Tema 1 LFA 2019-2020
Automate finite deterministe

George Daniel MITRA

13 ianuarie 2020

Rezumat

Tema constă în implementarea unui program care analizează un automat finit determinist și răspunde la întrebări despre structura lui și limbajul acceptat

1 Specificații temă

1.1 Cerință

Să se implementeze un program care, primind o reprezentare a unui automat finit determinist, să răspundă la următoarele întrebări despre el:

- Este limbajul acceptat de automat vid?
- Se află șirul vid în limbajul acceptat de automat?
- Este limbajul acceptat de limbaj infinit?
- Care sunt stările accesibile, productive, respectiv utile?

1.2 Conținutul arhivei

Arhiva trebuie să conțină:

- surse, a căror organizare nu vă e impusă
- un fișier Makefile care să aibă target de build și run
- un fișier README care să conțină maxim 20 de linii de maxim 80 de caractere în care să descrieți sumar reprezentarea și algoritmii aplicați. Cu cât mai scurt, cu atât mai bine!

Arhiva trebuie să fie zip. Nu rar, 7z, ace sau alt format ezoteric. Fișierul Makefile și fișierul README trebuie să fie în rădăcina arhivei, nu în vreun director.

Fișierul trebuie să se numească README, nu readme, ReadMe, README.txt, readme.txt, read-me.doc, rEADME, README.md sau alte variante asemănătore sau nu.

Nerespectarea oricărui aspect mai sus menționat va duce la nepunctarea temei.

1.3 Specificații program

1.3.1 Intrări

Programul va citi dintr-un fișier numit „dfa” automatul finit determinist, în formatul specificat în secțiunea 3

Programul va primi ca argument în linia de comandă întrebarea la care trebuie să răspundă:

- -v : Verifică dacă limbajul acceptat de automat este vid
- -e : Verifică dacă limbajul acceptat de automat conține șirul vid
- -a : Afișează care sunt stările accesibile
- -u : Afișează care sunt stările utile
- -f: Verifică dacă limbajul acceptat de automat este finit (bonus)

Modul în care e dat parametrul prin intermediul make este (exemplu pentru -e):

```
make run arg=-e
```

Intrările se consideră corecte.

1.3.2 Ieșiri

Programul afișează răspunsul la ieșirea standard și nu afișează nimic la ieșirea de eroare.

Dacă argumentul în linia de comandă este -v, -e sau -f, atunci programul afișează „Yes” sau „No”, în funcție de răspuns.

Dacă argumentul în linia de comandă este -a sau -u, atunci programul afișează fiecare stare care respectă condițiile pe câte o linie. Ordinea nu contează. Explicațiile pentru ce înseamnă stări accesibile, respectiv utile, se găsesc la secțiunea 3

1.3.3 Suport

Arhiva samples.zip conține cod care face analiza lexicală pentru C și Java în flex, respectiv jflex. Îl puteți folosi ca punct de pornire. Codul vine cu Makefile.

Pentru C, care nu are structuri ca hashtable în bibliotecile standard, aveți la dispoziție o implementare de alfabet, un arbore ternar de căutare pentru a stoca stringuri(stări) și o listă circulară generică.

Pentru C++, puteți adapta lexer.lex din csample.

Pentru Python, puteți folosi python-ply pentru a genera un analizor lexical.

1.3.4 Versiuni

Mașina de test are instalat Ubuntu 18.04 LTS si următoarele versiuni:

- gcc/g++: 9.2.1 20191102
- clang: 7.0.0
- flex: 2.6.4

- bison: 3.0.4
- javac: 11.0.5
- jflex: 1.6.1
- python: 2.7.17, 3.8.0
- python numpy: 1.16.6(2.7), 1.18.1(3.8)
- python ply: 3.11

1.3.5 Limbaje acceptate

Limbajele acceptate sunt C, C++, Java și Python.
Se recomandă flex [3], jflex [6] pentru analiza lexicală.
Singurele limbaje care vin cu schelet de cod sunt C și Java.

1.3.6 Încărcare temă

Tema trebuie încărcată pe vmchecker [1].
Materialele se găsesc pe elf [2]

2 Noțiuni introductive

2.1 Limbajul de descriere

Limbajul este descris printr-o gramatică BNF și folosește următoarea convenție de culori:

- albastru - neterminali
- verde - operatori ai limbajului BNF și paranteze ajutătoare
- rosu - terminali (elemente care fac parte efectiv din limbajul descris)

Pentru a simplifica sintaxa, Se folosesc operatorii $*$, $+$ și ? cu semnificația din expresiile regulate.

2.2 Simbol, Alfabet, Șir

2.2.1 Simbol

Un simbol poate fi literă, cifră sau caracter special.
<symbol> ::= <lower-case letter> | <digit> | <other>
<lower-case letter> ::= (a | b | c | d | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z)
<digit> ::= (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
<other> ::= (‘ | - | = | [|] | ; | ´ | \ | . | / | ˜ | ! | @ | # | $ | % | ˆ | & | * | _ | + | : | ¨ | | | < | > | ?)
<other> reprezintă toate simbolurile printabile care nu sunt alfanumerice de pe o tastatură standard cu layout US International. Nu copiați din document simbolurile, pentru că s-ar putea să nu se potrivească.

2.2.2 Alfabet

Un alfabet este orice mulțime de simboluri.

Atenție, alfabetul poate fi vid și apare ca {} în cazul ăsta.

În teste nu există niciun caz cu mai puțin de două simboluri în alfabet.

<alphabet> ::= { (<terminal> (, <terminal>)*)? }

3 Automate finite deterministe

3.1 Descriere

Un automat finit este un model de calcul care primește la intrare o bandă de simboluri și își modifică starea internă în funcție de ce are la intrare. Acesta poate fi privit ca o cutie neagră cu niște stări între care face tranziții în funcție de intrare, la fiecare tranziție mișcând capul de citire la dreapta.

Formal, un automat finit determinist este un tuplu $M = (K, \Sigma, \delta, s, F)$, cu următoarele proprietăți:

- K este mulțimea stărilor. K este finită, de unde vine și numele automatului. K nu poate fi vidă;
- Σ este alfabetul din care sunt formate cuvintele acceptate de automat;
- δ se numește funcție de tranziție. $\delta : K \times \Sigma \to K$. $\delta(p, a) = q; p, q \in K; a \in \Sigma$ înseamnă că automatul, dacă se află în starea p și primește pe bandă a, trece în starea q. Fiindcă δ este funcție, toate tranzițiile din fiecare stare pentru fiecare simbol trebuie să fie definite.
- $s \in K$ este starea de start. Aceasta este starea în care se află automatul înainte de a primi ceva pe bandă.
- $F \subseteq K$ este mulțimea stărilor finale. Dacă automatul se află într-o stare $f \in F$ după ce a terminat de citit șirul înseamnă că acceptă șirul.

Notăm cu $\vdash$ operatorul de trecere între configurații.

$\vdash: K \times \Sigma \to K$

$\forall p, q \in K, \forall a \in \Sigma, \delta(p, a) = q \Leftrightarrow (\forall w \in \Sigma^*, (p, aw) \vdash (q, w))$

Notăm cu $\vdash^*$ închiderea reflexivă și tranzitivă a operatorului $\vdash$.

Un șir w este acceptat de un automat $M = (K, \Sigma, \delta, s, F)$ dacă și numai dacă $\exists f \in F, (s, w) \vdash^* (f, e)$.

3.2 Specificații

În temă, un automat finit determinist este dat ca un tuplu, conform definiției:

<DFA> ::= (<states> , <alphabet> , <transitions> , <state> , (<states> | {}))

<states> ::= { <state> (, <state>)* }

<state> ::= <name>

<name> ::= (<upper-case letter> | <lower-case letter> | <digit> | _) +

<transitions> ::= (<transition> (, <transition>) *)

<transition> ::= d(<state> , <symbol>)= <state>

3.3 Definiții

Fie $M = (K, \Sigma, \delta, s, F) \in AFD$

3.3.1 Apartenența șirului vid (-e)

$e \in \mathcal{L}(M) \Leftrightarrow (\exists f \in F, (s, e) \vdash^* (f, e)) \Leftrightarrow s \in F$

Pentru că într-un automat determinist orice tranziție consumă un simbol, pentru ca șirul vid să facă parte din limbaj trebuie ca starea inițială să fie finală.

3.3.2 Stări accesibile (-a)

Notăm cu $\mathcal{A}(M)$ mulțimea stărilor accesibile în automatul M.

$\forall p \in K, p \in \mathcal{A}(M) \Leftrightarrow (\exists w \in \Sigma^*, (s, w) \vdash^* (p, e)$

O stare p este accesibilă dacă și numai dacă există un drum din starea inițială până în p.

3.3.3 Stări productive

Notăm cu $\mathcal{P}(M)$ mulțimea stărilor productive din M.

$\forall p \in K, p \in \mathcal{P}(M) \Leftrightarrow (\exists f \in F, \exists w \in \Sigma^*, (p, w) \vdash (f, e))$

O stare p este productivă dacă și numai dacă există un drum din p într-o stare finală.

3.3.4 Stări utile (-u)

Notăm cu $\mathcal{U}(M)$ mulțimea stărilor utile din M.

$\forall p \in K, p \in \mathcal{U}(M) \Leftrightarrow p \in \mathcal{A}(M) \wedge p \in \mathcal{P}(M)$

O stare p este utilă dacă este accesibilă și productivă.

$\mathcal{U}(M) = \mathcal{A}(M) \cap \mathcal{P}(M)$

3.3.5 Limbaj vid (-v)

$\mathcal{L}(M) = \emptyset \Leftrightarrow F \cap \mathcal{A}(M) = \emptyset \Leftrightarrow \mathcal{U}(M) = \emptyset \Leftrightarrow S \notin \mathcal{P}(L)$

Limbajul acceptat de M este vid dacă și numai dacă nicio stare finală nu e accesibilă, dacă nu există stări utile sau dacă starea inițială nu e productivă.

3.3.6 Limbaj finit (-v, bonus)

$|\mathcal{L}(M)| \neq \infty \Leftrightarrow (\forall p \in \mathcal{U}(M), \forall w \in \Sigma^* \setminus \{e\}, (p, w) \nvdash (p, e)$

Limbajul acceptat de M este finit dacă și numai dacă nu există un drum nevid de la o stare utilă p către ea însăși. Dacă ar exista un astfel de drum, automatul ar putea ajunge în p, ($p \in \mathcal{A}(M)$), ar putea da ture prin p consumând un șir nevid, și apoi ar putea accepta o infinitate de șiruri ($p \in \mathcal{P}(M)$).

Altfel spus, nu pot exista în M cicluri care includ stări utile.

4 Punctaj

4.1 Checker

Checker-ul (test.sh(oferă un punctaj între 0 și 125. Ce este peste 100 intră în categoria bonus. Testele sunt publice.

100 de puncte de checker înseamnă un punct din nota finală.
Pentru fiecare funcționalitate, punctajul e distribuit în felul următor:

- -e: $25 = (3_+ + 3_-) \cdot 4_d + 1_b$
- -a: $21 = 5 \cdot 4_d + 1_b$
- -u: $21 = 5 \cdot 4_d + 1_b$
- -v: $33 = ((4_+ + 4_-) \cdot 4_d + 1_b)_{c(a,u)}$
- -f: $25 = ((3_+ + 3_-) \cdot 4_d + 1_b)_{c(a,u)}$

Notația folosită este următoarea:

- $3_+, 4_+$ reprezintă 3, respectiv 4 teste pozitive, $3_-, 4_-$ reprezintă 3, respectiv 4 teste negative.
- 4_d se referă la 4 cazuri de dificultate diferită, 2 pentru $|K| = 20$, 1 pentru $|K| = 200$, 1 pentru $|K| = 2000$.
- 1_b se referă la 1 punct bonus dacă ați trecut cel puțin un test din grupul ăla.
- $x_{c(a,u)}$ se referă la faptul că punctajul x e condiționat de obținerea a cel puțin jumătate din punctajul pe testele pentru -a și -u.

Motivul pentru existența condiționării este că stările accesibile si stările utile sunt necesare pentru a verifica dacă un limbaj e vid sau finit și un răspuns binar ar fi prea ușor de falsificat pentru punctaj gratuit.

Arhiva cu checker conține un generator de teste individuale (checker/gen.py) și un generator de suite de teste (checker/testsetgen.py).

Checker-ul poate fi folosit pentru pentru a evalua testele implicite (./test.sh) sau un set de teste personalizat (./test.sh <testset >).

gen.py și testsetgen.py își afișează modul de utilizare dacă sunt rulate fără parametri. Explicații despre alte fișiere se găsesc în checker/README

4.2 Depunctări

Implementările în care sunt hardcodate teste sau care sunt obfuscate și fac dificilă verificarea hardcodării, vor primi 0 puncte.

Un exemplu de hardcodare este stocarea rezultatelor testelor și afișarea lor din memorie sau afișarea aceluiași răspuns binar indiferent de intrare.

Un exemplu de obfuscare ar fi cod care arată așa sau folosirea de instrucțiuni goto îmbârligate, fie că forma e folosită sau nu pentru a ascunde soluții care nu rezolvă problema

Deadline: 19.01.2020, 23:59. Upload-ul va rămâne deschis pâna la ora 05:00 a doua zi.

Bibliografie

[1] vmchecker LFA

[2] elf tema

[3] flex homepage

[4] Lexical Analysis with Flex

[5] Using flex

[6] jflex homepage

[7] jflex user manual

[8] jflex user manual in japanese

[9] Laborator 1 SO: Makefile