul 1: foloseste algoritmi paraleli optimali
1: construieste arborele de parsare
2: numeroteaza nodurile arborelui, in preordine

Pasul 2: crearea starilor
1: parcurge arborele in preordine, atasand fiecarui nod
    vizitat si necorespunzator concatenarii, un numar de
    ordine pornind de la k=1, k++
2: parcurge arborele si ataseaza fiecarui nod
    necorespunzator concatenarii, perechea de stari (
    stareInitiala, stareFinala) calculand stareInitiala=2k
    -1, stareFinala=2k.
3: ataseaza fiecarui nod corespunzator concatenarii,
    perechea de stari (stareInitiala_fiuStanga, stareFinala_
    fiuDreapta)

Pasul 3: construirea automatului
1: declara procedura gen(s, sim, n1, n2)
    daca sym este caracter, diferit de cuvantul vid, atunci
        simbol[s] = sim
    daca n2 este dat ca parametru, atunci next2[s] = n2
    next1[s] = n1
2: executa in paralel, pentru fiecare nod v din arbore,
    procedura potrivita de construire a automatului - sunt
    folosite perechile (stareInitiala, stareFinala) ale
    fiecarui nod:
        * daca v corespunde operatorului de uniune si fiul
            stanga este vs, fiul dreapta este vd
        gen( vi,   ,vsi, vdi)
        gen(vsf,   ,vf,     )
            gen(vdf,   ,vf,     )
        * daca nodul v corespunde operatorului de
            concatenare
            gen(vsf,    ,vdi,   )
        * daca nodul v corespunde operatorului de inchidere
            gen(vi,     ,vsi, vf)
            gen(vsf,    ,vf,  vsi)
        * daca v corespunde unui caracter oarecare a
            gen(vi, a, vf, )
```

Listing 1: Algoritmul lui Rytter [9]

## 1.2 EXEMPLIFICARE A ALGORITMULUI LUI RYTTER

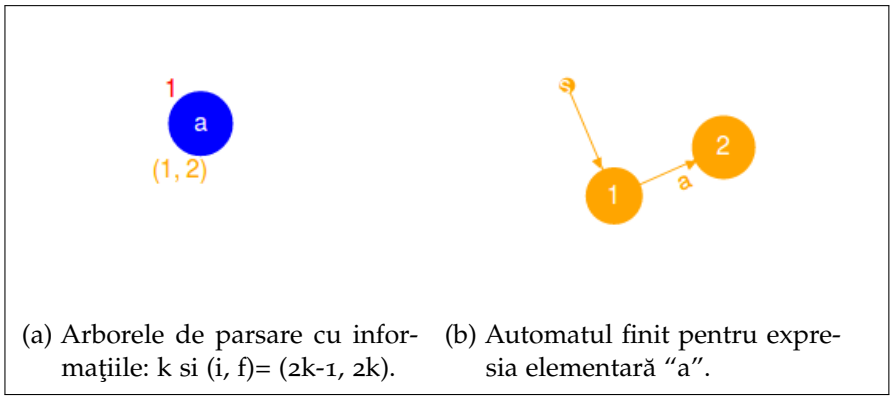
În cele ce urmează este prezentat algoritmul lui Rytter, pentru expresiile regulate: a, a|b, ab și a∗, folosind pentru vizualizare, screenshoturi ale aplicaţiei curente.

### 1.2.1 *Expresia regulată elementară "a"*

Conform algoritmului, arborele de parsare al expresiei regulate **a**, prezentat în Figura 3a, are ataşată perechea de stări (1,2). În pasul final al algoritmului este apelată procedura:

- gen(1, "a", 2, ), care completează informaţiile $symbol[1] =$ "a", $next1[1] = 2$.

Pentru desenarea automatului, în acest caz, se construiesc două stări **1** şi **2** în Figura 3b, unite de o muchie adnotată „a", care pleacă din starea 1 şi se opreşte în starea 2.




(a) Arborele de parsare cu informaţiile: k si (i, f)= (2k-1, 2k).

(b) Automatul finit pentru expresia elementară "a".

Figura 3: Expresia elementară "a" - screenshot Refiner.

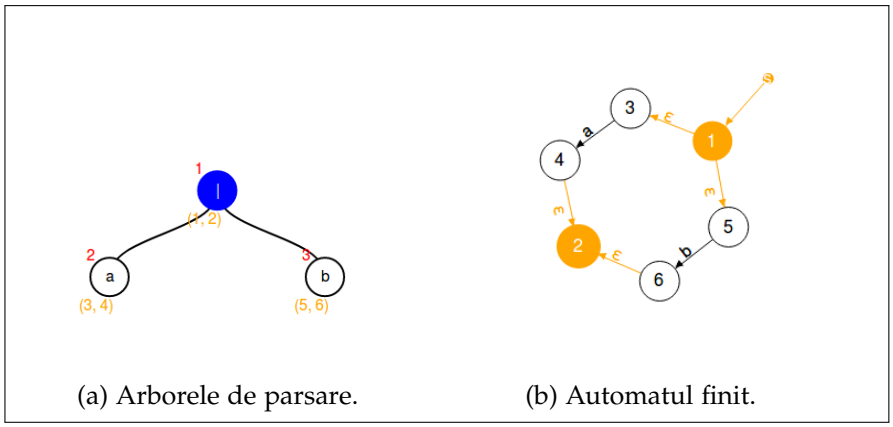
### 1.2.2 *Expresia regulată "a | b" sau "a+b"*

Pentru obţinerea informaţiilor automatului echivalent cu expresia **a | b**, sunt necesare toate cele 3 perechi de stări prezente în arborele din Figura 4a: ale nodului etichetat cu operatorul de uniune - (1,2), ale fiului stânga - (3,4), şi ale fiului dreapta - (5,6). În ultimul pas al algoritmului sunt procesate urmatoarele:

- $gen(1,,3,5) -> next1[1] = 3, next2[1] = 5$
- $gen(4,,2,) -> next1[4] = 2$
- $gen(6,,2,) -> next1[6] = 2$

Reprezentarea automatului ca graf, în Figura 4b, necesită desenarea a două stări noi: **1** şi **2**. Apoi sunt desenate două muchii

cu sursa în noua stare inițială 1 şi destinaţiile în stările inițiale ale automatelor pentru expresiile a şi b, mai exact stările 3 şi 5. Alte două muchii au ca surse stările 4 şi 6, stări finale pentru subautomatele echivalente cu a, respectiv b, iar ca destinaţie noua stare finală a automatului - starea 2.



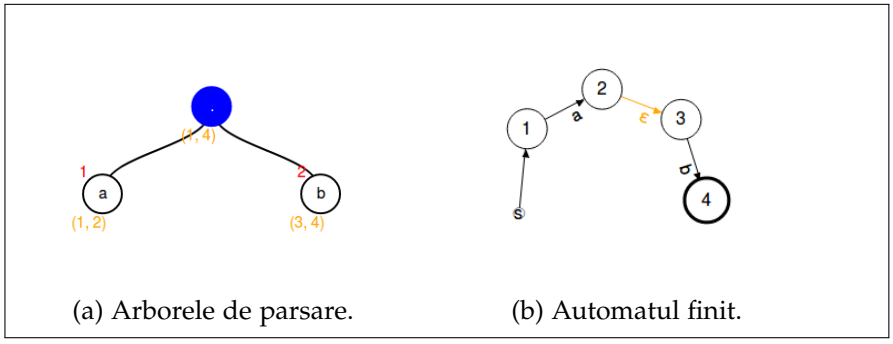

(a) Arborele de parsare. (b) Automatul finit.

Figura 4: Expresia regulată "a | b" - screenshot Refiner.

### 1.2.3 *Expresia regulată "ab" sau "a.b"*

Concatenarea a două expresii regulate elementare – a şi b cu perechile de stări (1, 2) pentru a şi (3, 4) pentru b – va crea în noul automat o muchie cu sursa în starea finală a automatului echivalent cu a şi destinaţia în starea inițială a automatului echivalent cu b. Automatul construit poate fi vizualizat în Figura 5.

- $\text{gen}(2, ,3, ) -> \text{next1}[2] = 3$




(a) Arborele de parsare. (b) Automatul finit.

Figura 5: Expresia regulată "ab" - screenshot Refiner.

### 1.2.4 *Expresia regulată "a*"*

Pentru nodul etichetat cu operatorul de închidere, din Figura 6a, sunt luate în calcul perechea de stări (1, 2) a nodului operatorului şi perechea (3, 4) a nodului frunză. Din acestea se completează tablourile astfel:

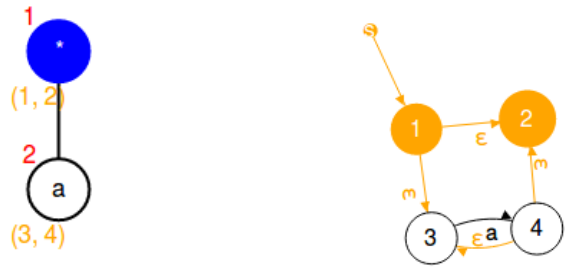
- $next1[1] = 3, next2[1] = 2$, după apelarea gen(1, , 3, 2)
- $next1[4] = 2, next2[4] = 3$, obţinute din gen(4, , 2, 3)

În reprezentarea grafică, 1 este noua stare iniţială şi totodată sursa pentru 2 muchii:

- una cu destinaţie în starea iniţială 3 a automatului echivalent cu a
- cealaltă oprindu-se în noua stare finală 2

Alte două muchii au ca sursă starea finală a automatului echivalent cu a şi ca destinaţii:

- noua stare finală 2
- starea iniţială 3 a automatului echivalent cu a




(a) Arborele de parsare cu informaţiile: k si (i, f) = (2k-1, 2k).

(b) Automatul finit nedeterminist pentru expresia regulataă "a*".

Figura 6: Expresia regulată "a*" - screenshot Refiner.

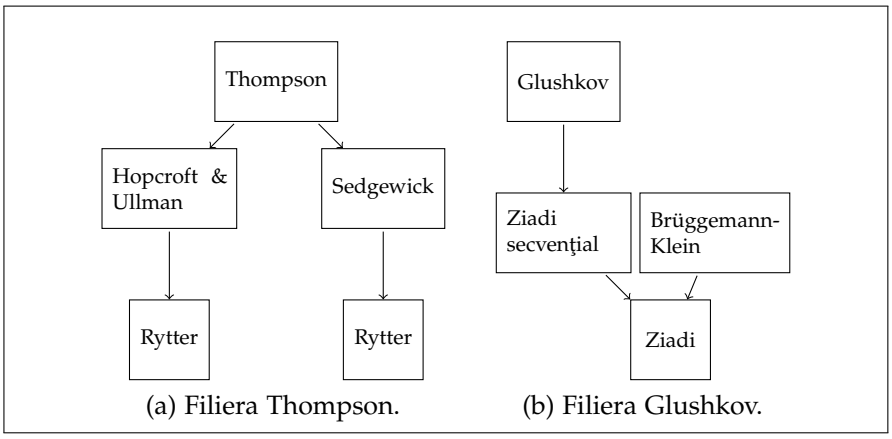
## 1.3 COMPARAŢIE CU ALŢI ALGORITMI PARALELI

### 1.3.1 *Rytter - Sedgewick*

Deşi obţinerea automatului echivalent cu o expresie regulată este tratată de mulţi algoritmi, după cum se poate citi în lucrarea lui Watson [23], totuşi paralelizări optimale ale acestei transformări au fost doar două, până în 1996 - după cum remarca Ziadi în [24] - , ambele datorându-se lui Rytter. Cei doi algoritmi, deşi pornesc de la aceeaşi abordare, a lui Thompson, descrisă în [20], câştigă diferenţe prin prelucrările realizate de către Hopcroft & Ullman [12], respectiv Sedgewick [19].

Figura 7a sumarizează aceste deveniri ale agoritmilor, de la varianta clasică a lui Thompson şi până la paralelizările dezvoltate de Rytter.

Ambii algoritmi sunt optimali, obţinând timpi de procesare de $O(\log\ n)$ pentru expresii regulate de dimensiune $n$, folosind $O(n/\log\ n)$ procesoare pe maşini cu acces random paralel, fără conflicte de scriere (CREW-PRAM).




(a) Filiera Thompson. (b) Filiera Glushkov.

Figura 7: Genealogia primilor 3 algoritmi paraleli optimali pentru problema transformării unei expresii regulate în automat finit.

Paralelizarea algoritmului lui Sedgewick, prezentată de Rytter în [9], este mai complexă în operaţii, în comparaţie cu paralelizarea lui Hopcroft & Ullman, construieşte un automat cu mai puţine stări şi în mai multe etape.

```
    Varianta paralela optimala a algoritmului lui Sedgewick

1&2: aceiasi ca la algoritmul dupa Hopcroft & Ullman
3: executa in paralel, contractia starilor corespunzatoare
   inchiderii si uniunii
4: executa in paralel, fuziunea starilor corespunzatoare
   concatenarii
5: executa in paralel, pentru fiecare nod v din arbore,
   procedura potrivita de construire a automatului,
   respectand regulile din Figura 1 si Figura 2 si
   urmatoarele:
   * daca v corespunde operatorului de uniune
               gen( vi,   ,vsi, vdi)
               gen(vdf,   ,vsf,  )
   * daca v corespunde operatorului de inchidere
               gen(vsf,    ,vf,  vsi)
   * daca v este nod frunza, etichetat cu un caracter
      oarecare "a"
               gen(vi, a, vf, )
6: modifica reprezentarea pe vectori a automatului, in asa
   fel incat, sa ramana doar intrarile pentru starile
   nemodificate in pasii 3 si 4
```

Listing 2: Varianta paralelă optimală a algoritmului lui Sedgewick [9]

### 1.3.2 *Ziadi şi Champarnaud*


Figura 8: Parcursul uzual pentru construirea unui program care să recunoască un pattern descris de o expresie regulată.

Spre deosebire de paralelizările lui Rytter, algoritmul paralel optimal descris de Ziadi şi Champarnaud [24] are la bază o altă metodă clasică de sintetizare a automatului echivalent, anume aceea a lui Glushkov [10]. Acest lucru face posibilă atingerea unui pas mai înaintat în parcursul din Figura 8, ei obţinând un automat nedeterminist fără ϵ-tranziţii.

Schiţa din Figura 7b sumarizează faptul că paralelizarea evidenţiată de Ziadi şi Champarnaud porneşte de la un algoritm secvenţial dezvoltat de Ponty, Ziadi şi Champarnaud [17], folosind totodată şi rezultatele despre forma normală stea a unei expresii regulate studiată de Brüggemann-Klein [3]. Forma normală stea este tot o expresie regulată, echivalentă cu cea iniţială, deci descriind acelaşi limbaj şi generând acelaşi automat Glushkov ca şi expresia iniţială.

Pentru fiecare nouă apariţie a unui aceluiaşi simbol în expresia regulată, automatul Glushkov are o stare distinctă. Muchiile acestuia conectează oricare două stări - apariţii de simboluri identice sau diferite -, care pot fi consecutive într-o parcurgere a expresiei regulate, mai exact într-un cuvânt desemnat de expresie [3].

Pentru descrierea automatului sunt folosite 4 seturi în care sunt stocate:

- poziţiile simbolurilor iniţiale ale tututor cuvintelor din limbajul desemnat de automat şi de expresia regulată E - first(E)
- poziţiile simbolurilor finale ale tututor cuvintelor din limbaj - last(E)
- poziţiile imediat consecutive unei anumite poziţii x - follow(E, x)
- cuvântul vid ϵ, dacă el aparţine limbajului descris de E, sau altfel, mulţimea vidă ϕ - null(E)

Algoritmul lui Ziadi&co. obţine forma normală stea a expresiei şi construieşte seturile amintite anterior, pentru această formă.

```
        Algoritmul semnat de Ziadi & Champarnaud

Pasul 1:
1: construieste arborele de parsare pentru expresia regulata
     E
2: construieste T(E_stea) - arborele de parsare al formei
    normale stea - E_stea - pornind de la cel al lui E

Pasul 2: pentru fiecare nod din arborele de parsare al
    formei normale stea construieste setul Null(E_stea)

Pasul 3: Construieste, pentru toate nodurile din T(E_stea),
    seturile first si last reprezentate ca multimi de arbori

Pasul 4: Construieste  pentru fiecare nod din T(E_stea),
    setul follow reprezentand muchiile cu sursa in nodul
    respectiv
```

Listing 3: Algoritmul semnat de Ziadi şi Champarnaud [24]

Complexitatea timp a algoritmului Ziadi&Champarnaud depinde de lungimea $n$ a expresiei regulate şi este de ordinul $O(\log\ n)$ atunci când este rulat pe un model CREW-PRAM cu $O(n^2/\log\ n)$ procesoare.

## 1.4 CONCLUZII LA ALGORITMUL LUI RYTTER

Algoritmul lui Rytter este primul algoritm paralel pentru problema conversiei expresilor regulate în automate finite cu $\epsilon$-tranziţii şi este un algoritm optimal, rezolvând cu succes conversia unei expresii de dimensine $n$ în $O(\log\ n)$ timp, cu $O(n/\log\ n)$ procesoare care partajează aceeaşi memorie.

Algoritmul acesta poate fi înţeles mai uşor decât varianta paralelizată a lui Sedgewick şi decât algoritmul paralel al echipei Ziadi şi Champarnaud care au etape mai complexe de obţinere a automatului.

Plusul de claritate şi simplitate al algoritmului lui Rytter îl face pe acesta preferabil ca material didactic în cadrul experimentării construirii unui analizor lexical sau chiar a unui compilator şi deci, preferabil ca subiect al unei animaţii cu scop didactic.

# 2

# IMPLEMENTAREA VIZUALIZĂRII

## 2.1 INTERFAŢA

Interfaţa aplicaţiei are un design simplu, cu o cromatică redusă şi elemente grafice strict necesare. Această parcimonie a elementelor vizuale este motivată de studii [21], [14] şi scrieri [16] din sfera designului minimalist al siteurilor web. Acestea susţin că simplitatea face parte din considerentele estetice ale unui website şi îmbunătăţeşte în acelaşi timp interacţiunea utilizator - aplicaţie.

Iniţial, aplicaţia web descrisă în această lucrare pune la dispoziţia utilizatorului:

- un spaţiu de editare a expresiei regulate,
- un buton 'start' şi
- spaţiul destinat vizualizării.

După introducerea expresiei regulate în câmpul text, prin apăsarea butonului 'start' sau a tastei 'Enter' este afişat arborele de parsare al expresiei şi se observă totodată în Figura 9

- butoanele 'next' şi 'prev' care permit navigarea prin paşii algoritmului.

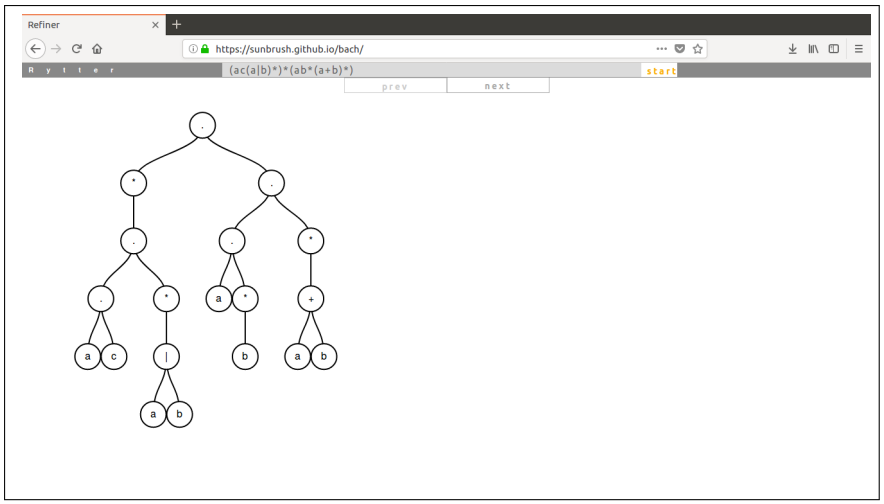



Figura 9: Arborele rezultat în urma parsării expresiei regulate $(ac(a|b)^*)^*(ab^*(a+b)^*)$

## 2.2 UTILIZARE

Un posibil scenariu de rulare a aplicaţiei implică următorii paşi:

1. deschiderea fişierului index.html într-un browser web sau accesarea link-ului `https://sunbrush.github.io/bach/`,
2. introducerea unei expresii regulate valide, în câmpul text,
3. apăsarea tastei 'Enter' sau a butonului 'start',
   - expresia regulată este parsată şi este afişat arborele de parsare
4. apăsarea butonului 'next' până când toate nodurile arborelui de parsare au vizibilă informaţia k,
   - după prima apăsare a nodului 'next', devine accesibil şi butonul 'prev'
   - nodul curent este evidenţiat prin culoarea roşie, iar
   - această primă parcurgere a arborelui de parsare se face în preordine
5. apăsarea butonului 'next' până când toate nodurile arborelui de parsare au vizibilă informaţia (i, f),
   - această a doua parcurgere a arborelui este în postordine
   - nodul curent este evidenţiat prin culoarea orange
6. apăsarea butonului 'next' până când au fost generate automatele pentru toate nodurile arborelui de parsare,
   - nodul curent din arborele de parsare este evidenţiat prin culoarea abastru
   - pune în vedere, pas cu pas, nodurile şi muchiile automatului create pentru fiecare nod în parte
   - în această etapă, algoritmul descris de Rytter obţine informaţiile ce reprezintă automatul finit, prin calcule în paralel, stocându-le în mod independent în 3 vectori diferiţi. Pentru a simplifica înţelegerea acestui pas, am ales ca animaţia să arate construirea automatului final pe parcursul unei traversări în postordine a arborelui de parsare.
7. apăsarea butonului 'prev' până când acest buton devine inactiv.

NOTĂ: Butoanele 'next' şi 'prev' pot fi apăsate atâta timp cât navigarea prin paşii algoritmului este justificată.

## 2.3 INTERN

### 2.3.1 *PEG.js*

Parserul acestui proiect este generat cu ajutorul bibliotecii PEG.js. Această bibliotecă se bazează pe un formalism nou de generare a limbajelor, anume parsing expression grammar - PEG - descris de Bryan Ford [8]. Autorul declară că acest formalism este mai potrivit limbajelor maşină decât cele descrise în ierarhia lui Chomsky care au în vedere limbajul uman, în primul rând.

Pentru generarea online a parserului sunt de parcurs etapele descrise în cele ce urmează:

- Accesarea sitului `http://pegjs.org/online`
- Copierea gramaticii din fişierul pegjs_grammar(din folderul proiectului curent) în câmpul text din secţiunea: "1. Write your PEG.js grammar"
- Completarea textului "REFINER.parser" în câmpul "Parser variable" din sectiunea "3. Download the parser code"
- Apăsarea butonului "Download parser".
- Salvarea fişierului parser.js în folderul cu sursele proiectului curent

Gramatica folosită împreună cu biblioteca PEG.js pentru generarea parserului din proiectul curent descrie aritmetica expresiilor regulate. Acest tip de gramatică nu permite obţinerea a mai mult de un arbore de parsare, cu alte cuvine, nu permite ambiguităţi şi reuşeşte aceasta prin prioritizarea regulilor. În gramatica din Listing 4 se observă că o expresie regulată poate fi privită ca o uniune de concatenări de închideri de paranteze de operanzi.

Gramatica aceasta are 7 reguli, fiecare regulă avand un nume, un şablon şi definind acţiunea care trebuie executată în cazul în care parserul găseşte şablonul în textul de intrare. Prima regulă este şi prima pentru care parserul va căuta o potrivire în input. Dacă în expresia regulată nu este găsită nici o uniune, parserul va căuta o concatenare, datorită prezenţei caracterului '//' la sfârşitul regulii. Dacă este gasită concatenarea, parserul va returna un obiect cu 3 câmpuri: "info" stocând simbolul concatenării, "left" - primul operand al concatenării şi "right" - cel de-al doilea operand.

```
Expression = Union

Union
  = head:Concatenation tail:(_ ("+"/"|") _ Concatenation)+ {
    return tail.reduce(function (previous, current) {
        return {
            info: current[1],
            left : previous,
            right: current[3]
            };
    }, head);
  }
  / Concatenation

Concatenation
  = head:Closure tail:(_ ("."/ _) _ Closure)+ {
    return tail.reduce(function (previous, current) {
        return {
            info: ".",
            left : previous,
            right: current[3]
            };
    }, head);
  }
  /  Closure

Closure
  = head:Parenth op:(("*")_) {
      return {
          info: "*",
          left: head
      };}
  / Parenth

Parenth
  = "(" _ expr:Expression _ ")" {
      return expr;
    }
  / Operand

Operand
  = [a-z]/[A-Z] {
       return {
           info:text()
       };}

_ "whitespace" = [ \t\n\r]*
```

Listing 4: Aritmetica expresiilor regulate în format PEG

### 2.3.2 *JavaScript*

```
var REFINER = function () {
    'use strict';
    // private variables
         var input,
            parseTree={};
        // private methods
    [...]
    // the public object returned to REFINER
    return Object.freeze({
        [...]
        getParseTree: getParseTree,
        parse: REFINER.parser.parse,
        [...]
        rytterize: REFINER.rytter.rytterize,
        d3DrawRytter: REFINER.rytter.drawRytter
    });
} ();
```

Listing 5: Obiectul REFINER

Proiectul descris în această lucrare implementează algoritmul lui Rytter în JavaScript, făcând uz de latura orientată obiect a acestui limbaj. Beneficiile programării orientate obiect urmărite aici sunt încapsularea, mentenanţa şi uşurinţa detectării greşelilor. Acestea sunt asigurate de aplicarea pattern-ului 'module' descris de Crockford, care permite ascunderea informaţiilor private, oferă acces privilegiat către acestea prin intermediul închiderii şi poate crea obiecte securizate [6]. În cazul de faţă, un modul este un obiect ale cărui metode expuse nu pot fi modificate (imutabil), care nu foloseste "new" sau "this" (durabil [6]) şi care este rezultatul execuţiei unei funcţii anonime. Implementarea în JavaScript a aplicaţiei este conţinută într-un singur obiect - denumit REFINER. Acesta impiedică expunerea variabilelor vizibile în spaţiul proiectului, ca variabile globale ale spaţiului JavaScript, deci şi coliziunea de nume cu alte obiecte sau funcţii din acest spaţiu şi oferă o serie de metode care servesc interacţiunii cu utilizatorul. Listing 5 prezintă schematic structura acestui obiect. Arborele de parsare ca variabilă a acestui obiect este cunoscut la nivelul întregului proiect, dar este inaccesibil din exteriorul spaţiului proiectului.

Automatul final şi arborele cu informaţiile k şi (i, f) - numit aici şi arborele Rytter - şi procedurile care ţin de algoritmul lui Rytter şi de vizualizarea acestuia sunt cuprinse într-un singur modul vizibil în Listing 6, facilitând astfel extinderea proiectului către vizualizarea mai multor algoritmi comparabili cu acesta.

```
REFINER = REFINER || {};

REFINER.rytter = (function () {
    'use strict';
        var rytterTree;
        var nfa = {
                symbol: [],
                next1: [],
                next2: []};
//******* OBTAINING THE NFA , FOLLOWING THE RYTTER'S
    ALGORITHM *************
    var getNFA = function (tree) {[...]};  // end of getNFA
    //apply the Rytter algorithm to the parse tree
    var rytterize = function () {[...]};
// ************ DRAWING AND ANIMATING THE PARSING TREE AND
    THE NFA ****************
    // change the form of Rytter's tree from {info,n,k,i,f,
        left:{}, right:{}}
    // to {info,n,k,i,f, children:[]}
    // so that it can be read and used by d3.layout.tree()
    var jsonizeT = function () {[...]}; //end of jsonizeT
    [...]
    // adds nodes and edges to the NFA, acording to the '
        next' button
    var addToNFA = function (d_elem, gnodes, glinks)
        {[...]};
    // removes nodes and edges from NFA, according to the '
        prev' button
    var removeFromNFA = function (d_elem, gnodes, glinks)
        {[...]};
    // firing all up
    var drawRytter = function() {[...]}; //end of drawRytter

        return Object.freeze({
                        rytterize: rytterize,
            drawRytter: drawRytter
        });
}());
```

Listing 6: Obiectul REFINER.rytter

### 2.3.3 *D3.js*

Numele întreg al bibliotecii D3.js este Data-Driven Documents. Murray explică faptul că *datele* sunt procurate de utilizator, *documentele* sunt documente web, precum HTML sau SVG, care pot fi redate de un browser web, iar partea de *driven* revine bibliotecii care crează legături între date şi documente [15].

```
var drawRytter = function() {
    [...]
    drawTree();
    d3.select('#nextbtn').on("click", function () {
        [...]});//end of next button listener
    d3.select("#prevbtn").on("click", function () {
        [...]
        force.stop();
        removeFromNFA(currentTreeNode, gnodes, glinks);
        force.nodes(gnodes);
        force.links(glinks);
        force.start();
        drawNFA();
        [...]
        colorize(previd, "blue");
        [...]});//end of prev button listener}; //end of
            drawRytter
```

Listing 7: Funcţia drawRytter - eliminarea nodurilor şi a muchiilor din automat

D3.js conţine funcţii specializate, cunoscute ca layout-uri, care intermediază pentru o formatare a datelor, potrivită cu forma de afişare dorită [13], de exemplu un arbore de parsare sau un graf reprezentând un automat finit.

În cadrul acestui proiect, pentru arborele de parsare a fost folosit d3.layout.tree(), iar pentru automatul finit - d3.layout.force(). Layout-ul tree() acceptă date în format {info_nod, children:[]}, iar layout-ul force() cere ca datele să fie de forma {nodes:[], links:[]}. Pe baza acestor formatări, D3.js generează şi manipulează vizualizarea ca SVG - Scalable Vector Graphics - un format de imagine bazat pe text cu elemente proprii de marcare.

Pentru fiecare click pe butonul *next* sau *prev*, ca răspuns la acest eveniment,

- nodul corespunzător din arbore va fi evidenţiat prin culoare,
- în funcţie de etapa aferentă din algoritmul lui Rytter, arborele este reactualizat cu informaţia k sau (i,f), pentru nodul curent,
- iar în cazul ultimului pas al algoritmului, sunt redate şi evidenţiate prin culoare muchiile şi nodurile din automat, care corespund nodului curent din arbore, sau sunt eliminate Listing 7.

De un real folos au fost în această etapă exemplificările online ale dezvoltatorului D3.js - Mike Bostock şi articolul https://gist.github.com/rdpoor/3a66b3e082ffeaeb5e6e79961192f7d8.

## 2.4 VERSIUNI

Lucrul cu D3.js a determinat existenţa a două versiuni ale proiectului.

În prima versiune, biblioteca D3.js era responsabilă doar de evidenţierea nodurilor prin culoare, aşezarea lor în pagină şi tranziţiile din cadrul animaţiei. Acea versiune folosea pentru vizualizare, în plus, biblioteca Viz.js şi utilitarul DOT.

Arborele de parsare împreună cu informaţiile k și (i,f) şi automatul erau transcrise în limbaj DOT, apoi şirul de caractere conţinand această notaţie era transmis ca prim parametru funcţiei Viz(str, engine, output) a librariei Viz.js. Această funcţie returna un SVG ca şir de caractere, iar şirul acesta era inclus în interfaţa web.

Implementarea actuală utilizează JSON pentru importul de date în D3, facilitând astfel o mai simplă şi mai rapidă manipulare a elementelor de vizualizare şi chiar accesarea unor biblioteci precum three.js, pentru afişare tridimensională, într-o versiune ulterioară.

De asemeni, în atenţia versiunii următoare intră şi

- redimensionarea arborelui şi a automatului în funcţie de numărul de noduri
- evidenţierea prin culoare a nodurilor şi a muchiilor care urmează a fi eliminate din automat
- mărirea duratei tranziţiilor pentru a face operaţiile mai vizibile.

# CONCLUZII

Vizualizarea algoritmului lui Rytter, descrisă în prezenta lucrare, permite utilizatorului să oprească şi să reia parcursul algoritmului, are un design simplu şi concis, fără a distrage atenţia de la paşii algoritmului şi subliniază informaţia care necesită maximum de atenţie la un moment dat. Conform studiilor [5], [4], [22], această animaţie poate îmbunătăţi gradul de înţelegere al algoritmului lui Rytter, de către studenţii care participă la cursul de Construcţia Compilatoarelor, unde se cere implementarea acestui algoritm ca etapă intermediară în dezvoltarea unui analizor lexical.

Această aplicaţie web, disponibilă online la adresa `https://sunbrush.github.io/bach/`, a fost dezvoltată şi testată folosind browserul Firefox, rulat pe Ubuntu 16.04. Trebuie menţionat faptul că este posibil ca, la rularea aplicaţiei în alte configuraţii, să se descopere bug-uri ale acesteia.

# BIBLIOGRAFIE


[1] Alfred V. Aho şi Jeffrey D. Ullman. *Foundations of Computer Science*. W.H.Freeman, 1992. Chap. 10.

[2] Guy E. Blelloch şi Bruce M. Maggs. "Algorithms and Theory of Computation Handbook: Special Topics and Techniques". In: ed. by Mikhail J. Atallah şi Marina Blanton. 2nd ed. Chapman & Hall/CRC, 2010. Chap. Parallel Algorithms, pp. 25–25.

[3] Anne Brüggemann-Klein. "Regular expression into finite automata". In: *Theoretical Computer Science* 120 (1993), pp. 197–213.

[4] Michael D. Byrne, Richard Catrambone şi John T. Stasko. "Evaluating Animations as Student Aids in Learning Computer Algorithms". In: *Computers & Education* 33 (1999), pp. 253–278.

[5] Richard Catrambone şi A. Fleming Seay. "Using Animations to Help Students Learn Computer Algorithms". In: *Human Factors* 44.3 (2002), pp. 495–511.

[6] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[7] Loris D'Antoni, Matthew Weaver, Alexander Weinert şi Rajeev Alur. "Automata Tutor and what we learned from building an online teaching tool". In: *Bulletin of the European Association for Computer Science* 117 (2015), pp. 143–160.

[8] Bryan Ford. "Parsing Expression Grammars: A Recognition-based Syntactic Foundation". In: *ACM SIGPLAN Notices* 39.1 (2004), pp. 111–222.

[9] Alan Gibbons şi Wojciech Rytter. *Efficient Parallel Algorithms*. Press Syndicate of the University of Cambridge, 1988. Chap. 3.4.

[10] Victor Mikhailovich Glushkov. "The Abstract Theory of Automata". In: *Russian Mathematical Surveys* 16 (1961), pp. 1–53.

[11] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Press Syndicate of the University of Cambridge, 1997.

[12] John E. Hopcroft şi Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computations*. Addison–Wesley, 1979. Chap. 2.

[13] Elijah Meeks. *D3.js in Action*. Manning Publications Co., 2015.

[14] Morten Moshagen şi Meinald T. Thielsch. "Facets of Visual Aesthetics". In: *International Journal of Human-Computer Studies* 68.10 (2010), pp. 689–709.

[15] Scott Murray. *Interactive Data Visualization for the Web*. O'Reilly Media, Inc., 2013.

[16] Jakob Nielsen. *Designing Web Usability: The Practice of Simplicity*. Indiana: New Riders Publishing, 1999.

[17] Jean-Luc Ponty, Djelloul Ziadi şi Jean-Marc Champarnaud. "Automata Implementation. First International Workshop on Implementing Automata, WIA '96 London, Ontario, Canada, August 29–31, 1996 Revised Papers". In: ed. by D. R. Raymond, D. Wood şi S. Yu. Vol. 1260. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1997. Chap. A New Quadratic Algorithm to Convert a Regular Expression into an Automaton, pp. 109–119.

[18] Susan H. Rodger şi Thomas W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc., 2006.

[19] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983. Chap. 20.

[20] Ken Thompson. "Regular Expression Search Algorithm". In: *Communications of the ACM* 11.6 (1968), pp. 419–422.

[21] Alexandre N. Tuch, Eva E. Presslaber, Markus StoCklin, Klaus Opwis şi Javier A. Bargas-Avila. "The Role of Visual Complexity and Prototypicality Regarding First Impression of Websites: Working towards understanding aesthetic judgments". In: *International Journal of Human-Computer Studies* 70 (2012), pp. 794–811.

[22] Barbara Tversky, Jullie Bauer Morrison şi Mireille Betrancourt. "Animation: Can it Facilitate?" In: *International Journal of Human-Computer Studies* 57 (2002), pp. 247–262.

[23] Bruce William Watson. "Taxonomies and Toolkits of Regular Language Algorithms". PhD thesis. Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.

[24] Djelloul Ziadi şi Jean-Marc Champarnaud. "An Optimal Parallel Algorithm to Convert a Regular Expression into its Glushkov Automaton". In: *Theoretical Computer Science* 215 (1999), pp. 69–87.