Capitolul 1

Automate finite.
Rolul lor în modelarea activităţilor din analiza lexicală[1]

Un *program de recunoaştere* pentru un limbaj este acel program care primeşte la intrare un şir "X" şi răspunde "DA" dacă "X" este o secvenţă a limbajului, respective"NU", în caz contrar. Pentru a analiza programul de recunoaştere corespunzător unei expresii regulate (program gen analizor lexical) trebuie contruită, în prealabil, o diagramă de tranziţii generalizată numită **automat finit** (AF).

Un AF poate fi determinist (AFD) sau nedeterminist (AFN). Nedeterminarea constă, în principiu, în faptul că, din cel puţin o stare sunt posibile mai multe tranziţii pentru acelaşi simbol de intrare.

Atât AFD cât şi AFN pot recunoaşte limbaje specificate prin expresii regulate. De obicei AFD conduc la programe mai rapide dar numărul lor de stări este mai mare.

Există metode de transformare (conversie) a expresiilor regulate în ambele tipuri de automate. Conversia este mai simplă în cazul AFN.

În exemplele ce urmează în acest paragraf se va utiliza următoarea expresie regulată[1]:

`(a|b)*abb` – mulţimea tuturor şirurilor compuse din caractere a şi b care se termină cu secvenţa `abb` (ex: `abb, aabb, babb, aaabb, …`)

1.1 Definiţia unui AFN

Un AFN este un model matematic reprezentat prin următorul cvintet:

$AFN = <S,A,f_t,s_o,F>$, unde:

- **S** este **mulţimea stărilor**;
- **A** reprezintă mulţimea simbolurilor de intrare (**alfabetul** de intrare);
- **f_t** se numeşte **funcţie de tranziţie** şi pune în corespondenţă perechi „stare-simbol" cu "stări", adică: $f_t : S \times A \rightarrow S$;
- **s_o** este **starea iniţială** (de start); $s_o \in S$;
- **F** este **mulţimea stărilor finale** (acceptoare); $F \subseteq S$.

Un AFN se poate reprezenta ca un graf orientat etichetat numit **graf de tranziţie** în care nodurile corespund stărilor automatului iar etichetele descriu funcţia de transfer (f_t). Se poate remarca asemănarea dintre un *graf de tranziţie* şi o *diagramă de tranziţie*. Deosebirile de principiu sunt următoarele:

- acelaşi caracter se poate utiliza pentru a eticheta două sau mai multe tranziţii din aceaşi stare;
- este permisă utilizarea ca etichetă a simbolului ε (şirul vid).

În fig. 1.1 se prezintă un *exemplu* de AFN corespunzător expresiei regulate: **`(a|b)*abb`**. Nedeterminismul se manifestă în starea *0* în care, pentru simbolul de intrare *a*, sunt posibile 2 tranziţii: se poate rămâne în starea *0* sau se poate trece în starea *1*.

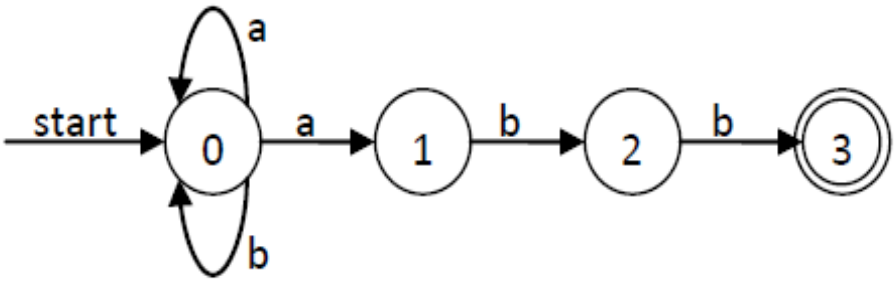


Fig. 1.1 Un prim exemplu de AFN pentru expresia: **`(a|b)*abb`**

În calculator, funcţia de tranziţie se poate implementa în mai multe moduri. O modalitate potrivită este sub forma unei **tabele de tranziţii** care au câte o linie pentru fiecare stare, câte o coloană pentru fiecare simbol de intrare (inclusiv pentru simbolul ε, daca este necesar) (fig. 1.2). Intrarea în tabelă corespunzătoare liniei i şi simbolului de intrare a este mulţimea de stări pentru care există tranziţii din starea i etichetate cu simbolul a.

Stare	Simbol intrare	
	a	b
0	{0,1}	{0}
1	-	{2}
2	-	{3}
3	-	-

Fig 1.2 Tabela de tranziţii corespunzătoare AFN din fig. 1.1

Se spune că un AFN *acceptă* un şir de intrare ”X” dacă şi numai dacă există un drum în graful de tranziţii începând din starea de start până la o stare acceptoare astfel încât etichetele pe drumul respectiv să formeze şirul ”X”. Drumul respectiv se numeşte **cale de acceptare** pentru şirul ”X”. Aceluiaşi şir de intrare îi pot corespunde mai multe căi de acceptare într-un AFN. Totalitatea şirurilor acceptate de un AFN reprezintă **limbajul definit de automatul respectiv.**

1.2 Definiţia unui AFD

Un AFD este un caz particular de AFN la care se impun următoarele restricţii asupra funcţiei de transfer f_t.

1) din nici o stare nu pleacă tranziţii etichetate cu ε;
2) pentru orice stare s şi pentru orice simbol de intrare a, există cel mult un arc etichetat cu a care pleacă din s.

În concluzie, un AFD va avea cel mult o tranziţie din fiecare stare pentru orice simbol de intrare. Aceasta înseamnă ca AFD conţine cel mult o cale de acceptare (recunoaştere) pentru un şir de intrare dat.

În fig. 1.3 se prezintă un algoritm pentru simularea (parcurgerea) unui AFD, având starea iniţială s_0, mulţimea stărilor acceptoare **F** şi funcţia de transfer f_t. Algoritmul se aplică asupra unui şir de intrare ”X”, terminat prin caracterul special **eof** şi generează la ieşire

raspunsul ”DA” în cazul în care şirul ”X” este acceptat şi ”NU” în caz contrar. Funcţia `carurm` furnizează următorul caracter din intrare iar procedura `gen` generează rezultatul final.

```
s:=s0;
c:= carurm;
while (c≠eof) and (∃ tranziţie de ieşire din s pt c) do begin
   s:= ft(s,c);
   c:= carurm;
end while;
if (s ∈ F) and (c=eof) then
    gen (“da”)
else
    gen(“nu”);
```

Fig. 1.3 Simularea unui AFD

1.3 Construirea unui AFD echivalent cu un AFN dat

Datorită faptului că funcţia de transfer f_t este multiplu definită pentru anumite stări, simularea prin program a funcţionării unui AFN este dificilă. Din acest motiv se preferă intercalarea unei etape intermediare de transformare a **AFN** într-un **AFD echivalent** (ambele recunosc acelaşi limbaj) urmând ca, în final, să se realizeze programul care simulează funcţionarea AFD.

Ideea pe care se bazează algoritmul de transformare este aceea că fiecare stare din AFD corespunde unei mulţimi de stări din AFN reprezentând toate stările în care s-ar putea ajunge din starea de start pe toate căile posibile, pentru un şir de intrare”X”. În continuare se va nota cu N automatul nedeterminist şi cu D, cel determinist echivalent.

Se va contrui tabela de tranziţii pentru D notată **Dtranz** şi, implicit, mulţimea stărilor AFD, notată **Dstări**. Fiecare stare a lui D corespunde unei mulţimi de stări din N. ”Dtranz” va fi astfel contruită încât va simula, în paralel, toate tranziţiile care se pot efectua în N pentru un şir de intrare dat. Evidenţa mulţimilor de stări din AFN se realizează prin intermediul operaţiilor definite în tab. 1.1.

Operatia	Descriere
ε_închidere(s)	Mulţimea stărilor AFN caere pot fi atinse din starea „s” utilizând numai tranziţii *ε*.
ε_închidere(T)	Mulţimea stărilor AFN care pot fi atinse numai prin tranziţii *ε* din toate stările *s* componente ale mulţimii de stări T.
f_t(T,a)	Mulţimea stărilor AFN la care există o tranziţie din oricare stare *s* a mulţimii T, pentru simbolul de intrare *a*.

Tab. 1.1. Operaţii asupra starilor AFN

Înainte de a vedea primul simbol de intrare, automatul N poate să fie în oricare din stările mulţimii *ε_închidere(s_o).* Să presupunem că pentru o secvenţă dată de simboluri de intrare, s-a ajuns la stările din mulţimea *T*. Daca *a* este următorul simbol de intrare, N se poate deplasa, la citirea acestui sinbol, în oricare din stările mulţimii $f_t(T,a)$ şi, implicit, la oricare din stările conţinute în *ε_închidere($f_t(T,a)$).* Algoritmul de determinare a mulţimii ”Dstări” şi de construire a tabelului ”Dtranz” este prezentat în fig. 1.4.

* se calculează *ε_închidere(s_o)* şi se consideră această mulţime ca fiind prima stare, nemarcată, a mulţimii ”Dstări”;

```
while * ∃ o stare nemarcată T ∈ Dstări do begin
   * marchează T;
   for * orice simbol de intrare a do begin
        U:= ε_închidere(f_t(T,a));
        if  * U ∉ Dstări then
             *adaugă U la Dstări, ca nemarcată;
        end if;
        Dtranz[T,a] := U;
   end for;
end while;
```

Fig. 1.4 Determinarea mulţimii stărilor şi a tabelei de tranziţii corespunzătoare AFD echivalent cu un AFN dat

O stare din *D* este o stare acceptoare dacă ea reprezintă o mulţime de stări AFN de care aparţine cel puţin o stare acceptoare a lui N. Pentru calculul lui *ε_închidere(T)* se poate aplica algoritmul schiţat în fig. 1.5.

```
* introdu toate stările din T în stivă;
* iniţializează ε_închidere(T) cu T;
while * stiva nu este goală do begin
  * extrage starea din vârful stivei, notată t;
  for * orice stare u, cu arc de la t la u etichetat cu ε do
     if * u ∉ ε_închidere (T) then begin
          *adaugă u la ε_închidere (T);
          * introdu u în stivă;
     end if;
  end for;
end while;
```

Fig. 1.5 Algoritmul pentru calculul mulţimii **ε_închidere (T)**

Exemplu: Se consideră AFN din figura 1.6

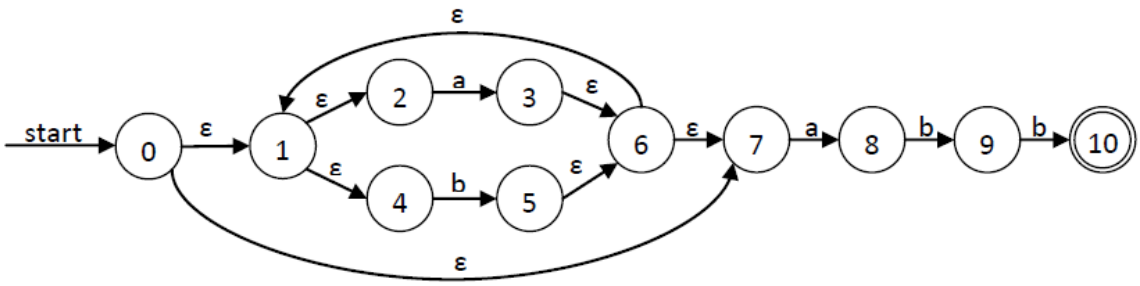


Fig. 1.6 AFN corespunzător expresiei **(a|b)*abb**

Se aplică algoritmul din figura 1.4:
ε_închidere(0)={0,1,2,4,7}
A={0,1,2,4,7} – starea de start a AFD
Dstări = {A} A, nemarcată

Ciclul 1

- marchează A => Dstări={A}

1) Pentru "A" şi "a":

f_t(A,a)={3,8}
U = ε_închidere ({3,8})= {1,2,3,4,6,7,8} – nu există în Dstări
B = {1,2,3,4,6,7,8}
Dstări = {A,B}
Dtranz [A,a]=B

2) Pentru „A" şi „b":

f_t(A,b) = {5}
U= ε_închidere ({5}) = {1,2,4,5,6,7} – nu există în Dstări
C = {1,2,4,5,6,7}
Dstări = {A,B,C}
Dtranz[A,b] = C

Ciclul 2

- marchează B => Dstări = {A, B, C}

1) Pentru „B" şi „a":

f_t(B,a)={3,8} => conduce la B, existentă în Dstări
Dtranz[B,a]=B

2) Pentru „B" şi „b":

f_t(B,b)={5,9}
U = ε_inchidere({5,9})={1,2,4,5,6,7,9}- nu exista în Dstări
D = {1,2,4,5,6,7,9}
Dstări = { A, B, C, D}
Dtranz [B,b]=D

Ciclul 3

- marchează C => Dstări = { A, B, C, D }

1) Pentru "C" şi "a":

f_t(C,a) = {3,8} => conduce la B, existentă în Dstări
Dtranz[C,a]=B

2) Pentru "C" şi "b":

f_t(C,b)={5} => Conduce la C, existentă în Dstări
Dtranz[C,b] = C

Ciclul 4

- marchează D => Dstări = { A, B, C, D}

1) Pentru "D" şi "a":

f_t(D,a) = {3,8} => conduce la B, existentă în Dstări
Dtranz[D,a]=B

2) Pentru "D" şi "b":

f_t(D,b)={5,10}
U = ε_închidere({5,10})={1,2,4,5,6,7,10}- nu exista în Dstări
E = {1,2,4,5,6,7,10}
Dstări = { A, B, C, D, E}
Dtranz [D,b]=E

Ciclul 5

- marchează `E => Dstări = { A, B, C, D, E}`

1) Pentru "E" şi "a":

f_t`(E,a) = {3,8} => conduce la B, existentă în Dstări`
`Dtranz[E,a]=B`

2) Pentru "E" şi "b":

f_t`(E,b)={5} => Conduce la C, existentă în Dstări`
`Dtranz[E,b] = C`

Concluzie:

Toate stările din `Dstări` sunt marcate => algoritmul se încheie.

`Dtranz` este prezentată sub forma tabelei de tranziţii îi fig. 1.7

Stare	Simbol intrare	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Fig. 1.7 Tabela de tranziţii `Dtranz` pentru AFD echivalent cu AFN din fig 1.6

În Fig. 1.8 este prezentat AFD corespunzător tabelei `Dtranz` care este echivalent cu AFN iniţial. Singura stare finală a AFN, starea *10* care aparţine de starea *E* a AFD. Rezultă că *E* este singura stare finală a AFD.

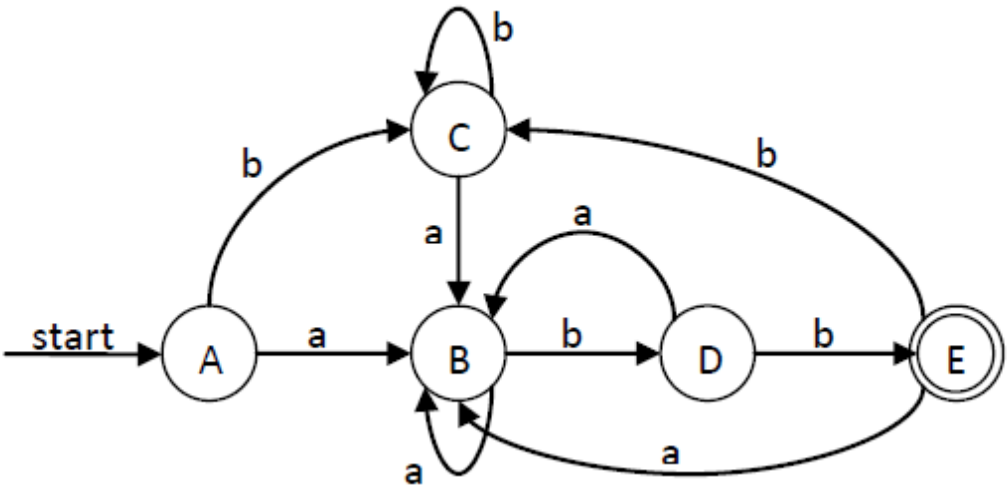


Fig. 1.8 AFD rezultat, echivalent cu AFN din fig. 1.6

1.4 Construirea unui AFN echivalent cu o expresie regulată

Există mai multe modalităţi de realizare a unui program de recunoaştere corespunzător unei expresii regulate. O primă modalitate, descrisă şi analizată în acest paragraf, constă în construirea unui AFN echivalent cu expresia regulată urmată de simularea comportării AFN pentru un şir de intrare dat. Dacă viteza de execuţie este importantă se poate include, ca etapă intermediară, transformarea AFN într-un AFD echivalent conform algoritmului din fig. 1.4, urmată de simularea comportării AFD utilizând algoritmul din § 1.2 (fig. 1.3).

O altă alternativă, principal diferită, care va fi prezentată în § 1.10, constă în construirea AFD direct din expresia regulată, fară a mai apela la construirea, în prealabil, a unui AFN echivalent.

Algoritmul prezentat în continuare urmăreşte cazurile parcurse la definiţia unei expresii regulate. La început se construiesc AFN-uri care recunosc simbolul **ε** şi orice simbol din alfabet. Pe baza acestor AFN elementare se realizează în continuare AFN-uri pentru expresii care conţin operaţii de reuniune (alternativă), de produs (concatenare) şi de închidere Kleene (operaţia stea). Fiecare pas în construcţie va avea un număr de stări cel mult egal cu dublul numărului de simboluri şi de operatori din expresia regulată iniţială.

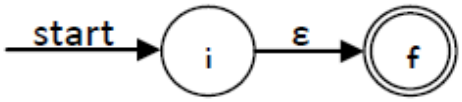
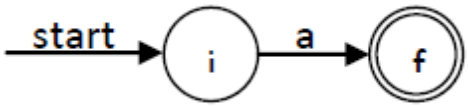
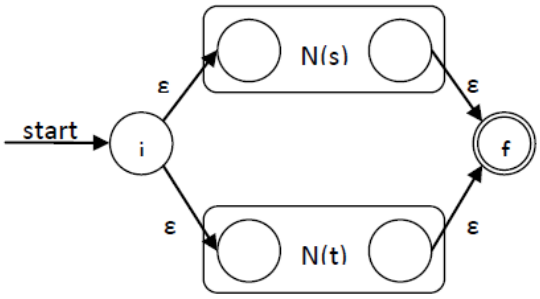
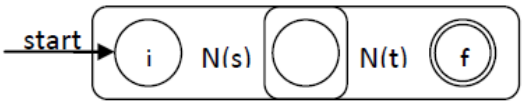
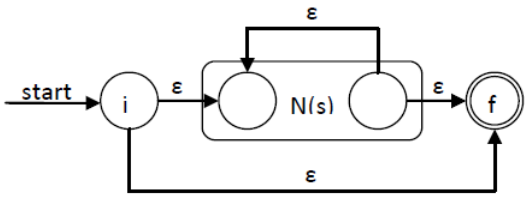
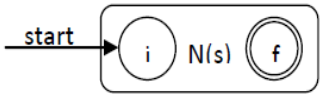
Regulile utilizate pentru construirea automatelor elementare, cât şi cele corespunzătoare operaţiilor de bază sunt prezentate sintetic în tab. 1.2. `i` şi `f` reprezintă starea iniţială respectiv starea finală a automatului construit. Notaţia **N(r)** desemnează automatul corespunzător expresiei regulate `r`.

Metoda de construire propriu-zisă a AFN, denumită **Construcţia lui Thompson**, constă din următoarele operaţii:

- se descompune expresia dată, `r`, în subexpresiile constituente;
- se utilizează regulile 1 şi 2 (tab. 1.2) pentru a construi AFN corespunzător tuturor simbolurilor de bază din `r`;
- Pornind de la structura sintactică a expresiei `r`, se combină AFN construite anterior, utilizând regulile 3-6 (tab. 1.2), până se ajunge la AFN pentru întreaga expresie.

De fiecare dată cand construcţia introduce o nouă stare, aceasta trebuie să primească un nume (număr) distinct. Se poate verifica faptul că AFN produs pentru o expresie `r` are următoarele proprietăţi:

1) N(r) are cel mult de *2 ori* atâtea stări cât este *numărul de simboluri plus numărul de operatori din* `r`.
2) N(r) are exact o stare de start şi o stare finală (acceptoare); niciun arc nu intră în starea de start şi nici un arc nu părăseşte starea finală (starea finală nu are tranziţii de ieşire). Această proprietate este valabilă atât pentru automatul final cât şi pentru automatele componente.
3) Fiecare stare din N(r) are cel mult o tranziţie de ieşire etichetată cu un simbol din alfabet sau cel mult două tranziţii de ieşire etichetate cu ε.

Regula nr:	Expresia regulată	AFN-rezultat (echivalent)	Observaţii
1	**ε** (simbolul vid)	 N(**ε**)	
2	**a** (orice simbol din alfabet,a є A)	 N(a)	Daca acelaşi simbol apare de mai multe ori în expresia regulată, se construieşte care un asemenea AFN, distinct, pentru fiecare apariţie a simbolului.
3	**s|t** (reuniune) limbajul acceptat L(s)|L(t)[L(s)UL(t)]	 N(s|t)	Se introduc doua stari noi, starile de start si acceptoare din N(s) si N(t) isi pierd statutul care l-au avut
4	**st** (concatenare) limbajul acceptat L(s)L(t)	 N(st)	Starea de start a lui N(s) devine stare de start pentru N(st) iar starea finală a lui N(t) devine stare finală N(st); nu se introduc stari noi ci numărul de stări din automat scade cu unu prin contopirea stării finale a lui N(s) cu cea iniţială a lui N(t).
5	**s*** (închidere Kleene) limbajul acceptat (L(s))*	 N(s*)	„i” şi „f” sunt stări noi
6	**(s)** Limbajul acceptat: L(s)	 N((s))	Acelaşi automat ca şi pentru expresia „s”.

Tab. 1.2 Regulile de construire a unui AFN echivalent cu o expresie regulată

Exemplu: **r = (a|b)*abb**

Mai întâi se realizează arborele sintactic al expresiei, reprezentat în fig. 1.9.

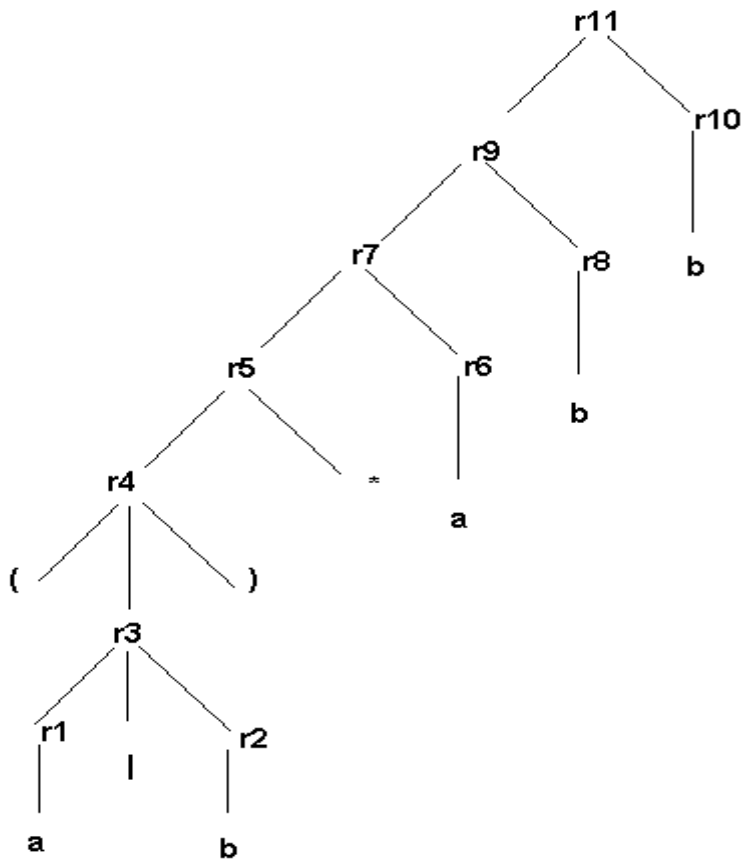
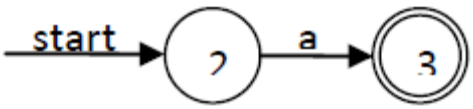


Fig 1.9 Arborele sintactic corespunzător expresiei **(a|b)*abb**

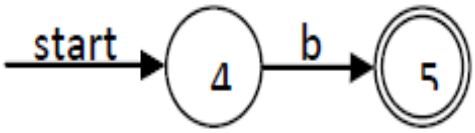
În continuare, se contruiesc succesiv AFN corespunzător expresiilor constituente: $r_1, r_2, ..., r_{11}$.

r1=a



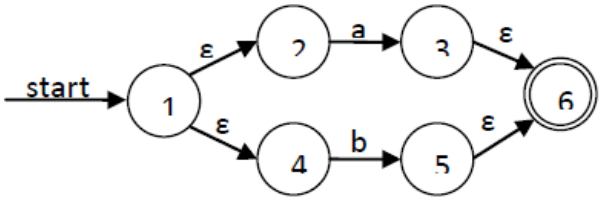
N(r1):

r2=b



N(r2):

r3=r1|r2

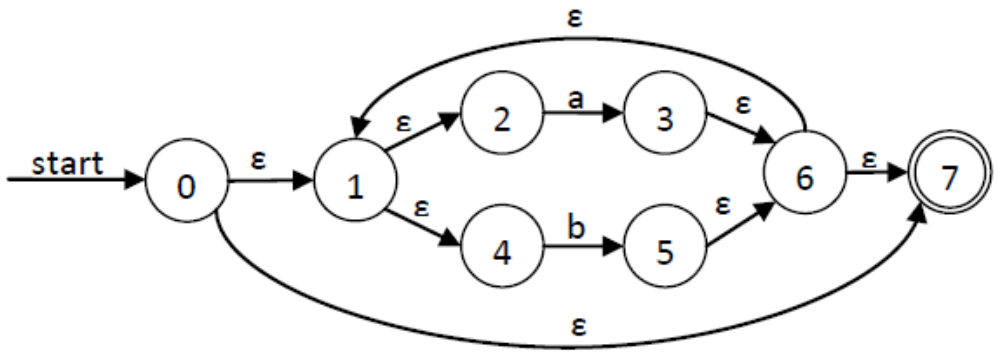


N(r3):

r4=(r3)

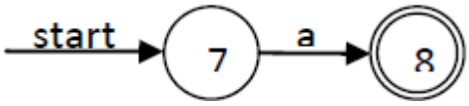
N(r4)≡N(r3)

r5=r4*



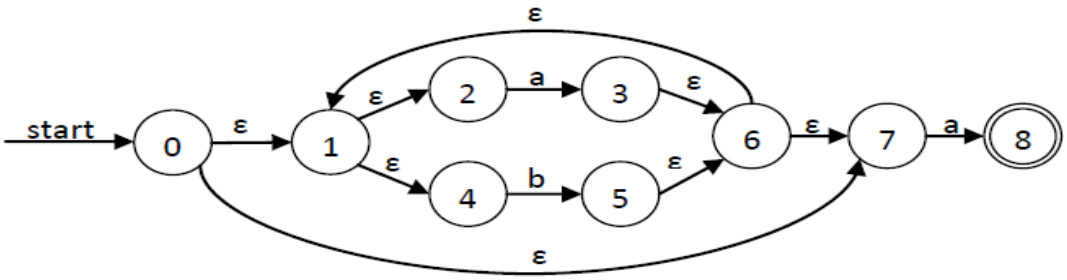
N(r5):

r6=a



N(r6):

r7=r5r6

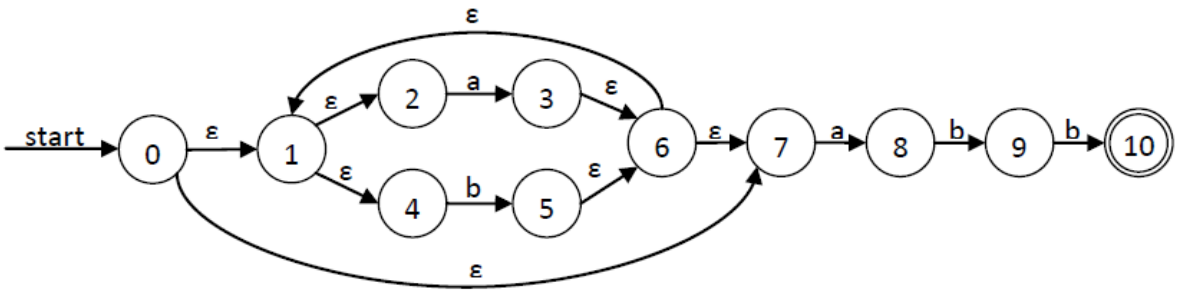


N(r7):

AFN corespunzător expresiilor r_8 şi r_{10} sunt similăre cu N(r_6) iar N(r_9) se obţine analog cu N(r_7). În final:

r_{11}= r_9r_{10}=(a|b)*abb

N(r11):



Urmărind structura sintactică a expresiei regulate de la care se porneşte, metoda prezentată este un algoritm **dirijat de sintaxă**. AFN obţinut pe această cale are relativ multe

stări în comparaţie cu cele obţinute prin alte metode. Acest dejavantaj este însă compensat de simplitatea şi **naturaleţea** mecanismului de contrucţie.

1.5 Algoritm pentru simularea comportării unui AFN

Se consideră un AFN notat N, construit dintr-o expresie regulată conform algoritmului prezentat în **§** 1.4. Se va prezenta un algoritm care stabileşte dacă N acceptă (recunoaşte), sau nu, un şir de intrare dat, X. Pentru a mări eficienţa acestui calcul, se utilizează proprietăţile suplimentare ale unui AFN construit pentru o expresie regulată conform algoritmului anterior. Starea iniţială a automatului este notată cu s_0 iar mulţimea stărilor finale este *F*. Se consideră de asemenea, că şirul de intrare se termină cu caracterul special **eof**. Algoritmul este prezentat în fig. 1.10.

```
S:= ε_închidere (s₀);
a:=carurm;
while (a≠eof) and (∃tranziţie de ieşire din s pt. c) do
   begin
      s:= ε_închidere (fₜ(S,a);
      a:= carurm;
   end;
if (S∩F ≠ Φ) and (a=eof) then
     gen („da”)
else
     gen(„nu”);
```

Fig. 1.10 Descrierea algoritmului care simulează comportarea unui AFN

Structurile de date necesare pentru implementarea eficientă a acestui algorim sunt două stive şi un şir de cifre binare, indexat de stările lui N. Într-o stivă se ţine evidenţa mulţimii curente a stărilor nedeterministe iar a doua, se utilizează pentru calculul mulţimii de stări următoare. Pentru calculul mulţimii *ε_închidere*, se poate aplica algoritmul din fig. 1.5. Vectorul de cifre binare înregistrează dacă o stare este prezentă într-o stivă pentru a se evita dublare ei. Timpul de căutare a unei stări în vector este constant. Dupa calculul complet al stării următoare se poate schimba rolul stivelor. Deoarece fiecare stare din N are cel mult doua tranziţii de iesire, rezultă că fiecare stare produce, după efectuarea unei tranziţii, cel mult două stări noi. Notăm cu **|N|** numărul de stări din automat şi cu **|X|** numărul de simboluri din sirul X (lungimea sa). Timpul necesar pentru calculul mulţimii de stări următoare este proporţional cu **|N|** iar timpul total de simulare va fi proporţional cu **|N|x |X|.**

Exemplu:
Se consideră AFN din fig 1.6, şirul *X* fiind format dintr-un singur caracter, *a*.

```
S= ε_închidere(0) = {0,1,2,4,7}
ft(S,a)={3,8} => S= ε_închidere({3,8})={1,2,3,4,6,7,8}

F={10} => S∩F=Φ => algoritmul generează („nu”).
```

1.6 Consideraţii de eficienţă a algoritmilor prezentaţi

Dându-se o expresie regulată r şi un sir de intrare X există, din cele prezentate până acum, două metode pentru a determina dacă $X \in L(r)$ şi anume:

1) Aplicând algoritmul prezentat în § 1.4, se construieşte AFN corespunzător lui r. Notând cu $|r|$ lungimea expresiei r, timpul necesar pentru construirea unui AFN este de ordinul lui $|r|$ si se noteaza $\theta(|r|)$. Tabela de tranziţii pentru N poate fi înregistrată într-un spatiu de memorie, deasemenea proportional cu $|r|$. Utilizând algoritmul din fig. 1.10, se poate stabili daca N acceptă şirul X, într-un timp $\theta(|r| \times |X|)$, cu un consum de spaţiu de memorie $\theta(|r|)$.
2) AFN rezultat pentru o expresie regulată r se transformă într-un AFD echivalent, utilizând algoritmul din § 1.3. Implementând funcţia de tranziţie printr-o tabelă de tranziţie, se poate determina dacă şirul de intrare X este recunoscut de automat, aplicând algoritmul din fig.1.3, într-un timp $\theta(|x|)$ – proporţional cu lungimea şirului şi independent de numărul de stări din AFD. Rezultă că procedeul este foarte rapid şi trebuie aplicat atunci când timpul de execuţie este critic. Consumul de spaţiu de memorie este $\theta(2^{|r|})$ şi în anumite situaţii particulare, poate să fie foarte mare. Pentru exemplificare, se consideră expresia regulată:

$$(a|b)^*a\ \underbrace{(a|b)(a|b)\ldots(a|b)}_{n-1 \text{ paranteze}}$$

care descrie şirul cu proprietatea că al n-lea caracter, numărat de la capătul din dreapta, este a. La această expresie nu se poate renunţa la lungime (nu poate fi scrisă mai concentrat). În această situaţie, numărul de stări din AFD este 2^n, pentru că trebuie ţinut cont de ultimele n caractere din şirul de intrare pentru a identifica poziţia caracterului a. Astfel de expresii sunt totuşi rare.

O variantă a metodei 2), bazandu-se tot pe AFD, constă în construirea parţiala a tabelei de tranziţii, utilizând tehnica numită „evaluarea leneşă a tranziţiei”. În acest caz tranziţiile sunt calculate în timpul execuţiei, analog cu simularea AFN dar o tranziţie de stare pentru un anumit caracter de intrare nu se calculează decât atunci când este absolut necesar. Tranziţiile calculate se înregistreaza într-o zonă care funcţionează ca o **memorie cache**: de fiecare dată cand urmează să se realizeze o tranziţie, se consultă mai întâi memoria cache. Dacă tranziţia nu este găsită, ea se calculează şi se înregistrează. În momentul în care memoria cache se umple, se şterge o tranziţie calculată anterior. În felul acesta, cerinţele de spaţiu sunt proporţionale cu lungimea expresiei regulate, la care se adaugă dimensiunea memoriei cache, fiind de acelaşi ordin cu cele de la AFN. Performanţele de viteză sunt apropiate de cele ale AFD (nu se strică în mod spectaculos) pentru că, în majoritatea cazurilor, tranziţia necesară va fi găsită în memoria cache.

1.7 Proiectarea analizoarelor lexicale bazată pe automate de tip AFN

Se consideră tiparele reprezentate prin expresiile regulate $r_i, i=1,n$. AFN corespunzătoare se notează cu $N(r_i), i=1,n$. Aceste automate parţiale se combină într-un automat unic, în care se introduce o unică stare de start, cu tranziţii ε spre fiecare din cele n automate parţiale (fig. 1.11).

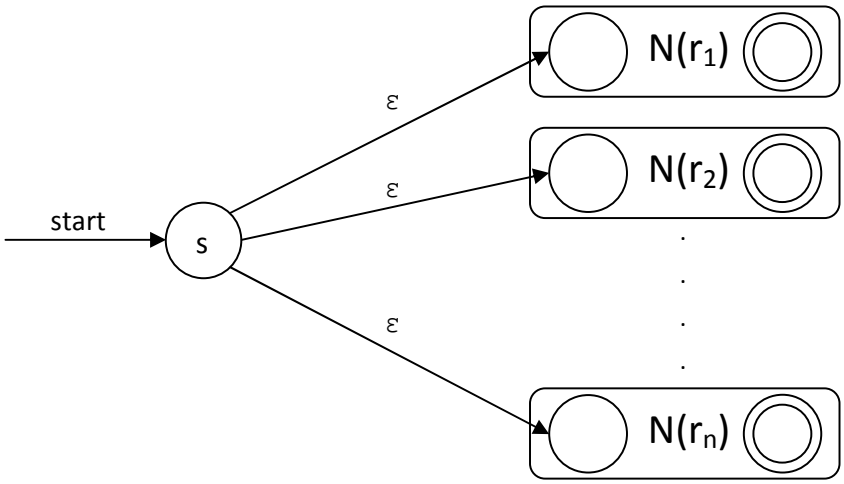


Fig. 1.11 Structura AFN combinat

Pentru a simula comportarea AFN combinat, algoritmul din fig. 1.10, elaborat pentru simularea unui singur AFN, se modifică astfel încât să se asigure recunoaşterea celui mai lung prefix din fişierul de intrare care corespunde unui tipar. Pentru aceasta, atunci când se găseşte o mulţime de stări ce include o stare acceptoare, se continuă simularea până se ajunge la "terminare", adică la o mulţime de stări din care nu mai există tranziţii de ieşire corespunzătoare simbolului de intrare curent. Dacă se adaugă o stare acceptoare la mulţimea curentă de stări, se înregistrează poziţia curentă de intrare şi tiparul r_i corespunzător acestei stări acceptoare (fiecare AFN parţial are o singura stare acceptoare). În cazul în care mulţimea curentă de stări conţine o stare acceptoare, se păstrează ultima stare acceptoare întâlnită. La realizarea condiţiei de terminare, pointerul de anticipare este retras la ultima poziţie de stare acceptoare înregistrată. Tiparul cu care s-a făcut corespondenţa pentru acea stare, identifică atomul găsit iar lexema este constituită de şirul dintre cei doi pointeri care gestionează tamponul de intrare. Specificarea poate fi astfel făcută încât să existe întotdeauna un tipar care să se pună în corespondenţă cu intrarea (eventual cel de eroare).

Exemplu: Se dă programul Lex de mai jos, constând din 3ER si nici o definiţie regulată:

```
a     {}    /* se omit acţiunile */
abb   {}
a*b+  {}
```

Cei trei atomi de mai sus sunt recunoscuţi de automatele din figura 1.12(a).

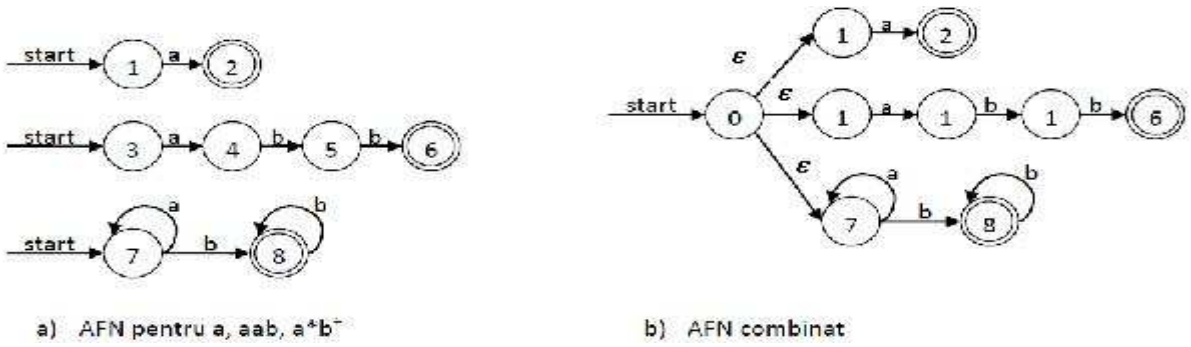


Fig. 1.12 AFN care recunoaşte cele trei tipare diferite

Al treilea automat este simplificat faţă de cel care ar rezulta prin *Construcţia lui Thompson.* Conform metodei de mai sus, cele trei automate se transforma in AFN combinat din fig. 1.12(b).

În continuare se analizează comportarea AFN combinat pentru şirul de intrare aaba utilizând algoritmul de simulare din figura 1.10, modificat conform propunerii anterioare.

În fig. 1.13 se prezintă corespondenţa mulţimilor de stări şi tipare pe măsură ce se prelucrează caracterele din intrare.

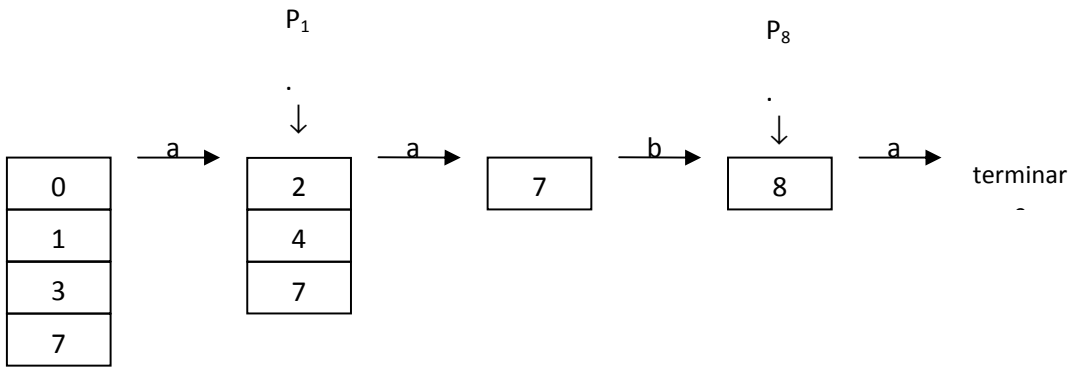


Fig. 1.13 Secvenţa de mulţimi de stării la prelucrarea intrării aaba

Se observă că mulţimea iniţială de stări este {0, 1, 3, 7}. Stările 1, 3 şi 7 au fiecare o tranziţie la a în stările 2, 4 şi respectiv 7. Deoarece starea 2 este stare acceptoare pentru primul tipar, după ce s-a citit primul a se înregistrează faptul că primul tipar poate fi recunoscut.

Deoarece există o tranziţie din starea 7 în starea 7, la al doilea caracter din intrare şi o alta din starea 7 în starea 8 la al treilea caracter din intrare, b, se continuă efectuarea tranziţiilor. Starea 8 este stare acceptoare pentru al treilea tipar. Din starea 8 nu mai sunt posibile tranziţii la următorul caracter din intrare, a, astfel că s-a ajuns la „terminare". Deoarece ultima corespondenţă a apărut după ce s-a citit al treilea caracter din intrare, se va recunoaşte al treilea tipar, corespunzător cu lexema aab.

Rolul lui **acţiune$_i$**, asociat cu tiparul p_i în specificarea lex este următorul: când se recunoaşte un exemplar al lui p_i, ANLEX execută programul asociat, **acţiune$_i$**. Trebuie remarcat, totuşi, faptul că **acţiune$_i$** nu va fi executat automat la intrarea AFN într-o stare care include starea acceptoare pentru p_i, ci numai atunci cand p_i se dovedeşte a fi tiparul care produce cea mai lungă corespondenţă.

1.8 Proiectarea analizoarelor lexicale bazată pe automate de tip AFD

O altă abordare pentru construirea unui ANLEX dintr-o specificare Lex este utilizarea, pentru a realiza corespondenţa de tipare, a unui AFD. Situaţia este complet analoagă cu simularea modificată a AFN din paragraful precedent. La conversia unui AFN în AFD, într-o submulţime dată de stări nedeterministe, pot exista mai multe stări acceptoare. Într-o astfel de situaţie, are prioritate starea acceptoare corespunzătoare tiparului listat primul in specificarea Lex. Ca şi în cazul simulării AFN, singura modificare ce trebuie realizată este efectuarea în continuare a tranziţiilor de stare până se ajunge într-o stare care, pentru simbolul curent de

intrare, nu mai are stare următoare. Lexema căutată este cea corespunzătoare ultimei poziţii din intrare pentru care AFD a intrat într-o stare acceptoare.

Exemplu: Prin conversia AFN din fig. 1.12 în AFD se obţine tabela de tranziţii din fig. 1.14. Stările AFD au fost numite prin liste de stări ale AFN. Ultima coloană indică unul din tiparele recunoscute la intrarea în acea starea a AFD. De exemplu, între stările AFN `2, 4` şi `7`, numai `2` este acceptoare şi anume este starea acceptoare a automatului pentru ER **`a`**. Deci starea AFD `247` recunoaşte tiparul **`a`**.

STARE	SIMBOL INTRARE		Tipar anunţat
	a	**b**	
0137	247	8	nimic
247	7	58	a
8	-	8	$a*b^+$
7	7	8	nimic
58	-	68	$a*b^+$
68	-	8	abb

Fig. 1.14 Tabela de tranziţii pentru AFD

Se observă că şirul **`abb`** corespunde cu 2 tipare, `abb` si $a*b^+$, recunoscute în stările AFN `6` şi respectiv `8`. Din acest motiv, starea `68` a AFD din ultima linie a tabelei de tranziţii, include `2` stări acceptoare ale AFN. Deoarece, în regulile de traducere pentru specificarea Lex, **`abb`** apare inainte de $\mathbf{a*b^+}$, în starea AFD `68` se va recunoaşte tiparul **`abb`**.

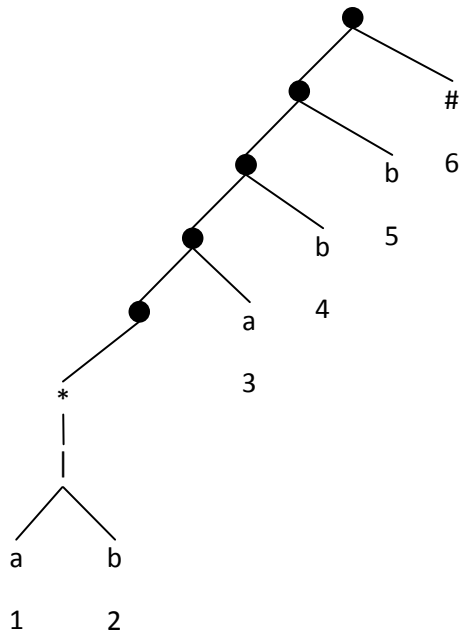
Pentru şirul de intrare `aaba`, AFD intră în stările sugerate de simularea AFN prezentată în fig. 1.13. Pentru un al doilea exemplu (şirul de intrare **`aba`**), AFD din fig. 1.14 porneşte din starea `0137`. La intrarea `a` trece în starea `247`, apoi, pentru `b`, trece în `58`, iar la intrarea `a`, nu are stare următoare. Astfel se ajunge la terminare, trecînd prin stările AFD `0137, 247` şi `58`. Ultima dintre acestea include starea AFN `8`, care este acceptoare. Deci, în starea `58`, AFD anunţă că s-a recunoscut tiparul $a*b^+$ şi selectează ca lexemă prefixul **`ab`** al intrării, care a condus la starea `58`.

1.9 Stări importante ale unui AFN

O stare a unui AFN este **importantă** dacă are o tranziţie de ieşire diferită de ε. De exemplu, la conversia unui AFN în AFD, se utilizează numai stările importante din submulţimea `T` când se detenmină ε_`închidere` ($f_t(T,a)$) (mulţimea de stări accesibile din `T`, la intrarea `a`). Mulţimea $f_t(s,a)$ este nevidă numai dacă starea **s** este importantă. Pe parcursul construcţiei, două submulţimi pot fi identificate (se pot confunda) dacă au aceleaşi stări importante şi dacă, fie amândouă includ, fie nici una nu include stări acceptoare ale AFN.

Când construcţia submulţimilor este aplicată unui AFN obţinut din ER prin *algoritmul lui Thompson*, se pot exploata proprietaţile speciale ale AFN. În acest algoritm se construieşte o stare importantă atunci când în ER apare un simbol din alfabet. De exemplu, pentru expresia `(a|b)*abb` se construiesc stări importante pentru fiecare `a` şi pentru fiecare `b`.

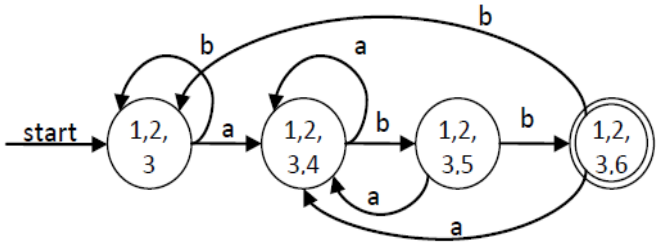
În plus, AFN rezultat are exact o stare acceptoare care însă nu este importantă pentru că nu are tranziţii care să o părăsească . Această unică stare acceptoare a automatului se poate transforma în stare importantă prin concatenarea expresiei regulate, la dreapta, cu simbolul **#** şi prevăzând o tranziţie de la starea acceptoare la cea corespunzătoare **#** -lui. Expresia **E#** senumeşte **augmentată.** În acest fel se pot neglija stările neimportante în timpul construcţiei. Când construcţia este gata, oricare stare a AFD cu o tranziţie la **#** trebuie să fie stare acceptoare.



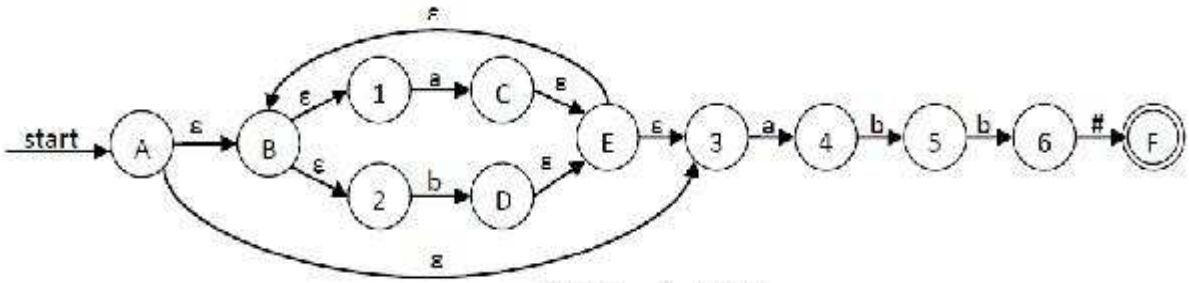
(a|b)* a b b

1 2 3 4 5

a) Arborele de sintaxă pentru (a|b)*abb#



b) AFD rezultat



c) AFN subinteles

Fig. 1.15 AFD şi AFN construite din (a|b)*abb pe baza stărilor importante

O *ER augmentată* se reprezintă printr-un arbore de sintaxă cu simbolurile de bază ca frunze şi operatorii ca noduri interioare. Un nod interior se va numi **nod-cat, nod-sau** sau **nod-stea** după cum este etichetat de un operator de concatenare, de | sau, respectiv de *****. În fig. 1.15(a) se prezintă un astfel de arbore de sintaxă cu *noduri-cat* marcate prin puncte.

Frunzele din arborele sintactic pentru ER sunt etichetate prin simboluri din alfabet sau prin **ε**. Fiecărei frunze neetichetate prin **ε** îi ataşăm un număr întreg unic care va reprezenta poziţia frunzei precum şi poziţia simbolului în expresie. Un simbol repetat va avea mai multe astfel de poziţii.

Stările numerotate în AFN din fig. 1.15(c) corespund poziţiei frunzelor în arborele de sintaxă din fig. 1.15(a). Aceste stări sunt stări importante ale AFN. Stările neimportante sunt desemnate prin litere mari. Aplicând transformarea **AFN => AFD** în aceste condiţii, rezultă AFD din fig. 1.15(b), cu o stare mai puţin decât anterior.

AFD din fig. 1.15(b) poate fi obţinut din AFN din fig. 1.15(c) construind submulţimile de stări şi identificând submulţimile ce coţin aceleaşi stări importante. Faţă de AFD obţinut anterior (§ 1.3) din acelaşi AFN, rezultă o stare mai puţin.

1.10 Construirea unui AFD echivalent cu o expresie regulată

1.10.1 Funcţii utilizate în procesul de construcţie

În acest paragraf se arată modul de construire a unui AFD direct dintr-o ER augmentată **(r)#.** Se începe prin a consrui un arbore sintactic T pentru (r)# şi apoi, prin traversări peste T, se calculează patru funcţii: **anulabil, primapoz, ultimapoz** şi **pozurm**. AFD se va construi din funcţia **pozurm**. Primele trei funcţii sunt definite pe nodurile arborelui sintactic şi se utilizează pentr a calcula pozurm, care este diferită pe mulţimea poziţiilor.

Pe baza echivalenţei între stările importante ale AFN şi poziţiile frunzelor din arborele de sintaxă al ER, se poate scurtcircuita construirea AFN, construind direct AFD ale cărui stări corespund mulţimilor de poziţii din arbore.

Tranziţiile **ε** conţin informaţii referitoare la situaţiile posibile ale simbolului, în sensul că fiecare simbol din şirul de intrare la AFD poate fi pus în corespondenţă cu anumite poziţii. Mai concret, un simbol de intrare c poate fi pus în corespondenţă numai cu poziţii la care există un c, dar nu orice poziţie având un c poate fi pusă în corespondenţă cu o anumită apariţie a lui c în şirul de intrare.

Noţiunea de poziţie pusă în corespondenţă cu un simbol de intrare va fi definită prin funcţia **pozurm**, pe poziţiile arborelui de sintaxă, şi anume: daca **i** este o poziţie, atunci **pozurm(i)** este mulţimea poziţiilor **j** pentru care există un şir de intrare de forma **...** cd **....** astfel încât **i** corespunde acestei apariţii a lui **c**, iar **j**, acestei apariţii a lui **d.**

Exemplu: În fig. 1.15(a), pozurm(1) = {1, 2, 3} pentru că, dacă apare la intrare un **a** corespunzător poziţiei 1 atunci, în continuare, poate apare din nou un **a** corespunzător următoarei aplicări a operatorului ***** sau, poate apare un **b** (din acelaşi motiv) sau un **a** corespunzător începutului şirului **abb.**

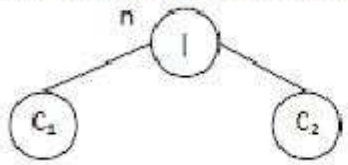
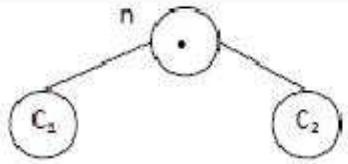
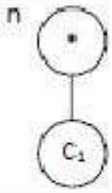
Pentru a calcula funcţia **pozurm**, trebuie să se cunoască ce poziţii pot fi puse în corespondenţă cu primul sau cu ultimul simbol al unui şir generat de o subexpresie dată a unei ER. De exemplu , dacă subexpresia este de forma **r*,** atunci fiecare poziţie care poate fi

prima în r, va urma fiecărei poziţii care poate fi ultima în r. Pentru o subexpresie de forma **rs**, fiecare primă poziţie din s urmează fiecărei ultime poziţii din r.

La fiecare nod **n** al arborelui sintactic corespunzător ER, se defineşte o funcţie `primapoz(n)` care dă submulţimea poziţiilor corespunzătoare cu primul simbol al unui şir generat de subexpresia cu rădăcina în `n`. Analog, se defineşte o funcţie `ultimapoz(n)`, care furnizează mulţimea poziţiilor corespunzătoare ultimului simbol dintr-un astfel de şir. De exemplu, dacă n este rădăcina întregului arbore (fig. 1.15(a)), atunci `primapoz(n) = {1, 2, 3}` si `ultimapoz(n)={6}`. În continuare se prezintă algoritmii de calcul pentru aceste funcţii.

Pentru a calcula `primapoz` şi `ultimapoz` trebuie să se cunoască nodurile care sunt rădăcini ale unor subexpresii generând limbaje ce includ şirul vid. Astfel de noduri se numesc *anulabile* şi vor fi precizate cu ajutorul funcţiei logice `anulabil(n)` care este *adevărată* dacă nodul este în această categorie şi *falsă* în caz contrar.

În tab. 1.3 se prezintă regulile de calcul pentru funcţiile **`anulabil`** şi **`primapoz`**. Există o regulă de bază referitoare la expresiile formate dintr-un simbol de bază şi trei reguli indictive care permit să se determine valorile funcţiilor parcurgând arborele sintactic de la frunze.

Nod n	anulabil(n)	primapoz(n)
n este o frunză cu eticheta ε	TRUE	Φ
n este o frunză cu eticheta cu poziţia i	FALSE	{i}
	anulabil(c_1) OR anulabil(c_2)	primapoz(c_1) U primapoz(c_2)
	anulabil(c_1) AND anulabil(c_2)	IF anulabil(c_1) THEN primapoz(c_1) U primapoz(c_2) ELSE primapoz(c_1)
	TRUE	primapoz(c_1)

Tab. 1.3 Regulile pentru calculul funcţiilor `anulabil` şi `primapoz`

Regulile pentru **`ultimapoz`** sunt aceleaşi ca şi cele pentru `primapoz`, dar cu C_1 şi C_2 inversate.

Ultima regulă pentru **`anulabil`** arată că dacă `n` este un *nod-stea* cu fiul C_1, atunci `anulabil(n)` este *true*, deoarece închiderea unei expresii generează un limbaj care-l include, cu certitudine pe **ε**. Ca un alt exemplu, a patra regulă pentru `primapoz` arată faptul că, dacă într-o expresie `rs`, `r` generează ε (`anulabil(`c_1`)` este `true`), atunci primele poziţii ale lui `s` „se văd prin” `r`, fiind, de asemenea, prime poziţii şi pentru `rs`; în caz contrar numai primele poziţii ale lui `r (primapoz(`c_1`))` sunt şi primepoziţii pentru `rs`. Celelalte

reguli pentru **anulabil** şi **primapoz**, ca şi cele pentru **ultimapoz**, se utilizează în mod similar.

Funcţia **pozurm(i)** arată ce poziţii pot urma poziţiei **i** în arborele de sintaxă. Toate modalităţile în care o poziţie o poate urma pe alta se definesc cu următoarele două reguli:

- Dacă **n** este un *nod-cat* cu fiul stâng C_1 şi fiul drept C_2 iar **i** este o poziţie în **ultimapoz(C_1),** atunci toate poziţiile din **primapoz(C_2)** se includ în **pozurm(i).**
- Dacă n este un *nod-stea* iar i este o poziţie in **ultimapoz(n),** atunci toate poziţiile din **primapoz(n)** se includ în **pozurm(i).**

Dacă s-au calculat primapoz şi ultimapoz pentru fiecare nod dintr-un arbore; funcţia pozurm pentru fiecare poziţie se poate calcula făcând o traversare spre adâncime a arborelui sintactic.

Exemplu: În fig. 1.16 se arată valorile funcţiilor **primapoz** şi **ultimapoz** în toate nodurile arborelui din fig. 1.15(a); primapoz(n) este scrisă în stânga nodului, iar ultimapoz(n), în dreapta. De exemplu, primapoz de la frunza din extremitatea stângă etichetată cu a este {1} deoarece această frunză este etichetată cu poziţia 1. Similar, primapoz la a doua frunză este {2}. Conform celei de a doua reguli de mai sus, primapoz pentru părintele acestor frunze este {1, 2}.

Nodul etichetat * este singurul nod „anulabil”. Deci, prin condiţia din **IF,** a celei de-a patra reguli, primapoz pentru părintele acestui nod (cel care reprezintă (a|b)* a), este reuniunea lui {1, 2} cu {3} care sunt primapoz a fiilor drept şi stâng. Pe de altă parte, condiţia **ELSE** se aplică pentru ultimapoz a acestui nod, deoarece frunza din poziţia 3 nu este “anulabilă”. Deci ultimapoz pentru parintele *nodului-stea* este doar {3}.

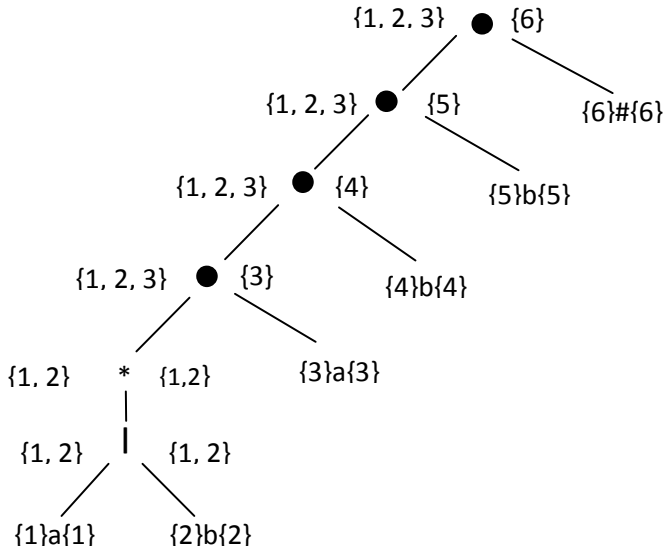


Fig. 1.16 **primapoz** şi **ultimapoz** pe nodurile din arborele de sintaxă pentru (a|b)*abb#

Funcţia **pozurm** se calculează de jos în sus, pentru fiecare nod al arborelui sintactic. La *nodul-stea* se adaugă 1 şi 2 atât la **pozurm(1)** cât şi la **pozurm(2),** pe baza regulii 2. La nodul părinte al *nodului-stea*, se adaugă 3 atât la **pozurm(1)** cât şi la **pozurm(2),** utilizând regula 1. La următoarele 2 *noduri-cat* se adaugă 5 la **pozurm(4)** şi respectiv 6 la **pozurm(5),** utilizând aceiaşi regulă. Construcţia completă a lui **pozurm** este prezentată în tab. 1.4.

Nod	pozurm
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-

Tab. 1.4 Funcţia `pozurm`

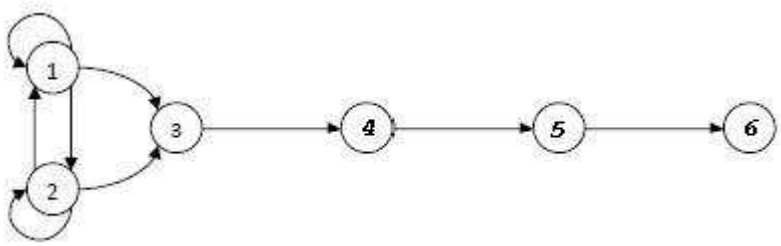


Fig. 1.17 Graf orientat ilustrând funcţia ***pozurm***

Funcţia **`pozurm`** se poate reprezenta ca un graf orientat având câte un nod pentru fiecare poziţie şi un arc orientat de la nodul **`i`** la nodul **`j`** dacă **`j`** este în **`pozurm(i)`** (fig. 1.17).

Este interesant de remarcat faptul că această diagramă ar deveni un AFN fără tranziţii **ε** pentru ER respectivă, dacă:

1. toate poziţiile din **`primapoz`** pentru rădăcină devin stări de start;
2. fiecare arc orientat `(i,j)` este etichetat prin simbolul din poziţia `j`;
3. poziţia asociată cu **`#`** devine singura stare acceptoare.

1.10.2 Algoritm pentru construcţia unui AFD dintr-o expresie regulată `r`

Intrare: O expresie regulată `r`.
Ieşire: Un AFD notat cu `D` care recunoaşte `L(r)`
Metoda:

1. Se construieşte arborele de sintaxă `T`, pentru expresia regulată augmentată`(r)#.`
2. Se construiesc funcţiile **`anulabil`**, **`primapoz`**, **`ultimapoz`** şi **`pozurm`** prin traversări în adâncime ale arborelui `T`.
3. Se construieşte $\mathbf{D_{stări}}$ (mulţimea de stări ale lui `D`) şi D_{tranz} (tabela de tranziţii pentru `D`) conform procedurii din fig. 1.18.

Stările din $D_{stări}$ sunt mulţimi de poziţii, fiecare fiind iniţial nemarcată. O stare devine marcată la tratarea tranziţiilor sale de ieşire. Starea de start a lui `D` este **`primapoz`**`(răd)`, iar stările acceptoare sunt toate cele care conţin poziţii asociate cu marcajul de sfârşit, **#**.

```
* iniţial, singura stare nemarcată în Dstări este primapoz(răd), unde
      răd este rădăcina arborelui sintactic pentru(r)#;
while * există stare nemarcată T în Dstări do begin
   * marchează T;
   for * orice simbol de intrare a do begin
      * fie U mulţimea poziţiilor care sunt în pozurm(p) pentru
      orice poziţie p∈T, astfel că simbolul din poziţia p este a;
      if * (U ≠ ϕ) and (U ∉ Dstări) then
         *adaugă U ca stare nemarcată la Dstări;
      Dtranz[T,a]:=U
   end
end
```

Fig. 1.18 Construcţia unui AFD

Exemplu: Să se construiască un AFD pentru expresia **(a|b)*abb**. Arborele de sintaxă pentru **((a|b)*abb)#** este prezentat în fig. 1.15(a). Funcţia **anulabil** este adevărată numai pentru nodul etichetat *****. Funcţiile **primapoz** şi **ultimapoz** sunt prezentate în fig. 1.16 iar **pozurm**, în tab. 1.4.

Din fig. 1.16 => **primapoz**(răd)={1, 2, 3}. Se notează acestă mulţime cu A şi se introduce ca primă stare nemarcată în $D_{stări}$.

Se consideră simbolul de intrare a. Poziţiile corespunzătoare lui a din A sunt 1 si 3; => Starea B= pozurm(1) U pozurm(3) = {1, 2, 3, 4} care nu este în $D_{stări}$ şi, ca atare, se adaugă. De asemenea, Dtranz[A, a]=B.

Se consideră simbolul de intrare b. Dintre poziţiile din A, numai poziţia 2 este asociată cu b. => pozurm(2)={1, 2, 3}=A care este în $D_{stări}$. Dtranz[A, b]=A.

Se continuă apoi cu B={1, 2, 3, 4}. Stările şi tranziţiile care se obţin în final sunt cele din figura 1.15(b).

1.11 Minimizarea numărului de stări ale unui AFD

Se poate determina teoretic faptul că fiecare mulţime regulată este recunoscută de un AFD cu un număr minim de stări, unic până la numele stărilor. În continuare se va arăta cum se poate construi acest AFD minimal, reducând numărul de stări dintr-un AFD dat, la un minimum posibil, fără a afecta limbajul necunoscut. Se presupune că se dă un AFD notat D, care are mulţimea de stări S, alfabetul de intrare A şi include tranziţii de ieşire din fiecare stare pentru toate simbolurile din alfabet. Dacă această ultimă condiţie nu este îndeplinită, se introduce o stare suplimentară notată **m**, numită **stare moartă**, prevăzută cu tranziţii de la **m** la **m** pentru oricare simbol de intrare şi adăugând tranziţii de la **s** la **m** pentru simbolul de intrare a dacă nu există tranziţie de ieşire din **s** pentru acest simbol de intrare (a∈S).

Se spune că un şir **w distinge** starea s de starea t dacă, pornind AFD din starea s şi furnizându-i la intrare şirul **w** se ajunge într-o stare acceptoare iar când se porneşte din t, pentru acelaşi şir de intrare, se ajunge într-o stare neacceptoare, *sau invers*. De exemplu, **ε** distinge oricare stare acceptoare de orice stare neaceptoare iar în AFD din fig. 1.8, stările A şi B sunt distinse de şirul de intrare **bb**, deoarece, pentru şirul **bb**, A conduce la starea neacceptoare C iar din B, se merge în starea acceptoare E.

Funcţionarea algoritmului de minimizare se bazează pe găsirea tuturor grupelor de stări care pot fi distinse de un anumit şir de intrare. Fiecare grup, format din stări care nu pot

fi distinse de nici un şir de intrare, va fi reprezentat în automatul minimizat printr-o singură stare. Algoritmul lucrează prin rafinarea unei partiţii a mulţimii de stări. Fiecare grup de stări din partiţie constă din stări care încă nu au fost distinse una de alta, iar oricare pereche de stări alese din grupuri diferite, au fost distinse de un anumit şir de intrare avut în vedere anterior.

Iniţial, partiţia constă din două grupuri: stările acceptoare (F) şi starile neacceptoare (S-F). Pasul fundamental al algoritmului constă în considerarea unui grup oarecare de stări, de exemplu $A=\{s_1, s_2, \ldots, s_k\}$ şi a unui simbol de intrare a şi verificarea tranziţiilor din stările $s_1, s_2, \ldots, s_k$ pentru intrarea a. Dacă aceste tranziţii se efectuează la stări care cad în două sau mai multe grupe diferite ale partiţiei curente, atunci A trebuie divizată astfel încât tranziţiile din submulţimile obţinute, pentru simbolul de intrare a, să se limiteze la un singur grup al partiţiei curente. *Exemplu*: s_1 şi s_2 merg, pentru simbolul de intrare a, la t_1 şi t_2 care sunt în grupe diferite ale partiţiei curente => A trebuie divizat în cel puţin 2 sumulţimi astfel încât o submulţime va conţine pe s_1, iar cealaltă pe s_2. Se remarcă faptul că t1 şi t2 au fost distinse anterior, de un şir notat de exemplu **w**, iar s_1 şi s_2 sunt distinse de şirul **aw**.

Acest proces de divizare a grupurilor din partiţia curentă se repetă până când nici un grup nu mai trebuie împărţit (nu se mai pot crea grupuri noi). Se poate arăta că stările care nu sunt divizate în grupuri diferite prin acest procedeu, nu vor fi distinse, cu certitudine, de orice şir de intrare. În acest moment s-a ajuns la partiţia finală, fiecare grup de stări din această partiţie reprezentând câte o stare din automatul minimizat. Se renunţă, de asemenea, la *starea moartă* şi la stările care nu pot fi atinse din starea de start. Se poate arăta că **AFD obţinut prin acest procedeu, acceptă acelaşi limbaj de intrare şi este minim din punct de vedere al numărului de stări.**

Algoritmul pentru minimizarea numărului de stări a unui AFD:

Intrare: Un AFD notat D, cu mulţimea de stări S, alfabetul de intrare A, cu tranziţii definite pentru toate stările şi toate elementele alfabetului, cu starea de start s_0 şi mulţimea de stări acceptoare F.

Ieşire: Un AFD notat D_{min}, care acceptă acelaşi limbaj ca şi D şi are un numar minim posibil de stări.

Metoda:

1) Se construieşte o partiţie iniţială $\prod$ a mulţimii de stări, cu 2 grupuri: stările acceptoare F şi stările neacceptoare S-F.
2) Din partiţia $\prod$, pe baza procedurii din fig. 1.19, se construieşte o nouă partiţie, $\prod_{nou}$.
3) **Dacă** $\prod_{nou}=\prod$, se consideră $\prod_{final}:=\prod$ şi se trece la pasul 4; **altfel** , se consideră $\prod:=\prod_{nou}$ şi se repetă pasul 2.
4) Se alege câte o stare din fiecare grup al partiţiei $\prod_{final}$ ca reprezentativă pentru acel grup. Aceste stări reprezentative vor fi stările AFD redus, D_{min}. Fie s o altfel de stare. Presupunem ca există o tranziţie în D, pentru intrarea a, de la s la t. Fie r reprezentantă grupului t (în particular r poate fi chiar t). În aceste condiţii, D_{min} va avea o tranziţie de la s la r pentru simbolul de intrare a. Se alege ca stare de start pentru D_{min} reprezentanta grupului ce conţine starea de start s_0 a lui D, iar stările acceptoare ale lui D_{min} sunt acelea care conţin stări din F. De remarcat faptul că, datorită modului în care s-a alcătuit partiţia iniţială, fiecare grup din $\prod_{final}$ constă fie numai din stări din F, fie nu au nici o stare din F.
5) Dacă D_{min} are o *stare moartă*, adică o stare **m** care nu este acceptoare şi are tranziţii de ieşire spre ea însăşi pentru toate simbolurile de intrare, atunci se elimină **m** din D_{min}. De asemenea se îndepărtează toate stările care nu pot fi atinse din starea de start. Toate tranziţiile spre **m**, din alte stări, devin nedefinite.

```
for * oricare grup G din ∏ begin
   * partiţionată G în subbrupe astfel încât 2 stări s şi t din G
     sunt în acelaşi subgrup doar dacă, pentru oricare simbol de
     intrare a, stările s şi t au traziţii de ieşire etichetate a
     spre stări din acelaşi grup al lui ∏;
   * înlocuieşte G, în ∏nou, prin mulţimea subgrupelor obţinute
end
```

Fig. 1.19 Construirea lui $\prod_{nou}$

Exemplu: Se condideră AFD din fig. 1.8. Partiţia iniţială ∏ constă din două grupe: starea acceptoare, (E) şi stările neacceptoare (A,B,C,D). Rezultă ∏={(E),(A,B,C,D)}. Pentru a construi $\prod_{nou}$ conform algoritmului din fig. 1.19, se consideră mai întâi (E). Deoarece acest grup constă dintr-o singură stare, nu mai poate fi partiţionat => (E) este inclus în $\prod_{nou}$. Algoritmul consideră apoi grupul (A,B,C,D). Pentru simbolul de intrare a, fiecare dintre aceste stări are o tranziţie la B => ele ar putea rămâne toate în acelaşi grup. Pentru simbolul de intrare b în schimb, A, B şi C merg la membri ai grupului (A,B,C,D) din ∏ iar D merge la E, membru al altui grup => în $\prod_{nou}$, grupul (A,B,C,D) trebuie divizat în 2 noi grupuri (A,B,C) şi (D) => $\prod_{nou}$={(E),(A,B,C),(D)}

La următoarea aplicare a procedurii din fig. 1.19, în continuare, simbolul de intrare a nu provoacă noi divizări. Deoarece, pentru simbolul de intrare b, A şi C au tranziţii la C iar B are tranziţie la D, membru al altui grup din partiţie, (A,B,C) trebuie divizat în 2 noi grupuri (A,C) şi (B) => următorul $\prod_{nou}$={(E),(A,C),(B),(D)}

În continuare, nici unul din grupurile care constau dintr-o singură stare nu mai poate fi divizat. Singura posibilitate privind o nouă partiţionare ar fi divizarea grupului (A,C). Dar atât A cât şi C merg în aceeaşi stare B pentru simbolul de intrare a şi merg la aceeaşi stare C pentru simbolul b => $\prod_{nou}=\prod=\prod_{final}$.

Pentru grupul (A,C) se alege ca reprezentant pe A, iar B, D şi E vor reprezenta grupurile singulare respective. Se obţie automatul redus a cărui tabelă de tranziţii este prezentată în fig 1.20, iar AFD corespunzător este in fig. 1.21.

Stare	**Simbol intrare**	
	a	**b**
A	B	A
B	B	D
D	B	E
E	B	A

Fig. 1.20 Tabela de tranziţii pentru AFD redus

Starea A este starea de start iar starea E este singura stare acceptoare. Nu exista o *stare moartă* iar toate stările pot fi atinse din starea de start A.

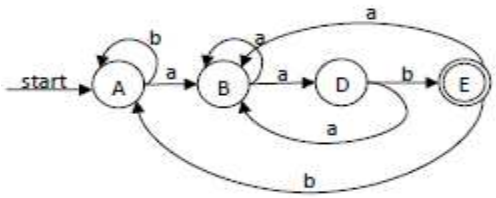


Fig. 1.21 AFD redus