IRINA ATHANASIU

DIANA RAICIU   RADU SION   IRINA MOCANII

Universitatea "Politehnica" din București
Catedra de Calculatoare

LIMBAJE FORMALE
Și
AUTOMATE

(îndrumar pentru aplicații)

©MATRIX ROM
CP. 16 - 162
77500 - BUCUREȘTI
tel. 01.4113617, fax 01.4114280
e-mail: matrix@fx.ro

Descrierea CIP a Bibliotecii Naționale a României

Limbaje formale și automate. îndrumar pentru aplicații/ Irina Athanasiu, Diana Raiciu, Radu Sion, Irina Mocanu, București, Matrix Rom, 2002
98 pagini, 25 cm
Bibliogr.
ISBN 973-685-407-8
Athanasiu, Irina
Raiciu, Diana
Sion, Radu
IV. Mocanu, Irina
004.43

ISBN 973 - 685 - 407 - 8

Despre autori,

Acest îndrumar are mulți autori cu o contribuție secvențială. A început cu niște foi scrise de mână care conțineau probleme și schițe de rezolvări, a mai existat și un text introductiv în lex în format electronic. în 1992 Diana Raiciu (în prezent Diana Mărculescu, profesor în Department of Electrical and Computer Engineering din Universitatea Carnegie Mellon, SUA) a realizat prima formă electronică a acestui îndrumar. Următorul autor (1999) a fost Radu Sion (în prezent doctorand în departamentul de calculatoare din Universitatea Purdue, SUA). A urmat în 2001 doamna asistenta Irina Mocanu. Fiecare dintre autori a corectat textul anterior și a mai adăugat probleme noi. Cu fiecare nou autor textul părea că devine gata. Am hotărât să public textul în formatul curent care desigur este încă departe de ce putea să fie, pentru că se aniversează anul acesta 10 ani de când credeam că îndrumarul va fi foarte curând gata de publicare.

Martie 2002 Irina Athanasiu

1

/ Elemente de teoria limbajelor formale 2

1.1 Gramatici 2

1.1.1 Ierarhia Chomsky 3

Probleme 4

1.1.2 Lema de pompare 16

Probleme 17

1.1.3 Transformări asupra GIC 18

1.1.3.1 Eliminare recursivitate stânga 18

1.1.3.2 Eliminare *X* producții 18

1.1.3.3 Eliminare simboli neutilizați 19

1.1.3.4 Eliminare simboli inaccesibili 20

Probleme 20

1 2 Mulțimi regulate. Expresii regulate 23

Probleme 24

1.3 Acceptoare 27

1.3.1 Automate finite 27

Probleme 28

1.3.2 Automate cu stivă (push-down) 44

Probleme 45

1.3.3 Mașina Turing 52

Probleme 54

2 Lex - generator de analizoare lexicale 65

2.1 Expresii regulate. Structura unei specificații lex. 66

2.2 Elemente avansate 70

2.2.1 Funcționarea analizorului lexical generat de lex 71

2.2.2 Stări de start 72

2.2.3 Macrodefiniții, funcții și variabile predefmite 73

2.2.4 Fișiere de intrare multiple .73

2.3 Exemple comentate 74

3 Teme pentru acasă 87

4 Bibliografie 91

1 Elemente de teoria limbajelor formale

Definiția 1.1. Se numește *alfabet* orice mulțime finită T de simboli.
Definiția 1.2. Se numește *limbaj peste un alfabet T* orice submulțime a mulțimii T*.

1.1 Gramatici

Definiția 1.1.1. O *gramatică* este un cvadruplu G = (N, T, P, S) unde:

- N este o mulțime finită de simboli numiți simboli neterminali
- T este o mulțime finită de simboli numiți terminale (N n T = 0)
- P este o submulțime a mulțimii (N u T)* N (N u T)* x (N u T)* și reprezintă mulțimea producțiilor gramaticii G. Un element p e P , p = (a,P) se reprezintă sub forma: oc -> p
- S e N se numește simbolul de start al gramaticii

Definiția 1.1.2. Se numește *formă propozițională în gramatica G* orice șir din (N u T)* obținut prin aplicarea recursivă a următoarelor reguli:

1. S este o formă propozițională;
2. Dacă ocpy este o formă propozițională și există o producție p -> 8 atunci și a5y este o formă propozițională.

Definiția 1.1.3. Se numește *propoziție generată de gramatica G* o formă propozițională în G care conține numai terminale.

Definiția 1.1.4. Se spune că între două forme propoziționale a și p în gramatica G există *relația de derivare* notată cu a =>p dacă există două șiruri wl, w2 și o producție y —> 8 astfel încât a = wlyw2 și P = wl8w2. Se spune că între formele propoziționale a și P există relația de derivare notată a $=>^k$ P dacă există formele propoziționale 8_0, Si, 82,..., §k astfel încât **a** = 80 => 81 =>... =>8k = p. închiderea tranzitivă a relației de derivare se notează cu =>"*

Definiția 1.1.5. Fie G o gramatică. Se numește *limbaj generat de gramatică G* și se notează L(G) mulțimea propozițiilor generate de G. Altfel spus:

L(G)={weT*|S=>"^{w}w}

Definiția 1.1.6. Două gramatici G1, G2 se numesc *echivalente* dacă generează același limbaj adică L(G1) = L(G2).

limbaje formale si translatoare Seminar LFA

1.1.1 Ierarhia Chomsky

Gramaticile pot să fie clasificate conform complexității producțiilor în următoarea ierarhie:

- gramatici de tip 0 (fără restricții) - au producțiile de forma:

a -» P cu a e (N u T)* N (N u T)* , P e (N u T)*

- gramatici de tip 1 (dependente de context - GDC) - au producțiile de forma:

aAP->ayp, a, Pe(NuT)*, A e N, y e (N U T)$^+$
sau de forma
S -> *k*.
în al doilea caz S nu apare în membrul drept al nici unei producții.

- gramatici de tip 2 (independente de context - GIC) au producțiile de forma:

A-^a,AsN,ae(NU T)*.

- gramatici de tip 3 (regulate la dreapta - GR) - au producțiile de forma:

A-H>ocBcuAeN,Be(Nu {A,}) și a e T*.

Corespunzător mulțimii gramaticilor G[k], k = 0,1, 2, 3 există mulțimea L[k] care reprezintă mulțimea limbajelor ce pot fi generate de o gramatică din G[k]. L[0] este mulțimea limbajelor generale care pot fi generate de o gramatică de tip 0 (fără restricții), L[1] este mulțimea limbajelor dependente de context (generate de o GDC), L[2] mulțimea limbajelor independente de context (generate de o GIC), iar L[3] mulțimea limbajelor regulate (generate de o GR). Se poate arăta că are loc incluziunea:

L[3]cL[2]cL[1]cL[0].

Termenul de "dependent de context" provine din faptul că producția ocAp —> ayP, poate să fie interpretată ca - neterminalul A poate să fie înlocuit cu șirul y doar în contextul dat de șirurile a și p. Similar termenul "independent de context" provine din faptul că o producție de forma A —>• a se poate interpreta ca - neterminalul A poate să fie înlocuit cu șirul a indiferent de context (indiferent de simbolii între care apare).

Se observă că într-o gramatică independentă de context prin aplicarea unei producții de formă A —» a asupra unei forme propoziționale, lungimea acesteia poate să devină mai mică (dacă a este șirul vid), egală (dacă **a** este un terminal sau neterminal) sau mai mare (dacă **a** este un șir de lungime mai mare decât 1). Acest lucru nu este valabil pentru gramaticile dependente de context unde lungimea unei forme propoziționale în urma aplicării unei producții de forma

aAP->ayP,a, P€(NuT)*, A e N, y e (N U T)$^+$

devine mai mare (dacă y este un șir de lungime mai mare decât 1), cel mult egală (dacă y este un șir format dintr-un neterminal sau un terminal). Prin urmare, nu este evident că orice gramatică independentă de context este și dependentă de context. Se poate demonstra că pentru orice gramatică independentă de context există o altă gramatică independentă de context echivalentă obținută prin transformări, astfel încât modul de creștere a lungimii formelor propoziționale să fie similară cu cea a gramaticilor dependente de context.

Intuitiv, o caracterizare a gramaticilor de mai sus poate fi făcută din punctul de vedere al "numărării" simbolilor. Astfel, gramaticile regulate pot "număra" până la o limita finită. De exemplu, limbajul specificat prin L = { 0 T | 0 < i < 5 } poate fi generat de o gramatică regulata

G astfel: S-> 0000011111 | 00001111 | 000111 | 0011 | 01 | *X*

De asemenea, o gramatică independentă de context poate păstra un singur contor (limita nu este cunoscută sau fixată). De exemplu, limbajul L = { 0T j n > 0 } poate să fie generat de gramatică independentă de context G = ({ S }, {0, 1}, P, S) unde P = { S -» OS 1 | *X* }.

Probleme

Să se construiască gramaticile care generează următoarele limbaje:

proHeiTu 1.1-1

$L = \{ a^n b^n j n > 0 \}$

Soluție:

Orice șir nevid w e L începe cu simbolul a și se termină cu simbolul b, astfel încât poate să fie scris sub forma w = a x b unde x este un șir de terminale care fie este vid, fie este de aceeași formă cu șirul w. Prin urmare o gramatică G care generează limbajul L poate să fie: G = ({ S }, { a, b }, P, S) unde: P = { S-» a S b | *X* }.

Verificare:

Pentru șirul vid *(X)* secvența de derivări este S => L Pentru șirul aaabbb secvența de derivări este S ; = > a S b = > a a S b b = > a a a S b b b = > a a a b b b

Comentarii:

Gramatica G este independentă de context. Se poate arăta că în acest caz nu se poate construi o gramatică regulată care să genereze același limbaj, adică să fie echivalentă cu prima.

$L = \{ a^n b^n \ i\ n > 0 \}$

Soluție:

Orice șir nevid w e L începe cu simbolul a și se termină cu simbolul b, astfel încât poate să fie scris sub forma w = a x b unde x este un șir de terminale care fie este șirul ab (n > 0, nu se acceptă șirul vid), fie este de aceeași formă cu șirul w. Prin urmare o gramatică G care generează limbajul L poate să fie: G = ({ S }, { a, b}, P, S) unde: P = { S-> a S b | ab }.

Verificatei

Pentru șirul aaabbb secvența de derivări este S = > a S b = » a a S b b = > a a a b b b

Comentarii: . . . . -

Gramatica G este independentă de context

problem» I.î-3

$L = \{ a^n b^n c^m d^m j n > l\ \text{și}\ m > l \}$

Soluție:

Observăm că pentru orice șir w din L există două subșiruri independente wl = $a^n b^n$ respectiv w2 = $c^{ra} d^m$ care pot fi generate separat. Vom utiliza două neterminale A,B pentru generarea celor două subșiruri independente wl și w2. Primul subșir începe întotdeauna cu a și se termină cu b, adică wl = $a^n b^n$ se poate scrie sub forma wl = a v b cu v un șir care este de aceeași formă cu wl, și va fi generat în același mod, sau este șirul vid. în mod analog, subșirul w2 = $c^m d^m$ poate fi generat astfel: w2 începe în mod sigur cu simbolul c și se termină cu simbolul d. Prin urmare, w2 se poate scrie w2 = c v d, unde v este de aceeași formă cu w2 sau este șirul vid. Astfel, mulțimea producțiilor va conține producții de forma: A -» aAb | ab și B -> cBd | cd. Pentru concatenarea celor două subșiruri, vom introduce producția S -» AB. Prin urmare o gramatică care generează limbajul L este: G = ({ S, A, B }, { a, b, c, d }, P, S) unde: P = { S->AB, A-»aAb | ab, B ->cBd | cd }.

Verificare;

Pentru șirurile a b c c d d (n = 1 și m = 2), respectiv a a b b c d (n = 2 și m = 1) se obțin următoarele secvențe de derivări: S=> A B=> a b B=> a b c B d=> a b c c d d respectiv S=> A B=> a A b *B*=> a a b b *B*=> a a b b c d.

Comentarii:

Gramatica G este independentă de context

* problema-4,1^- '• "_ ^>l\:-\'-f'f%V--/./T'T."ij"-j~V°."""""*'i""--"---"t^".<v"> *ST.'*

$L = \{ a^n b^n c^m d^m \mid n > 0\ \text{și}\ m > 0 \}$

Soluție:

Observăm că pentru orice șir w din L există două subșiruri independente wl = $a^n b^n$ respectiv w2 = $c^m d^m$ care pot fi generate separat. Vom utiliza două neterminale A,B pentru generarea , celor două subșiruri independente wl și w2. Primul subșir începe întotdeauna cu a și se termină cu b, deci wl = $a^n b^n$ se poate scrie sub forma wl = a v b cu v un șir care este de aceeași formă cu wl, și va fi generat în același mod; deoarece *n > 0* wl poate fi și șirul vid. In mod analog, subșirul w2 = $c^m d^m$ poate fi generat astfel: w2 începe în mod sigur cu simbolul c și se termină cu simbolul d. Prin urmare, w2 se poate scrie w2 = c v d, unde v este de aceeași formă cu w2; deoarece *m > 0* w2 poate fi și șirul vid. Astfel, mulțimea producțiilor va conține producții de forma: A -> aAb | *X* și B -» cBd | *X*. Pentru concatenarea celor două subșiruri, vom introduce producția S -» AB. Prin urmare o gramatică care generează limbajul L este: G = ({ S, A, B }, { a, b, c, d }, P, S) unde: P = { S->AB, A->aAb | A B

Verificare:

Pentru șirurile a b (n = 1 și m = 0), respectiv c d (n = 0 și m = 1) se obțin următoarele secvențe de derivări: S=> A *B*=> a A b *B*=> a b B => ab și S=> A B=> B=> c B d => c d.

problema 1.1-5

L = { $a^nb^mc^md^n$ | n > 1 și m y 1 }

Soluție:
Orice șir w din L începe cu simbolul a, și se termină cu simbolul d. w se poate scrie ca w = a u d unde u este un șir care fie este de tipul celui din mulțimea L, fie este de forma b^mc^m, m > 1. Acesta din urmă se poate scrie sub forma b v c cu v de aceeași formă sau v este șirul vid. Astfel, ținând seama de observațiile anterioare, o gramatică ce generează limbajul L este: G = ({ S, A-}, { a, b, c, d }, P, S) unde P = { S - * a S d | a A d , A - > - b A c | b c } .

Verificare:
Pentru șirul a a b c d d (n = 2 ș i m = 1) s e obține următoarea secvență de derivări: a S d=> a a A d d=> a a b c d d.

Comentarii:
Se observă că neterminalul A este utilizat pentru a genera subșiruri de forma b^mc^m. Gramatica G este independentă de context.

problema 1.1-6

L = {w e { 0,1 }* | w conține un număr egal de 0 și 1 }

Soluție:
Fie w un șir oarecare astfel încât w e { 0, 1 }* și $\#_0(w)$ = #,(w) unde prin $\#_a(w)$ am notat numărul de apariții ale simbolului a în șirul w. Dacă w începe cu simbolul 0, atunci există u și v, u,v e { 0, 1 }* cu u și v având fiecare un număr egal de 0 și 1 (eventual zero) astfel încât șirul w se poate scrie w = 0 u 1 v, cu u, v e { 0,1 }* cu număr egal de 0 și 1 astfel încât w = 0 u 1 v. Producția care generează acest tip de șiruri este: S -> 0 S 1 S. De asemenea, dacă w e L începe cu 1, se arată în mod analog că w se poate scrie w = l u 0 v c u u , v șiruri cu număr egal de 0 și 1. Pe baza celor de mai sus, rezultă că o gramatică care generează limbajul L este G = ({ S }, { 0, 1 }, P, S) unde: P = { S - > O S 1 S | 1 S O S | A . } .

Verificare:
Pentru șirul 0 1 0 1 1 0, se obține următoarea secvență de derivări: S => 0 S 1 S 01 SOS 1 S = > 0 1 0 S 1 S = ^ 0 1 0 1 S = ^ 0 1 0 1 1 S O S = > b 1 0 1 1 0 S = * 0 1 0 110.

Comentarii:
Gramatica G este o gramatică de independentă de context.

problema 1.1.-7 •_ ! ..

L = { w e { 0,1 }* | w nu conține subșirul 011 }

Soluție:
Presupunem că parcurgem un șir din limbajul L începând cu simbolul cel mai din stânga. Se observă că pot să existe mai multe situații în ceea ce privește simbolul curent:
1 Simbolul curent este 1 și cel anterior a fost tot 1. Atâta timp cât simbolul curent este 1 nu suntem la un început al unui sub șir de forma 011.
2. Simbolul analizat este 0 adică suntem la începutul unei posibile secvențe 011 care trebuie rejectată. Atâta timp cât simbolul curent este 0 este aceași situație. Dacă simbolul curent este 1 se trece în situația următoare (3).
3. Simbolul curent este 1 și cel precedent a fost 0, adică suntem în interiorul unei posibile secvențe 011 care trebuie rejectată. De aici, singurul simbol posibil este 0 (1 ar genera secvența 011), care ar putea să înceapă o nouă posibila secvență 011. Ca urmare, pentru un simbol 1 suntem în situația 2.
Vom codifica cele trei situații posibile prin neterminalele S, A, B. Corespunzător celor de mai sus o gramatică G care generează limbajul L este: G = ({ S, A, B }, { 0,1 }, P, S), P = { S -> ÎS | 0A | *X*, A -» 0A 11B | *X*, B -» 0A | *X*}.

Verificare:
Pentru șirurile: 10 1, respectiv 0 10 1, secvențele de derivări sunt: S=> 1 S=> 1 0 A=> 10 1 B=> 1 0 1 și S=> 0 A=> 0 1 B=> 0 1 0 A=> 0 1 0 1 B=> 0 1 0 1.

Comentarii:
Gramatică G este regulată, iar limbajul L este de un limbaj regulat.

pichk-iis.i I.1-K
L = { w 6 { 0 , 1 } * |w| divizibilă cu 3 }

Soluție:
Vom presupune că parcurgem șirul w de la stânga la dreapta considerând un prefix al acestuia: w = u a v unde u este prefixul, a este simbolul curent (0 sau 1), iar v este restul șirului (u, v e { 0,1 }*). Sunt posibile următoarele situații:
1. Numărul de simboli din prefixul u este de forma 3 *n, n e N
2. Numărul de simboli din prefixul u este de forma 3*n + 1
3. Numărul de simboli din prefixul u este de forma 3*n+2
Asociem neterminalul S primei stări (care este de altfel cea corespunzătoare generării unui Șir din limbajul L), iar celorlalte două stări neterminalele A, respectiv B. Din prima stare se trece în a doua după ce se "citește" simbolul curent (numărul de simboli citiți devine 3*n+1), iar de aici, după "citirea" unui nou simbol, se trece în starea 3 și apoi din nou în starea 1 după "citirea" următorului simbol din șir, etc. Gramatica care generează limbajul L este: G = ({ S, A, B }, { 0,1 }, p, S) unde P = { S -» 0A | IA | *X* , A -> 0B 11B, B -> OS | ÎS }.

Verificare:
Pentru șirul 010011, se obține secvența de derivări S=> 0A=> 01B=> 010S=> 0100A=> 01001B=> 010011S=> 010011.

Comentarii:
Altă gramatică regulată echivalentă este Gl = ({ S }, { 0, 1 }, P, S) unde: P = { S - > 0 0 0 S | 0 0 1 S 10 1 0 S I 0 1 1 S I 1 0 0 S | 1 0 1 S 11 1 0 S 11 1 1 S | *X*}.

I'rnliluii.i I I-'>

Soluție:

Șirurile din L pot fi de două tipuri: cu i > j, respectiv i < j. Astfel, un șir w e L poate fi scris fie sub forma w = $0^j0^n1^j$ dacă i > j și n = i - j, fie sub forma w = OT1' dacă i < j și n = j - i. în ambele cazuri, n > 0. Pentru generarea primului șir, vom observa că acesta începe cu 0 și ; se termină cu 1 și poate fi scris sub forma w = 0 u 1 unde u este fie de aceeași formă, fie de formă 0", cu n > 0. în mod analog, cel de-al doilea șir începe cu 0 și se termină cu 1 și poate i fi scris sub formă w = 0 u 1 unde u este fie un șir de aceeași formă, fie de forma 1", cu n > 0. i Ținând cont de observațiile de mai sus, o gramatică ce generează limbajul L este G = ({ S, A, |B}, {0,1 }, P, S) unde P = {S -» 0 S l | 0 A | 1 B, A -> 0 A | A., B -> IB | A.}.

Verificare:

Pentru șirul 0 0 0 1 se obține următoarea secvență de derivări: S => 0 S 1 ⟹ 0 0 A l=> 0 0 0 A l = > 0 0 0 1 .

Comentarii:

G este o gramatică independentă de context.

Problema 1.1-10

L = { x^my^n | n < m sau 2*m < n, n, m > 1 }

Soluție:

Limbajul L conține două tipuri de șiruri: șiruri de forma x^my^n cu 2*m < n, respectiv cu n < m. , Primul tip de șir poate să fie scris sub forma $x^my^ry^{2*m}$ unde n = 2 * m + r, cu r > 0 și m > 1 : iar pentru generarea lui pot să fie utilizate producțiile: Sj -> x A y y, A -> x A y y | y C ș i C i -> y C | A. Al doilea tip de șir poate să fie scris sub forma $x^nx^ty^n$ unde m = n + r c u r > 0 ș i n > 1. Pentru generarea acestui șir se pot utiliza producțiile: S_2 -> x B y, B -> x B y | x D, D ; -> x D | *X*. Prin urmare, o gramatică care generează limbajul L este: G = ({ S, Si, S_2, A, B, C, \ D }, { x, y }, P, S) unde P = { S -> Si | S_2, S, -> x A y y, A -» x A y y | y C, S_2 -» x B y, B -> x B y [x D , C -> y C | A, D -> x D | A }.

Verificare:

Pentru șirul x x x y y (pentru care m = 3 și n = 2, adică m > n) se obține secvența de derivări: s = * S2=> x B *y*=> *x* x B y *y*=> x x x D y y=> x x x y y. Fie șirul x x y y y y y (pentru care m = 2 și n = 5 adică 2 * m< n). Pentru acest șir se obține secvența de derivări: S=> Si=> x A y y=> x x A y y y y=> x x y C y y y y = > x x y y y y y .

Comentarii:

G este o gramatică independentă de context

pinlil'.lll.l 1.1-l 1

L = { $a^mb^nc^pd^q$ | m + n = p + q , m, n, p, q > 0 }

Soluție:

Relația din enunț poate fi scrisă sub forma m - q = p - n. Pot apare două cazuri:

1 m > q- Notăm r = m - q = p - n , unde r > 0 și m = q + r & p = n + r. Fie LI limbajul generat în acest caz. Atunci LI poate fi descris prin: LI = { $a^qa^rb^nc^nc^rd^q$ | n,q > 0, r > 0 }.Orice șir de aceasta formă, poate să fie scris ca w = a u d unde u este de aceeași formă cu w sau de forma $a^rb^nc^nc^r$ și în acest caz poate fi scris sub forma: u = a v c unde v este fie de aceeași formă cu u, fie este de forma b^nc^n. în acest caz, v poate fi scris sub formă v = b z c unde șirul z este de aceeași formă cu v, fie șirul vid. Prin urmare, vor fi utilizate producțiile: S -> aSd | A, A -> aAc | B, B -» bBc | A

2. m < q. Notăm r = q - m = n - p , unde r > 0 și prin urmare n = p + r & q = m + r. Fie L2 limbajul generat în acest caz. Prin urmare L2 = {$a^mb^rb^pc^pd^rd^m$ | m, p>0, r>0}.Un șir | oarecare w eL2, poate fi scris sub formă w = a u d, unde u este de aceeași formă cu w, j sau w este de formă $b^rb^pc^pd^r$ și în acest caz u fie se poate scrie sub forma u - b v . d c u v ; un șir de aceeași formă cu u, sau u este de forma x = b^pc^p. Șirul nou obținut poate fi ; generat observând că se poate scrie sub forma x = b z c unde z este un șir de aceeași ' I formă cu x. Pot să fie utilizate, producțiile: S -»• aSd | bCd, C -» bCd | D, D -> bDc | *X*. \ Se observă că L = LI u L2 și că cele două neterminale B și D se comportă la fel (adică i generează același limbaj), și putem renunța la neterminalul D. |

Prin urmare, o gramatică G care generează limbajul L este: G = ({ S, A, B, C }, { a, b, c, d }, j P, S) unde P = { S -> aSd | A | bCd, A -» aAc | B, B -> bBc | *X*, C H> bCd | B }.

Verificare:

Vom considera două exemple de derivare: a a b c c c (cazul m > q), respectiv a b b c d d ! (cazul m < q). Pentru șirul a a b c c c se obține secvența de derivări: S => A => a A c=> a a ; • A c c=> a a B c c=> a a b B c c c = > a a b c c c , respectiv pentru cel de-al doilea șir: S=> a S |

G este o gramatică independentă de context.

Problema li-42-7 "1 * ;"' <^*«f.'

: L = { a^mb^n | n < m < 2 * n , n , m > l } ;

Soluție:

Notăm c u p = m - n ș i q = 2 * n - m unde p > 0, q > 0. Prin urmare, n = p + q ș i m = 2 * p + i : q Și limbajul poate fi scris sub forma: L = { $a^{2*p}a^qb^qb^p$ | p, q > 0 } : ; Un șir care aparține acestui limbaj, poate să fie scris sub forma wl = a a u b unde u este de ; : aceeași formă cu wl sau de formă a^qb^q. j

Șirul a^qb^q poate să fie generat în mod asemănător lui w1, observând că poate să fie scris sub i forma w2 = a v b unde v este de aceeași formă cu w2 sau este șirul ab. Prin urmare o [gramaticaeste G = ({ S, A }, { a, b }, P, S)unde P = { S -> aaSb | aaAb, A -» aAbij ab}.................

Verificare:

Fie șirul: a a a b b (n = 2, m = 3). Șirul poate să fie obținut utilizând următoarea secvență de derivări: S=> a a A b=> a a a b b.

Comentarii:

. G este o gramatică, independentă de context

problema 1.1-13

L = { wcw^R | w e {a,b}* } unde prin u = w^R am notat șirul care are proprietatea u, = $w_{n'i+}$i pentru orice i = 1, 2,…, n unde n = |wj = |u| și w = wi w_2… w_n, u = ui U2… u_n (șirul reflectat corespunzător lui w).

Soluție:

Orice șir din limbajul I, are proprietatea că începe și se termină cu același simbol sau este doar simbolul c. în primul caz șirul poate să fie scris sub forma w = x u x unde x e { a, b }, iar u are aceeași proprietate ca w. Luând în considerare cele de mai sus, gramatica G care generează limbajul L este G = ({ S }, { a, b, c }, P, S) unde P = { S -> aSa | bSb | c }. într-adevăr, orice șir u generat de G are proprietatea că există w e {a,b}* astfel încât u = wcw^R și pentru orice șir u e L, există o derivare S =*> u.

Verificare:

Pentru șirul a b a c a b a, se obține următoarea secvență de derivări: S => a S a: a b S b a = > a b a S b a b a = > a b a c a b a

Comentarii:

G este o gramatică independentă de context

problema 1.1-14

L = {w e {a,b}* w = w^R}

Soluție:

Un șir w care respectă condiția de mai sus poate să aibă una din următoarele forme: v v^R,vav^R, vbv^R unde v e { a, b }* este un șir oarecare. Pentru a genera aceste șiruri, putem să utilizăm gramatica G = ({ S }, { a, b }, P, S) cu P = { S ->• aSa | bSb | a | b | *X}*.

Verificare:

Fie șirul a b a a b a a b a . Pentru obținerea acestui șir putem să considerăm următoarea secvență de derivări: S = > a S a = > a b S b a = > a b a S a b a = > a b a a S a a b a = > a b a a b a a b a.

Comentarii: . , .

G este o gramatică independentă de context

IMLIII.I I.! 15

L =

Soluție:

Orice șir nevid w e L conține la începutul șirului toate simbolurile a, după care urmează toate simbolurile b și se termină cu simbolurile c. Pentru a genera toate a-urile se va folosi neterminalul Si, pentru a genera toate b-urile se va folosi neterminalul S_2, iar pentru a genera toate c-urile se va folosi neterminalul S_3. Prin urmare, gramatica G care generează limbajul L poate să fie: G = ({ S, S,, S_2, S_3 }, { a, b, c }, P, S) unde: P = {S-> S,S_2S_3, S, -» a S, | a, S_2 -> b S_2|b, S_3 -> c S_3|c}.

]erificare:

Fie șirul aabbbcccc. Pentru obținerea acestui șir putem să considerăm următoarea secvență de derivări: S=> SiS_2S_3=> a SiS_2S_3 => a a S_2S_3 => a a b S_2S_3=> a a b b S_2S_3 => a a b b b S_3=> a a ; b b c S 3 = > a a b b c c S_3 = > a a b b c c c S_3 = > a a b b b c c c c :

Comentarii:

G este o gramatică independentă de context ,

problema 1.1-16

L= *{aVc^k* i#j sauj #c, i,j,k>0 } i

Soluție:

Pot apare 4 cazuri:

', 1. Dacă i < j, notăm p = j - I, rezultă j = p + i, p > 0. Fie LI limbajul generat în acest caz. j
: Atunci LI poate fi descris prin: LI = { aibibpck | p > 0, i, k > 0 }. Vom folosi j
I netermmalul S1 pentru generarea subșirului aibi, pentru a genera subșirul bp se va folosi |
neterminalul S2, iar pentru a genera subșirul ck se va folosi neterminalul S3. Prin urmare |
: vor fi utilizate producțiile S -» SI S2 S3, SI -» a SI b | a b, S2 -» b S2 j b,.S3 -> c S3 '| j
c. i

: 2. Dacă i > j, notăm p = i - j, rezultă i = p + j, p > 0. Fie L2 limbajul generat în acest caz. :
: Atunci L2 poate fi descris prin: L2 = { a'VbV | p > 0, j, k > 0 }. Vom folosi neterminalul j
; Si pentru generarea subșirului a^p, pentru a genera subșirul a'b' se va folosi neterminalul ;
i S2, iar pentru a genera subșirul c^k se va folosi neterminalul S_3. Prin urmare vor fi utilizate j
producțiile S ^ Si S_2 S_3, SI -» a S, | a, S_2-» a S_2 b | ab, S_3->cS_3|c. j

3. Dacă j < k, notăm p = k - j, adică k = j + p, p > 0. Fie L3 limbajul generat în acest caz. j
; Atunci L3 poate fi descris prin: L3 = { aWc[15]1 p > 0, i,j > 0 }. Vom folosi neterminalul j
Sj pentru generarea subșirului a^l, pentru a genera subșirul bV se va folosi neterminalul j
S_2, iar pentru a genera subșirul c^p se va folosi neterminalul S_3. Prin urmare vor fi utilizate j
… .producțiile S-» Si S_2S_3, Si-»aȘi |a, S_2-»bS_2c|bc, S_3-^cS_3|c……………………I

4. Dacă j > k, notăm p = j - k, și j = p + k, p > 0. Fie L4 limbajul generat în acest caz Atunci L4 poate fi descris prin: L4 = { a'b^{p}b^{k}c^k | p > 0, i, k > 0 }. Vom folosi neterminalul Si pentru generarea subșirului a', pentru a genera subșirul b^p se va folosi neterminalul S_2, iar pentru a genera subșirul b^{k}c^k se va folosi neterminalul S_3. Prin urmare vor fi utilizate producțiile S -» S, S_2 S_3, S, -* a Si | a, S_2 ->• b $S_2$1 b, S_3 -» b S_3 c | b c.

Corespunzător celor 4 cazuri notăm neterminalele Si, S_2, S_3 cu Si_a, $S]_b$, Si_c, $S_{!d}$, S_{2a}, S_{2b}, S_{2c}, S_2d, S_{3a}, S_{3b}, S_{4c}, S_{3d}. Prin urmare gramatica ce generează limbajul este: G = ({ S, Si_a, S_{2a}, S_{3a}, $S_{,b}$, S_{2b}, S_{3b}, $S_{,c>}$ S_{2c}, S_{3c}, Sid, S_{2d}, S_{3d} }, { a, b, c }, P, S) unde: P = { S-» $Si_a S_{2a} S_{3a}$ | $Si_b S_2 b S_3 b$ | $Si_c S_{2c} S_{3c}$ j $Si_d S_{2d} S_3 d$, Si_a —>• a Sj_a b | a b, S_{2a} —> b S_{2a} | b, S_{3a} —• c S_{3a} | c , Sjb —> a Sib | a , S_{2b} -> a S_{2b} b | a b, S_{3b} -» c S_{3b} | c, Si_c -> a Si_c | a , S_{2c} -> b S_{2c} c | b c, S_{3c} -» c S_{3c} | c, Sid -> a Sid | a , S_{2d} -> b S_{2d} | b, S_{3d} H> b S_{3d} c | b c }.

G este o gramatică independentă de context

L={a^{i}b^{j}c^k|i<j,i,j,k>0}

Notăm p = j - i, adică j = i + p, p > 0. Atunci L poate fi descris prin: L = { a'b'b^{p}c^k j p>0, i, k > 0 }. Vom folosi neterminalul A pentru generarea subșirului a'b^i, pentru a genera subșirul b^p se va folosi neterminalul B, iar pentru a genera subșirul c^k se va folosi neterminalul C. Prin urmare, gramatica G care generează limbajul L poate să fie: G = ({ S, A, B, C }, { a, b, c }, P, S) unde: P = { S->ABC, A -> a A b | A, B -> b B | b, C -> c C | A.}.

G este o gramatică independentă de context

L={a^{i}b^{i}c^k|i+j=k,i,j,k>0}

Dacă k = i + j atunci L poate fi descris prin: L = { a'Wc'c' | i, j > 0 }. Orice șir w din limbaj începe cu simbolul a, și se termină cu simbolul c. Deci, w se poate scrie ca w = a u c unde u este un șir care fie este de tipul celui din mulțimea L, fie este de forma *Vd,* j > 0 . Acesta din urmă se poate scrie sub forma b v c cu v de aceeași formă sau v este șirul bc. Prin urmare gramatica G care generează limbajul L poate să fie: G = ({ S, A }, { a, b, c }, P, S) unde: P = {S n>a S c|a A c, A->b A c|b c}.

G este o gramatică independentă de context

1.1-1')

I \\ I ÎMI

Soluție:

Vom presupune că pai curgem .-.irul \\ de la aiânga ia Jivapia considerând un prefix al acestuia: w = u a v unde u este prefixul, a este simbolul curent (a sau b), iar v este restul șirului (u, v € { a, b }*). Sunt posibile următoarele situații:

1. Numărul de simboli a din prefixul u este par
2. Numărul de simboli a din prefixul u este impar

Asociem neterminalul S primei stări (care este de altfel ceea corespunzătoare generării unui șir din limbajul L), iar celeilalte stări neterminalul A. Din prima stare se trece în a doua dacă simbolul curent a fost un a și se rămâne în starea S dacă simbolul curent a fost un b. Din starea A se trece în starea S dacă simbolul curent a fost un a și se rămâne în starea A dacă simbolul curent a fost un b. Deci gramatica care generează limbajul L este: G =.({ S, A }, { a, b }, P, S) unde P={S->aA|bS|Â/, A^bA|aS}.

Comentarii:

G este o gramatică regulată

jiiuiiliin.i 1.1 *IU*

L={we{a,b}*|$\#_a$(w) = 2*$\#_b$(w)}

Soluție:

Pentru a genera șirurile din limbaj de câte ori apare un simbol b trebuie sa apară doi simboli a. Nu este importantă ordinea în care apar simbolii. Deci gramatica care generează limbajul L este: G = ({ S }, { a, b }, P, S) unde P = {S->aSaSbS|aSbSaS|bSaSaS|Â,}.

Comentarii:

G este o gramatică independentă de context

Limbajul parantezelor bine formate

Soluție:

Gramatica care generează limbajul L este: G = ({ S },{(,)}, P, S) unde P = { S -> S S | (S) !()} sau G = ({ S },{(,)}, P, S) unde P = { S ^ S S | (S) | X } dacă este acceptat șirul vid.

Comentarii:

G este o gramatică independentă de context

problema 1.1-22

L={wwjwe {0, 1 }* }

Soluție:

Analizând forma generală a șirurilor ce fac parte din limbajul L se observă că în generarea acestor șiruri este necesară " memorarea " primei jumătăți a șirurilor pe măsură ce se face această generare, pentru *a* putea utiliza apoi informația memorată în generarea celei de-a doua jumătăți a șirului. în mod evident, această "memorare" nu se poate realiza utilizând mecanismele oferite de gramaticile regulate sau independente de context. Am văzut că se pot genera șiruri de formă w / c u ajutorul unor gramatici independente de context.
Vom genera un șir în care unui simbol 1 din prima jumătate a șirului îi va corespunde un simbol notat U în a doua jumătate a șirului. în mod corespunzător, unui simbol 0 din prima jumătate a șirului îi va corespunde un simbol notat Z în a doua jumătate a șirului. Pentru a marca sfârșitul șirului vom utiliza un neterminal B. Se observă că în acest mod s-a memorat prima jumătate a șirului într-o formă inversată. în continuare, din modul în care se utilizează următoarele derivări ar trebui să rezulte și restul șirului. Astfel, transformările pe care le vom opera asupra unui șir generat ca mai sus, pot să fie exprimate în termeni intuitivi astfel: orice Z sau U se transformă în 0, respectiv 1 dacă sunt urmate de terminatorul B, iar orice simbol 0' sau 1 se deplasează către capătul din stânga, respectiv orice simbol Z sau U se deplasează către dreapta (pentru a ajunge la terminatorul B).
De exemplu: 001UZZB => 001UZ0B => 001UOZB => 001U00B => 0010U0B => 00100UB => **001001B** => **001001.** Astfel, mulțimea producțiilor gramaticii va conține S -> A B cu neterminalul A utilizat pentru crearea unui șir de forma w_t w_2 unde wi e { 0, 1 }*, w_2 e { U, Z}*, iar între wi și w_2 există relația: wi = w_2^R în care U = 1, Z = 0.
Acest șir este generat prin producțiile: A -> 0 A Z | 1 A U | *X.* Pentru generarea limbajului L mai sunt necesare producțiile care realizează înlocuirea neterminalelor U, Z cu 1, respectiv 0 dacă sunt urmate de neterminalul B și dubla deplasare a simbolilor: Z B -> 0 B, U B -> 1 B, Z 0 -> 0 Z, U 0 -» 0 U, Z 1 -> 1 Z, U 1 -> 1 U. De asemenea, este necesară producția B -» A. pentru a elimina simbolul utilizat ca terminator. Deci o gramatică G care generează limbajul L este G = ({ S, A, B, U, Z }, { 0, 1 }, P, S) unde: P={S^AB, A->OAZ|1AU|A., Z B->0B, UB->1B, Z0-»0Z, U0->0U, Z1->1Z, U1->1U, B->A.}.

' criticare:

Pentru șirul 0 1 0 1 se obține șirul de derivări: S=>AB: • 0AZB=>0 1AUZB=>0 ZB=>01U0B=>010UB=>0101B=> 0101

Comentarii:

Se observă că G este o gramatică de tip 0 (tară restricții).
Dacă se utilizează producția B -* *l* într-un șir de derivări înaintea transformării tuturor neterminalelor U și Z, se poate obține o formă propozițională din care nu se mai poate obține un șir din limbajul L.

De exemplu: S => AB => 0AZB => 0AZ și nu mai există nici o producție care să poată să fie aplicată. Din forma propozițională obținută nu se mai poate obține un șir care să aparțină limbajului L. Astfel de rezultate țin de aspectul nedeterminist al gramaticii respective.

1.1-23

L={a^nbV|n>1}

Soluție:
Problema poate să fie abordată într-o manieră similară cu cea anterioară adică se „memorează" prima parte a șirului (a^n), însă generarea nu poate continua verificând că există un număr corespunzător de simboli b și de simboli c. Prin urmare, va trebui să „memorăm" prima parte a șirului (a^{11}) și apoi să generăm același număr de perechi de forma (BC) unde B, C sunt neterminale asociate cu terminalele b, c. în continuare, din derivările ulterioare, va rezulta
transformarea celei de-a doua părți a șirului în șirul b^nc^n. Vom considera deci, șirurile de forma $a^n(BC)^n$ unde B,C sunt neterminale prin intermediul cărora se va face generarea subșirului b^nc^n. Pentru aceasta este necesară substituția oricărui subșir de forma CB cu șirul BC, respectiv înlocuirea neterminalelor B,C cu b, respectiv c. Astfel, o gramatică G care generează limbajul L este G = ({ S, B, C }, { a, b, c }, P, S) unde: P = { S-» aSBC | abC, CB -> BC, bB -» bb, bC ->' bc, cB -> Bc, cC -> ce }.

Verificare:
Pentru șirul $a^3b^3c^3$ de exemplu, se obține șirul de derivări: S=>aSBC=>aaSBCBC=> aaabCBCBC=>aaabBCCBC=>aaabBCBCC=>aaabBBCCC=>aaab bBCCC=>aaabbbCCC=>aaabbbcCC=> aaabbbccC=> aaabbbccc = $a^3b^3c^3$.

Comentarii:
G este o gramatică fără restricții

1.1-24

! L = {$a^mb^nc^md^n$|m, n>1}

Soluție:
Vom genera întâi forme propoziționale de tipul $a^mb^nD^nC^m$ unde C, D sunt neterminale asociate terminalelor c, d. într-adevăr, o formă propozițională ca mai sus poate fi generată de producțiile: S->aSCjaAC, A->bAD|bD.
Următorul pas este deplasarea simultană a simbolilor C spre stânga și respectiv D spre dreapta, înlocuirea cu terminalele corespunzătoare c, d făcându-se de la stânga la dreapta: DC -> CD, bC -> bc, cC -» ce, cD -> cd, dC -> Cd, dD -> dd. Astfel, o gramatică G care generează limbajul L este G = ({ S, A, C, D }, { a, b, c, d }, P, S) unde: P={SH>aSC|a ^ Q A -> b A D | b D, DC -> CD, bC -> bc, cC -> ce, cD-> cd, dC ->• Cd, dD -> dd }.

Verificare:

Pentru șirul $a^2b^3c^2d^3$ se obține următoarea secvență de derivări: S => aSC =?>aaACC=>aa bADCC=>aabbADDCC=*aabbbDDDCC=>aabbbDDCDC=>aabbb DCDDC=>aabbbCDDDC=>aabbbcDDDC=>aabbbcDDCD=^aabb bcdDCD=>aabbbcddCD=>aabbbcdCdD=>aabbbcCddD=>aabbbc cddD=>aabbbccddd=$a^2b^3c^2d^3$.

Comentarii:

Această gramatică *realizează* abstractizarea corespondenței numărului de parametri cu care a : fost definită o procedură și numărul de parametri cu care aceasta se apelează. Deoarece în procesul de compilare analiza sintactică utilizează gramatici independente de context, această verificare (care comportă o gramatică fără restricții) se va face în faza de analiză semantică și nu în cea de analiză sintactică.

problem;! 1.1-25

L={aVc"d"|n>l }

Soluție:

Vom considera deci, șirurile de forma a"(bCD)" unde C, D sunt ncterminale prin intermediul cărora se va face generarea subșirului c"d". Pentru aceasta este necesară substituția oricărui subșir de forma DC cu șirul CD, respectiv înlocuirea neterminalelor C, D cu c, respectiv d. Astfel, o gramatică G care generează limbajul L este G = ({ S, C, D}, { a, b, c, d }, P, S) unde: P = { S-> aSbCD | abCD, DC -» CD, Db ->bD. Cb -> bC, bC -> bc, cC H> CC, CD -» cd,dD->dd}.

('oinenlarii:

G este o gramatică fără restricții

1.1.2 Lema de pompare

Propoziție. Pentru orice limbaj L independent de context există un număr natural kj caracteristic limbajului, astfel încât dacă z **e** L și |z| > k, atunci z poate fi scris sub forma z = uvwxy îndeplinind următoarele condiții:

2. |vwx|<k
3. uv'wx'y e L pentru orice i

Probleme

Să se demonstreze că următoarele limbaje nu sunt independente de context.

problem» 1.1-26

L={$a^nb^nc^n$|n>0}

Soluție:

Presupunem prin absurd că L este independent de context, deci este aplicabilă lema de pompare. Fie n = k și fie șirul: z = $a^kb^kc^k$. într-adevăr, |z| > k, deci există șirurile u, v, w, x, y i astfel încât z = uvwxy cu vx *4 X* și |vwx| < k.

; Deoarece |vwx| < k, rezultă că vwx nu poate conține și a și c.

1) Dacă vwx nu conține a, atunci u trebuie să conțină toți simbolii a din șirul z (adică a^k). Pentru i = 0, rezultă uwy e L, adică: uwy = $a^kb^kc^k$ și deci |uwy| = 3*k. Dar, [uvwxy| = jz| = i 3*k și deci |v| = |x| = 0, ceea ce contrazice condiția vx *4 X.*

. 2) Dacă vwx nu conține c, atunci toți simbolii c trebuie conținuți în șirul y (adică c^k). Pentru ' i = 0, rezultă uwy e L, adică: uwy = $a^kb^kc^k$ și |uwy| = 3*k. Cum |uvwxy| = 3*k, rezultă că |v = |x| = 0 (contradicție). Prin urmare, L nu poate fi independent de context.

pi'ob'eiitu 1.1-27

L={a^i|i = 2^jJ>0}

Presupunem prin absurd că L este independent de context, și este aplicabilă lema de pompare. Fie k = n și fie șirul: z = *a^l".* într-adevăr, |z| > n, atunci există șirurile u, v, w, x, y . : astfel încât z = uvwxy cu vx *4 X* și |vwx| < n.

Considerăm z = a^paV$a^s$$a^{2"'p"q"r"s}$, p,q, r, s > 0. Rezultă u = a^p, v = a^q, w = a^r, x = a^s și y = ; $a^{p\ q"'\ "s}$ de unde rezultă |vwxj = q + r + s<nși |vx| = q + s > lceea ce înseamnă vx *4 h.*

i Dacă i = 2 atunci uvW'y e L, adică $a^{2"+q+s}$ ***e** L,* de unde rezultă 2^n + q + $_s$ = 2^{n+j} = *2"* +(2* ; : - 1)*2^n adică q + s = (2^j - 1)* 2" pentru j > 1=> q + s > 2" (deoarece 2^j-1>1)=i>q + s>n | \=> contradicție (q + r + s < n). Prin urmare, L nu poate fi independent de context. . I

1.1.3 Transformări asupra GIC

Gramaticile independende de context (GIC) sunt utilizate pentru analiza sintactică în cadrul,] compilatoarelor. Pentru că procesul de analiza sintactică se simplifică mult dacă gramaticile 1 utilizate satisfac anumite condiții, este. util să se facă transformări asupra acestui tip dej gramatici, transformări care să producă gramatici echivalente (care generează același limbaj).

1.1.3.1 Eliminare recursivitate stânga

Spunem că o gramatică este *recursivă stânga* dacă există un neterminal A astfel încât existai o derivare A =>+ Ax pentru A e N, x e (N UT)* . Dacă pentru un neterminal A există] producțiile

AB₂f... | AB_m|y,| ... | y_n

unde Yi nu începe cu A, 1 <i<n, se spune că avem *recursivitate stânga imediată* și] producțiile anterioare se pot înlocui cu:

A-»Y,A'|y₂A'|...|y_nA'
A'-»BiA'|B₂A'|...|B_mA'|X

Această construcție elimină recursivitatea stângă imediată. Dacă gramatica nu permite i derivări de tipul A =>+ A (nu are cicluri) și nu conține *X* producții poate să fie transformată în: vederea eliminării recursivității stânga utilizând următorul algoritm.

Intrare: o gramatică fără cicluri și *X* producții
Ieșire; o gramatică echivalentă fără recursivitate stânga.
Se alege o ordine a neterminalelor, fie ea: Ai, Aj, ..., A„
pentru i = **2 ..** n **execută**
  pentru j = 1 .. i-1 **execută**
    înlocuiește fiecare producție de forma Ai -^ r cu producțiile-
    Ai- ->-.Ui r | u, r I ... | u_k r unde A, -> ui I u₂ .. | u_k
    sunt toate producțiile pentru Aj.
  elimină recursivitatea stângă imediată între producțiile Ai.

1.1.3.2 Eliminare A producții

O gramatică nu are *X* producții dacă satisface una din-următoarele condiții:

- Nu există nici o producție care să aibă în partea dreaptă șirul vid sau
- Există o singură producție care are în partea dreaptă șirul vid și anume producția S -> *X.* Simbolul S nu apare în acest caz în partea dreaptă a nici unei producții.

Algoritmul de transformare este:


```
Intrare: o gramatică G = (N, T, P, S)
Teșire: o gramatică G' = (N', T, P', S') care satisface condițiile L(G) = L(G')
                          și G' nu are X producții
i = 0                                                         *
No = { A 1 A ^ X 6 P}
repetă

  N1 = { A I A ->> a e ?, a<EN,_*} \J Ni-i
până când Ni = Nj-i

dacă S e Nf
   N' = N U {S'}
   P' = {S' -> A, S' ->
altfel
  N' = N
  S' = S"
  P' = O
CI
pentru fiecare p e P execută
  dacă p este de forma
           ^->aoB1a1...Bkak,k>O,B,eNf,l<i<k,aje((N-Nf)uT)*,O<j<k
      P' = P u({A-»a0 X| <Xj •••Xk ak Xj 6{B;,Â}} - {A->Â})
  altfel
       P' = P' U P
  D
.
```

1.1.3.3 Eliminare simboli neutilizați

Un simbol neterminal este nefinalizat dacă din el nu se poate deriva un șir format numai din simboli terminali.

Algoritmul de transformare este:

Intrare: o gramatică G = (N, T, P, S)
Ieșire: o gramatică G' = (N', T, P', S) care satisface condițiile L(G) = L(G') și VA 6N, A =>+ w, w 6T*.
i = 0;
No = {A-> *a* , *aeT'}*
repetă
  i++

  Ni = {A | A ->'/îeP, /?e(Nwur)*}u N^
Până când Nt = Nj-i
N' =N.
P' c: P și este format numai producțiile din P care au
  în partea stângă simboli din N'

1.1.3.4 Eliminare simboli inaccesibili

Un simbol neterminal este inaccesibil dacă nu poate apare într-o formă propozițională. Algoritmul de transformare este:

```
Intrare : o gramatică G = (N, T, P, S)
Ieşire: o gramatică G' = (N', T, P', S) care satisface condiţiile L(G) = L(G')
          şi VA eN' există w e (N U T) *, s =>w şi A apare în w.
i = 0;
N_o = {S}
repetă

   N^*= { Ae N | A apare în partea dreaptă a producţiilor pentru un neterminal
          din Nj-j} UN,
până când Ni = N^
N' = Ni
P' c: P conţine numai ;producţiile corespunzătoare neterminalelor din N'
```

Probleme

problema 1.1-28

Să se elimine recursivitatea-stângă pentru următoarea gramatică: G = ({S, L},{ a,,,(,)}, P, S) unde P = {S -> (L) | a, L ->• L, S| a}

Soluție:

Se alege o ordine pentru neterminale, fie ea: S < L . Pentru producția S -> (L) | a nu se face nici o modificare. La fel pentru S -» a.

Pentru producția L —» L,S | a se elimină recursivitatea imediată și rezultă:

L -» aL'
L'-> ,SL' | *X*

In final după eliminarea recursivității stânga gramatica este:
G = ({ S, L, L'}, { a,,, G)}, P, S) unde: P = { S-> (L) | a, L,-> aL', L' -> ,SL' *X}*

pro bit mu 1.1-29

Să se elimine recursivitatea stângă pentru următoarea gramatică:
G = ({ S, A, B }, { a, b }, P, S) unde P = {S -> A | B, A -> Sa | Bb, B -> Sb | Aa}

Soluție:

Se alege o ordine pentru neterminale S < A < B
Pentru producția S —> A | B nu se face nici o modificare.
Pentru producția A -> Sa | Bb avem S < A deci se va folosi S -> A | B | *X* și rezultă

A -> Aa | Ba | a |Bb din care se elimină recursivitatea imediată și rezultă:
A -> BaA' | aA' | BbA', A' ->• aA' | *X*
Pentru producția B -» Sb | Aa se substituie cu S -> Sb | Aa | *X* și rezultă B H> Ab | Bb | b| Aa iar apoi se înlocuiește A -> BaA' | aA' | BbA' și rezultă B -» BaA'b | aA'b j BbA'b | Bb | b | B a A' a | a A' a | B b A' a. Din care se elimină recursivitatea imediată și rezultă:
; B _> a A' b B' | bB' | a A' a B', B' -> aA'b B' | b A' b B' | b B' | a A' a B' | b A'a B' | *X*
în final după eliminarea recursivității stânga gramatica este:
; G = ({ S, A, A', B, B' }, { a, b }, P, S) unde: P = { S-» A | B, A -> BaA' | aA' | BbA', A' -> aA' i *X*, B -> a A' b B' | bB' | a A' aB' ,B' -» aA'b B' | b A' b B' | b B' | a A' a B' | b A'a B'

probli-Pia 1.1-30

Să se elimine recursivitatea stângă pentru următoarea gramatică: G = ({ A, B, C }, { a, b, c }, P, A) unde P = {A -> BC | a, B -» Ca | Ab, C -> Ab | cC | b}

Soluție:

Se alege o ordine pentru neterminale A < B < C. Pentru producția A -» BC | a nu se face nici o modificare. Pentru producția B -> Ca] Ab deoarece A < B se va folosi A -» BC j a și rezultă B -» Ca | BCb | ab din care se elimină recursivitatea imediată și rezultă: B -> CaB' | jabB', B'->CbB'|L
• Pentru producția C -» AB j cC | b se substituie cu A -» BC | a și rezultă C -» BCb | aB | cC | b iar apoi se înlocuiește B -> CaB' | abB' și rezultă C -» CAB'CB | abB'Cb | aB | cC | b
: Din care se elimină recursivitatea imediată și se obține: C -> abB'CbC | aBC | cCC | bC, C -> AB'CBC [*X*. în final după eliminarea recursivității stânga gramatica este:
G = ({ A, B, B\ C, C }, { a, b, c }, P, A) unde: P = { A-» BC | a, B -> CaB' | abB', B" -+ CbB' | X, C -> abB'CbC | aBC j cCC | bC, C -» AB'CBC j *X}*

1.1-31

s i ->• elimine ' piiHluciiili.- Jiu y.nn iik\i li i,[s. \. li. 1 ',. / J. 1' ; I'. M unde P = {S -» ABC, A -> BB | *X*, B -> CC | a, C -> AA | b}

Soluție:

№ = {A}, Ni = {A, C}, N_2 = {A, B, C}, N_3 = {S, A, B, C} = N_f
: S e Nf deci se introduc producțiile S' -> S și S' -> *X*
\ Producția S -> ABC devine S '-> ABC | AB | AC | BC | A |B | C
, Producția A -» BB devine A -» BB | B
: Producția B -> CC \ a devine B -> CC | C | a
! Producția CC -> AA | b devine C -> AA | A | b

; După eliminarea *X* producțiilor gramatica este G = ({ S', S, A, B, C }, { a, b }, P, S') unde: P = { S' -> S , S' -> *X*, S ->' ABC | AB | AC | BC | A |B | C, A ->• BB 1 B, B ->€C | C | a, C -> ,AA|A|b}

probleniii 1.1-32

Să se elimine simbolii nefinalizați, iar apoi cei inaccesibili pentru gramatica G = ({ S, A, B }, {a, b }, P, S) unde P = {S-> A | a, A-> AB, B-> b}

Soluție:

Eliminare simboli nefinalizati: N_o = {S, B}, Ni = {S, B}, N_f=Ni.
Rezultă că A este simbol nefinalizat, se vor elimina producțiile corespunzătoare neterminalului A și gramatica va deveni: G = ({S, B}, {a, b }, P, S) unde P = {S -> a, B ->• b}.
Eliminare simboli inaccesibili :No = {S},Ni = {S},Nf=Ni
B este inaccesibil, d rămâne producția S —»a. După eliminarea simbolilor nefinalizati și inaccesibili G = ({S}, {a},P,S)undeP={S->a}

Coaientatiî:

Ordinea corectă de aplicare a celor doi algoritmi este eliminare simboli nefinalizați și apoi eliminare simboli inaccesibili.

problema 1.1-33

Să se elimine simbolii nefinalizați, iar apoi cei inaccesibili pentru gramatica G = ({S, A, B, C, D}, { a, b, d}, P, S) unde P={S-»bA|aB,A-»a|aC|aD|aS|bAA, B -> b | Cb | Db | bS | aBB, C -» Cb | Db, D -> Cb | Db, D -» Cd | dAC}

Soluție:

Eliminare simboli nefinalizati:
N_0={A,B},N, = {S,A,B},N_f=Ni
Rezultă că C, D £ Nf, adică sunt simboli nefinalizati, se vor elimina producțiile corespunzătoare lor și gramatica va deveni: G = ({S, A, B }, {a, b, d}, P, S) unde P = {S ->• bA | aB, A -* a | aS | bAA, B->b|bS|aBB}
Eliminare simboîi inaccesibili:
No={S},N,= {S,A,B},N_f=N,
Nu există simboli inaccesibili.
în final gramatica este: G = ({ S, A, B }, {a, b, d}, P, S) unde P = {S -> bA | aB, A -» a | aS | bAA, B-»b|bS|aBB}

problema 1.1-34

Să se elimine recursivitatea stângă, *X* producțiile, simbolii neutilizați și inaccesibili pentru gramatica G = ({S, B, C, D, E}, {a, b}, P, S) unde P = {S-> aBa, B -> Sb | bCC |DaB, C -> abb | DD, E -> aC, D ->• aDB}

Soluție:

Eliminare recursivitate stânga. Se alege ordinea neterminalelor S < B < C< E< D
Pornind de la B ->• Sb | bCC | DaB, S -»aBa. Rezultă B -> aBab | bCC | DaB care nu este recursivă stânga-
Eliminare Xproducții. Gramatica nu are *X* producții.
Eliminare simboli nefinalizați. N_o = {C}, Ni = {B, C, E}, N_2 = {S, B, C, E} = N_f
D <£ Nf, se elimină toate producțiile corespunzătoare neterminalului D.
Gramatica devine G = ({S, B, C, E}, {a, b}, P, S) unde P = {S H> aBa, B -> Sb | bCC, C -> abb, E

Eliminare simboli inaccesibili. N_o = {S}, Ni = {S, B}, N_2 = {S, B, C} = N £ E « N_f, rezultă E simbol inaccesibil.
în final rezultă: G = ({S, B, C}, {a, b}, P, S) undeP = {S -> aBa, B-> Sb [bCC, C -> abb}

problema 1.1-35

Să se elimine recursivitatea stânga, *X* producțiile, simbolii neutilizati și inaccesibili pentru gramatica G = ({S, T}, {a, b, c}, P, S) unde P = {S-> TbT, T -> TAT |ca}

Soluție:

Eliminare recursivitate stânga. Alegem ordinea neterminalelor S < T. Se consideră producția T -> TaT | se elimină recursivitatea stânga și rezultă T -» caT' și T' —> aTT' | *X*
RezultăG = ({S, T, T'}, {a, b, c}, P, S) undeP = {S -> TbT, T -> caT', T' -> aTT' | *X}*
Eliminare Xproducții. No = {T'}, Ni = {T'}, N_f=Ni, Rezultă G = ({S, T, T'}, {a, b, c}, P, S) unde P={S->TbT, T-»caT'|ca,T'->aTT'|aT}
Eliminare simboli nefinalizați. N_o = {T}, Ni = {S, T, T'}. Nu exisistă simboli nefinalizați.
Eliminare simboli inaccesibili. N_o = {S}, Ni = {S, T}, N_2 = {S, T, T'}, Nf = N_2. Nu există simboli inaccesibili.
în final se obține G = ({S, T, T'}, {a, b, c}, P, S) unde P = {S -» TbT, T -» caT' | ca, T' ->• aTT' | aT}.

1.1 Mulțimi regulate. Expresii regulate

Definiția 1.2.1. Fie T un alfabet finit. Se numește mulțime regulată asupra alfabetului T mulțimea definită recursiv astfel:
1- 0 este o mulțime regulată asupra alfabetului T.
2. Dacă a **e** T, atunci { a } este o mulțime regulată asupra alfabetului T.
3. Dacă P, Q sunt mulțimi regulate, atunci P u Q , PQ, P* sunt mulțimi regulate asupra alfabetului T.
4. O mulțime regulată asupra alfabetului T nu se poate obține decât aplicând regulile 1-3.
Am notat prin P u Q , PQ, respectiv P* reuniunea, concatenarea a două mulțimi, respectiv închiderea tranzitivă a unei mulțimi.
Definiția 1.2.2. O expresie regulată este definită recursiv în modul următor:
1. *X* este o expresie regulată care generează mulțimea 0.
2. Dacă a e T, atunci a este o expresie regulată care generează mulțimea { a }.
3. Dacă p, q sunt expresii regulate care generează mulțimile P, Q, atunci:
(p + q) sau (p | q) este o expresie regulată care generează mulțimea P u Q .

(pq) este o expresie regulată care generează mulţimea PQ.
(p)* este o expresie regulată care generează mulţimea P*.
4. O expresie regulată nu se poate obţine decât prin aplicarea regulilor 1-3.

Proprietăţi ale expresiilor regulate:

1. a | P = B|oc	(comutativitate reuniune)
2. a|(B|y) = (a|P)|y	(asociativitate reuniune)
3.ct(py) = (a p) y	(asociativitate concatenare)
4. a(P|y) = aP|ay	(distributivitate la stânga a concatenării faţă de reuniune)
5. (a|P)y=ay|py	(distributivitate la dreapta a concatenării faţă de reuniune)
6. a^. = A,a = a	(element neutru pentru concatenare)
7. a* = a I a*	
8. (a *)* = a*	
9. a | a = a	
10.(a*p*)* = (a + P)*	

Utilizând expresii regulate se pot construi ecuaţii regulate. Soluţia generală a ecuaţiei de forma: X = a X + b unde a, b, X sunt expresii regulate, este: X = a* (b + y) unde y este o expresie regulată oarecare. Soluţia minimală (punctul fix al ecuaţiei) este: X = a* b.

Propoziţie Fie G o gramatică regulată. L(G) este o mulţime regulată.

Definiţia 1.2.3. Două expresii regulate se numesc echivalente dacă descriu aceeaşi mulţime regulată.

Probleme

problema 1.12-1

Să se rezolve sistemul de ecuaţii:
X = a1X + a2Y + a3
Y = b1X + b2Y + b3

Soluţie:
Pentru prima ecuaţie, soluţia este X = a1* (a2 Y + a3), înlocuind în cea de-a doua ecuaţie, obţinem Y = bl al* (a2 Y + b3) + b2 Y + b3 sau echivalent, folosind proprietăţile expresiilor regulate: Y = bl al*a2 Y + bl al*b3 +b2 Y + b3 sauY = (bl al*a2+b2) Y + (bl al*b3 +b3).Deci,pentruYse obţine soluţia Y = (bl al* a2 + b2)* (bl al* b3 + b3). în mod corespunzător, înlocuind în prima ecuaţie, se obţine următoarea soluţie pentru X: X = al* a2 Y + al* a3 sau X = al* a2 (bl al* a2 + b2)*(blal*b3+b3) + al*a3.

problema 1.12-2

Să se rezolve sistemul de ecuaţii:
X1=OX2+1X1+Â,
X2 = OX3 + 1X2
X3 = OX1 + 1X3

Soluţie:
Din ultima ecuaţie obţinem X3 = 1* 0 XI. înlocuind în a doua ecuaţie obţinem X2 = 1 X2+0 1* 0 xl, de unde rezultă X2 = 1 * 0 1 * 0 XI. Dacă se înlocuieşte în prima ecuaţie rezultă XI = 0 1 * 0 1 * 0 XI + 1 XI + X. Deci XI = (01*01*0 + 1)*. Dacă se înlocuieşte rezultatul obţinut pentru XI în formulele corespunzătoare lui X2 şi X3 obţinem X2 = 1* 0 1* 0 (0 1* 0 1* 0 + 1)* şi X3 = 1* 0 (0 1*01*0+1)*.

Comentarii:
Se poate demonstra că în expresia regulată XI numărul de simboli 0 este divizibil cu 3. Prin urmare, XI poate fi scris sub formă echivalentă: XI =(1*0 1*0 1*0)* 1*.

problema 1.12-3

Să se construiască expresia regulată care generează mulţimea regulată egală cu limbajul generat de gramatica regulată cu producţiile: P-> 0 Q11 P, Q-» 0 R11 P, R-» 0 R11 R10

Soluţie:
Asociem fiecărui netcrminal o expresie regulată şi fiecărei producţii de forma A-> a | p, o ecuaţie de forma A = a + p unde a şi p sunt tot expresii regulate. Corespunzător setului de producţii de mai sus, se obţine următorul sistem de ecuaţii:

```
P=0Q+1P
Q=0R+1P
R=0R+1R+0
```

Din ultima ecuaţie, se obţine folosind proprietatea de distributivitate a concatenării faţă de reuniune R = (0 + 1) R + 0 care are soluţia R = (0 + 1)* 0. înlocuind în a doua ecuaţie, obţinem: Q = 0 (0 + 1)* 0 + 1 P. înlocuind Q în prima ecuaţie: P = 0 0 (0 + 1) * 0 + 0 1 P + 1 P adică P = (01 + 1) P + 00 (0 + 1)* 0. Această ecuaţie are soluţia P = (0 1 + 1)* 0 0 (0 + 1) * 0.

Să se construiască expresia regulată care generează mulţimea regulată egală cu limbajul regulat generat de gramatica regulată:
S-» 0 A [1 S | X
A-» 0 B|1 A
B->0S|1B

Soluţie:

Corespunzător setului de producţii de mai sus, se obţine următorul sistem de ecuaţii:
S = 0 A + 1 S + X
A = 0 B + 1 A
B = 0 S + 1 B

! Din ultima ecuație, obținem B = 1 * 0 S. înlocuind în a doua ecuație, aceasta devine: A = 0 1 * : 0 S + I A care are soluția A = 1 * 0 1 * 0 S . Prima ecuație devine în urma înlocuirii expresiei : regulate A, S = O 1 * O 1 * O S + 1 *S* + *X* sau, folosind distributivitatea concatenării față de reuniune. S = (0 1* 0 1* 0 + 1) S + *X* care are soluția S = (0 1* 0 1* 0 + 1)* *X* = (0 1* 0 1* 0 + !)*·'·

problema 1.2-5

Să se construiască gramatica regulată care generează limbajul generat de expresia regulată: (a I b)* a* b+ a*

Soluție:

Notăm S = (a | b)* a* b+ a* și A = a* b+ a*. Corespunzător, se poate scrie ecuația ce are soluția S: S = (a + b) S + A. De asemenea, dacă notăm B' = b+ a* și B = b* a*, putem scrie următoarea relație B' = b B (deoarece b+ = b b*).

Pe de altă parte, A = a* B' și deci se poate scrie relația A = aA + B'sauA = aA + bB unde B = b* a*. Notând C = a*, B devine B = b* C care este soluția ecuației B = b B + C. De asemenea, C = a* este soluția ecuației C = a C + *X*. Prin urmare, corespunzător expresiei regulate de mai sus se obține următorul sistem de ecuații:

S = a S + b S + A
A = a A + b B
B = b B + C
C = a C + A,

și respectiv următorul set de producții:

S - > a S b S j A
A - > a A | b B
B - > b B

Gramatica G care generează limbajul descris de expresia de mai sus, este G = ({ S, A, B, C }, { a, b }, P, S) cu P mulțimea de producții de mai sus.

problema 1.2-6

Să se construiască gramatica regulată care generează limbajul descris de expresia regulată: : (a | b) * a (a | b)

Soluție:

Notăm S = (a | b)* a (a | b) și A' = a (a | b). Deci, S este soluția ecuației S = (a + b) S + A' iar A' poate fi scris sub forma A' = a A unde A = a + b. Deci, corespunzător expresiei regulate S, poate să fie scris următorul sistem de ecuații: S = a S + b S + A', A' = a A, A = a + b sau S = aS + bS + aA, A = a + b. Gramatica ce generează limbajul generat de expresia S este G = ({ S, A }, { a, b }, P, S) unde P conține producțiile: { S -» a S | b S | a A, A-> a | b }.

problema 1.2-7

§ă se construiască gramatica care generează limbajul descris de expresia regulată: (a | b)* a (a | b) (a 1 b)

Soluție: .

Notăm S = (a | b)* **a** (a | b) (a | b) și respectiv A' = a (a | b) (a | b). S este soluția ecuației: S = (a + b) S + A' care poate să fie scrisă echivalent S = a S + b S + A' (1). Dar A' = a A unde A = (a | b) (a S b). Prin urmare, relația (1) devine S = aS + bS + aA (2). Dacă notăm B = (a | b), atunci A devine - A = a B + b B (3) și B = a + b (4) Corespunzător relațiilor (2) - (4), setul de producții al gramaticii G care generează limbajul descris de expresia regulată S este :

S - » a S | b S | a A
A-> a B | b B
B-> a | b

Gramatica ce generează limbajul descris de expresia regulată dată este G = ({S, A, B}, {a, b}, P, S) unde P conține producțiileJS-» a S | b S [a A, A-> a B | b B, B-> a | b}

1.3 Acceptoare

Spre deosebire de gramatici și expresii regulate care generează limbaje formale acceptparele sunt dispozitive care sunt în strare să „recunoască" dacă un șir de simboli face parte dintr-un limbaj pentru care mecanismul este acceptor.

1.3.1 Automate finite

Definiția 1.3.1. Se numește automat finit un obiect matematic AF = (Q, T, m, sO, F) unde:

- Q reprezintă mulțimea finită a stărilor
- T este o mulțime finită de elemente numită alfabet de intrare
- m este funcția parțială a stării următoare m: Q x (T U { *X* }) -» P(Q) unde prin P(Q) s-a notat mulțimea părților lui Q
- sO e Q este o stare numită starea de start
- F c Q este o mulțime de stări numită mulțimea stărilor finale sau de acceptare

Definiția 1.3.2. Se numește graf de tranziție pentru automatul finit AF = (Q, T, m, sO, F) un graf orientat cu arce etichetate G = (N, A) în care nodurile (mulțimea N) reprezintă stările automatului finit, iar arcele (mulțimea Ac N x N) sunt definite astfel: (s_i; Sj) **e** A dacă există a e T astfel încât Sj e m(s_i; a). în acest caz, arcul (SJ, sj) este etichetat cu simbolul a.

Definiția 1.3.3. Se spune că un șir w G T* este acceptat de automatul finit AF dacă există o cale în graful de tranziție între starea de start și o stare finală, astfel încât șirul simbolilor care etichetează arcele este șirul w. Mulțimea șirurilor acceptate de un automat finit formează limbajul acceptat de automatul finit respectiv.

Definiția 1.3.4. Se numește automat finit determinist un automat finit AF = (Q, T, m, sO, F) pentru care funcția de tranziție m: Q x T-> Q. Se observă că în acest caz:

nu există X-tranziții

pentru orice s e Q și orice a e T există o unică stare s' e S astfel încât m(s, a)=s'.

Propoziție. Pentru orice automat finit nedeterminist (AFN) există un automat finit determinist (AFD) care acceptă același limbaj.

Propoziție Limbajele acceptate de automate finite sunt limbaje regulate (generate de gramatici regulate).

Având în vedere că limbajele regulate sunt generate și de expresii regulate, există o echivalență ca putere între gramatici regulate, expresii regulate și automate finite. Automatele; finite deterministe sunt utilizate pentru implementarea analizoarelor lexicale. Expresiile regulate' sunt utilizate pentru specificare atomilor lexicali recunoscuți de un analizor lexical în mod; corespunzător au fost elaborați algoritmi pentru construirea de automate finite deterministe direct din expresii regulate.

Probleme

problema 1.3-1

i Să se construiască automatul finit care accepta limbajul generat de gramatica:

P->0 Q|1 P

Q-> 0 R| 1 P

R-> 0 R 11 R| 1

Să se reprezinte graful de tranziție corespunzător.

Soluție;

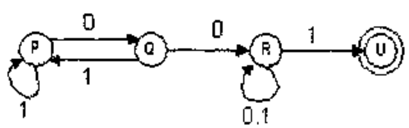
Gramatica este în mod evident regulată, prin urmare există un automat finit care acceptă limbajul generat de aceasta. Vom construi acest automat asociind fiecărui neterminal o stare și fiecărei producții o tranziție. Rezultă că putem construi următoarea funcție de tranziție:

m(P,0) = {Q},m(P, 1) = {P}

m(Q,0) = {R},m(Q,1)={P}

m(R,0)={R},m(R, 1) = { R, U }

unde U este o stare nou introdusă, în care se va produce acceptarea (deci va fi singura stare finală). Prin urmare, automatul finit care acceptă limbajul generat de gramatica de mai sus este: AF = ({ P, Q, R, U }, { 0, 1 }, m, P, { U }). Corespunzător, graful de tranziție asociat este:



Comentarii:

Se observă că în acest caz, automatul finit obținut este nedeterminist (pentru starea R și simbolul 1 există două stări succesoare posibile: R și U).

problema 1.3-2

ga se construiască automatul finit care acceptă limbajul generat de gramatica G:

S-> 0 A | 1 S *X,* A-» 0 B | 1 A, B-> 0 S | 1 B

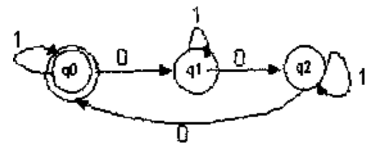
Să se reprezinte graful de tranziție asociat.

Soluție:

Fie următoarea mulțime de stări Q = { qO, ql, q2 } astfel încât asociem fiecărui neterminal o stare din Q: lui S i se asociază qO, lui A i se asociază ql, lui B i se asociază q2. Corespunzător construim funcția de tranziție în modul următor:

m(qO, 0) = ql, m(qO, 1) = qO, m(ql, 0) = q2, m(ql, 1) = ql, m(q2, 0) = qO, m(q2, 1) = q2

Automatul finit este AF = ({ qO, ql, q2 }, { 0, 1 }, m, qO, { qO }). Se observă că AF este determinist. Graful de tranziție corespunzător este:



Comentam;

Șirurile acceptate conțin un număr de simboli 0 divizibil Cu 3. Automatul obținut este determinist.

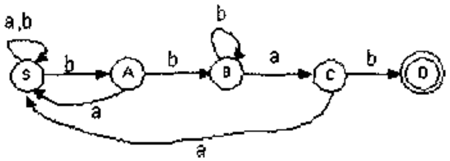
proMcma 1,3-1

\ Să se construiască gramatica regulată pentru limbajul L = { w e {a,b}* [w = ubbab, u E {a, j b}* } construind mai întâi graful de tranziție asociat.

Soluție:

Fie S starea inițială a automatului. Trebuie să accepte șirurile care se termină cu bbab. Din i starea S trecem în starea A în momentul în care apare un b (am recunoscut primul b din subșirul bbab). Din starea A se trece în starea B dacă s-a întâlnit un b (s-a recunoscut șirul bb) și se trece în starea S dacă s-a primit un a (în continuare se încearcă să se identifice i subșirul bbab).

Din starea B se trece în starea C dacă s-a întâlnit un a (s-a recunoscut bba) și se rămâne în B ! pentru b. Din starea C se trece în starea D dacă s-a întâlnit un a, adică în starea D arii găsit \ bbab, deci D este stare finală. Din starea C se revine în S dacă s-a întâlnit un a, se reîncearcă i să se identifice subșirul bbab. în starea S se poate rămâne dacă s-a întâlnit a sau b (pentru a ; se putea accepta șirul u).



Pentru a construi gramatica fiecărei stări i se asociază un neterminal. Rezultă gramatica G = ' ({ S, A, B, C}, { a, b }, P, S) unde: P = { S-» a S | b S | b A, A -> b B | a S, B -> b B | a C, C '

! .3-4

Să se construiască gramatica regulată care generează limbajul L = { aia_2 ... a_n | n > 3, a, e {x, y}, a_n.2 = y } construind mai întâi graful de tranziție asociat.

Soluție:

Fie S starea inițială a automatului. în starea S se acceptă x și y pentru a se genera orice prefix. De asemenea din S se trece în starea A dacă s-a întâlnit y (s-a recunoscut $a_{n\cdot 2}$). Din starea A se trece în starea B dacă s-a întâlnit x sau y. Din starea B se trece în starea C dacă s-a întâlnit x sau y (C este și stare finală).

; Pentru a construi gramatica, fiecărei stări i se asociază un neterminal. Rezultă gramatica G = i ({ S, A, B}, { x, y }, P, S)unde: P = { S-> x S | y S | y A, A -»x B | y B, B -» x j y).

problema !.3-5

i Să se construiască automatul finit care acceptă limbajul descris de expresia regulată:
(a | b)* a* b+ a*
Să se reprezinte graful de tranziție corespunzător.

Soluție:

i Expresia regulată (a | b)* a* b+ a* generează limbajul generat și de gramatica G determinată i- în Problema 1.2.5, adică: S-» a S | b S | A, AH> a A | b B, B-> b B | C, C-> a C | *X*
' Construim automatul finit care acceptă limbajul descris de expresia regulată de mai sus.
: Funcția de tranziție parțială pentru automatul finit care acceptă limbajul descris de gramatica i G este: m(S, a) = { S }, m(S, b) = { S }, m(S, *X)* = { A }, m(A, a) = { A }, m(A, b) = { B }, | m(B, b) = { B }, m(B, *X)* = { C }, m(C, a) = { C }
! Deoarece există producția C -» *X,* starea asociată neterminalului C este de acceptare (sau finală). Același lucru s-ar fi întâmplat pentru orice producție de forma A —> a, cu a e T. Automatul finit care acceptă limbajul descris de expresia (a | b)* a* b+ a* este AF = ({ S, A, B, C }, { a, b, c }, m, S, { C })...

Comentarii:

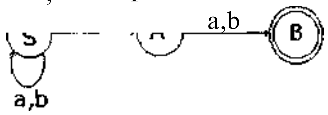
Automatul AF astfel construit este nedeterminist (are X-tranziții).

prolrfcnta 1,3-6

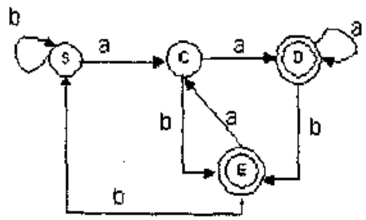
Să se construiască automatul finit care acceptă limbajul descris de expresia regulată:
(a i b)* a (a I b)

Soluție:

Gramatica G care generează limbajul descris de expresia regulată (a | b)* a (a | b) este (vezi Problema 1.2.6.): S - » a S | b S | a A , A-» a | b. Asociem fiecărui neterminal o stare și fiecărei producții o tranziție. Corespunzător, obținem următoarea funcție de tranziție parțială: m(S, a) = { S, Â }, m(S, b) = { S }, m(A, a) = { B }, m(A, b) = { B } unde B este o stare de acceptare (finala) nou introdusă. Se observă ca automatul finit este nedeterminist. Graful de tranziție corespunzător este:



Dacă dorim să construim automatul finit determinist care acceptă limbajul descris de (a | b)* a (a | b), putem să efectuăm transformări asupra gramaticii G astfel încât din mulțimea producțiilor să rezulte o funcție de tranziție de tipul: m: Q' x *T*-> Q'. Pentru aceasta, transformăm gramatică G prin *factorizare stânga* (terminalul a începe și în producția S—•. a S și producția S-» a A). Obținem: S-» a C | b S, A-» a | b, C-> S | A.
Prin *substituție de începuturi* (" corner substitution "), obținem: S-> a C | b S, A-> a | b, C-> a C b S a | b, unde observăm că neterminalul A este *neutilizat* (deci îl putem elimina). Aplicând din nou factorizarea stânga urmată de substituție de începuturi, obținem: SH> a C | b S, C-> a D | b E, *D-> C | X,* E-> S | *X,* respectiv, S-> a C | b S, C-> a D | b E, D-> a D | b E | *X,* E-> a C | b S | *X.* Se observă că în acest caz se poate construi o funcție de tranziție totală de forma: m(S, a) = C, m(S, b) = S, m(C, a) = D, m(C, b) = E, m(D, a) = ; D, m(D,'b) = E , m(E, a) = C, m(E, b) = S
; Graful de tranziție pentru acest automat finit determinist este:



Stările asociate neterminalelor D, respectiv E sunt stări de acceptare datorită existenței producțiilor: DH> *X* și E -* *X.*

Comentarii:

Se poate construi automatul finit determinist care acceptă limbajul de mai sus pornind direct de la.expresiaregulată.

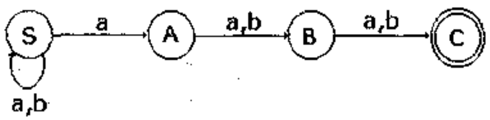
proMwii i **I** '-"

Să se construiască automatul finit care acceptă limbajul generat de expresia regulată
(a* I *b) * a (a \ b) (a | b)

Soluție.

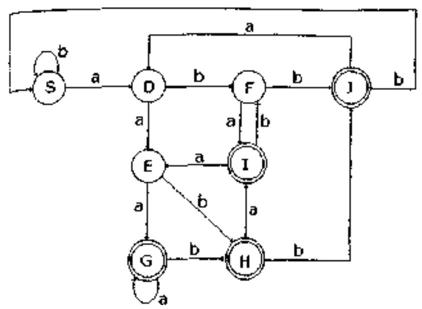
Fie gramatica G care generează limbajul descris de expresia regulata (a | b)* a (a | b) (a | b) (vezi Problema 1.2.7.): S-> a S | b S | a A, A-» a B | b B, B-> a | b. Construim pe baza mulțimii de producții de mai sus următoarea funcție de tranziție: m(S, a) = { S, A }, m(S, b) = { S }, m(A, a) = { B }, m(A, b) = { B }, m(B, a) = { C }, m(B, b) = { C } unde C este o stare de acceptare nou introdusă.

Se observă că automatul finit obținut: AF = ({ S, A, B, C }, { a, b }, m, S, { C }) este nedeterminist. Graful, corespunzător este:



Pentru a obține automatul finit determinist care acceptă limbajul generat de gramatica G de mai sus, se pot face transformări asupra gramaticii. Prin *factorizare stânga* (terminalul a începe și producția S-» a S și producția S-> a A) se obține: S-»aD|bS, A -» a B | b B, B -> a | b, D -> S | A. Prin *substituție de începuturi* (" corner substitution ") , obținem: S -> a D |bS, B -> a [b, D->aD[bS|aB|bB, unde observăm că neterminalul A este *neutilizat* (deci îl putem elimina). Aplicând din nou factorizarea stânga urmată de substituție de începuturi, obținem: S -> a D | b S, B ^ a | b, D -» a E | b F, E -> D | B, F -> S | B, respectiv, S -» a D | b S, D -» a E | b F, E->aE|bF|a|b, F->aD|bSja|b. Aplicând din nou factorizarea stânga urmată de substituție de începuturi, obținem: S -» a D | b S, D -> aE|bF, E -> a G | b H, F -> a 11 b J, G -» E | A, H -» F | A, I -» D | A, J ->S | A, respectiv, S -»aD|bS, D->aE|bF, E-»aG|bH, F->aI|bJ, G->aG|bH|X, H->aI|bJ|A., I ~>aE|bF|A,, J->aD|bS|A..

Se observă că în acest caz se poate construi o funcție de tranziție totală de forma:
m(S, a) = D, m(S, b) = S, m(D, a) = E, m(D, b) = F, m(E, a) = G, m(E, b) = H , m(F, a) = I, m(F, b) = J, m(G, a) = G, m(G, b) = H, m(H, a) = I, m(H, b) = J, m(I, a) = E, m(I, b) = F, m(J, a) = D, m(J, b) = S. Graful de tranziție al automatului determinist este:

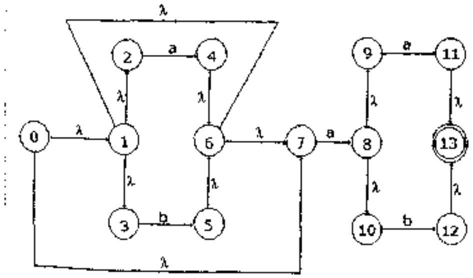


problema î;3># ,

Să se construiască AFN (automatul finit nedeterminist) care generează același limbaj ca și ; următoarea expresie regulată și apoi să se construiască AFD-ul (automatul finit determinist) corespunzător. Să se facă și construcția directă: expresie regulată-AFD
(a|b)*a(a|b)

Soluție:

AFN corespunzător expresiei regulate (a | b)* a (a | b) este:



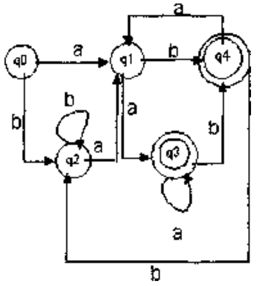
Construcția AFD se face pe baza algoritmului de construire a unui AFD echivalent unui AFN [da]t (vezi curs pentru notații și algoritm). Starea inițială a AFD va fi: qO = *X* inchidere({ 0 }) = **i 0» 1,2,3,7}.**
n continuare, celelalte stări precum și tranzițiile corespunzătoare se vor determina astfel:
(x) = qj unde qj = Ă_inchidere(move(qi, x)), obținem:

q! = A._inchidere(move(qO, a)) = A,_inchidere({ 4, 8 }) = { 1,2, 3, 4, 6, 7, 8, *9,* 10 }
q2 = _inchidere(move(qO, b)) = Ăjnchidere({ 5 }) = { 1, 2, 3, 5, 6, 7 }
q[3] = Mnchidere(move(ql, a)) = A._inchidere({4, 8,11}) = {1,2, 3, 4, 6, 7, 8, 9, 10, 11,13 }
'3[4=] Unchidere(move(ql, b)) = ljnchidere({ 5, 12 }) = { 1, 2, 3, 5, 6, 7,12, 13 }

Se obţine următoarea tabelă de tranziţii:

stare	stare următoare	
	a	b
qO	q i	q2
q l	q3	q4
q2	q l	q2
q3	q3	q4
q4	q l	q2

şi corespunzător, graful de tranziţie:



Dacă dorim acum să construim direct automatul finit determinist care acceptă expresia de mai sus, trebuie să utilizăm algoritmul de construire a AFD direct din expresia regulată (vezi curs). Considerăm arborele corespunzător expresiei regulate, terminate cu terminatorul #:

```
(a | b)* a (a | b) #
 1  2    3  4   5  6
```

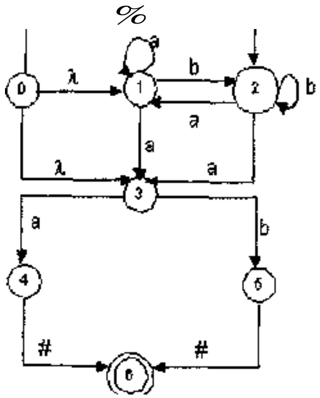
Arborele este:

```
                  {1,2,3}  .  {6}
                          / \
                         /   \
              {1,2,3}  .  {4,5} \
                       / \ {6} # {6}
                      /   \     6
                     /     \
                    /       \
       {1,2,3} . {3}    {4,5}[ {4,5}
              / \           / \
             /   \         /   \
            / {3} a {3}   /     \
           *       3     /       \
     {1,2}  |  {1,2}    /         \
           / \    {4} a {4}  {5} b {5}
          /   \        4          5
         /     \
        /       \
 {1} a {1}  {2} b {2}
     1          2
```


Am notat în stânga fiecărui nod firstpos(nod), iar în dreapta lastpos(nod). Calculând pentru fiecare nod frunză followpos(i) (i codul unui nod frunză), se obţine:

nod	followpos
a 1	{ 1, 2, 3 }
b 2	{ 1, 2, 3 }
a 3	{4 , 5}
a 4	{ 6 }
b 5	{ 6 }
# 6	-

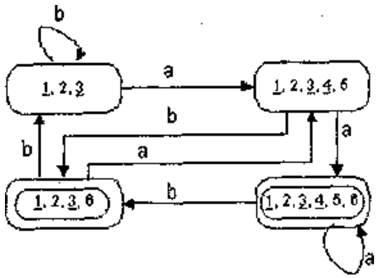
Corespunzător tabelei de mai sus, se obţine automatul finit nedeterminist reprezentat prin următorul graf de tranziţie (vom asocia fiecărui nod frunză o stare în graful de tranziţie), arcele sunt stabilite după următoarea regulă: există un arc între nodul i şi nodul j dacă j 6 followpos(i). Arcul se etichetează cu simbolul corespunzător codului j. De asemenea, se introduce o stare iniţială 0 din care exista Â.-tranziţii în stările din firstpos(rad) (rad este rădăcina arborelui corespunzător expresiei regulate). Vom obţine următorul graf de tranziţie:



Se observă că în stările 4, respectiv 5 este acceptat sufixul aa, respectiv ab. Automatul finit determinist corespunzător expresiei regulate se obţine considerând că stare iniţială firstpos(rad) = { 1, 2, 3}. Tranziţiile se determină astfel:

$$m(qi, x) = \bigcup_{i = cod(x)} \{ followpos(i)\} = qj$$

Se obţine următorul graf de tranziţie pentru AFD:



Am subliniat în fiecare mulțime nodurile i pentru care este îndeplinită condiția i = cod (a).

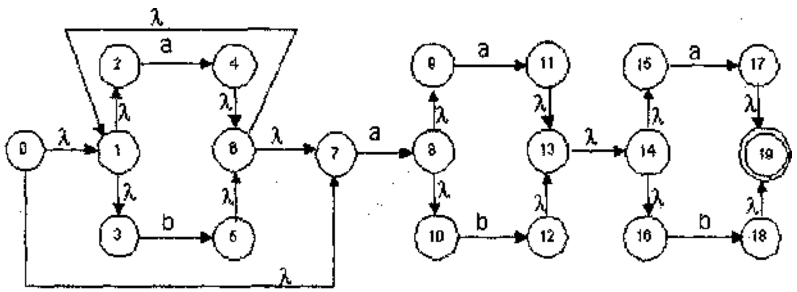
Comentarii:

Se observă că în starea { 1,2,3,4,5,6 } se acceptă sufixul aa, iar în starea {1,2,3,6} se acceptă! sufixul ab. Ambele sunt stări terminale deoarece conțin codul pentru # (terminatorul expresiei] regulate).

Să se construiască AFN pentru următoarea expresie regulată și apoi să se construiască AFD-ul corespunzător. Să se facă și construcția directă: expresie regulată-AFD:
(a|b)*a(a|b)(a|b)

Soluție:

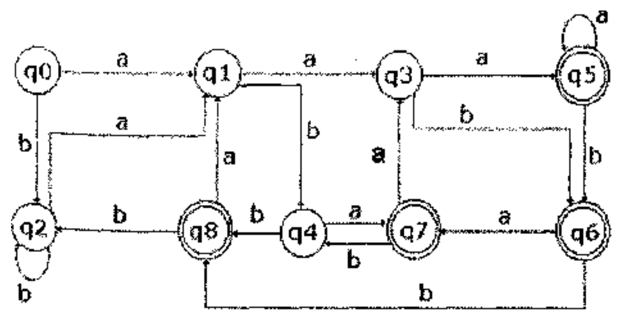
AFN corespunzător expresiei regulate de mai sus este:



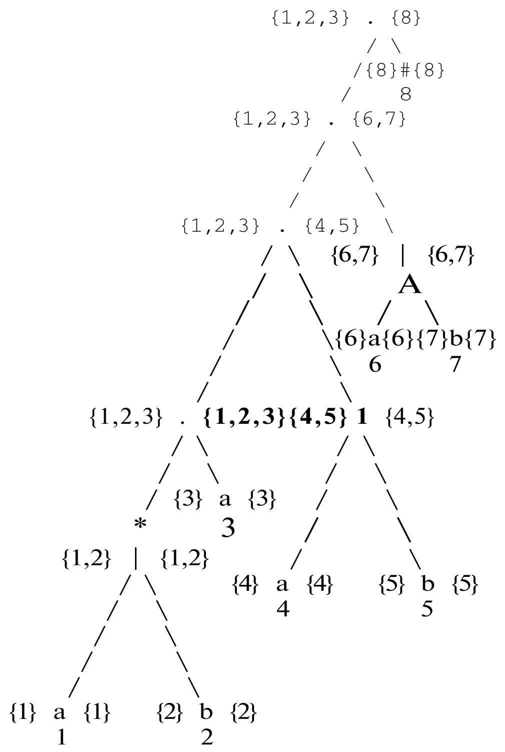
Pentru a obține AFD din AFN-ul de mai sus, starea inițială a noului automat determinist va fi: qO = A._inchidere({ 0 }) = { 0, 1, 2, 3, 7 } deoarece starea inițială a AFN este starea 0. în continuare, vom determina stările următoare si tranzițiile utilizând:
m(qi, x) = ^_inchidere(move(qi, x))

Vom obține: m(qO, a) = A_inchidere(move(qO, a)) = X_inchidere({ 4, 8 }) = { 1, 2, 3, 4, 6, 7, 8, 9, 10 } = ql, m(qO, b) = A,_inchidere(move(qO, b)) = A,_inchidere({ 5 }) = { 1, 2, 3, 5, 6, 7 } = q2, m(ql, a) = X_inchidere(move(ql, a)) = X_jnchidere({ 4, 8, 11 }) = { 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16 } = q3, m(ql, b) = _inchidere (move(ql, b)) = ÂJnchidere ({ 5^ : 12 }) = { 1, 2, 3, 5, 6, 7, 12, 13, 14, 15, 16 } = q4, m(q2, a) = A._inchidere(move(q2, a)) = : X_inchidere({ 4, 8 }) = ql, m(q2, b) = A,_inchidere(move(q2, b)) = A._inchidere({ 1, 2, 3, 5, ; 6, 7 }) = q2, m(q3, a) = X_inchidere(move(q3, a)) = X_inchidere({ 4, 8, 11, 17 } = { 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 19 } = q5, m(q3, b) = Î\._inchidere(move(q3, b)) = X_mchidere({ 5, 12, 18 }) = { 1, 2, 3, 5, 6, 7, 12, 13, 14, 15, 16, 18, 19 } = q6, m(q4, a) = X_inchidere(move(q4,a)) = X_inchidere({ 4, 8, 17 })={ 1, 2, 3, 4, 6, 7, 8, 9, 10, 17, 19 } = : q7, m(q4, b) = A._mchidere(move(q4, b)) = A,_inchidere({ 5, 18 }) = { 1, 2, 3, 5, 6, 7, 18, 19 } = q8, m(q5, a) = A,_inchidere(move(q5, a)) = A,_inchidere({ 4, 8, 11, 17 } = { 1, 2, 3, 4, 6, 7, : 8, 9, 10, 11, 13, 14, 15, 16, 17, 19 } = q5, m(q5, b) = X_inchidere(move(q5, b))' = ; X_inchidere({ 5, 12, 18 }) = { 1, 2, 3, 5, 6, 7, 12, 13, 14, 15, 16, 18, 19 } = q6, m(q6, a) = X_inchidere(move(q6, a)) = \jnchidere({ 4, 8, 17 }) = { 1, 2, 3, 4, 6, 7, 8, 9, 10, 17, 19 } = \ q7, m(q6, b) = ^_inchidere(move(q6, b)) = A,_inchidere({ 5, 18 }) = { 1, 2, 3, 5, 6, 7, 18, 19 } \ = q8, m(q7, a) = X_inchidere(move(q7, a)) = ^_inchidere({ 4, 8, 11 }) = { 1, 2, 3, 4, 6, 7, 8, i 9, 10, 11,13, 14, 15, 16 } = q3, m(q7, b) = ;Wnchidere(move(q7, b)) = X_inchidere({ 5, 12 j }) = { 1, 2, 3, 5, 6, 7, 12, 13, 14, 15, 16 } = q4, m(q8, a) = X_inchidere(move(q8, a)) = i X,_inchidere({ 4, 8 }) = { 1, 2, 3, 4, 6, 7, 8, 9, 10 } = ql, m(q8, b) = A._inchidere(move(q8, b)) i ; = A,_inchidere({ 5 }) = { 1, 2, 3, 5, 6, 7 } = q2. j

' O altă soluție se obține pornind direct de la expresia regulată construind AFN, respectiv AFD ! corespunzătoare. în mod corespunzător funcției de tranziție de mai sus, obținem următorul i graf de tranziție asociat AFD. Stările finale vor fi toate stările care conțin starea finală 19 din i i AFN inițial: i



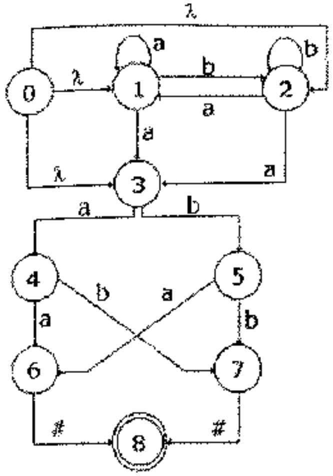
Arborele corespunzător expresiei regulate este:



în stânga fiecărui nod se află firstpos(nod), iar în dreapta lastpos(nod). Calculând pentru fiecare frunză followpos(i) (i este codul frunzei), obținem:

nod	followpos
a 1	{ 1, 2, 3 }
b 2	{ 1, 2, 3 }
a 3	{4,5}
a 4	{6,7}
b 5	{6,7}
a 6	{ 8 }
b 7	{ 8 }
# 8	-

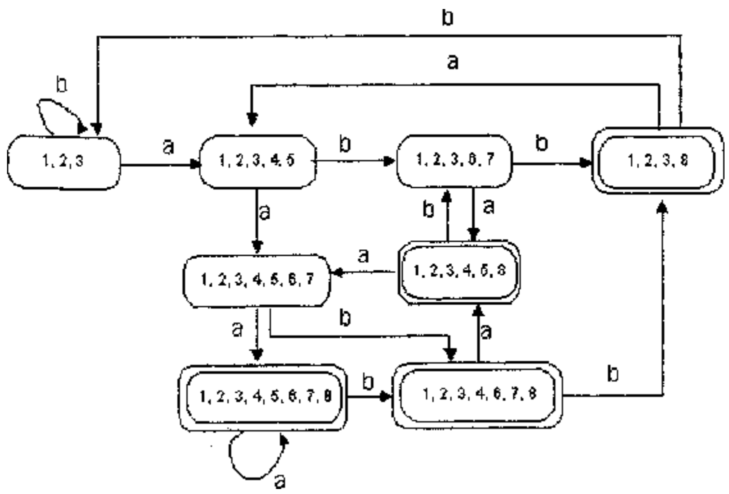
Procedând în mod analog problemei precedente obținem următorul AFN:



Dacă dorim să obținem AFD corespunzător automatului de mai sus, considerăm ca stare inițială firstpos(rad) = { 1, 2, 3 }. Tranzițiile se obțin astfel:

m(qi, x) = v followpos(j) = qk
j = cod(x)

Obținem următorul automat finit determinist (în care sunt marcate ca fmale stările ce conțin codul 8 corespunzător simbolului terminator #):



problem;! 1.3-10

Să se construiască automatul finit determinist pentru expresia regulată: (a|b)*.

Soluție;

Considerăm expresia regulată terminată cu simbolul # și codificăm în mod corespunzător simbolii:

(a | b)* #
 1 2 3

Pentru expresia regulată de mai sus, arborele corespunzător este:

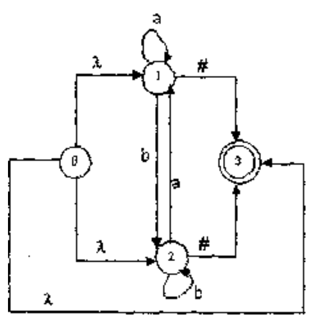
```
                {1,2,3}  .  {3}
                          / \
                         /   \


           {1,2}*{1,2}{3}  #  {3}
           {1,2}|{1,2}     3
                  / \
                 /   \
                /     \
        {1} a {1}  {2} b {2}
            1          2
```

Determinând pentru fiecare cod i followpos(i) pe baza mulțimilor firstpos(nod) (specificată în stânga nodurilor), respectiv lastpos(nod) (specificată în dreapta nodurilor), se obține următorul tabel:

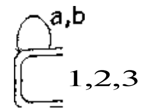
nod	followpos
a 1	{ 1, 2, 3 }
b 2	{ 1, 2, 3 }
# 3	-

Graful de tranziție corespunzător automatului finit se determină asociind fiecărui cod o stare, iar arcele se determină astfel: există un arc între starea i și starea j dacă j e followpos(i), iar eticheta arcului este simbolul cu codul j. Obținem:



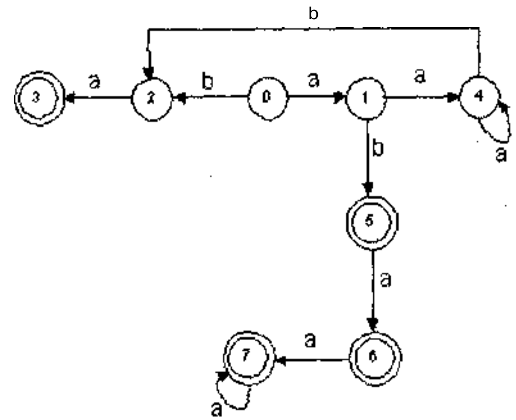
40

Am introdus o stare de start (0) din care există ^-tranziții în stările corespunzătoare • elementelor din mulțimea firstpos(rad) = { 1, 2, 3 }. De asemenea, starea 3 corespunzătoare simbolului terminal # este stare finală. Automatul fmit obținut astfel este nedeterminist.
Dacă dorim să obținem automatul finit determinist, vom considera starea de start firstpos(rad) = { 1, 2, 3 }, iar tranzițiile se determină ca în problema anterioară. Obținem următorul automat finit determinist:



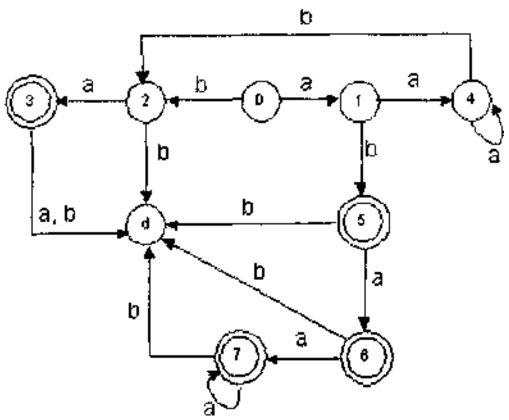
Să se minimizeze automatele specificate prin graf urile de tranziție:

prolik'ma 1.3-11

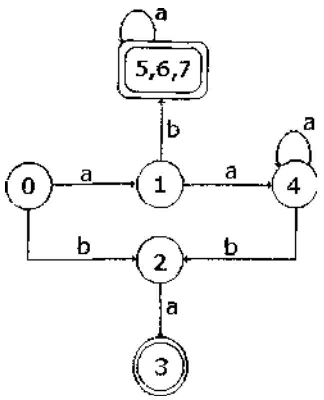


Soluție;

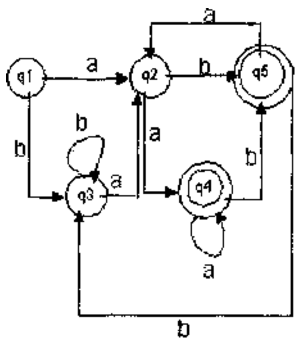
Introducem o nouă stare d astfel încât funcția de tranziție să fie totală, adică pentru orice stare q pentru care m(q, x) nu este definită (x e{ a, b }), adăugăm m(q, x) = d. Noul graf de tranziție devine:



Fie partiția disjunctă pe mulțimea stărilor: P = { { 0, 1, *2,* 4 }, { 3, 5, 6, 7 }, { d } }unde prima mulțime este mulțimea stărilor nefmale, iar a doua este mulțimea stărilor finale. în continuare, pentru fiecare mulțime Q e P si toate stările q e Q pentru care: m(q, x) g Q' astfel încât există q' cu m(q', y) e Q' (1). Se construiește o nouă partiție: P = (P \ { Q }) u { Q \ new, new } unde new este mulțimea stărilor care îndeplinesc condiția (1). într-adevăr, în cazul nostru obținem. Pentru Q = { 0, 1, 2,4 }, tranzițiile stărilor componente sunt:
m(0, a) = 1 e Q m(0, b) = 2 e Q, m(1, a) = 4 e Q m(1, b) = 5 g Q deci 1 e new
m(2, a) = 3 £ Q deci 2 e new, m(4, a) = 4 e Q m(4, b) = 2 e Q
Prin urmare, noua partiție pe mulțimea stărilor este: P = {{0,4}, {1,2}, {3,5,6,7}, {d}}.
Pentru Q = { 0, 4 }, tranzițiile stărilor sunt: m(0, a) = 1 e { 1, 2 } m(0, b) = 2 e { 1, 2 }, m(4, a) = 4 g { 1, 2 } deci 4 e new. Partiția obținută pe mulțimea stărilor devine: P = {{0 }, { 4 }, { 1, 2 }, { 3, 5, 6, 7 }, { d } }. Pentru Q = { 1, 2 } (este prima mulțime din partiție care conține mai mult de un element) rezultă: m(1, a) = 4 e {4 } m(1, b) = 5 e { 3, 5, 6, 7 }, m(2, a) = 3 <£ { 4 } deci 3 e new.
Partiția astfel obținută este: P = { { 0 }, { 1 }, { 2 }, { 4 }, { 3, 5, 6, 7 }, { d } }
Pentru Q={3,5,6, 7} tranzițiile stărilor componente sunt: m(3, a) = d e { d } m(3, b) = d e { d }, m(5, a) = 6 *i* { d } deci 5 e new, m(6, a) = 7 g { d } deci 6 e new, m(7, a) = 7« { d } deci 7 e new.
Partiția obținută în acest caz este: P = { { 0 }, { 1 }, { 2 }, { 3 }, {4}, { 5, 6, 7 }, { d } }. Fie acum Q = { 5, 6, 7 }. Tranzițiile stărilor componente sunt: m(5, a) = 6 e { 5, 6, 7 } m(5, b) = d e { d }, m(6, a) = 7 e { 5, 6, 7 } m(6, b) = d e { d }
m(7, a) = 7 € *{5,6,1}* m(7, b) = d e { d }, adică nu mai apare o nouă partiție pe mulțimea stărilor. Prin urmare, automatul finit minimizat este:

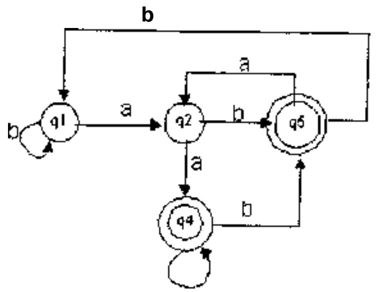


|.i.ihluii.il.*-12



Soluție;

Funcția de tranziție este totală, deci putem considera următoarea partiție pe mulțimea stărilor: ; P={jql,q2,q3},{q4,q5}}
Aplicând același algoritm de la punctul a), considerăm mulțimea: Q = { ql, q2, q3 } pentru care tranzițiile stărilor componente sunt: m(ql, a) = q2 e { q2 }, m(ql, b) = q3 e { ql, q3 }, m(q2, a) = q4 £ { ql, q2, q3 } deci q2 e new, m(q3, a) = q2 6 { q2 }, m(q3, b) = q3 e {ql,q3 }. Noua partiție pe mulțimea stărilor devine: P= {{ql, q3}, q2}, {q4, q5} }. Fie Q = { ql, q3 }. Tranzițiile pentru această mulțime de stări: m(ql, a) = q2 € { q2 }, m(ql, b) = q3 *e {* ql, q3 }, m(q3, a) = q2 e { q2 }, m(q3, b) = q3 e { ql, q3 }
Fie acum Q = {q4, q5}. Tranzițiile pentru stările componente sunt: m(q4, a) = q4 € {q4, q5 } m(q4, b) = q5 € { q4, q5 }, m(q5, a) = q2 £ { q4, q5 } deci q5 e new. Noua partiție pe mulțimea stărilor este: P = { { ql, q3 }, { q2 }, { q4 }, { q5 } }. Automatul finit minimizat, Pentru care starea corespunzătoare mulțimii { ql, q3 } este notată cu ql, este:



1.3.2 Automate cu stivă (push-down)

Definiția 1.3.5. Se numește automat cu stivă un obiect matematic definit în modul următor: P = (Q, T, Z, m, qO, zO, F) unde:

- Q - este o mulțime finită de simboli ce reprezintă stările posibile pentru unitatea de control a automatului;
- T - este mulțimea finită a simbolilor de intrare;
- Z - este mulțimea finită a simbolilor utilizați pentru stivă;
- m - este o funcție, m : Q x (T u *{X})* x Z -> P(S x Z*) (prin P(Q) s-a notat mulțimea părților lui Q) este funcția care descrie modul în care se obține starea următoare și j informația care se introduce în stivă pentru o combinație (stare, intrare, conținut stivă) | dată;
- qO s Q este starea inițială a unității de control;
- zO e Z este simbolul aflat în vârful stivei în starea inițială;
- F c Q reprezintă mulțimea finită a stărilor finale.

Definiția 1.3.6. O configurație de stare a automatului este un triplet (q, w, a) e Q x T* x Z* unde:

- q - reprezintă starea curentă a unității de control;
- w - reprezintă partea din banda de intrare care nu a fost încă citită. Dacă w = *X* înseamnă că s-a ajuns la sfârșitul benzii de intrare;
- a - reprezintă conținutul stivei.

automatului este relația |- definită asupra mulțimii [următor: (q, aw, za) |- (q^F, w, Pa) unde (q^l, P) **e** m(q, a, z), q_e Q, a e T u {1}, w e T*, z e Z, a e Z*, *fie Z**.

Dacă a * *X* înseamnă că, dacă unitatea de control este în starea q, capul de citire este pe simbolul a iar simbolul din vârful stivei este z atunci automatul poate să își schimbe configurația în modul următor: starea unității de control devine q', capul de citire se deplasează cu o poziție la dreapta iar simbolul din vârful stivei se înlocuiește cu p.

Dacă a = *X* înseamnă că avem o A.-tranziție pentru care simbolul aflat în dreptul capului de citire pe banda de intrare nu contează (capul de citire nu se va deplasa), însă starea unității de control și conținutul memoriei se pot modifica. O astfel de tranziție poate să aibă loc și după ce s-a parcurs întreaga banda de intrare.

Dacă se ajunge într-o configurație pentru care stiva este goală nu se mai pot executa tranziții.

Relația j- se poate generaliza la |-\ |-*, |-*, într-o manieră similară relației de derivare pentru forme prepoziționale.

Definiția 1.3.8. O configurație inițială pentru un automat cu stivă este o configurație de forma (qO, w, zO) unde w e T*. O configurație finală este o configurație de forma (q, *X,* a) cu q $_e$ F, a **e** Z*.

Definiția 1.3.9. Spunem că un șir w este acceptat de un automat cu stivă prin stări finale dacă (qO, w, zO) |-* (q, *X, a)* pentru q e F ș i a e Z * . Mulțimea șirurilor acceptate de un automat cu stivă se notează cu L(P).

Definiția 1.3.10. Spunem că un șir w este acceptat de un automat cu stivă prin stivă goală dacă (qO, w, zO) |-* (q, *X, X)* pentru q **e** Q. Limbajul acceptat de un automat pushdown P în acest mod se notează cu Le(P).

Propoziție. Dacă L(P) este limbajul acceptat de automatul P prin stări finale atunci se poate construi un automat pushdown P' astfel încât L(P) = Le(P').

Propoziție. Dacă Le(P) este limbajul acceptat de automatul P prin stivă goală atunci se poate construi un automat pushdown P' astfel încât **Le(P)** = L(P').

Propoziție. Mulțimea limbajelor acceptate de automate cu stivă este mulțimea limbajelor independente de context.

Probleme

Să se determine limbajul acceptat de următorul automat cu stivă:

PD = ({qO, ql, q2, q3}, {a, b, c}, {a, b, c, z}, m, qO, z, {q3 })
m(qO, a, z) = { (qO, az) }
m(qO, a, a) = {(qO, aa)}
m(qO,c,a)={(ql,a)}
m(ql,b,a)={(ql,A.)}
m(ql,b,z)={(ql,bz)}
m(ql,b,b)={(ql,bb)}
m(ql,c,b)={(q2,b)}
m(q2,b,b)={(q2,X)}
m(q2, *X,* z) = { (q3, *X) }*

Să;

Soluție:

Evoluția automatului PD pentru un anumit șir de intrare poate fi caracterizată în modul următor: din starea inițială qO se poate face o tranziție dacă șirul începe fie cu simbolul a fie cu simbolul c. Se observă că simbolii a sunt introduși în stivă. Fie n numărul de simboli a din prefixul șirului de intrare, deci șirul de intrare poate fi de forma: w = a" ev unde n > 0, iar v este un sufix al șirului de intrare. în continuare, se trece în starea ql în care poate urma doar simbolul b în șirul de intrare și pentru fiecare simbol b de pe banda de intrare, se descarcă un simbol a din vârful stivei.

Deci, numărul de simboli b care poate urma după c este tot n, iar w este de forma; w = a"cb^n u, u este un sufix al șirului de intrare. Simbolul care poate urma pe banda de intrare în momentul în care stiva este vidă (s-au descărcat toți simbolii a din stivă) este tot b.
în continuare, pot apare doar simbolii b care se introduc în stivă. Fie m numărul acestora. Configurația în acest moment a prefixului analizat din șirul de intrare este: w = a" c b^n b^m x, unde x este restul din șir rămas neanalizat. Se observă că din starea q1, dacă pe banda de intrare nu urmează un b, poate urma doar un c. Se trece în starea q2 în care sunt acceptați doar simboli b pe banda de intrare. Ori de câte ori apare un b pe banda de intrare, se descarcă un b din vârful stivei. în momentul în care stiva s-a golit, se trece în starea finala q3. Deci șirurile acceptate de automat sunt de forma: w = a^{n}cb^{n}b^{m}cb^{m}cum,n>0.
Pentru șirul de intrare aacbbbcb, mișcările efectuate de automat sunt:
(qO, aacbbbcb, z) |- (qO, acbbbcb, az) |- (qO, cbbbcb, aaz) |- (ql, bbbcb, aaz)
|- (ql, bbcb, az) |- (ql, bcb, z) |- (ql, cb, bz) |- (q2, b, bz)

Hq2,A,z)|-(q3,Aa)

deci șirul este acceptat.

prohlcm» 1.3-14

Să se construiască automatul cu stivă care acceptă următorul limbaj:
L={wcw^R|we {a,b}*}

Soluție:
Vom folosi în construcția automatului PD care acceptă limbajul L faptul că șirul de intrare este simetric și are număr impar de simboli, simbolul din centru fiind c. Prin urmare, pentru a putea stabili dacă șirul de intrare poate fi acceptat sau nu, trebuie memorată prima jumătate a șirului (până se întâlnește simbolul c) în stivă și apoi comparată cu a doua jumătate a șirului (vom compara simbolul analizat în mod curent de pe banda de intrare cu simbolul din vârful stivei; dacă sunt egali, se descarcă stiva). Prin urmare, automatul cu stivă care acceptă limbajul L, este: PD = ({qO, ql, q2 }, {a,b,c}, {a,b,c,z},m,qO,z, {q2})
m(qO,d,e)={(qO,de)} Vde {a,b},Vee {a,b,z}
m(qO,c,d)={(ql,d)}Vde{a,b,z}
m(ql,d,d)={(ql,A)}Vde{a,b}
m(qU,z)={(q2,A.)}

Comentarii:
Se observă ca automatul este determinist și acceptarea este prin stări finale.

Să se construiască automatul cu stivă care acceptă următorul limbaj:
L={O^{i}l^j|O<j i}

Soluție:
]Numărul de simboli 0 este mai mare sau egal decât numărul de simboli 1. Vom proceda în mod analog exemplului precedent introducând inițial în stivă toți simbolii 0 din prima parte a șirului și ulterior descărcând stiva pentru fiecare simbol 1 întâlnit pe banda de intrare. Condiția ca un șir să fie acceptat este ca stiva să nu se golească înainte de a ajunge la sfârșitul șirului. Rezultă deci automatul:

= ({q0,ql,q2}, { 0,1 }, { 0,z },m, qO,z, { q2

m(q0,0,a)={(q0,0a)}Vae {O,z}
m(q0,1,0)={(ql,X)}
m(ql,1,O)={(ql,X)}
m(ql,^a)={(q2,X)}Vae{0,z}

Verificare:
Pentru șirul de intrare 00011, evoluția automatului este:

(qO, 0001 1,z)i-(q0,0011, Oz)
|-(q0,011,00z)
j-(qO, 11,000z)
|-(ql, 1,00z)
|-(ql,A,0z)
|-(q2,A,z)

prin urmare șirul este acceptat. De asemenea, pentru șirul de intrare 011, evoluția automatului este:

funcția de tranziție nu este definită în acest caz, prin urmare șirul nu este acceptat (starea ql nu este de acceptare).

Comentarii:
Automatul PD este nedeterminist deoarece există două tranziții din starea ql nedistinctibile:

m(qU,0)={(q2,A.)}

problema 1.3-16

Să se construiască automatul cu stivă pentru limbajul generat de următoarea gramatică:
S -» a S b | a A b
A->c

Soluție:

Putem construi un automat cu stivă care acceptă limbajul generat de gramatică prin stivă goală: PD' = ({ q0 }, { a, b, c }, { a, b, c, S, A }, m, q0, S, { }), m(q0, X, S) = { (q0, a S b), (q0, a A b) }, m(qO, Â, A) = {(qO,c)}, m(qO,x,x) = {(q(U)}Vxe{a,b,c}

O altă soluție se poate obține considerând un automat cu stivă extins (în acest caz considerăm că stiva crește spre dreapta): PD" = ({qO, ql}, {a, b, c}, {a, b, c, z, S, A}, m, qO, z, {ql }), m(qO, x, *X)* = {(qO, x)} V x e {a, b, c}, m(q0, X, aSb) = {(qO, S)}

m(q0, X, aAb) = {(qO, S)}, m(q0, X, c) = {(qO, A)}, m(q0, X, zS) = {(ql, *X)}*

Verificare;

Pentru șirul de intrare aacbb, se obțin evoluțiile celor două automate:

(qO, aacbb, S) |- (qO, aacbb, aSb) |- (qO, acbb, St) |- (qO, acbb, aAbb) |- (qO, cbb, Abb)
|- (q0, cbb, cbb) |- (q0, bb, bb) |- (q0, b, b) |- (q0, X, X)

iar pentru a doua variantă:

(qO, aacbb, z) |- (qO, acbb, za) |- (qO, cbb, zaa) |-(qO, bb, zaac) |- (qO, bb, zaaA)
|-(qO, b, zaaAb) |- (qO, b, zaS) |-(qO, X, zaSb) |- (qO, X, zS) |-(ql, X, z)

problema 1.3-17

Să se construiască automatul cu stivă care acceptă următorul limbaj:
L = {w | w conține un număr egal de a și b}

Soluție:

Vom utiliza aceeași tehnică ca și în problemele precedente și anume: deoarece numărul de simboli a este egal cu numărul de simboli b din șirul de intrare, putem introduce în stivă toți simbolii a, dacă șirul începe cu a, respectiv toți simbolii b, dacă șirul începe cu b. în continuare, de fiecare dată când pe banda de intrare se află un simbol diferit de cel din vârful stivei se descarcă stiva, altfel se introduce în stivă. Șirul este acceptat dacă în final stiva este vidă. Rezultă: PD = ({qO},{a,b},{a, b,z},m,qO,z, {qO}), m(qO, x, z) = {(qO, xz)} V x e {a,b}, m(qO,x,x) = {(qO,xx)} Vx e {a,b},m(qO,x,y)={(q(U)}Vx,ye {a,b}x #y,m(qO,*X,z)*= {(qO,z)}

O altă soluție se poate construi observând că limbajul L este generat de gramatica: S-> a S b S | b S a S | X(vezi, de exemplu, Problema 1.1.6 din paragraful 1.1.1.). Prin urmare putem construi un automat cu stivă care acceptă limbajul L prin sivă goală: PD' = ({qO}, {a, b}, {a, b, S}, m, qO, S, {}), m(qO, X, S) = {(qO, a S b S), (qO, b S a S),(qO, *X)*}, m(qO, c, c) = {(qO, *X)*} V c e {a, b}

în fine, o a treia soluție se poate obtine considerând un automat cu stivă extins (în acest caz considerăm stiva crește spre dreapta): PD" = ({ qO, ql }, { a, b }, { a, b, z, S }, m, qO, z, { ql }), m(qO, x, *X)* = {(qO, x)} V x 6 { a, b }, n # , *X, X)* = {(qO, S)}, m(qO, X, aSbS) = {(qO, S)}, m(qO, X, bSaS) = {(qO, S)}, m(qO, X, zS) = {(ql, *X)*}

limbaje formale si translatoare Seminar LFA

Verificare:

Pentru șirul de intrare abbbaa, se obțin evoluțiile celor trei automate:

(qO, abbbaa, z) |- (ql, bbbaa, az) |- (ql, bbaa, z) |- (ql, baa, bz) |- (ql, aa, bbz)
|

sau:

(qO, abbbaa, S) |- (qO, abbbaa, aSbS) |- (qO, bbbaa, SbS) |- (qO, bbbaa, bS)
|- (qO, bbaa, S) |- (qO, bbaa, bSaS) |- (qO, baa, SaS) |- (qO, baa, bSaSaS)
|- (qO, aa, SaSaS) |- (qO, aa, aSaS) |- (qO, a, SaS) |- (qO, a,aS) |- (qO, X, S)

iar pentru ultima variantă:

(q0, abbbaa, z) |- (q0, bbbaa, za) |- (q0, bbbaa, zaS) |- (q0, bbaa, zaSb)
|- (q0, baa, zaSbb) |-(q0, aa, zaSbbb) |- (q0, aa, zaSbbbS)
|-(q0, a, zaSbbbSa) |- (q0, a, zaSbbbSaS) |-(q0, a, zaSbbS)
|-(q0,A,,zaSbbSa) |- (q0, X, zaSbbSaS) |- (q0, X, zaSbS)
|-(q(U,zS)|-(ql,A,z)

Comentarii:

Considerând noțiunea de derivare putem să introducem două noi notații. *Derivarea stânga* este o derivare în care se înlocuiește întotdeauna cel mai din stânga neterminal în timp ce într-o *derivare dreapta* se înlocuiește întotdeauna cel mai din dreapta neterminal. Ultimele două variante corespund derivărilor stânga, respectiv dreapta,^le șirului de intrare.

S => aSbS => abS => abbSaS => abbbSaSaS => abbbaSaS => abbbaaS =» abbbaa

Respectiv:

S => aSbS => aSbbSaȘ => aSbbSa => aSbbbSaSa => aSbbbSaa=> aSbbbaa => abbbaa

1.3-18

Să se construiască automatul cu stivă care acceptă următorul limbaj:
L={a"b"'|n 1,m 1,n msaum<2*n}

Soluție:

Deoarece între numărul de simboli a (n), respectiv numărul de simboli b (m) există fie relația n < m (cazul 1), fie relația m < 2 * n (cazul 2), vom trata cele două cazuri în același mod: se depun în stivă toți simbolii a de pe banda de intrare prin simpla copiere în stivă (cazul 1), deci vom avea în stivă cei n simboli a sau prin duplicare (cazul 2), caz în care vor exista în stiva 2 * n simboli a. în continuare, atât în cazul 1 cât și în cazul 2 se descarcă stiva pentru fiecare simbol b de pe banda de intrare. în cazul 1 se trece în starea de acceptare dacă nu s-a ajuns la sfârșitul șirului înainte de a se goli stiva, iar în cazul 2 daca la terminarea șirului, stiva este nevidă. Prin urmare, un automat PD care acceptă iimbajulLeste:

PD = ({ $_{q0}$, ql, $_{q2}$, q3, q4, q5, q6 }, {a,b}, {a, z},m, qO, z, {q3, q6})

m(q0, a, z) = {(ql, az), (q4, aaz)}

(tratare cazul 1:)
m(ql, a, a) = { (ql, aa) }, m(ql, b, a) = { (q2, A)}, m(q2, b, a) = { (q2, A) *}*
m(q2, *X,* z) = { (q3, z)} cazul n = m, m(q2, b, z) = { (q3, z) } cazul n < m
m(q3, b, z) = { (q3, z) }
(tratare cazul 2:)
m(q4, a, a) = {(q4, aaa) }, m(q4, b, a) = { (q5, *X)},* m(q5, b, a) = {(q5, A) }
m(q5, *X,* a) = { (q6, *X)}*

Verificare:

Pentru șirul de intrare aabbb se obține următoarea evoluție a automatului cu stivă:
(qO, aabbb, z) |- (ql, abbb, az) |- (ql, bbb, aaz) |- (q2, bb, az) |- (q2, b, z) |- (q3, *X,* z)
sau (qO, aabbb, z) |- (q4, abbb, aaz) |- (q4, bbb, aaaaz) j- (q5, bb, aaaz) |- (q5, b, aaz)
|- (q5, *X,* az) |- (q6, *X,* z)

Deci șirul aabbb este acceptat de automat. Pentru șirul de intrare aabb, evoluția automatului PD este: (qO, aabb, z) |- (ql, abb, az) |- (ql, bb, aaz) |- (q2, b, az) |- (q2, *X,* z) |- (q3, *X,* z)
sau
(qO, aabb, z) |- (q4, abb, aaz) |- (q4, bb, aaaaz) |- (q5, b, aaaz) |- (q5, *X,* aaz) |- (q6, *X,* az)

Deci șirul aabb este acceptat de automat conform ambelor condiții. Pentru șirul aabbbb evoluția automatului este: (qO, aabbbb, z)J- (ql, abbbb, az) |- (ql, bbbb, aaz) |- (q2, bbb, az) |- (q2,bb,z)|-(q3,b,z)|-(q3,A,z)

Deci șirul este acceptat de automat (conform primei condiții). De asemenea, pentru șirul de intrare aab, evoluția automatului este: (qO, aab, z) |- (ql, atf, az) |- (ql, b, aaz) |- (q2, A, az) și în acest caz nu mai este aplicabilă nici o tranziție sau: (qO, aab, z) |- (q4, ab, aaz) |- (q4, b, aaaaz) j- (q5, *X, aaaz)* |- (q6, *X, aaz).* Adică șirul este acceptat de automat (conform condiției a doua).

problema 1.3-19

Să se construiască automatul cu stivă care acceptă următorul limbaj:
L = { an bm an | n, m >= 1 } u { an bm cm | n, m >= 1 }

Soluție:

Deoarece limbajul specificat de L este de fapt reuniunea a doua limbaje, va trebui ca în specificarea automatului PD să realizăm niște mecanisme prin care să fie acceptate șiruri din ambele limbaje. Astfel, șirurile nu pot începe decât cu simbolul a. Toți simbolii a din prefixul șirului se introduc în stivă. De asemenea, toți simbolii b care urmează sunt introduși în stivă. In continuare: Dacă următorul simbol de pe banda de intrare (după ce s-au epuizat toți simbolii b) este a, atunci se încearcă acceptarea unui șir de forma: $a^n b^m a^n$, caz în care se descarcă toți simbolii b din stivă și se verifică dacă numărul de simboli a din stivă este egal cu numărul de simboli a rămași pe banda de intrare.

Dacă următorul simbol de pe banda de intrare este c, se încearcă acceptarea unui șir de forma: $a^n b^m c^m$, caz în care se verifică dacă numărul de simboli c din sufixul șirului este egal cu numărul de simboli b din stivă. în final, se descarcă din stivă toți simbolii a.

Rezultă automatul:
PD = ({ qO, ql, q2, q3, q4, q5, q6}, {a, b, c}, {a, b, z}, m, qO, z, { q5 })
(încarcăsimbolii a și b în stivă:)
m(qO, a, z) = {(ql, az)}, m(ql, a, a) = {(ql, aa)}, m(ql, b, a) = {(ql, ba)}
m(ql,b,b) = {(ql,bb)}
(varianta 1:)
rn(ql, a> b) = { (i^, A,)} am consumat un simbol a de la intrare
m(q2, *X,* b) = {(q2, *X)}* descarcă simbolii b din stivă
m(q2, a, a) = {(q3, *X)}*
m(q3, A, ,a) = {(q4, A-)} scot un a din stivă ce corespunde simbolului a consumat în starea ql
m(q4!a,a)={(q4,A)}
m(q4, *X,* z) = {(q5, z)} stare finală
(varianta 2:)
m(ql, c, b) = {(q6, A) }, m(q6, c, b) = { (q6, A) }, m(q6, *X,* z) = { (q5, z) }
(descarcă simbolii a din stivă:)
m(q4, A, a) = {(q4, A.)}, m(q4, A, z) = {(q5, z)} (stare finală)

problem:! 1.3-20

Să se construiască automatul cu stivă care acceptă limbajul expresiilor aritmetice cu paranteze

Soluție:

Vom considera că există două stări ql, respectiv q2 pentru care sunt valabile următoarele semnificații: ql este starea de start. Atât timp cât pe banda de intrare simbolul analizat este "(", se copiază în stivă. Dacă simbolul de pe banda de intrare este a, se trece în starea q2.
în starea q2, dacă simbolul de pe banda de intrare este operator (+, *), se trece în starea.ql.
Dacă simbolul este ")", se descarcă din stivă un simbol "(" •
Rezultă automatul PD = ({ ql, q2, q3 }, { a, +, *, (,)}, { z, (}, m, ql, z, { q3 }),
m(ql, (, x) = { (ql, (x) } x e { (, z }, m(ql, a, x) = { (q2, x) } x e { (, z }
m(q2,),()={ (q2, A) }, m(q2, y, x) = { (ql, x)} y e {+,*}, m(q2, *X,* z) = { (q3, *X)* }

Verificări::

Pentru șirul de intrare a+a*(a+(a)*a), evoluția automatului este:

(ql, a+a*(a+(a)*a), z) |- (q2, +a*(a+(a)*a), z) |- (ql, a*(a+(a)*a), z)
|- (q2, *(a+(a)*a), z) |- (ql, (a+(a)*a), z) |- (ql, a+(a)*a),(z)
|- (q2, +(a)*a),(z) |- (ql, (a)*a), (z) |- (ql, a)*a), ((z) j-(q2,)*a), ((z)
|- (q2, *a), (z) |- (ql, a), (z) |-(q2,), (z) |-(q2, A, z) |-(q3, A, *X)*

Comentarii:

Șirurile sunt acceptate prin stivă goală.

1.3.3 Mașina Turing

Definiția 1.3.11. Se numește mașină Turing obiectul matematic: MT = (Q, T, m, q_0) unde

- Q este o mulțime finită a stărilor mașinii Turing, hg Q;
- T este alfabetul finit de intrare care conține simbolul # dar nu conține simbolii L și R;
- m este funcția de tranziție m : Q x T -> (Q u {h}) x (T u {L, R}).
- qo e Q este starea inițială a mașinii Turing.

Dacă q e Q, a e T și m(q, a) = (p, b) înseamnă că fiind în starea q, având simbolul a sub capul de citire mașina trece în starea p. Dacă b e T atunci simbolul a va fi înlocuit cu b, altfel capul se va deplasa cu o poziție la stânga sau la dreapta în funcție de valoarea simbolului b 6 {L, R}. Se observă că mașina Turing a fost definită ca un acceptor determinist.

Definiția 1.3.12. O configurație pentru o mașină Turing este (q, ***a***, a, /?) notată și (q, aay?)undeq e (Qujh}), *ae* T*, a e T, *J3e* (T* (T - {#}) u *{X})* în care

- q este starea curentă
- *a* este șirul aflat pe banda de intrare la stânga capului de citire / scriere
- a este simbolul aflat sub capul de citire / scriere
- /? este șirul aflat la dreapta capului de citire / scriere.

Definiția 1.3.13. Relația de tranziție |- se definește asupra configurațiilor în modul următor. Fie (ql, wl, al, ui) și (q2, w2, a2, u2) două configurații. Atunci:

(ql,wl,al,ul) |- ($q2_s$w2,a2,u2) dacă și numai dacă există b e T u{L, R} astfel încât m(ql, al) = (q2, b) și este îndeplinită una din următoarele condiții:

```
1. b e T, wl = w2, ui = u2, a2 = b;

2. b = L, wl = w2a2
   dacă al * # sau ui * X
      u2 = al ui
   altfel u2 = X

3. b = R, w2 = wlal
   daca ui = X
      u2 = X
      a2 = #
   altfel ui = a2 u2
```

Un *calcul efectuat de o mașina Turing* este o secvență de configurații cO, ci, ..., cn astfel încât n > 0 și cO |- ci |-... |- cn. Se spune despre calcul că are loc în n pași.

O mașină Turing poate să fie utilizată și ca *acceptor de limbaj.* Și anume să considerăm un limbaj L definit asupra alfabetului T care nu conține simbolii #, D și N. Fie dL : T* -* {D, N} o funcție definită în modul următor - pentru VweT*

Ddacăw GL

dL(w) =

\

N dacă w 0 L

Se spune ca limbajul *L este decidabil în sens Turing* (Turing decidable) dacă și numai dacă funcția dL este calculabilă Turing. Dacă dL este calculată de o mașina Turing MT spunem ca *MT decide L.*

Noțiunea de acceptare este mai largă decât noțiunea de decidabilitate în legătura cu limbajele. Și anume dacă L este un limbaj asupra alfabetului T, spunem ca limbajul L este *Turing acceptabil* dacă există o mașina Turing care se oprește dacă și numai dacă w e L.

Propoziție. Limbajele Turing decidabile sunt și Turing acceptabile. Reciproca nu este însă adevărată.

Definiția 1.3.14. O schemă de mașină Turing este tripletul *F = (M, *r*, MO) unde

- M este o mulțime finită de mașini Turing care au toate un alfabet comun T și mulțimi de stări distincte;
- MO e M este mașina inițială;
- T] este o funcție parțială, *r* : M x T-> M.

O schemă de mașină Turing reprezintă o mașina Turing T compusă din mașinile care formează mulțimea M. Funcționarea acesteia începe cu funcționarea mașinii MO. Dacă MO se oprește atunci ¥ poate continua eventual funcționarea conform altei mașini din M. Dacă MO se oprește cu capul de citire / scriere pe caracterul a atunci există următoarele situații:

- t|(M0, a) este nedefinită, în acest caz ¥ se opreşte;
- -n(M0, a) = M', în acest caz funcționarea continuă cu starea inițială a mașinii M'.

4> = (M, r|, MO) o schemă de mașină Turing reprezintă mașina MT = (Q, T, m, s) unde

- Q = QOu...uQku{qO,..., qk}
- s = sO
- m este definită în modul următor:
  a. dacă q e Qj, 0 < i < k, a e T și nij(q,a) = (p, b), p * h atunci m(q, a) = îîij(q, a) = (p, b);
  b. dacă q e Q_b 0 < i < k, a e T și mi(q, a) = (h, b) atunci m(q, a) = (qi, b);
  c. dacă r|(Mi, a) (0 < i < k, a e T) este nedefinită atunci m(qi, a) = (h,a);
  d. dacă r)(Mi, a) = Mj (0 < i < k, a e T) și mj(sj, a) = (p, b) atunci

(p,b)

m(qi, a) =

\

(qj,b)p = h

Propoziție. Adăugarea unor facilități la definiția Mașinii Turing ca de exemplu :

- banda de intrare infinită la ambele capete;
- mai multe capete de citire / scriere;
- mai multe benzi de intrare;
- bandă de intrare organizată în două sau mai multe dimensiuni (eventual ca o memorie);

nu crește puterea de calcul oferită. Nu se schimbă nici clasa funcțiilor care pot să fie calculate și nici a limbajelor acceptate sau decise.

Probleme

problema 1.3-21

Fie maşina Turing MT = ({qO, ql, q2}, {a, #}, m, qO) cu

m(qO,a) = (ql,L)
m(qO, #) = (qO, *#)*
m(ql,a) = (q2,#)
m(ql,#) = (h,#)
m(q2, a) = *(ql,* a)
m(q2,#) = (qO,L)

care este evoluţia maşinii dacă pleacă din configuraţia iniţială: (qO, $\#a^n$ a) n > 0 ?

Soluţie:

Să presupunem că n este un număr impar de exemplu n = 1. în acest caz evoluţia maşinii Turing este:

(qO, #aa) |- (ql, #aa) j- (q2, ##a) |- (qO, ##a) se agaţă

Dacă n este par, de exemplu 2 atunci:

(qO, $\#a^2$a) j- (ql, #aaa) |- *(ql,* #a#a) |- (qO, #a#a) j- (ql, #a#a) |- (h, #a#a)

Deci maşina se opreşte pentru şiruri de forma: $\#a^{2k}$a şi le transformă în #a#a...#a. Pentru şiruri de forma $\#a^{2k+1}$a maşina nu se opreşte.

prohkma 1.3-22

Să se construiască maşina Turing care acceptă limbajul L = { w e {a, b}* j w conţine doi simboli a consecutivi}.

Soluţie:

Se pleacă din starea iniţială qO şi se merge spre dreapta, dacă se întâlneşte un simbol a se va trece într-o stare nouă şi se continuă deplasarea la dreapta. Maşina se va opri pe al doilea simbol a întâlnit.

MT = ({qO,ql},{a,b,#},m,qO)cu

m(qO,a) = (ql,R)
m(qO, x) = (qO, R) x e {b,#}
m(ql, a) = (h, a)
m(ql,x) = (qO, R) x e {b, #}

Descrieţi în cuvinte transformarea efectuată de următoarea maşina Turing.

MT = ({qO, ql, q2}, {a, b, #}, m, qO) cu

m(qO, x) = (qO, x) x e {a, b} arbitrar
m(qO,#) = (ql,L)
m(ql,a) = (ql,L)
m(ql,b) = (ql,L)
m(ql,#) = (q2,b)
m(q2,a) = (q2,,R)
m(q2,b) = (q2,R)
m(q2,#) = (h,#)

Soluţie:

Să considerăm de exemplu comportarea pentru configuraţia iniţială (qO, #ab#).

(qO, #ab#) |- (ql, #ab#) $|-^+$ (ql, #ab#) |- (q2, bab#) $|-^+$ *(ql,* bab#) |- (h, bab#)

Maşina transformă o bandă de intrare de forma #w# în bw#.

Considerând reprezentarea unară a numerelor naturale, care este funcţia pe numere naturale calculată de următoarea maşină Turing:

MT = ({qO, ql, q2, q3}, {I, #}, m, qO) cu

m(qO,#) = (ql,L)
m(qO, I) = (qO, I) arbitrar
m(ql,#) = (h,R)
m(ql,I) = (q2,#)
m(q2,#) = (q3,L)
m(q2,1) = (q2,1) arbitrar
m(q3,#) = (h,R)
m(q3,I) = (h,#)

Soluţie:

Să considerăm câteva exemple de evoluţie.

(qO, ##) |- (ql, ##) |- (h, ##), (qO, #I#)|- (ql, #I#)|- *(ql,* ###)|- (q3, ##)|- (h, ##)

(qO, $\#I^n\#$)|- (ql, #I""I#)l- (q2, #r##)\- (q3, $\#I^{n"2}I\#$) |- (h, $\#I^{n"2}\#$)

0 n < 2

Se observă că f(n) = "!

n - 1 n >= 2

proMt-niii 1.3-2^

Să se construiască mașina Turing care decide limbajul:
L = {w e {a, b}* | w conține cel puțin un simbol a}.

Soluție:
Mașina Turing trebuie să lase pe banda de intrare răspunsul D sau N după cum șirul aparține sau nu limbajului.

: MT = ({qO, q1, q2, q3, q4, q5, q6, q7, q8}, {a, b, #}, m, qO) cu

m(qO,#) = (q1,L)
m(q1,a) = (q3, #) a găsit
:m(q1,b) = (q2,#)
; m(q1, #) = (q7, R) capătul din stânga nu a găsit
m(q2, #) = (q1, L) mai caută
m(q7, #) = (q8, N) a scris răspuns NU
; m(q8, N) = (h, R) gata
' m(q3, #) = (q4, L) mai șterge
i m(q4, #) = (q5, R) capătul din stânga a găsit
''. m(q4, x) = (q3, #) x e {a, b} mai șterge
i m(q5, #) = (q6, D) a scris răspuns DA
; m(q6, D) = (h, R) gata

problema 1.3-26

Să se construiască mașina Turing care acceptă limbajul:
L = {w e {a, b} | w conține cel puțin un simbol a}.

Soluție:
Mașina Turing trebuie să se oprească dacă și numai dacă șirul de pe banda de intrare aparține limbajului. Inițial pe banda de intrare se află șirul sub forma #w# iar în final dacă șirul aparține limbajului pe banda de intrare trebuie să se găsească #w#.

MT = ({qO, q1, q2}, {a, b, #}, m, qO) cu

m(qO, #) = (q1, L)
m(q1,b) = (q1,L)
m(q1, a) = (q2, R)
m(q2, x) = (q2, R) x **e** {a, b}
m(q2, #) = (h, #)

pinlili-ni.i 1 ţ-2"

Să se construiască mașina Turing care decide limbajul:
L = {w e {a, b} |]w| este divizibilă cu 2}.
Soluție;

MT = ({qO, q1, q2, q3, q4, q5, q6, q7, q8}, {a, b, #}, m, qO) cu

m(qO,#) = (q1,L)
m(q1,x) = (q2,#) x e {a,b}
m(q2,#) = (q3,L)
m(q3,x) = (q4,#) x E {a,b}
m(q4,#) = (q1,L)
m(q3, #) = (q7, R) scrie N pe banda de intrare
m(q7,#) = (q8,N)
m(q8,N) = (h,R)
m(q1, #) = (q5, R) scrie D pe banda de intrare
m(q5,#) = **(q6,D)**
m(q6,D) = (h,R)

Să se construiască mașina Turing care având alfabetul de intrare T = {a. b, #}, înlocuiește a-urile cu b-uri și invers. Inițial pe banda de intrare se află șirul #w#, iar în final trebuie să fie #w'# (w' șirul modificat).

Soluții':
M 1 • i :q<!. qi. ql. u.\). !a. I\ "1. *r.*\. qiu cu

m(qO,#) = (q1,L)
m(q1, a) = (q2, b) înlocuiesc a cu b
m(q1, b) = (q2, a) înlocuiesc b cu a
m(q2, x) = (q1, L) x e {a, b}, mă deplasez spre stânga după înlocuire
m(q1, #) = (q3, R) am ajuns la marginea din stânga a șirului, mă întorc la marginea din dreapta șirului
m(q3, x) = (q3, R) x e {a, b}
m(q3,#) = (h,#)

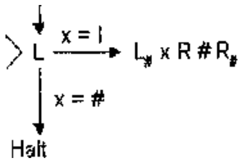
problema 1.3-29

Să se construiască MT compusă care calculează f(x, y) = x + y, unde x și y sunt numere naturale

Soluție:

Considerăm că pentru x, y și rezultat se utilizează reprezentarea unară utilizând simbolul 1. Inițial banda de intrare conține #x#y#, în final banda va conține #x+y#.

Mașina Turing se va deplasa cu o poziție la stânga. Dacă simbolul curent este # (y = 0) atunci i ea se va opri (banda de intrare va conține rezultatul). Dacă simbolul curent este I (y * 0), ; atunci fiecare I din y va fi deplasat cu o poziție la stânga.



Verificare:

(q, ###) I- (q, i- (h, ###;

x = 0 și y = 2

(q, ##II#) (q, ##II#)
(q, ##II#)
(q, #|II#)
(q, #IH#)
(q, #III#)
(q, #I#I#)
(q, #I#I#)
(q, #I#I#)
(q,
(q,
(q,
(q, |
(q, #II##)
(q, #II|#)
(h, #II#)

x = 2 și y = 2

(q, #II#II#) (q, #II#II#)
(q,
(q,
(q,
(q, #III#I#)
(q, #III#I#)
(q, #III#I#)
(q, #III#I#)
(q,
(q,
ţq, #IIII##)
(q, #IIII##)
(q,
(h,

58

Fie x și y numere naturale. Să se construiască mașina Turing compusă care calculează funcția

x —y x > y

H

0 x< y

Considerăm că pentru x, y și rezultat se utilizează reprezentarea unară utilizând simbolul I. Inițial pe banda de intrare se află #x#y#, iar în final va trebui să fie #x-y# pentru x y și ## pentru x < y. Pentru a realiza operația de scădere mașina Turing trebuie să șteargă, de la dreapta la stânga, câte un simbol I din y. Corespunzător simbolului I șters din y trebuie șters un simbol I din x, tot de la dreapta la stânga. După care procedeul se repetă. După ce s-a șters (a fost înlocuit cu #) un simbol I din y, respectiv x, nu mai știm care este simbolul # ce delimita cele două numere. Corespunzător pentru fiecare înlocuire este necesar să cunoaștem capătul din dreapta a fiecărui șir (x și y), pentru care șirurile de pe banda de intrare sunt transformate înainte de efectuarea operației de scădere sub forma axaya_(se introduce un nou simbol a care va delimita cele două numere).

După ce s-a șters un simbol I din y și se încercă ștergerea unui simbol I din x se poate întâmpla să nu mai existe I în x, caz în care trebuie șterși toți simbolii I din y (cazul y > x). în final simbolurile a ce au fost introduse pentru a delimita cele două numere trebuie refăcute cu #. Mașina Turing compusă care calculează funcția f(x, y) este:

y = I **V = j**

= a **x = a**

R, # L, # L, # R', # R, # R **x = I**

x =

R_3#

prnhli-m.i 1.3-31

Să se construiască mașina Turing care acceptă limbajul:

L = { w e

Soluție:

Pentru a se verifica $\#_a(w) = \#_t(w) = \#_c(w)$ se caută de la stânga la dreapta primul a, după care tot de la stânga la dreapta primul b, respectiv primul c. La fiecare căutare (un simbol a, b, respectiv c) dacă simbolul respectiv este găsit, atunci acesta va fi șters. Pentru a șterge fiecare simbol găsit a, b, respectiv c se va folosi un simbol d g {a, b, c} (dacă ștergerea s-ar realiza cu # nu am mai ști care este capătul din dreapta al șirului inițial).
Mașina Turing se oprește dacă șirul de pe banda de intrare aparține limbajului L. Pentru aceasta înainte de a căuta a, b, respectiv c trebuie să vedem dacă primul simbol căutat la dreapta (diferit de d) nu este #, caz în care șirul aparține limbajului, deci mașina Turing se oprește. Pentru șiruri care nu aparțin limbajului mașina va cicla căutând la dreapta un simbol a sau b sau c.

x!=#

Halt

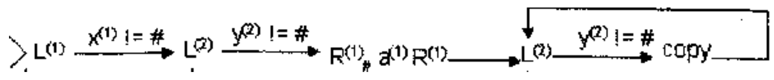
Să se construiască mașina Turing care calculează funcția *f(x, y) = x * y,* unde x și y sunt numere naturale

Soluție:

Considerăm că pentru x, y și rezultat se utilizează reprezentarea unară utilizând simbolul 1. Vom construi mașina Turing cu două benzi: bl conține #x# și b2 conține #y#, rezultatul va fi pe bl #x*y#.
Sunt posibile următoarele situații:
1. Dacă x = O și y 0 rezultatul înmulțirii este 0.
2. Dacă x 0 și y=0 trebuie ștearsă banda 1.
Dacă x 0 și y 0 atunci va trebuie copiat de y - 1 ori conținutul inițial al benzii 1 la sfârșitul benzii 1. Pentru a păstra valoarea inițială de pe banda 1 se marchează sfârșitul șirului cu a, iar în final după terminarea calculării înmulțirii se înlocuiește a cu I și se mută cu o poziție la stânga sfârșitul șirului de pe banda 1.

Astfel mașina Turing testează mai întâi dacă x = 0 sau y = 0, caz în care rezultatul este 0.

limbaje formale si translatoare Seminar LFA



= #

RCI) tfDLCO

= #

R0)

Pentru a copia o singură dată pe x, după simbolul a pe banda bl vom scrie mașina Turing compusă copy, care lucrează numai cu banda b1. Inițial bl conține #wa#, iar în final #waw# sau inițial bl conține #waw'#, iar în final #waw'w#_(w' reprezintă șirul w copiat de un număr de ori).
Mașina Turing compusă care realizează copierea este:

problem» 1.3-33

Să se construiască mașina Turing care decide limbajul: L- {w\ j w e {a. b] *

Soluție:

Pentru a determina în mod determinist mijlocul șirului se folosește un caracter c care se propagă spre stânga șirului. Pentru a verifica dacă cele două "jumătăți" ale șirului, delimitate de simbolul c, sunt egale se vor folosi două benzi.

1. se propagă simbolul c pe banda bl spre stânga cu o poziție și apoi se copiază pe banda bl pe banda b2 (copierea se face parcurgând banda bl de la stânga la dreapta, iar banda b2 de la dreapta la stânga)
2. se compară cele două jumătăți găsite (banda bl este parcursă de la stânga la dreapta pornind inițial de la simbolul c, banda b2 este parcursa de la dreapta la stânga). Compararea are loc până se întâlnește pe banda 1 #. Dacă cele două jumătăți astfel determinate sunt egale trebuie ștearsă banda 1 și se scrie D. Dacă cele două jumătăți determinate nu sunt egale se propagă cu încă o poziție la stânga pe banda 1 simbolul c, după care se repetă copierea pe banda 1 și apoi compararea. Propagarea la stânga cu o poziție a simbolului c pe banda 1 are loc până se întâlnește # (marginea din stânga a șirului), caz în care banda 1 este ștearsă și trebuie scris N.

Pentru aceasta se vor construi mașinile Turing compuse:

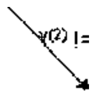
copy pentru a realiza copierea pe banda 2
cmp pentru a compara cele două jumătăți găsite

copy (mașina Turing compusă de copiere)
De exemplu:
Inițial
bl:#abbabcb#
b2:##
după copiere
bl: #abbabcb#
b2: #bcbabba#

| *〈〉=

cmp (mașina Turing compusă de comparare)
Fie cmp mașina Turing compusă care compară cele două jumătăți delimitate de simbolul c. Cele două jumătăți se află pe banda bl, respectiv banda b2.
Inițial:
bl:#abbabcb#
b2: #bcbabba#
Banda bl este parcursă de la stânga la dreapta, iar banda b2 de la dreapta la stânga.
în final:
bl:#abbabcb#
b2: #bcbabba#
Dacă cele două jumătăți sunt egale:
Inițial
bl:#abbcabb#
t>2: #bbacbba#
în final:
bl:#abbcabb#
b2: #bbacbba#

62



I= c

Mașina Turing care decide limbajul L este:

!

cmp

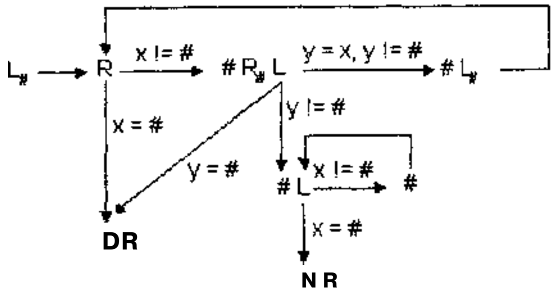
R(1) NO) RtD

R(1) 00) RC)

problema 1.3-34

Să se construiască mașina Turing compusă care decide limbajul:
L = { w e {a, b} * | w= w^R }

Soluție:
Inițial banda de intrare conține #w#, iar în Ikul !^ sau #N#, după cum w aparține sau nu limbajului L. Se verifică egalitatea simbolilor de pe poziții egale față de capetele șirului. Pe măsură ce se parcurge șirul caracterele se șterg și se înlocuiesc cu #.



2 Lex - generator de analizoare lexicale

lex este un generator de analizoare lexicale prezent în orice versiune UNIX. Față de varianta standard (anii 70 - '80) există numeroase variante mai noi. Cea mai cunoscută variantă se numește *flex* (Fast lexical analyzer generator) și face parte dintre instrumentele realizate de FSF (Free Software Foundation) în cadrul proiectului GNU (GNU is Not UNIX). Spre deosebire de *lex* pentru *flex* (ca și pentru alte programe din domeniu public) programul sursă este disponibil, și în același timp performanțele exprimate în timp de execuție și memorie ocupată sunt mai bune. Din acest motiv în cele ce urmează vom prezenta de fapt varianta *flex.*

lex poate să fie utilizat în programare și pentru alte aplicații decât scrierea de compilatoare. Astfel, ori de câte ori un program trebuie să identifice în fișierul de intrare șiruri de caractere având o structură de tip "atom lexical" (adică numere întregi sau reale în diferite formate, cuvinte cheie, identificatori, caractere speciale, etc.) pentru care se execută acțiuni speciale, putem să folosim o funcție generată de lex.

Un generator de programe este de fapt un translator care realizează traducerea unui text care reprezintă specificațiile programului ce trebuie generat (sursa) într-un text scris într-un limbaj de programare (obiect). în cazul lex-ului generarea analizorului lexical utilizează și un cod fix care reprezintă emulatorul automatului determinist corespunzător. Cu alte cuvinte algoritmul de analiză propriu-zis. în cazul lex-ului acest cod este scris în C, și programele generate vor fi programe C.

Formularea "execuția programului lex" se referă printr-un abuz de limbaj la execuția programului generat pe baza specificațiilor și nu la execuția programului care realizează traducerea.

Pentru a obține un program executabil din textul generat de către lex, textul C generat trebuie să fie compilat (utilizând un compilator de C) obținându-se un modul obiect.

Specificațiile sursă pentru lex constau dintr-o descriere a unităților lexicale ce urmează să fie recunoscute de către programul generat. Pentru fiecare unitate lexicală se specifică eventual și o acțiune reprezentată de o secvență de cod C care urmează să se execute la recunoașterea unui șir ce corespunde formei generale. Dacă specificația este corectă atunci se va genera un fișier lex.yy.c (lexyy.c sub MS-DOS). în cadrai acestui fișier apare cel puțin o funcție numită yylexO c a r e reprezintă de fapt analizorul lexical. Execuția acesteia realizează căutarea în șirul de intrare a unui subșir care să "se potrivească" cu unul din modelele descrise în specificații. Dacă un astfel de șir este găsit atunci se va executa acțiunea asociată modelului respectiv.

Pentru a înțelege modul în care se utilizează programul lex trebuie să precizăm câteva lucruri. Și anume, funcționarea oricărui analizor lexical are o parte fixă care nu depinde de modelele de atomi lexicali căutate în șirul de intrare. Algoritmul corespunzător este de fapt simularea unui automat finit determinist. Această parte constituie scheletul analizorului lexical. Acest schelet este un text scris în limbajul C. Partea care variază de la un analizor lexical la altul are de fapt două componente și anume modelele de atomi lexicali căutate în șirul de intrare și acțiunile pe care eventual analizorul lexical trebuie să le execute la identificarea unui șir care se "potrivește" cu un model de atom lexical. Specificațiile lex conțin descrierea părții variabile. Deoarece ce se generează în final este un text C, acțiunile sunt descrise ca secvențe de instrucțiuni C. în aceste secvențe pot să apară orice instrucțiuni C inclusiv apeluri de funcții. Secvențele respective vor "îmbrăca" scheletul părții fixe. Specificarea modelelor de atomi lexicali se face sub formă de expresii regulate. Pornind de la ele programul lex va genera tabelele care descriu funcţia de tranziţie a automatului determinist corespunzător. Aceste tabele reprezintă o parte din structurile de date pe care lucrează "scheletul". Utilizând "scheletul" şi secvenţele de instrucţiuni oferte de către programator programul lex va genera un text care reprezintă analizorul lexical. Programul lex "nu înţelege" nici scheletul şi nici acţiunile specificate de către programator, aceste elemente reprezintă numai şiruri de caractere pe care lex ştie să le combine. în mod corespunzător cel care scrie specificaţiile lex trebuie să aibă în vedere aspecte ca domeniul de valabilitate al variabilelor utilizate în programul generat. Şi asta ţinând cont că există de fapt două tipuri de variabile, variabile pe care le controlează numai cel care scrie specificaţiile lex şi pentru ele se face atât declararea cât şi utilizarea în cadrul secvenţelor specificate de către programator şi variabile care sunt declarate în cadrul scheletului (sunt predefmite din punct de vedere al celui care scrie specificaţiile). Declararea acestor variabile apare în textul generat înainte de începutul funcţiei yylex() Referirea acestora se poate face atât în secvenţele de cod care se execută asociat diferitelor modele de atomi (aceste secvenţe vor fi interne funcţiei yylex()) cât şi în funcţii care vor fi incluse în textul generat după textul funcţiei yylex().

2.1 Expresii regulate. Structura unei specificaţii lex.

în cele ce urmează vom numi program lex (flex) textul ce conţine descrierea unităţilor lexicale şi a acţiunilor corespunzătoare. Să considerăm un exemplu foarte simplu pentru a oferi o idee despre cum arată specificaţiile programele lex.

rcma *2*-1

```
 /* declaraţii de variabile utilizate in
 programul generat
 */
  int numar_cifre = 0, numar__litere = 0;
%}
/* declaraţii de macrodefiniţii utilizate in specificarea regulilor */

cifra [0-9]
litera [A-Za-z]


 {cifra} { numar__cifre++; }
 {litera} { număr litere++;}


main () {
yylex();
 printf("\n număr cifre = %i, număr litere = %i\n",
   numar_cifre, numar_litere) ;
```

Descriere; .

Programul numără literele şi cifrele care apar în fişierul de intrare.

Nu este cel mai simplu program lex posibil. Conţine însă majoritatea elementelor Unui program lex. Se observă că programul este format din trei secţiuni separate de caracterele %%. Prima secţiune poate conţine text de program C care va fi copiat nemodificat în textul programului generat şi macrodefiniţii. Textul care se copiază conţine declaraţiile de variabile şi funcţii ce vor fi utilizate în acţiunile asociate şirurilor recunoscute. Acest text trebuie inclus între caracterele %{ şi respectiv %} (fiecare apărând pe un rând separat). Macrodefinitiile sunt utilizate pentru a scrie mai compact descrierea modelelor de atomi lexicali. Pentru exemplul considerat au fost definite două nume cifra şi litera. Primul nume are asociată o descriere a unei liste de caractere care conţine numai cifre, în timp ce al doilea nume are asociată o listă care conţine orice literă (mare sau mică).

Pentru caractere se consideră ordonarea lexicografică obişnuită. Conform acestei ordonări cifrele sunt caractere care au valori într-un interval în care '0' este cel mai mic iar '9' este cel mai mare, literele mici sunt caractere care au valori mai mari decât literele mari, literele mari şi respectiv mici au valori în câte un interval astfel încât 'A' este caracterul cu valoarea cea mai mică dintre literele mari iar 'Z' este caracterul cu valoarea cea mai mare dintre literele mari, etc. Corespunzător definiţia unei cifre este un caracter având codul cuprins în intervalul închis [0-9] iar o litera este un caracter având codul cuprins în unul dintre intervalele [A-Z] şi respectiv fa-zi, în programele lex utilizarea caracterelor este destul de rigidă. Astfel dacă definiţia literei s-ar fi făcut sub forma:

`litera` [A-Z a-z]

atunci şi caracterul blanc ar fi fost considerat literă.

A doua secţiune conţine specificarea modelelor şi a prelucrărilor asociate. Fiecare model este scris pe o linie separată începând din prima coloană. Pe aceeaşi linie separată de un blanc sau un caracter de tabulare este specificată acţiunea corespunzătoare.

A treia secţiune conţine text C care va fi copiat în programul generat. De obicei aici apar funcţiile referite în acţiunile din secţiunea a doua şi care referă variabile predefmite. Deoarece în cazul nostru generăm un program care se va executa independent (adică nu generăm o funcţie care va fi referită din alte module compilate) în secţiunea a treia este conţinut codul pentru programul principal. în programul principal se apelează funcţia yylex() care va fi generată conform specificaţiilor din a doua secţiune. în exemplul considerat execuţia funcţiei yylex() se termină când se ajunge la sfârşitul fişierului de intrare.

în general dacă funcţiile apelate în cadrul acţiunilor sunt semnificative ca dimensiuni este de preferat ca acestea să apară în module de program compilate separat. în acest mod se simplifică depanarea programului generat.

Pentru descrierea unităţilor lexicale ce urmează să fie recunoscute se utilizează expresii regulate. în acest caz o expresie regulată este interpretată ca un şablon (model) care descrie o mulţime de şiruri. Un astfel de şablon poate să fie utilizat pentru clasificarea şirurilor de caractere în şiruri care corespund sau nu şablonului. Pentru lex se utilizează a doua interpretare a expresiilor regulate. Adică, programul generat va identifica într-un şir de intrare subşirurile care "se potrivesc" cu descrierile făcute în specificaţii.

în notaţia utilizată de lex pentru reprezentarea expresiilor regulate se utilizează caracterele obişnuite utilizate pentru limbajul C. O serie de caractere ca de exemplu: *, +, |, ? au semnificaţie specială în scrierea expresiilor regulate. Aceste caractere se numesc rnetacaractere. în afară de : *, +, I, ? se mai utilizează ca metacaractere:", \, {,}, ^, <, >, $, /, (,)•• în tabelul 2.1 sunt conţinute construcţiile lex în care se utilizează metacaractere.

notație	Semnificație
"șir"	șirul de caractere șir; chiar dacă conține numai un metacaracter, de exemplu "*", permite specificarea blancurilor și mctacaracterelor în expresii regulate (în mod normal un blanc încheie șirul de caractere care descrie o expresie regulată).
\x	dacă "x" este unul din caracterele speciale din limbajul C ca de exemplu t sau n arunci notația reprezintă caracterul respectiv, altfe) notația reprezintă caracterul "x" chiar dacă este un metacaracter.
[lista]	unul dintre caracterele din listă. Lista este formată din caractere individuale sau din intervale. De exemplu [ABC] reprezintă unul dintre caracterele A, B, C, iar [A-Za-z] reprezintă orice literă. Dacă primul caracter din listă este] sau - aiunci el este tratat ca atare (își pierde semnificația de metacaracter).
[^listă]	orice caracter mai puțin cele din listă. De exemplu [^0-9] înseamnă orice caracter, cu excepția cifrelor.
	orice caracter mai puțin caracterul linie nouă (\n).
^e	un șir reprezentat de expresia regulată e dar numai la început de linie.
e$	un șir reprezentat de expresia regulată e dar numai la sfârșit de linie.
<y>e	un șir reprezentat de expresia regulată e dacă analizorul este în starea y (noțiunea de stare va fi explicată ulterior).
e/f	un șir reprezentat de expresia regulată e dacă urmează un șir reprezentat de expresia regulată f. într-o expresie regulată poate să apară o singură dată această condiție (la sfârșitul expresiei).
e?	un șir reprezentat de expresia regulată e sau șirul vid.
e*	0,1,2,... apariții ale unor șiruri reprezentate de expresia regulată e.
e+	1,2,3,... apariții ale unor șiruri reprezentate de expresia regulată e.
e{m,n}	între m și n apariții ale unor șiruri reprezentate de expresia regulată e.
e{m,}	cel puțin m apariți i ale unor șiruri reprezentate de expresia regulată e.
e{m}	exact m apariții ale unor șiruri reprezentate de expresia regulată e.
e|f	un șir de caractere provenind din expresia regulată e sau din expresia regulată f
(x)	dacă x un caracter atunci are același efect ca și includerea caracterului între ghilimele. Dacă x este o expresie regulată atunci notația poate să fie utilizată pentru schimbarea priorității operatorilor utilizați.
{XX}	dacă xx este un nume definit în prima secțiune a unui program lex atunci notația reprezintă rezultatul substituirii numelui xx cu definiția asociata.

Tabelul 2-1

Mai există o construcție specială « E O F », care apare numai in Qex și care reprezintă condiția de sfârșit de fișier. Această construcție nu poate să intre în compunerea unei expresii regulate. în Tabelul 2-1 au apărut o serie de operații care sunt utilizate pentru formarea expresiilor regulate. Se pune problema care sunt regulile de precedență pentru aceste operații. Din păcate aici apar diferențe între lex și flex. în general ordinea operațiilor este în ordine descrescătoare:

```
()
* + ?
concatenare
|
/
```


De exemplu expresia:

[a-z][4-5]+/123+9 reprezintă șirul a444555555551233333339, dar nu reprezintă șirul a444555555551231231239, deoarece interpretarea expresiei este: ([a-z]([4-5]+))/(12(3+)9). Pentru a ușura citirea textului programului și pentru a obține programe portabile între diferitele variante de lex trebuie să se utilizeze paranteze pentru stabilirea ordinii de considerare a operatorilor.

Din Tabelul 2-1 mai rezultă faptul că semnificația unui metacaracter poate să depindă de contextul în care acesta apare. Astfel, în expresia:

cele patru puncte care apar nu au toate aceeași semnificație. Primul și ultimul reprezintă un caracter oarecare (diferit de linie nouă, dar care poate însă să fie și un punct). Al doilea punct indică faptul că pe a doua poziție în șirul căutat poate să apară și un punct. Al treilea caracter precedat de metacaracterul \ indică faptul că pe poziția 3 trebuie să apară un punct. Rezultă că atât șirul "a!.x" cât și șirul".... "se potrivesc" cu expresia regulată.

După cum am mai amintit un program lex este format din maxim 3 secțiuni separate de o linie care conține caracterele %%. Structura generală pentru un program este :

```
secvențe de cod care se copiază în programul generat
tracrodefiniții

reguli
aa
secvențe de cod care se copiază în programul generat
```

împărțirea secvenței de cod care se copiază în programul generat între cele două secțiuni se face în așa fel încât să se permită definirea variabilelor înainte de utilizare. Astfel toate variabilele definite de către programator și care urmează să fie referite în acțiuni trebuie să fie definite în prima secțiune. Dacă în funcțiile care sunt apelate din acțiuni apar referiri la variabile care vor fi definite în mod implicit în funcția yylex (de exemplu variabilele yytext sau yyleng) atunci aceste funcții trebuie să apară în a treia secțiune a programului. Existența unor secvențe de cod este opțională. Secvențele de cod care apar în prima secțiune sunt cuprinse între caracterele "%{" și respectiv "%}" conținute fiecare pe câte o linie separată. Este foarte important să se facă distincția între codul care este "înțeles" și prelucrat de către lex și cel care va fi înțeles de către compilatorul de C. Astfel lex nu face nici o verificare asupra textului cuprins între caracterele %{ și respectiv %}. Abia când se face compilarea programului generat de către lex se va face și verificarea acestor secvențe.

Prima secțiune conține macrodefmiții utilizate pentru simplificarea scrierii modelelor. Forma generală a unei macrodefiniții este:

nome expresie regulată

unde nume este un cuvânt care începe cu o literă sau caracterul"_" și continuă cu litere, cifre sau caractere "-", "_"; expresieregulată este construită pe baza regulilor din Tabelul 2-1. Utilizarea unei macrodefiniții se poate face în orice expresie regulată care apare ulterior prin utilizarea numelui macrodefmitiei între acolade.

Deoarece notaţiile utilizate în expresiile regulate sunt destul de greoaie, este indicată utilizarea unor macrodefiniţii pentru a uşura citirea acestora. De exemplu se poate considera pentru construirea expresiei regulate care reprezintă un număr real secvenţa de macrodefiniţii:

```
cifra  [0-9]
semn [+-]?
IntregFaraSemn  {cifra}+
Exponent  ([Ee]{semn}{IntregFaraSemn})
Real   ({semn}{IntregFaraSemn}\.{IntregFaraSemn}?{Exponent}?)
```

sau direct:

```
Real  [+-]?[0-9]+\. ([0-9]+)?([Ee] [+-] ?[0-9]+) ?
```

Cele două definiţii sunt echivalente ca efect, dar prima soluţie este mai uşor de urmărit.

Regulile au forma generală:

```
expresie  regulată  acţiune
```

Expresia regulată este scrisă începând din prima coloană a liniei pe care apare. Sfârşitul expresiei regulate este reprezentat de un caracter delimitator: blanc, linie nouă, un caracter de tabulare, etc. Acţiunea are forma unei instrucţiuni în limbajul C. Dacă pentru o acţiune sunt necesare mai multe instrucţiuni atunci ele vor fi grupate între caracterele "{" şi "}". De fapt pentru claritate şi pentru evitarea unor efecte laterale neplăcute este de preferat să se încadreze întotdeauna acţiunile între acolade. In orice caz acţiunea trebuie să înceapă pe linia pe care este scrisă expresia regulată.

Dacă pentru mai multe expresii regulate corespunde o aceeaşi acţiune atunci se poate utiliza o notaţie de forma:

```
expresie_regulată1 ]
expresie_regulată2 |
 ... |
expresie_regulatăn acţiune
```

Pentru toate cele n expresii regulate se va executa aceeaşi acţiune.

Dintre cele trei secţiuni este obligatorie numai secţiunea de reguli, care trebuie însă să fie precedată de perechea de caractere "%%". Cel mai simplu text de specificaţii conţine numai aceste caractere. în acest caz efectul funcţiei generate constă din tipărirea şirului de intrare.

2.2 Elemente avansate

Pentru a înţelege modul în care se utilizează programul lex trebuie să precizăm câteva lucruri. Şi anume, funcţionarea oricărui analizor lexical are o parte fixă care nu depinde de modelele de atomi lexicali căutate în şirul de intrare. Algoritmul corespunzător este de fapt simularea unui *automat finit determinist*. Această parte constituie scheletul analizorului lexical. Acest schelet este un text scris în limbajul C la care se adaugă o serie de variabile şi structuri de date. Partea care variază de la un analizor lexical la altul are de fapt două componente şi anume

modelele de atomi lexicali căutate în şirul de intrare şi acţiunile pe care eventual analizorul lexical trebuie să le execute la identificarea unui şir care se "potriveşte" cu un model de atom lexical. Specificaţiile lex conţin descrierea părţii variabile. Deoarece ce se generează în final este un text C, acţiunile sunt descrise ca secvenţe de instrucţiuni C. în aceste secvenţe pot să apară orice instrucţiuni C inclusiv apeluri de funcţii. Secvenţele respective vor "îmbrăca" scheletul părţii fixe. Specificarea modeleelor de atomi lexicali se face sub formă de *expresii regulate*. Pornind de la ele programul lex va genera tabelele care descriu funcţia de tranziţie a automatului determinist corespunzător. Aceste tabele reprezintă o parte din structurile de date pe care lucrează "scheletul". Utilizând "scheletul" şi secvenţele de instrucţiuni oferite de către programator, programul lex va genera un text care reprezintă analizorul lexical. Programul lex "nu înţelege" nici scheletul şi nici acţiunile specificate de către programator, aceste elemente reprezintă numai şiruri de caractere pe care lex ştie să le combine. în mod corespunzător cel care scrie specificaţiile lex trebuie să aibă în vedere aspecte ca domeniul de valabilitate al variabilelor utilizate în programul generat, şi asta ţinând cont că există de fapt două tipuri de variabile, variabile pe care le controlează numai cel care scrie specificaţiile lex şi pentru ele se face atât declararea cât şi utilizarea în cadrul secvenţelor specificate de către programator şi variabile care sunt declarate în cadrul scheletului (sunt predefinite din punct de vedere al celui care scrie specificaţiile). Declararea acestor variabile apare în textul generat înainte de începutul funcţiei yylexQ Referirea acestora se poate face atât în secvenţele de cod care se execută asociat diferitelor modele de atomi (aceste secvenţe vor fi interne funcţiei yylex()) cât şi în funcţii care vor fi incluse în textul generat după textul funcţiei yylexQ-

2.2.1 Funcţionarea analizorului lexical generat de lex

Analizorul lexical generat este de fapt o funcţie yylex(). în execuţia acestei funcţii se parcurge textul de intrare şi se caută un subşir care începe cu primul caracter şi "se potriveşte" cu o expresie regulată specificată în reguli. Dacă există mai multe soluţii se va lua în considerare cel mai lung subşir care "se potriveşte". Pentru mai multe soluţii având aceeaşi lungime se consideră soluţia care a fost găsită prima (în ordinea de parcurgere secvenţială a regulilor). în cazul expresiilor regulate care descriu şi contextul dreapta al şirului analizat (de exemplu expresiile regulate care termină cu $ sau care conţin condiţii de tip-/) în determinarea lungimii "de potrivire" participă şi contextul dreapta. De exemplu pentru expresiile:

```
"ab" {printf(" s-a recunoscut ab");}
"abc" {printf(" s-a recunoscut abc");}
"ab'V'cc" {printf(" s-a recunoscut ab urmat de ce");}
```

pentru şirul abec se va afişa textul:

```
s-a recunoscut ab urmat de ce
```

Variabila globală yytext este un pointer spre începutul şirului care "s-a potrivit". Lungimea şirului este memorată în yyleng. După ce se actualizează valorile variabilelor yytext şi yyleftg. conform identificării unei expresii regulate se va executa acţiunea asociată expresiei, care poate să utilizeze variabilele yytext şi yyleng.

După execuția acțiunii asociate unui subșir găsit se va continua parcurgerea șirului de intrare începând cu caracterul care urmează subșirului selectat. Dacă nu se găsește un subșir care să se potrivească cu o regulă se execută acțiunea implicită, adică se va copia caracterul curent în fișierul de ieșire. Din acest motiv dacă acest efect nu este dorit trebuie să se prevadă o ultimă regulă de forma:

```
|\n /* se potriveşte cu orice caracter */
```

care va "înghiți" orice caracter care nu "s-a potrivit". în continuare se va încerca găsirea unei "potriviri" începând cu următorul caracter, etc. A se vedea programele 2.2. - 2.4.

2.2.2 Stări de start

Pentru toate exemplele considerate până acum analizorul generat încearcă în ordine toate regulile specificate pentru a determina cel mai lung subșir care "se potrivește" cu o regulă. Uneori în funcție de context anumite reguli trebuie să fie ignorate. în acest scop într-un program lex se pot utiliza stări de start. Dacă nu se specifică altfel, analizorul este în starea 0 numită simbolic INIȚIAL. Se pot declara și alte nume de stări în prima secțiune sub forma:

hs numei nume$_n$

sau

%x nume$_n$

în primul caz dacă o acțiune este prefixată de <numep atunci acțiunea respectivă se va executa numai dacă starea curentă este numej. în același caz toate acțiunile care nu au nici un prefix vor fi luate în considerare indiferent de starea curentă. A doua notație nu este recunoscută de lex dar este recunoscută de flex și de alte variante ale programului lex. Dacă declararea numelui stării s-a făcut cu x (starea nume; este denumită în acest caz stare exclusivă) atunci acțiunile care nu au prefixul <numep nu vor fi executate dacă starea curentă este nume-j. Se poate prefixa o regulă și cu o listă de stări. în acest caz regula va fi considerată dacă starea curentă este una dintre stările din listă. Trecerea dintr-o stare în alta se face prin apelarea unei macrodefiniții predefmite BEGIN. Forma de apel a acestei macrodefmiții este:

```
BEGIN(nume);
```

unde nume este numele stării în care trebuie să treacă analizorul generat. A se vedea programele 2.5. - 2.6.

Utilizarea stărilor exclusive este de preferat celor obișnuite pentru că măresc claritatea programului. Stările exclusive există însă numai în flex. Pentru lex se poate obține o comportare de tip stare exclusivă utilizând o variabilă din programul generat care va fi testată înainte de fiecare verificare.

2.2.3 Macrodefiniții, funcții și variabile predefinite

în cadrul acțiunilor pot să apară orice instrucțiuni C valide, deoarece generatorul de programe lex nu face nici o analiză a acestora. Există însă câteva macrodefiniții lex predefmite care pot să apară în orice acțiune. în discuția despre stări am întâlnit deja una dintre aceste macrodefiniții (BEGIN). Mai există alte două macrodefiniții: ECHO și REJECT.

Efectul apelului macroinstrucțiunii ECHO constă din copierea valorii yytext la ieșire. Cu alte cuvinte, efectul acestei macroinstrucțiuni constă din copierea la ieșire a subșirului pentru care se execută acțiunea.

1

Macroinstrucțiunea REJECT este utilizată atunci când se dorește modificarea algoritmului de alegere a subșirului care "se potrivește". Și anume s-a precizat că dacă există mai multe soluții se alege cel mai lung subșir. în caz de egalitate, dictează ordinea în care au fost scrise expresiile regulate. Dacă se utilizează macroinstrucțiuea REJECT se renunță la cea mai bună soluție și se consideră a doua. A se vedea programele 2.7. - 2.9.

Se recomandă însă să se evite utilizarea macrodefmiției REJECT deoarece reduce viteza programului generat și în același timp conduce la programe greu de citit.

în acțiuni se pot utiliza și o serie de funcții predefmite:

- yymoreO adaugă ca prefix la următorul șir care "se potrivește" șirul pentru care se execută acțiunea curentă (șirul indicat de yytext). Vezi programul 2.10.
- yyless(n) se renunță la ultimele yyleng - n caractere din yytext care vor fi reutilizate pentru formarea subșirului următor. Vezi programul 2.11.
- unput(c) caracterul c este "pus" în bufferul de intrare devenind următorul caracter de intrare. Vezi programul 2.12.
- inputO citește un caracter din șirul de intrare. Această funcție permite amestecarea recunoașterilor de subșiruri făcută de către analizor cu cea făcută în cadrul acțiunilor. Vezi programul 2.13.
- yyterminate() - este de fapt o macroinstrucțiune care poate să fie redefinită, reprezintă o instrucțiune return. Ca efect se abandonează execuția funcției yylex(), adică se produce o revenire forțată din analizor. în mod implicit la întâlnirea sfârșitului de fișier se produce execuția acestei macroinstrucțiuni.

2.2.4 Fișiere de intrare multiple

în toate exemplele considerate până acum fișierul de intrare a fost considerat în mod implicit fișierul standard de intrare. Dar, pentru situațiile reale, fișierul de intrare este un fișier disc. Din acest motiv este necesar un mecanism de specificare a fișierului pe care analizorul generat îl va prelucra. Variabila yyin conține întotdeauna numele fișierului din care se citește textul prelucrat de către analizorul generat. Vezi programul 2.14.

Tratarea fișierelor multiple depinde de modul în care acestea sunt utilizate în timp. Evident există două situații. în prima situație este vorba de o înlănțuire de fișiere (adică după ce este tratat un fișier se trece la al doilea apoi la al treilea etc. în orice caz nu este necesară o revenire la un fișier anterior). în a doua situație trecerea la un nou fișier se face înainte de a se termina prelucrarea fișierului curent. La terminarea prelucrării unui fișier se face revenirea la fișierul anterior (ca de exemplu dacă se tratează directiva #include).

Să considerăm situaţia în care se face o înlănţuire a fişierelor. In acest caz ceea mai bună soluţie este utilizarea funcţiei yywrap. Această funcţie este apelată în mod automat de către analizorul generat în momentul în care se detectează un sfârşit de fişier. Dacă rezultatul apelului funcţiei este unu înseamnă că nu există un alt fişier de tratat şi execuţia funcţiei yylex s-a terminat. Dacă rezultatul apelului funcţiei este zero înseamnă că prelucrarea nu s-a terminat. în acest ultim caz funcţia yywrap realizează şi comutarea yyin pe noul fişier de intrare. Vezi programul 2.15.

2.3 Exemple comentate

11:111:1 *2-2*

```
blanc [ \t]+
%%
{blanc} { printf (" ");}.
{blanc}$ { printf("#");/* ignoră blancurile de la sfârşit */ }


main()
{ yylex();}
```

Descriere; ~

Programul înlocuieşte secvenţele de blancuri şi caractere de tabulare ("\t") cu câte un singur blanc. De asemenea şterge blancurile de la sfârşitul liniilor înlocuindu-le cu un caracter #. Adică pentru un fişier de intrare de forma:

```
a     b c
   a    b  d
```

se va obţine un fişier de ieşire de forma:

```
a b c#
a b d#          . . .  •:
```

Comentarii:

Se observă că o secvenţă de blancuri aflată la sfârşitul unei linii satisface ambele reguli, dar este luată în considerare cea de a doua regulă care se potriveşte subşirului mai lung (cel care conţine şi caracterul linie nouă, care nu este însă luat în considerare).

```
şterge
"alfa"      { printf("beta"); }
gama        { printf ("delta"); }
```



```
itiain() {
  yylex();
```

Descriere:

Şterge apariţiile şirului de caractere "şterge" din text şi înlocuieşte şirul "alfa" cu "beta", respectiv "gama" cu "delta". Dacă se utilizează un fişier de intrare de forma:

```
alfa şterge alfa
alfa beta gama delta
```

Se va obţine un fişier de ieşire de forma:

```
beta beta
beta beta delta delta
```

Comentarii: , ' ' . ' . :. .'-'"• . ...

în program se utilizează cele două forme echivalente de reprezentare a şirurilor de caractere. Să considerăm şi o modificare a programului anterior:

```
%%
şterge ;
"alfa"+     printf('beta");
gama+       printf{'delta");
9-2
main(){
  yylex ();
}  '    ..
```

De data aceasta dacă vom executa programul generat asupra fişierului de intrare:

```
alfaalfa şterge alfaaaaaaaaaaaaaaaa
gamagama şterge gamaaaaaaaaaaaaaaaa
```

Se va obţine fişierul de ieşire:

```
beta betaaaaaaaaaaaaaaaaa
deltadelta delta
```

Se observă că operatorul "+" se aplică în primul caz asupra întregului şir ("alfa"+) respectiv asupra ultimului caracter, în al doilea caz (gama+).

Acţiunile asociate expresiilor regulate pot să conţină instrucţiuni de forma return expresie. In cazul în care se produce o "potrivire" între un şir de caractere şi un astfel de model execuţia funcţiei yylex se termină şi se "întoarce" valoarea expresiei. în cazul în care nu se execută o astfel de instrucţiune se continuă parcurgerea şirului de intrare până când acesta se termină. Să considerăm tema 2-4. pentru care fiecare acţiune se încheie cu o instrucţiune return.

```
/* declarații de inacrodefiniții utilizate în specificarea
 regulilor
*/

cifra [0-9]
semn [+-]?
IntregFaraSemn {cifra}+
Exponent ([Ee]{semn}{IntregFaraSemn})
Real ({semn}{IntregFaraSemn}\.{IntregFaraSemn}?{Exponent}?)


{Real} { printf("real\n"); return yyleng;}
{IntregFaraSemn} { printf("intreg\n"); ; return yyleng;}
. { return -1; }
\n { return -2; }
%%
main () {
  printf("lungime = %i\n", yylex());
```

Descriere:

Executând programul generat asupra șirului de intrare:

123.56

Se va obține fișierul de ieșire:

```
real
lungime = 6
```

Comentarii:

în programul 2.4 sunt tratate toate cazurile. Adică dacă primul caracter care apare în fișierul de intrare nu reprezintă un început de număr întreg sau real atunci analiza se oprește cu rezultat -l iar pentru caracterul linie nouă rezultatul este -2.

#ina 2-5

```
 char * a;
 int numar_aparitii = 0, numar_cifre = 0;
%}
litera [A-Za-z]
cifra [0-9]
%s URMĂTOR
%%
<INITIAL>{litera}+ { a = malloc(yyleng);
 strcpy(a, yytext);
 numar_aparitii = 1;
 BEGIN (URMĂTOR);
```



```
<0KMATOR>{litera}+ { if ( !strcmp(a, yytext))
 număr aparitii++;

{cifra} {număr cifre++;}

main() {
 yylex();
 printf(„primul cuvânt %s apare de %i ori\n au fost %i cifre",
 a, număr aparitii, număr cifre);
 free(a);
}
```

Descriere:

Programul numără aparițiile primului cuvânt (șir care conține numai litere) dintr-un text în textul care urmează. De asemenea, programul contorizează și cifrele care apar în text.

Comentarii:

A fost declarată o stare numită URMĂTOR TIVIVIWI în jLvi si.iiv c^iv comandată prin execuția macroinstrucțiunii:

```
BEGIN (URMĂTOR) ;
```

apelată pentru acțiunea corespunzătoare găsirii primului cuvânt. Dacă se execută analizorul generat asupra fișierului de intrare: • •

```
123$ % &456 abbb bribnbnb abb nbnbnb abbb nbnbnbn abbbbbbbbbb
789 abbb hjhjhjhjh abb
```

se va obține fișierul de ieșire:

```
primul cuvânt abbb apare de 3 ori
 au fost 9 cifre
```

Dacă programul se schimbă prin declararea stării URMĂTOR ca stare exclusivă execuția programului generat pentru același fișier de intrare va produce:

789

```
primul cuvânt abbb apare de 3 ori
 au fost 6 cifre
```

Cifrele nu au fost luate în considerare decât atâta timp cât starea curentă a fost starea INIȚIAL.

ti 111.1 2-(i

```
x COMENTARIU
{
int NumarLinieComentariu = 0;
int NumarLinieProgram = 0;
```

```
„/*"  { BEGIN(COMENTARIU); ++NumarLinieComentariu; }

<CXMENTARIU>[^*\n]* /* se ignora orice nu este ,*'sau \n */
<CCMENTARIU>"*''+P*An]* {/* se ignora orice * care nu este
 urmat de ,/' sau linie noua */ }

<CCMENTARIU>\n ++NumarLinieComentariu;
<CCMENTARIU>"*"+"/" BEGIN (INIŢIAL) ;
\n ++NumarLinieProgram;


main() {
 yylex();
 printf(„au fost %i linii de comentariu \n",
 NumarLinieComentariu);
 printf(„au fost %i linii de program\n", NumarLinieProgram);
```

Beseriere:

Programul generat numără liniile ce conţin comentarii respectiv text „obişnuit" într-un program. Comentariile încep şi se termină cu caracterele „/*" respectiv „*/".

Comentarii:

La începutul comentariului şi la sfârşitul său se face comutarea între stările INIŢIAL şi COMENTARIU. în starea COMENTARIU se ignoră orice caracter mai puţin caracterele „*" şi respectiv „\n". Pentru caracterul „*" se verifică ce caracter urmează. Se ignoră şirurile de caractere „*" care nu sunt urmate de „/" sau de „\n". Cazul în care urmează caracterul „/" corespunde sfârşitului de comentariu, caracterele „\n" sunt tratate separat.. Pentru caracterele „\n" se face incrementarea numărului de linii din comentariu. Se observă că o linie este contorizată fie ca linie program fie ca linie comentariu după cum caracterul \n a fost întâlnit în starea INIŢIAL sau în starea URMĂTOR. Dacă comentariul este inclus într-o linie care conţine şi text de program după el atunci linia respectivă va fi contorizată atât ca linie program cât şi ca linie comentariu.

```
%%
„acesta" 1
„acesta este" I
„acesta este un" !
„acesta este un test" |
„acesta este un test ciudat" { ECHO; REJECT;}
        /* sg ignoră ce nu se potriveşte */

main(){
 yylex 0 r
}
```


Descriere:

Dacă se execută programul generat pentru programul asupra unui fişier de intrare care conţine textul:

```
acesta este un test ciudat
```

se va obţine textul:

```
acesta este un test ciudatacesta este un testacesta este unacesta esteacesta
```

C&mmKarik

Se observă că întâi s-a "potrivit" cel mai lung subşir pentru care s-a executat acţiunea ECHO care l-a tipărit apoi s-a executat acţiunea REJECT prin care s-a renunţat la soluţia respectivă. Se va considera următorul subşir (mai scurt). Şi pentru acesta se execută acţiunea { ECHO; REJECT} deci se va face afişarea, etc. Macrodefiniţia REJECT este utilă dacă pentru un acelaşi şir trebuie să se execute două acţiuni diferite. De exemplu în programul 2.8. se recunosc câteva cuvinte cheie, dar în acelaşi timp se face şi contorizarea tuturor cuvintelor care apar în text (inclusiv cuvintele cheie).

tenia 2-8

```
 int NumarCuvinte = 0;
}
cuvânt [A-Za-z][A-Za-zO-9]*
%%
bun |
rau |
urit |
frumos { ECHO; printf(" este adjectiv\n"); REJECT;}
maninca |
doarme I
scrie I
citeşte { ECHO; printf(" este verb\n"); REJECT;}
{cuvânt} { NumarCuvinte++;}
   /* ignoră orice altceva */

main () {
 yylex();
 printf("\n au fost %i cuvinte\n", NumarCuvinte);
}
```

Descriere:

Dacă se execută programul generat pentru programul asupra unui fişier de intrare care conţine textul:

```
aaa
bun
```

Se va obţine textul:

```
bun este adjectiv
au fost 2 cuvinte
```

Comentarii:

Subșirul "aaa" se potrivește cu a treia regula, deci el este numărat. Pentru subșirul "bun", acesta se potrivește cu prima regula, deci se afișează mesajul corespunzător și se execută ! macrodefmiția REJECT, adică se renunță la potrivirea găsită. Următoarea potrivire aleasă este regula a treia în care se numără cuvintele. Deci în final numărul de cuvinte este 2.

```
int NumarCuvinte = 0;
%}
cuvint [Â-Za-z][A-Za-zO-9]*
%%
{cuvint} { NumarCuvinte++; REJECT;}
bun I
rau I
urit I
frumos {
 ECHO;
 printfC este adjectiv\n") ;
 }
maninca I
doarme I
scrie |
citeşte { ECHO; printff" este verb\n");}
    /* ignora orice altceva */
%%
main (){
 yylex();
 printf("\nau fost %i cuvinte\n", NumarCuvinte);
```

Deseriere:

Să presupunem că schimbăm ordinea regulilor și poziția macroinstrucțiunii REJECT din 2. ca in programul curent. Pentru același text de intrare ca în tema 2-8:

aaa
bun

se va obține un text de ieșire ca:

```
bun este adjectiv
au fost 7 cuvinte
```

Comentarii:

Răspunsul pare ciudat, dar ceea ce se întâmplă de fapt este că întâi subșirul "aaa" se potrivește cu prima regulă, deci el este numărat, apoi se renunță la această soluție și se caută ! altă "potrivire". Se va găsi o potrivire pentru un șir mai scurt "aa", deci și acest cuvânt se • numără. Iar se renunță, urmează o potrivire cu subșirul "a", deci contorul ajunge la 3. în acest caz cu prima regulă nu mai există nici o potrivire, nici cu a doua, dar a treia regulă reușește să "înghită" un "a". Se continuă de la al doilea caracter "a". La terminarea parcurgerii subșirului "aaa" contorul este 6. în cazul subșirului "bun" după ce prima regulă a executat macrodefimția

REJECT se va încerca cu altă variantă și aceasta este potrivirea cu a doua regulă pentru care se consideră aceeași lungime. Această regulă "înghite" subșirul "bun" care deci este numărat o singură dată.

tema 2-10

```
%S DUPĂ
Cuvânt [A-Za-z]*
CuvintSpecial {Cuvânt}"$"
CuvintObisnuit {Cuvânt}[\t \n]+
%%
<INITIAL>{CuvintSpecial} { BEGIN(DUPĂ); putchar('(');}
<INITIAL>{CuvintObisnuit} /* se ignora */
<DUPA>{CuvintObisnuit} { yymoreO;}
<DUPA>{CuvintSpecial} { yytext[yyleng -1] = ')';
 ECHO; BEGIN(INIŢIAL);

main(){
 yylex();
```

Descriere:

Programul generat extrage șirurile de caractere cuprinse între caractere "$" din șirul de intrare și le copiază la ieșire între paranteze simple. Astfel pentru șirul:

a$a b$c
a $a b$c
aa$ a b$c
aa$ a b $c
aa$ a b $ c

se va obține:

(a b) (a b) (a b) (a b) (a b)

Comentarii:

Soluția din program nu este unică, nici măcar nu este cea mai simplă, dar înțelegerea ei reprezintă un bun exercițiu.

Unu 2-11

```
%s DUPĂ
Cuvânt [A-Za-z]*
CuvintSpecial {Cuvânt}"$"
CuvintObisnuit {Cuvânt}[\t \n]+
```

```
<INITIAL>"$" { BEGIN(DUPA); putchar ('('); }
<INITIAL>{CuvintObisnuit} /* se ignora ce a mai rainas */
<INITIAL>{CuvintSpecial} { yyless(yyleng - 1); }
<DUPA>"$"     { BEGIN(INIŢIAL);
 yytext[yyleng -1] = ')';
 ECHO;
 }
<DUPA>{CuvintObisnuit} { yymore(); }
%%
main(){
 yylex() ;
```

Descriere:

O soluţie mai complicată cu efect similar cu 2.10.

Comentarii:

Se foloseşte în acest caz funcţia yyless pentru a extrage caracterul "$

```
Cuvânt  ([a-zA-Z][a-zA-ZO-9]*)
CuvintSpecial  (\ ({Cuvânt}\))

{Cuvânt} {   '       .
 int i;
 printf("1 leng = %i, text = %s\n", yyleng,
 yytext);
    unput (')');
    for (i = yyleng - 1; i >= 0; —i)
    unput(yytext[i]);
 unput('(');
 printf("2 leng = %i, text = %s\n", yyleng, yytext);

{CuvintSpecial} |
\(\) ECHO;

main () {
   yylex();
```

Descriere:

Dacă se execută pentru şirul de intrare
abcde
se va obţine:

```
1 leng = 5, text = abcde
2 leng = -1, text = abcd)

(abcd)
```


- ***Comentarii:***

Se observă că subşirul recunoscut abcde a fost "prelungit" dar numai la un capăt, primul caracter care este "dat înapoi" (")") înlocuieşte ultimul caracter din subşir "e". Apelul yyless(yyleng -1) este echivalent cu unput(yytext[yyleng - 1]).

```
"/*" { int c;
   for(;;){
 while((c = inputO) != '*' && c != EOF)
 ; /* se ignora ce nu poate sa fie sfârşit */
   if (c = '*') {
   while ((c = inputO) = '*')

 if (c = '/') {
 break; /* s-a găsit sfârşit */
 printf(" a fost comentariu\n");


 if (c = EOF) {
     printf(" comentariu neterminat \n");
     break;
```

Descriere:

Un exemplu de utilizare al funcţiei input() în tratarea comentariilor cuprinse între caractere duble cum sunt cele din limbajul C.

2-14

```
%{
#define  LT      0
#define  LE      1
#define  EQ      2
#define  NE      3
#define  GT      4
#define  GE      5
#define  IF     10
#define  THEN   11
#define  ELSE   12
ttdefine  ID    13
#define  NUMĂR  14
```

```
#define   OPREL   15
#define   ATRIB   16

int yyval;
char x[30]; /* tabela de simboli  ( o intrare) */


/* macroinstrucţiuni */

DELIMITATOR   [ \t\n]
BLANC      {DELIMITATOR}+
litera     [A-Za-z]
 ifra      [0-9]
id         {litera}({litera}|{cifra})*
număr      {cifra}+(\.{cifra}+)?(E[+-]?{cifra}+)?



{BLANC}     { /* nici o acţiune */}
if          { printf(" IF"); }
then        { printf(" THEN"); }
else        { printf(" ELSE"); }
{id}        { yyval = Tratareld0;
              printf("%i, %i", yyval,ID);


{număr}     { yyval = TratareNumar();
              printf("%i, %i", yyval,NUMĂR);

            { yyval = LT; printf("%i, %j    yyval,OPREL);}
            { yyval = LE; printf("%i, %J    yyval,OPREL);}
            { yyval = EQ; printf(" li, %i    yyval,OPREL);}
"O"        '{ yyval = NE; printf("%i, %:    yyval,OPREL);}
            { yyval = GT; printf("%i, %:    yyval,OPREL);}
            { yyval = GE; printf("%i, li    yyval,OPREL);}
            { yyval = ATRIB; printf (" %i,    ", yyval, ATRIB),


int TratareNumar(void){
 printf("\n număr");
 return 1;


main( int argc, char **argv ){
 ++argv, —argc;
 if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
 else
   yyin = stdin;
  yylex ();
```


Descriere:

Programul reprezintă specificaţiile flex pentru un subset al limbajului Pascal. Analizorul va recunoaşte identificatori şi numere, pentru care va apela funcţii corespunzătoare şi o parte din operatorii şi cuvintele cheie care pot să apară într-un program Pascal.

Comentarii:

Funcţia TratareNumar ar fi putut să fie inclusă şi în prima secţiune de program deoarece nu referă variabile ce nu au fost încă definite. în schimb funcţia Tratareld utilizează variabilele yytext şi yyleng, deci nu poate să apară decât în a treia secţiune de program. Programul principal este mai complicat faţă de variantele anterioare şi anume, înainte de apelul funcţiei yylex se va stabili valoarea variabilei yyin care poate să fie stdin sau un nume specificat de utilizator.

linia 2-15

```
fundef yywrap
int NumarCaractere = 0, NumarCuvinte = 0, NumarLinii= 0;


BLANC    [ \t]
NOBLANC   [^ \t\n]



{NOBLANC}+NumarCaractere += yyleng;  ++NumarCuvinte;
{BLANCJ+ NumarCaractere += yyleng;
\n   ++NumarLinii; ++NumarCaractere;
«EOF» yyterminate();



char **ListaFisiere;
unsigned int FisierCurent = 0;
unsigned int NumarFisiere;

main( int argc, char **argv )  {
 FILE *fişier; ListaFisiere = argv + 1;
 NumarFisiere = argc -1;
 if (argc > 1)  {
    FisierCurent = 0;
    fişier = fopen(ListaFisiere[FisierCurent], "r");
    if (!fişier){
      printf("!!!eroare!!!");
      exit(1);
    }
    yyin = fişier;
    yylex();
```

```
/* Noua definiţie yywrap */

yywrap(){
  FILE *fisier;
  fişier = 0;
  fclose(yyin);
  printf("\nFisier %s", ListaFisiere[FisierCurent++]);
  printf("\nNumar caractere = %8d", NumarCaractere);
  printf("\nNumar cuvinte = %8d", NumarCuvinte);
  printf("\nNumar linii = %8d", NumarLinii);
  NumarCaractere = 0;
  NumarCuvinte = 0;
  NumarLinii = 0;
  if .(FisierCurent > NumărFişiere) return 1;
    fişier = fopen(ListaFisiere[FisierCurent], "r");
  if (!fişier){
    printf("!!! eroare!!!");
    exit(l);
  }
  yyin = fişier;
  return(fişier ? 0:1);
```

Descriere:

Programul calculează numărul de caractere, cuvinte şi linii conţinute într-o listă de fişiere transmise prin linia de comandă prin care este apelat programul generat. în programul principal se face deschiderea primului fişier. La fiecare apel al funcţiei yywrap se va face închiderea fişierului precedent, afişarea rezultatelor obţinute pentru acesta şi deschiderea fişierului următor. După fiecare deschidere de fişier variabila globală yyin este poziţionată pe indicatorul fişierului deschis. în momentul în care s-a ajuns la sfârşitul listei funcţia yywrap va avea rezultatul 1 şi execuţia funcţiei yylex se termină.

Comentarii:

Se observă că a fost necesară utilizarea directivei:

```
#undef yywrap
```

pentru a se putea defmii o nouă funcţie yywrapO deoarece există o macrodefinitie yywrap definită ca:

```
#define yywrap () 1
```


3 Teme pentru acasă

tema 3-1

Sase scrie gramaticile ce generează limbajele:

L={ $(ab)^n a'b^j c^k$ | n >= 0, i, j, k > 0}
L= { $a^i(ab)^n b^j c^k$|i>=0,n,j,k>0}
L={ aVcVb)" |j >= 0, n, i, k > 0}
L={ $a^i b^i (ab)^n c^k$ | k >= 0, n, j, i > 0}
L={ w e {a, b} $\#_a$(w) par si #b(w) impar}
L={ we {a,b}* $\#_a$(w) par si #b(w) par}
L= { we {a,b}* $\#_a$(w) impar si $\#_b$(w) par}
L-{ wG {a, b}* $\#_a$(w) impar si #b(w) impar}
L={ we {a,b} w nu conţine şirul baab }
L={ w e {a, b} w nu conţine şirul abba}
L={ we {a,b} w nu conţine şirul bbaa }
we {a, b}* w nu conţine şirul abbb }

\ Să se demonstreze folosind lema de pompare că următorul limbaj nu este independent de context:

|j}
: L= $\{a^i b^i c^k [i>j>k\}$
i L= $\{a^i b' c^k j0<=i<=j<=k\}$
^ L= $\{a^i b^j c^k|i,j,k>=O,i!=j,i!=k,j!=k\}$

Pentru următoarele gramatici să se elimine recursivitatea stângă. în gramaticile rezultate să se elimine *X* producţiile, după care să se elimine simbolii nefinalizaţi şi inaccesibili.

G = ({S, A, B}, {a, b}, P, S) unde P este:
S -> Ba | Ab
A -» Sa | Aab | a
B -> Sb I Bba | b ....

G = ({S, A, B, C}, {a, b}, P, S) unde P este:
S -» A | B
A -> Ba | Sb| b
B -» AB | Ba
C -> AS | b

G = ({S, A, B, C}, {a, b}, P, S) unde P este:
S -^ AB | AC
B ->• Bb | b
C -> Ac | BC
A - > a | C

G = ({S, A, B, C}, {a, b}, P, S) unde P este:
S -» AB | AC
A -> Sa | a
B -» BC | AB
C - > a B | b

tK m.i ţ-4

: Pentru următoarele gramatici, să se elimine simbolii nefinalizaţi şi inaccesibili.

I G = ({S, A, B}, {a, b}, P, S) unde P este
S -* SS | Aaa
A ->• aAB | aS | b
B - > b B

G = ({S, A, B, C}, {a, b}, P, S) unde P este
S ^ AB|CA
A - » a
B -> BC | AB
C -> aB | b

G = ({S, A, B}, {a, b}, P, S) unde P este
S - > a | A
A - > A B

G = ({S, A, B}, {a, b}, P, S) unde P este
S - > A
A -» bS | b
C ^ A S b

temu 3-5

Pentru expresile regulate:
(a|b)*ab(a|b)*
(ajb)*ab*(a|b)
(a|b)* a (a|b) b*
(a|b)+ a b*

Să se construiască automatele finit nedeterminist şi determinist şi să se minimizeze stările. Să se facă şi construcţia directă pornind de la expresia regulată la automatul finit determinist.

Uina 3-6

Să se construiască automatele cu stivă care acceptă limbajele generate de gramaticile:

G = ({S, A, B}, {a, b}, P, S) unde P este
S -> aB | bA
A -> a | aS | bAA
B -> b | bS | aBB
G = ({S, A}, {a, b}, P, S) unde P este
S - > a A A
A -> aS | bS| a

G = ({S, A, B, C}, {a, b}, P, S) unde P este
S - > A B C
A -» BB | *X*
B -> CC | a
C -» AA I b

u-niii '-"

Să se construiască maşinile Turing compuse care decide limbajele:

L = {w e {a, b}* | $\#_a$(w) par si $\#_b$(w) impar}
L = { w e {a, b}* | $\#_a$(w) par si #b(w) par}
L = { w € {a, b} * | $\#_a$(w) impar si $\#_b$(w) par}
L = {w e {a, b}* j $\#_a$(w) impar si $\#_b$(w) impar}

4 Bibliografie

1. Irina Athanasiu, Limbaje Formale si Compilatoare, Centrul de multiplicare, IPB, 1992
2. Alfred Aho, Jeffrey Ullman, The Theory of Parsing, Translation and Compiling,'voi. 1 Parsing
3. Harry R. Lewis, Christos H. Papadimitriou, Elements of the theory of computation