Tema 1 LFA 2017-2018
Simulator de mașini Turing

George Daniel MITRA

18 ianuarie 2018

Rezumat

Tema constă în implementarea unui program care citește și simulează o mașină Turing.

1 Specificațiile temei

1.1 Cerință

Să se implementeze un program care, primind o reprezentare a unei mașini Turing și o reprezentare a benzii ei, afișează banda după execuția mașinii.

Soluțiile care nu folosesc FLEX vor fi depunctate după cum e menționat în secțiunea 3.

1.2 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.

Cu magenta au fost colorate exemplele. Cele care nu sunt **bold** reprezintă variante extreme de folosire a spațiilor. Dacă vă e mai ușor în implementare, puteți presupune că astfel de cazuri extreme nu apar în fișierele de test.

De asemenea, au fost marcate cu galben elementele avansate ale limbajului, elemente care nu sunt necesare pentru descrierea unei mașini Turing, dar care pot face o astfel de descriere mai ușor de citit. Pentru un maxim de 8 puncte din 10 puteți ignora aceste elemente și să implementați doar limbajul de bază. Există fișiere de text speciale pentru limbajul de bază.

<source> ::= <alphabet-decl> (<comment> | <code>)*

Un fișier sursă conține declararea alfabetului, urmată de comentarii și/sau cod.

<alphabet-decl> ::= alphabet :: (<symbol>)+ ;

Declararea unui alfabet se face cu cuvântul cheie alphabet, separat prin :: de lista de simboluri, listă terminată cu ;.

Se presupune implicit că ; nu poate face parte din alfabet. Un alfabet nu poate conține duplicate.

Deși limbajul nu impune, declararea alfabetului se face în teste pe o singură linie, iar simbolurile sunt separate prin spații.

alphabet :: a b c # ;

alphabet::abc#;

alphabet:: ab

c # ;

<comment> ::= ; (<text> <end-of-line>

Un comentariu începe cu ; la început și se termină la sfârșitul liniei. Pentru simplitate, limbajul nu permite comentarii pe aceeași linie cu codul.

; acesta este un comentariu

<code> ::= <symbol-decl> | <set-decl> | <mt-decl>

Codul dintr-un fișier sursă contă în declararea de simboluri, mulțimi de simboluri sau mașini Turing.
<symbol-decl> ::= <name> = <symbol>

Se poate da un nume unui simbol. Simbolul trebuie să facă parte din alfabet, altfel, comportamentul programului este nedefinit. Nu este nevoie de mesaje de eroare, fișierele de intrare sunt corecte.

<set-decl> ::= <name> := { <symbol> (, <symbol>) + } ;

Se poate da un nume unei mulțimi de simboluri, sepcificată între acolade. Simbolurile sunt separate prin virgule și trebuie să facă parte din alfabet.

<any>= {a, b, c} ;

<any>={a,b,c};

<any>= {

a, b,

c } ;

<symbol> ::= <letter> | <digit> | <other>

Un simbol poate fi literă, cifră sau caracter special.

<other> trebuie să includă cel puțin #, $, * și @, dar puteți adăuga alte simboluri care nu sunt separatori sau terminatori în limbajul de descriere.

<name> ::= (<letter> | <digit> | _) +

Numele poate fi orice combinație de litere, cifre și _

<mt-decl> ::= <name> ::= (<mt>) + ;;

Se definește of nouă mașină Turing ca o concatenare de una sau mai multe mașini Turing.

<mt> ::= [<name> @] (<mt-call> | <mt-trans>) | & <name>

Operatorul binar @ se folosește pentru a da un nume temporar unei mașini Turing. Numele este valabil din momentul defnirii până la sfârșitul declarației curente de mașină Turing.

Numele dat cu @ se folosește cu operatorul unar & pentru scrierea mașinilor Turing ce conțin cicluri. Practic, numele dat cu @ reprezintă o etichetă, în timp ce operatorul & reprezintă goto (local definiției).

Nu se pun spații înainte și după @, respectiv după &

R_infinit ::= start@[R] &start ;;

Mașina care merge mereu la dreapta asociază o etichetă mașinii elementare care merge la dreapta, după execuția căreia sare la eticheta definită.

<mt-call> ::= [<name>] | [<elementary>]

Se pot apela mașini elementare sau mașini definite anterior în fișierul sursă.

[R]

[R(!#)]

[Copy]

<elementary> ::= L | R | <element> | L ([!] <element>) | R ([!] <element>)

Mașinile elementare sunt cele discutate la seminar, mașinile care se pot deplasa o poziție pe bandă, mașina care scrie un element pe bandă, mașinile care se deplasează la stânga/dreapta până la întâlnirea unui anumit element, respectiv cele care se deplasează până când ajung la un alt element.

Mașinile elementare cu paranteze ($M(x)$) se pot traduce ca repeat M until x. În cazult lui L(!a), ar fi repeat L until !a. Asta înseamnă că execută cel puțin o dată deplasarea, orice ar fi pe bandă.

Notația pentru mașini elementare nu conține spații.

Pentru alfabetul {a, b} mașinile elementare sunt: L, R, a, b, L(a), R(a), L(b), R(b), L(!a), R(!a), L(!b), R(!b).

<element> ::= <symbol> | < <name> > | & <name>

Un element reprezintă un simbol, accesat în diferite moduri: direct, prin numele lui între paranteze unghiulare, sau dintr-o „variabilă" folosind operatorul &.

Exemplu de mașină elementară care se deplasează la stânga până la întâlnirea simbolului definit cu numele „void": L(<void>).

Exemplu de mașină elementară care se deplasează la stânga până la întâlnirea simbolului care a fost memorat cu numele x (detalii despre memorare mai jos, la <set>): L(&x).

<mt-trans> ::= ((<transition>) *)

O mașină Turing care folosește simbolul de pe bandă este definită ca o listă de tranziții.

Dacă lista nu conține nicio tranziție, se obține mașina Turing care nu face nimic, halt.

Limbajul nu impune un coding style, dar tranzițiile se trec în fișierele de test pe linii distincte, indentate față de linia care conține (.

invert ::= (

{a} ->[b] ;

{b} ->[a] ;

) ;;

<transition> ::= <set> -> (<mt>) + ;

O tranziție precizează într-o mulțime toate simbolurile pentru care mașina execută aceeași tranziție.

Limbajul nu impune, dar în general se pune spațiu înainte și după ->, respectiv înainte de ;.

Mulțimea de tranziții nu acoperă obligatoriu toate simboluriloe din alfabet. Pentru simbolurile neacoperite, se consideră că mașina se oprește, ca în cazul halt.

<set> ::= [<name> @] [!] (< <name> > | { <symbol> (, <symbol>) * })

O mulțime nu poate conține duplicate.

Modul simplu de a scrie o mulțime este prin enumerarea elementelor din mulțime: {a, b, c}.

Un simbol poate fi înlocuit cu numele lui, dacă a fost definit anterior, sau extras din memorie, dacă a fost memorat anterior: {<void>, <separator>, &x }.

Apartenența la mulțime se poate inversa folosind operatorul unar !.

Memorarea simbolului de pe bandă cu numele x, dacă face parte din mulțime, se face folosind operatorul binar @: x@{a, b}.

Mulțimea poate fi înlocuită cu numele ei, dacă a fost definit anterior: <any>.

1.3 Exemplu complet de scriere a unei mașini

Pentru clarificarea sintaxei, vom considera un exemplu complet, mașina Turing care copiază un șir w. Pornind cu w pe bandă și capul de citire în stânga lui w ($\underline{\#}w\#$), mașina ajunge la $\underline{\#}w\#w\#$.

Cel mai simplu mod de a scrie mașina este următorul:

```
alphabet :: a b # ;

Copy ::= start@[R] (

    {a} ->[#] [R(#)] [R(#)] [a] [L(#)] [L(#)] [a] &start ;

    {b} ->[#] [R(#)] [R(#)] [b] [L(#)] [L(#)] [b] &start ;

    {#} ->[L(#)] ;

) ;;
```

Orice mașină Turing poate fi descrisă de limbajul de bază.

Se poate observa că tranzițiile pentru a și b sunt identice dacă notăm simbolul de pe bandă cu x. Le putem combina și atunci acțiunile sunt identice, $\forall x \in \{a, b\}$.

```
alphabet :: a b # ;

Copy ::= start@[R] (

    x@{a, b} ->[#] [R(#)] [R(#)] [&x] [L(#)] [L(#)] [&x] &start ;

    {#} ->[L(#)] ;

) ;;
```

Mai mult de atât, se observă că există o tranziție pe un simbol de oprire, #, și una pentru orice alt simbol. Dacă rescriem regulile în funcție de #, am putea adăuga simboluri în alfabet fără a schimba codul mașinii.

```
alphabet :: a b # ;

Copy ::= start@[R] (

    x@!{#} ->[#] [R(#)] [R(#)] [&x] [L(#)] [L(#)] [&x] &start ;

    {#} ->[L(#)] ;

) ;;
```

Pentru a obține o variantă generică, putem face astfel încât simbolul de oprire să poată fi ușor modificabil, dându-i un nume.

```
alphabet :: a b # ;

void = # ;

Copy ::= start@[R] (

    x@!{<void>} ->[<void>] [R(<void>)] [R(<void>)] [&x] [L(<void>)] [L(<void>)] [&x] &start ;

    {<void>} ->[L(<void>)] ;

) ;;
```

1.4 Domeniu de vizibilitate

În cazul în care faceți partea cu memorare, trebuie să aveți în vedere că variabilele au un domeniu de vizibilitate local și cea mai recentă „variabilă" are prioritate în mașina curentă.

Puteți face asta în mod tradițional și cinstit folosind o stivă de contexte, cu un context ținând toate variabilele din mașina curentă. La apelarea unei mașini noi, se creează un context nou. La terminarea execuțieiunei mașini se curăță contextul curent.

Probleme ar putea apărea la salt în aceeași mașină. Dacă saltul părăsește tranziția pentru care a fost definită variabila, atunci ea se elimină, altfel, rămâne valabilă și s-ar folosi o stivă pentru interiorul unei mașini.

Din fericire, nu e nevoie chiar de asta. Tranzițiile imbricate din aceeași mașină folosesc simboluri diferite întotdeauna. Pe lângă asta, la finalul unei tranziții, mașina se oprește, nefiind urmată de nimic.

Următoarele situații nu există în fișiere de test, chiar dacă limbajul le permite:

```
alphabet :: a b # ;
imb ::= (
    x@{a,b} ->[R][x] (
        x@{a,b} ->[R][x] ;
    ) ;
) ;;
conc ::= (
    x@{a,b} ->[R][x] ;
) (
    x@{a,b} ->[R][x] ;
) ;;
```

În primul caz, mașinile imbricate folosesc același simbol pentru memorare. Deși mașinile pot fi imbricate, simbolurile sunt mereu diferite. Altfel apare o problemă destul de complicată de domeniu de vizibilitate, unde trebuie să decidem ce x e vizibil, mai ales dacă facem un salt.

În al doilea caz, mașinile sunt una după alta. După o mașină cu tranziții nu urmează o altă mașină, și cu siguranță nu o alta cu tranziții.

1.5 Reprezentarea benzii

Banda este nemărginită la stânga și la dreapta. Aceasta se va reprezenta ca un șir de caractere care cuprinde toată informația utilă de pe bandă, poziția capului de citire/scriere, precum și câte un singur # la stânga și la dreapta pentru a marca restul benzii.

Poziția capului de citire/scriere va fi marcată prin simbolul > poziționat înaintea simbolului de pe poziția curentă.

Exemplu: dacă pe bandă avem două șiruri, abba și baab, separate prin #, putem avea următoarele cazuri:

#>#abba#baab# = capul de citire la stânga primului șir

#>####abba#baab# = capul de citire câteva poziții mai la stânga

#ab>ba#baab# = capul de citire pe al doilea b din abba

#abba#baab####># = capul de citire undeva după sfârșitul șirului

2 Trimiterea temei

2.1 Conținutul arhivei

Arhiva trebuie să conțină:

- surse, a căror organizare nu vă e impusă
- un fișier Makefile care să aibă target de build, clean și run
- un fișier README în care să descrieți arhitectura și abordarea pentru lexer. 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. Atenție! în special cei ce folosesc Max OS X!

2.2 Specificațiile programului

2.2.1 Limbaj

Tema poate fi implementată în c sau c++ folosind flex sau în java folosind jflex.

Mașina virtuală pe care se evaluează temele este un Debian care are următoarele versiuni instalate:

gcc/g++: 6.3.0 20170516

clang: 3.8.1-24

flex: 2.6.1

java: openjdk 1.8.0u151

JFlex: 1.6.1

Pentru că tema este de analiză lexicală, nu se vor folosi bison, yacc sau cup pentru parsare.

2.2.2 Intrări

Programul va citi dintr-un fișier cu extensia mt reprezentarea unor mașini Turing, în formatul specificat mai sus.

2.2.3 Ieșiri

Programul va afișa la ieșirea standard configurația benzii după execuția mașinii.

2.2.4 Argumente în linie de comandă

Programul va primi ca argumente în linia de comandă numele fișierului în care este definită mașina Turing, numele mașinii care trebuie rulată și configurația inițială a benzii.

Argumentele sunt date în comanda make run în variabilele $(arg1), $(arg2) și $(arg3). Pentru a evita evaluarea dolarilor din bandă, banda ar trebui folosită ca $(value arg3). Dacă aveți dubii, porniți de la exemplul pus la dispoziție.

2.3 Trimiterea temei

Tema se va încărca pe vmchecker [1].

3 Punctaj

3.1 Checker

Checker-ul oferă un punctaj între 0 și 150. 50 de puncte din 150 sunt bonus.

Testele sunt publice și punctajul lor e:

- 80p = teste care folosesc numai limbajul de bază (fără nume de simboluri, nume de mulțimi, mulțimi complementate, variabile)
- 20p = teste care folosesc limbajul extins, dar fără tranziții cu variabile (x@multime ->)
- 20p = (bonus) teste simple care folosesc și variabile
- 30p = (bonus) un test mai complicat care folosește în plus și alte simboluri speciale în afară de #

3.2 Materiale disponibile

La [2] găsiți materialele puse la dispoziție pentru rezolvarea temei:

- tlfa1.zip: Arhiva cu teste. test.sh e checker-ul
- slfa1.zip: O arhivă care conține un model de Makefile pentru C și un executabil compilat pentru un sistem GNU/Linux x64. Makefile are reguli pentru build, run, clean și pack. make pack împachetează fișierele într-o arhivă ce poate fi încărcată direct pe vmchecker [1]
- tema-0-lfa.pdf: Enunțul temei
- 7_8.zip: Starea benzii după execuția pas cu pas a produsului din ultimul test, pentru a calcula $7 \cdot 8 = 56$. Arhiva conține patru fișiere:
  1. 7_8_elem: conține starea benzii după execuția fiecărei mașini Turing elementare
  2. 7_8_neelem: conține starea benzii după execuția fiecărei mașini Turing neelementare
  3. 7_8_tranz: conține starea benzii după selectarea unei tranziții, dar înainte de a începe execuția
  4. 7_8_tot: conține toate informațiile de la punctele anterioare.

  Antetul este:
  - Contor: e un simplu contor pentru ordonarea evenimentelor monitorizate
  - Tranz: reprezintă indicele tranziției. Începe de la 0.
  - Tip: Reprezintă tipul evenimentului: Elem →mașină Turing elementară; Apel →mașină Turing neelementară; Tran →Alegerea unei tranziții.
  - Bandă: Banda
- 2_1.zip: Similar cu 7_8.zip, doar că pentru a calcula $2 \cdot 1 = 2$

3.3 Depunctări

Implementările care nu folosesc flex primesc maxim 30% din punctajul oferit de checker.

Implementările care folosesc flex insuficient (o expresie imensă pentru recunoașterea textului și tokenizare + parsare „de mână” în cod C sau Java) vor fi depunctate cu maxim 30% din punctajul oferit de checker.

Deadline: 21.01.2018, 23:59. Upload-ul va rămâne deschis pâna la ora 07:00. Nu există întârzieri sau depunctări pentru întârzieri

Pentru că primiți un executabil care ia maxim pe temă, temele care conțin fișiere binare vor fi considerate încercări de fraudă și depunctate în totalitate. Aveți grijă să curățați tot înainte de upload sau folosiți comanda make pack din makefile pus la dispoziție!

4 Întrebări frecvente

1. Q: Am inceput sa fac tema folosing JFlex cu Java. Problema este ca am generat un lexer.java cu jflex, dar nu stiu cum sa il folosesc.

   A: Pentru un exemplu aproape complet, există tema 0.

2. Q: primesc urmatoarea eroare la compilare

   lex: input rules are too complicated (>= 32000 NFA states)

   A: Eroarea apare când încerci să parsezi bucăți foarte mari de text sau elemente recurente folosind direct expresii regulate. Recomandarea e să folosești start conditions și să parsezi pe bucățele mici

3. Q: Este vreo problema daca afisez mai multe "#"-uri la sfarsit?

   A: Da. În enunț este specificat foarte clar la reprezentarea benzii că trebuie afișat un singur '#' la început și sfârșit pentru a marca restul benzii.

4. Q: As dori sa stiu daca se ofera punctaje partiale pe tema in cazul in care nu e finalizata. De exemplu daca am facut doar parsarea fisierelor de intrare.

   A: Dacă trece cel puțin un test, da. Altfel, nu.

5. Q: Definitia masinii turing este recursiva. Flex NU suporta definitii recursive.

   A: Deși sunt definite recursiv se pot folosi start conditions pentru a parsa cum trebuie chiar și regulile astea.

Bibliografie

[1] vmchecker

[2] Materiale

[3] Documentație GNU Flex

[4] Ghid de utilizare Flex

[5] Pagina jflex

[6] Manualul utilizatorului jflex

[7] Manualul utilizatorului jflex în japoneză

[8] Laborator 1 SO: Makefile