Limbaje formale si automate - Laboratorul 1
Grupele 143 & 144
Versiune 17.02.2020

Bogdan Macovei

Februarie 2020

Contents

1 Motivatie

Studiul limbajelor formale si, implicit, al automatelor reprezinta o necesitate in Informatica, avand diverse aplicatii in toate subramurile de activitate, cu accent in teoria compilatoarelor, pentru analizele lexicale si sintactice ale programelor (analiza lexicala implementata cu ajutorul limbajelor regulate - gramatici regulate si automate finite deterministe, scopul acestui laborator; analiza sintactica implementata cu ajutorul limbajelor independente de context - gramatici independente de context si automate stiva sau push-down).

In afara de teoria compilatoarelor, o alta aplicatie imediata o constituie intelegerea si utilizarea expresiilor regulate, care se regasesc in verificarea validitatii unei adrese de email, verificarea CNP-urilor, verificarea cerintelor minimale pentru parole (sa fie cel putin un caracter special, cel putin o litera mare etc.) cat si pentru a match-ui stringuri, a extrage anumite informatii (numerice, text, sau particularitati precum litere mari, caractere speciale) etc.

Modelul de lucru al automatelor finite este intalnit si in tipurile speciale de logici, care au la baza sistemele de tranzitie intre stari (ex. logici modale - epistemice, temporale, de incredere).

Studiul teoretic al limbajelor formale constituie baza pentru intelegerea modelului masinii Turing (limbaj formal de tip 0), care este un suport pentru analiza complexitatii si a calculabilitatii functiilor.

2 Notarea laboratorului

Laboratorul valoreaza 30p din nota finala, punctaj ce va fi obtinut din 3 proiecte. In plus, prezenta in cadrul laboratorului este **obligatorie**.

- fiecare nota $(N_i)_{i \in \{1,2,3\}}$ va fi obtinuta atat din implementarea propriu-zisa, cat si din cateva intrebari din teoria necesara implementarii (ponderea va fi 70-30 in favoarea implementarii);
- fiecare nota N_i variaza in intervalul [0, 10.5];
- pe fiecare laborator se poate acorda un punctaj bonus, pana in 0.5p, care va fi adunat la urmatoarea tema de laborator, acolo unde este cazul;
- nota pe laborator se calculeaza ca fiind media aritmetica a celor trei note obtinute;
- nota de laborator se transmite cu doua zecimale exacte, dar nu poate depasi 10.

Important. Proiectele pot fi dezvoltate in orice limbaj de programare, recomandarea fiind C++ (pentru a va obisnui cu programarea orientata pe obiecte), dar nu exista nicio depunctare in acest sens. In plus, pentru studentii care vor sa dezvolte si in limbaje functionale, recomandarea a fost Haskell, dar poate fi aleasa orice alta varianta functionala.

3 Limbaje regulate

3.1 Gramatici

3.1.1 Gramatici formale in general

Se numeste gramatica formala tuplul $G = (N, \Sigma, S, P)$, unde N este multimea neterminalelor sau a variabilelor (finita si nevida), Σ este multimea terminalelor (finita si nevida), $S \in N$ este axioma sau simbolul de start, iar $P \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ este multimea regulilor de productie (relatie binara peste $(N \cup \Sigma)^*$, cu $N \cap \Sigma = \Phi$.

Observatie: Notatia M^* pentru o multime finita se refera la monoidul liber generat de acea multime. De exemplu, pentru o multime $\Sigma = \{a, b\}$ de caractere, monoidul liber Σ^* este multimea tuturor combinatiilor de a si b (multime numarabila): $\Sigma^* = \{a, b, aa, ab, ba, bb, aaa, aab, ...\}$.

Un element $(u, v) \in P$ se noteaza $u \to v$ ("u trece in v" sau "u se poate inlocui cu v"); u se numeste membrul stang al productiei, iar v se numeste membrul drept al productiei.

Se numeste relatie de derivare intr-un pas relatia $\Rightarrow$ sau $\Rightarrow_G$ definita astfel: $\alpha \Rightarrow_G \beta$ daca si numai daca $\alpha = \alpha_1 u \alpha_2$, $\beta = \alpha_1 v \alpha_2$, $(u \to v) \in P$ si $\alpha_1\alpha_2 \in (N \cup \Sigma)^*$.

Prin $L(G)$ vom nota limbajul generat de gramatica G, si se defineste astfel:

$$L(G) = \{w \mid w \in \Sigma^*, S \Rightarrow^* w\}$$

Observatie: $\Rightarrow^*$ este inchiderea reflexiva si tranzitiva a relatiei $\Rightarrow$, si are semnificatie de "derivare intr-un numar de pasi ≥ 0".

3.1.2 Gramatici regulate

Se numeste gramatica regulata (sau gramatica de tip 3) o gramatica pentru care orice productie $(u \to v) \in P$ este de forma $u \in N$, iar $v \in \Sigma N \cup \Sigma \cup \{\lambda\}$, unde λ este notatia pentru cuvantul vid (de lungime 0).

Exemplu de gramatica regulata:

$$S \to aA \mid bB \mid \lambda$$
$$A \to aA \mid a$$
$$B \to bB \mid a$$

Exercitiu. Este cuvantul aa acceptat de gramatica? Dar cuvantul a? Dar cuvinte de forma a^*b? Dar cuvinte de forma b^*a? Notatia a^* inseamna oricate

aparitii ale lui a (≥ 0); notatia a^+ inseamna oricate aparitii ale lui a, dar > 0 (cel putin un a).

3.2 Automate finite

3.2.1 Automate finite deterministe

Un automat finit determinist DFA $M = (Q, \Sigma, \delta, q_0, F)$ unde Q este multimea starilor (finita si nevida), Σ este alfabetul de intrare (finit si nevid), δ este functia de tranzitie, $\delta : Q \times \Sigma \to Q$ cu extensia $\delta^* : Q \times \Sigma^* \to Q$, $q_0 \in Q$ este starea initiala a automatului, iar $F \subseteq Q$ este multimea starilor finale.

Referitor la δ^*, definitia acesteia este recursiva:

$$\delta^*(q, \lambda) = q$$

$$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

unde aw este un cuvant din Σ^*, cu primul caracter pus in evidenta - $a \in \Sigma$ (i.e. se aplica recursiv δ^*, facandu-se mereu o tranzitie normala cu starea curenta, prima litera din cuvant, iar din starea in care ajungem aplicam recursiv functia cu restul cuvantului).

Limbajul regulat generat de automatul finit se defineste astfel:

$$L(M) = \{w \mid w \in \Sigma^*, \delta^*(q_0, w) \in F\}$$

3.2.2 Automate finite nedeterministe

Un NFA M este identic cu un DFA, diferenta fiind doar in cazul functiei de tranzitie si, implicit, in cazul limbajului regulat ce poate fi generat.

$$\delta : Q \times \Sigma \to 2^Q$$

$$\delta^* : Q \times \Sigma^* \to 2^Q$$

cu definitiile

$$\delta^*(q, \lambda) = \{q\}$$

$$\delta^*(q, aw) = \delta^*(\delta(q, a), w) = \bigcup_{p \in \delta^*(q,w)} \delta(p, a)$$

Limbajul generat de un NFA este $L(M) = \{w \mid w \in \Sigma^*, \delta^*(q_0, w) \cap F \neq \Phi\}$

4 Implementare DFA

4.1 Utilizare C++11

Limbajul recomandat pentru laborator este C++ (varianta minima C++11), cu o eventuala abordare functionala (Haskell). Intrucat la Programare Orientata pe Obiecte se invata C++, in general demo-urile pentru laborator vor utiliza acest limbaj.

Ne propunem sa abstractizam cat mai bine modelul automatului finit determinist, astfel ca pentru $DFA\ M = (Q, \Sigma, \delta, q_0, F)$ ne propunem sa avem o clasa

```
class DFA { ... }
```

Analizam elementele componente ale automatului: Q si F sunt multimi (finite si nevide) de stari, iar codificarea starilor va fi prin numere intregi $0, 1, 2, ...$ (la seminar veti folosi, cel mai probabil, $q_0, q_1, q_2, ...$). In acest caz, tipul de date recomandat este $set < int >$. Starea initiala, notata q_0, va fi un intreg (int), iar aflabetul Σ va fi o multime de caractere, avand tipul $set < char >$.

Pentru functia δ reamintim signatura: $\delta : Q \times \Sigma \to Q$, adica trebuie sa primeasca o pereche $(stare, litera)$ careia sa ii atribuim o alta $stare$. Vom utiliza, in acest caz, tipul de date $map < pair < int, char >, int >$: perechii $(int, char)$ i se asocieaza un int.

Forma clasei pana in acest moment este:

```
class DFA
{
    set<int> Q, F;
    set<char> Sigma;
    int q0;
    map<pair<int, char>, int> delta;
}
```

Atentie la conventiile de programare OOP in C++! Membrii clasei sunt *private*, iar clasele isi definesc membrii *private* implicit, ceea ce inseamana ca Q, F, $Sigma$, q_0 si $delta$ sunt vizibili doar in interiorul clasei, nu si in exteriorul acesteia. Inainte de a defini functii specifice automatelor finite, vom crea doi constructori pentru clasa (unul fara parametri, unul cu 5 parametri) si getters. Constructorii, getter-ii si restul metodelor vor fi declarate public.

```
class DFA
{
    set<int> Q, F;
    set<char> Sigma;
    int q0;
    map<pair<int, char>, int> delta;
```

```
public:
    DFA() { this->q0 = 0; }
    DFA(set<int> Q, set<char> Sigma
       , map<pair<int, char>, int> delta
       , int q0, set<int> F)
    {
            this->Q = Q;
            this->Sigma = Sigma;
            this->delta = delta;
            this->q0 = q0;
            this->F = F;
    }

    set<int> getQ() const { return this->Q; }
    set<int> getF() const { return this->F; }
    set<char> getSigma() const { return this->Sigma; }
    int getInitialState() const { return this->q0; }
    map<pair<int, char>, int> getDelta() const { return this->delta; }
}
```

Vom mai adauga in clasa trei metode: supraincarcam operatorul de citire >>, si implementam doua metode, $isFinalState(int)$ si $deltaStar(int, string)$, unde $deltaStar = \delta^*$, functia extinsa pentru tranzitionarea in automat cand avem cuvinte de lungime strict mai mare decat 1. Definitia finala a clasei este:

```
class DFA
{
    set<int> Q, F;
    set<char> Sigma;
    int q0;
    map<pair<int, char>, int> delta;

public:
    DFA() { this->q0 = 0; }
    DFA(set<int> Q, set<char> Sigma
       , map<pair<int, char>, int> delta
       , int q0, set<int> F)
    {
            this->Q = Q;
            this->Sigma = Sigma;
            this->delta = delta;
            this->q0 = q0;
            this->F = F;
    }

    set<int> getQ() const { return this->Q; }
```

```
    set<int> getF() const { return this->F; }
    set<char> getSigma() const { return this->Sigma; }
    int getInitialState() const { return this->q0; }
    map<pair<int, char>, int> getDelta() const { return this->delta; }

    friend istream& operator >> (istream&, DFA&);

    bool isFinalState(int);
    int deltaStar(int, string);
};
```

Implementam functia *isFinalState*, declarand-o in afara clasei:

```
bool DFA::isFinalState(int q)
{
    return Q.find(q) != Q.end();
}
```

si functia *deltaStar*

```
int DFA::deltaStar(int q, string w)
{
    if (w.length() == 1)
    {
            return delta[{q, (char)w[0]}];
    }

    int new_q = delta[{q, (char)w[0]}];
    return deltaStar(new_q, w.substr(1, w.length() - 1));
}
```

Discutie. Cum a fost implementata functia *deltaStar*? Respecta definita matematica? Identificati metodele utilizate asupra structurilor de date. Recomandare: cautati si cititi documentatia pentru STL, in special pentru structurile de date foarte des utilizate: *vector*, *string*, *set*, *map*, *pair*.

Discutie. Fie urmatoarea implementare pentru functia de citire. Cum va arata inputul citit? Putem prelua informatia dintr-un fisier text?

```
istream& operator >> (istream& f, DFA& M)
{
    int noOfStates;
    f >> noOfStates;
    for (int i = 0; i < noOfStates; ++i)
    {
            int q;
            f >> q;
            M.Q.insert(q);
    }
```

```
    int noOfLetters;
    f >> noOfLetters;
    for (int i = 0; i < noOfLetters; ++i)
    {
            char ch;
            f >> ch;
            M.Sigma.insert(ch);
    }

    int noOfTransitions;
    f >> noOfTransitions;
    for (int i = 0; i < noOfTransitions; ++i)
    {
            int s, d;
            char ch;
            f >> s >> ch >> d;
            M.delta[{s, ch}] = d;
    }

    f >> M.q0;

    int noOfFinalStates;
    f >> noOfFinalStates;
    for (int i = 0; i < noOfFinalStates; ++i)
    {
            int q;
            f >> q;
            M.F.insert(q);
    }

    return f;
}
```

Exercitiu. Creati un proiect nou C++11 (minim) care sa contina toate informatiile de mai sus. Creati un fisier *dfa.txt* in care sa scrieti un input corect pentru a putea defini un DFA. Verificati corectitudinea programului, utilizand urmatoarea definitie in *main*():

```
int main()
{
    DFA M;

    ifstream fin("dfa.txt");
    fin >> M;
    fin.close();
```

```
    int lastState = M.deltaStar(M.getInitialState(), "ab");

    if (M.isFinalState(lastState))
    {
            cout << "Cuvant acceptat";
    }
    else
    {
            cout << "Cuvant respins";
    }

    return 0;
}
```

4.2 Utilizare Haskell

Limbajul Haskell este un limbaj functional, mai intuitiv de folosit pentru descrierea obiectelor matematice. Invatarea acestuia nu este scopul laboratorului curent, dar implementarile sunt, uneori, mai simple. Utilizarea Haskell in proiectele de laborator este optionala, si poate constitui un bonus pentru nota.

Cateva specificatii pentru Haskell sunt urmatoarele:

- putem redenumi tipurile de date, in loc sa avem Int, putem scrie type Q = Int, si sa utilizam Q cu aceeasi semnificatie;
- daca avem un tip de date Int, o lista de intregi se va scrie [Int];
- este un limbaj care, ca Python, accepta comprehensiunea listelor. De exemplu, daca vrem toate elementele pare din intervalul [0, 10], matematic am scrie $\{x \mid x \in \{0, 1, ..., 10\},\ x \mod 2 = 0\}$. In Haskell scriem similar: $[x \mid x \leftarrow [0..10],\ mod\ x\ 2 == 0]$;
- operatorii sunt, de regula, in forma prefixata: scriem $mod\ x\ 2$, sau $even\ x$ etc.;
- avem posibilitatea de a avea liste infinite, putem declara [0..] si va sti ca este lista elementelor naturale (0, 1, 2, ...);
- avem la dispozitie functia *zip* care se aplica pe doua liste, creand perechi de elemente de pe aceeasi pozitie in ambele liste, pana la lungimea celei mai scurte liste. De exemplu, $zip\ [0, 1, 2]\ ['A', 'B', 'C']$ creeaza lista formata din perechile de pe aceeasi pozitie: $[(0, 'A'), (1, 'B'), (2, 'C')]$. Efectul este identic daca scriem $zip\ [0..]\ ['A', 'B', 'C']$, pentru ca *zip*, asa cum am mentionat anterior, merge pana la lungimea celei mai scurte liste. Acesta este un mecanism usor pentru a afisa elementele care se situeaza pe pozitii pare, de exemplu. Sa presupunem ca vrem sa retinem toate literele mici

din alfabet care se afla pe pozitii pare, vom scrie, folosind o combinatie de comprehensiune si list: $list = [ch \mid (poz, ch) \leftarrow zip\ [0..]\ ['a'..'z'], even\ poz]$. Programul va face toate perechile de forma (0, 'a'), (1, 'b'), ..., (25, 'z'), si va selecta ch-ul (adica elementul din dreapta) pentru care poz este par;

- signatura functiilor este de forma f :: Int → Char (f primeste un intreg si returneaza un caracter). Pentru functii cu mai multe argumente, scrierea este similara. De exemplu, add :: Int → Int → Int inseaman ca add primeste doi intregi si returneaza un nou intreg;
- accesarea, dintr-o lista *list*, a unui element de pe o pozitie k se face prin scrierea $list!!k$. De exemplu, daca avem list = [0, 1, 2, 3] si vrem elementul de pe pozitia 3, vom scrie list !! 2 (si va fi egal cu 2).

Avand specificatiile de mai sus, definim:

```
type Q = Int
type Sigma = Char
```

Avem acum un tip de stari Q si un tip de litere din alfabet, *Sigma*. Daca avem nevoie de o multime de stari, vom scrie $[Q]$, iar daca avem nevoie de Σ^*, vom scrie $[Sigma]$. Definim si initializam starea initiala:

```
q0 :: Q
q0 = 0
```

si definim si doua tranzitii in functia δ:

```
delta :: Q -> Sigma -> Q
delta 0 'a' = 1
delta 1 'b' = 2
```

Vom defini si starile finale ale automatului:

```
finalStates :: [Q]
finalStates = [2]
```

In acest caz, tot ce ne ramane de facut este sa implementam functia $deltaStar = \delta^*$.

```
deltaStar :: Q -> [Sigma] -> Q
deltaStar q w
    | length w == 1 = delta q (w !! 0)
    | otherwise     = deltaStar (delta q (w !! 0))
                        [ch | (i, ch) <- zip [0..] w, not (i == 0)]
```

Pentru a testa daca un cuvant este acceptat, scriem in *main*:

```
main = do
    print $ elem (deltaStar q0 "ab") finalStates
```

Linkul care contine programul in Haskell: https://repl.it/repls/OpaqueWiltedProgrammer.

5 Implementare NFA - sarcina de laborator

In cadrul automatelor finite nedeterministe, modificarea (fata de DFA) consta in faptul ca functia δ nu mai ajunge cu un simbol, dintr-o stare, doar intr-o unica stare, ci poate ajunge simultan in mai multe stari. In acest caz, din signatura pentru DFA, vom avea o signatura de forma

$$map < pair < int, char >, set < int >> \ delta;$$

i.e. dintr-o stare, cu o litera, ajung intr-o multime de stari.

Se schimba si modalitatea de acceptare: nu accept daca starea finala in care ajungem este $q \in F$, pentru ca nu ajung doar intr-o stare finala, ci intr-o multime de stari finale. In acest caz, daca ajung in multimea S, accept daca si numai daca $S \cap F \neq \Phi$.

In plus, ar fi indicat sa consideram ca starea initala nu mai este un simplu $int\ q_0;$, ci un $set < int > \ q_0;$, intrucat la fiecare pas al aplicarii functiei δ^*, consideram ca starea din care plecam este, de fapt, intr-o multime a starilor.

Exercitiul 1. Creati o clasa

```
class NFA { ... }
```

similara clasei DFA si faceti modificarile necesare de mai sus.

Exercitiul 2. Modificati functia de citire, astfel incat sa retineti corect automatul nedeterminist in memorie. Indicatii: nu se citeste direct q_0, ci se va face un $q0.insert(x)$, unde x este citit din fisierul text, iar o linie pentru tranzitiile *delta* va fi de forma $stareIinitiala\ litera\ nrStari\ stare_1\ stare_2\ ...\ stare_{nrStari}$. De exemplu, scrierea 0 a 4 1 3 5 6 inseamna ca din starea 0, cu litera 'a', ajungem simultan in starile 1, 3, 5, 6.

Exercitiul 3. Modificati functia *deltaStar* astfel incat sa se comporte corect pentru automatele finite nedeterministe. Puteti implementa propria varianta (recomandat), sau puteti urmari codul de mai jos.

```
set<int> NFA::deltaStar(set<int> s, string w)
{
    int n = w.length();
    set<int> localFinalStates;

    // din prima stare in care suntem
    // (pentru ca suntem intr-un set<int> s)
    // adaugam in localFinalStates
    // toate tranzitiile cu prima litera din w
    for (int j : delta[{s[0], w[0]}])
    {
```

```
            localFinalStates.insert(j);
        }
        // am efectuat o tranzitie, micsorez n-ul
        n--;

        // daca n = 0, inseamna ca returnez starile finale adaugate
        if (n == 0)
        {
            return localFinalStates;
        }

        int contor = 0;

        // altfel, cat timp n != 0
        while (n)
        {
            set<int> auxiliar;

            // ma mut in starile finale in care am ajuns
            // efectuand o tranzitie
            // adica daca din 0 cu a am ajuns in 1 3 4 5
            // acum iterez in {1, 3, 4, 5}
            // pentru a face tranzitii cu urmatoarea litera
            // si stochez starile in auxiliar
            for (int i : localFinalStates)
            {
                for (int j : delta[{i, w[contor + 1]}])
                    auxiliar.insert(j);
            }

            n--;
            contor++;

            // golesc set-ul de localFinalStates
            localFinalStates.clear();

            // mut din auxiliar in localFinalStates
            for (int i : auxiliar)
                localFinalStates.insert(i);

            auxiliar.clear();
        }
        return localFinalStates;
    }
```

Exercitiul 4. Scrieti functia main si verificati daca un cuvant poate fi accep-

tat de automatul finit nedeterminist dat. Daca ati reusit sa parcurgeti cerintele si puteti explica in intregime ce ati facut, anuntati in cadrul laboratorului (in cadrul ORICARUI laborator din semestru) acest lucru, pentru a primi punctaj de activitate.

Exercitiul 5*. Optional, implementati in Haskell modificarile pentru NFA. Folositi, ca suport, link-ul de la implementarea DFA-ului.

6 Implementare λ-NFA - proiect de laborator

Un $\lambda - NFA$ este un automat finit nedeterminist care accepta si tranzitii vide. In acest caz, functia de tranzitie se defineste astfel:

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$$

$\delta(q, \lambda) := \lambda^*(q) = \{q\} \cup \{p \mid \exists p_0, p_1, ...p_n \in Q,\ p_0 = q,\ p_n = p,\ a.i.\ \forall i \geq 1 \Rightarrow p_i \in \delta(p_{i-1}, \lambda)\}$

i.e. $p \in \lambda^*(q)$ daca si numai daca exista un drum de la q la p format din λ-tranzitii. In acest caz, functia extinsa va avea urmatoarea definitie:

$$\delta^*(q, aw) = \lambda^*(\delta^*((\delta(q, a), w)))$$

unde

$$\delta(q, a) = \lambda^*(\delta(\lambda^*(q), a))$$

Cerinte:

- permiteti functiei de citire sa considere si λ-tranzitii. Folositi pentru λ caractere precum "@", "#" sau ".";
- definiti functia *lambdaInchidere* cu signatura

```
set<int> LNFA :: lambdaInchidere(int q);
```

  care calculeaza $\lambda^*(q)$;
- rescrieti functia *deltaStar*, astfel incat sa respecte definitia matematica de mai sus;
- scrieti functia main astfel incat sa puteti face un demo pentru acest proiect.

Important: acesta este un proiect de laborator care va fi notat (de la 0 la 10.5). Veti primi 7p pe implementare (1p rescriere corecta clasa (analog NFA, dar sa permiteti lambda-tranzitii), 2p scrierea corecta a functiei lambdaInchidere, 3p scrierea corecta a functiei deltaStar, 1p demo functional), 3p pe prezentare si teorie (echivalentele modelelor, definitia functiei de tranzitie, comportamentul automatelor etc.) si un bonus de 0.5p care se poate obtine sau din implementarea si in Haskell a programului, sau din implementari mai interesante (in general optimizate din punctul de vedere al timpului si spatiului - cu justificarea corectitudinii si complexitatii!). Termenul de predare este seara de dinaintea laboratorului urmator, pe adresa *bogdan.macovei.fmi@gmail.com*. Nu trebuie sa fiti prezenti la laborator pentru a obtine punctajul de implementare, insa cele 3p de teorie le obtineti doar daca sunteti prezenti. Nu este obligatoriu sa fiti, insa, prezenti la laboratorul imediat urmator, puteti primi partea de teorie in orice alt laborator, pana in saptamana 14.