

Capitolul 1

Automate finite.

Rolul lor în modelarea activităților din analiza lexicală[1]

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

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

Atât AFD cât și AFN pot recunoaște 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]:

$(a|b)^*abb$ – mulțimea tuturor șirurilor compuse din caractere a și b care se termină cu secvența abb (ex: abb, aabb, babb, aaabb, ...)

1.1 Definiția unui AFN

Un AFN este un model matematic reprezentat prin următorul cvintet:

AFN = $\langle S, A, f_t, s_o, F \rangle$, unde:

- **S** este **mulțimea stărilor**;
- **A** reprezintă mulțimea simbolurilor de intrare (**alfabetul** de intrare);
- **f_t** se numește **funcție de tranziție** și pune în corespondență perechi „stare-simbol” cu ”stări”, adică: $f_t : S \times A \rightarrow S$;
- **s_o** este **starea inițială** (de start); $s_o \in S$;
- **F** este **mulțimea stărilor finale** (acceptoare); $F \subseteq S$.

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

- același caracter se poate utiliza pentru a eticheta două sau mai multe tranziții din aceeași stare;
- este permisă utilizarea ca etichetă a simbolului ε (șirul 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 tranziții: se poate rămâne în starea 0 sau se poate trece în starea 1.

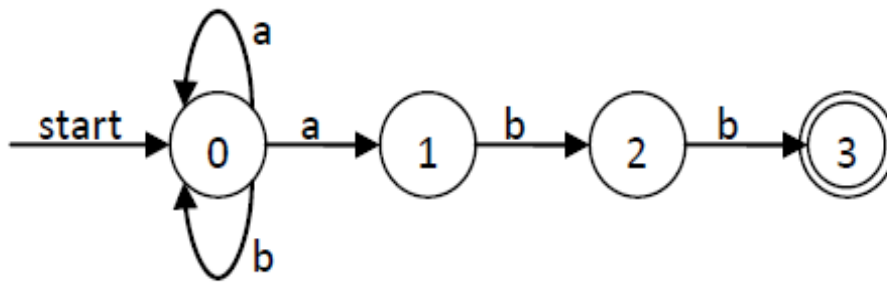


Fig. 1.1 Un prim exemplu de AFN pentru expresia: $(a|b)^*abb$

În calculator, funcția de tranziție se poate implementa în mai multe moduri. O modalitate potrivită este sub forma unei **tabele de tranziții** care au câte o linie pentru fiecare stare, câte o coloană pentru fiecare simbol de intrare (inclusiv pentru simbolul ε , dacă este necesar) (fig. 1.2). Intrarea în tabelă corespunzătoare liniei i și simbolului de intrare a este mulțimea de stări pentru care există tranziții 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 tranziții corespunzătoare AFN din fig. 1.1

Se spune că un AFN *acceptă* un șir de intrare "X" dacă și numai dacă există un drum în graful de tranziții începând din starea de start până la o stare acceptoare astfel încât etichetele pe drumul respectiv să formeze șirul "X". Drumul respectiv se numește **cale de acceptare** pentru șirul "X". Aceluiași șir de intrare îi pot corespunde mai multe căi de acceptare într-un AFN. Totalitatea șirurilor acceptate de un AFN reprezintă **limbajul definit de automatul respectiv**.

1.2 Definiția unui AFD

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

- 1) din nici o stare nu pleacă tranziții etichetate cu ε ;
- 2) pentru orice stare s și 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 tranziție din fiecare stare pentru orice simbol de intrare. Aceasta înseamnă ca AFD conține cel mult o cale de acceptare (recunoaștere) pentru un șir de intrare dat.

În fig. 1.3 se prezintă un algoritm pentru simularea (parcurea) unui AFD, având starea inițială s_0 , mulțimea stărilor acceptoare F și funcția de transfer f_t . Algoritmul se aplică asupra unui șir de intrare "X", terminat prin caracterul special **eof** și generează la ieșire

raspunsul "DA" în cazul în care șirul "X" este acceptat și "NU" în caz contrar. Funcția carurm furnizează următorul caracter din intrare iar procedura gen generează rezultatul final.

```

s:=s0;
c:= carurm;
while (c≠eof) and (∃ tranziție de ieșire 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 funcția de transfer f_t este multiplu definită pentru anumite stări, simularea prin program a funcționării unui AFN este dificilă. Din acest motiv se preferă intercalarea unei etape intermediare de transformare a **AFN** într-un **AFD echivalent** (ambele recunosc același limbaj) urmând ca, în final, să se realizeze programul care simulează funcționarea AFD.

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

Se va contrui tabela de tranziții pentru D notată **Dtranz** și, implicit, mulțimea stărilor AFD, notată **Dstări**. Fiecare stare a lui D corespunde unei mulțimi de stări din N. "Dtranz" va fi astfel contruită încât va simula, în paralel, toate tranzițiile care se pot efectua în N pentru un șir de intrare dat. Evidența mulțimilor de stări din AFN se realizează prin intermediul operațiilor definite în tab. 1.1.

Operatia	Descriere
$\varepsilon_inchidere(s)$	Mulțimea stărilor AFN caere pot fi atinse din starea „s” utilizând numai tranziții ε .
$\varepsilon_inchidere(T)$	Mulțimea stărilor AFN care pot fi atinse numai prin tranziții ε din toate stările s componente ale mulțimii de stări T.
$f_t(T, a)$	Mulțimea stărilor AFN la care există o tranziție din oricare stare s a mulțimii T, pentru simbolul de intrare a.

Tab. 1.1. Operații asupra starilor AFN

Înainte de a vedea primul simbol de intrare, automatul N poate să fie în oricare din stările mulțimii $\varepsilon_inchidere(s_o)$. Să presupunem că pentru o secvență dată de simboluri de intrare, s-a ajuns la stările din mulțimea T. Dacă a este următorul simbol de intrare, N se poate deplasa, la citirea acestui sinbol, în oricare din stările mulțimii $f_t(T, a)$ și, implicit, la oricare din stările conținute în $\varepsilon_inchidere(f_t(T, a))$. Algoritmul de determinare a mulțimii "Dstări" și de construire a tabelului "Dtranz" este prezentat în fig. 1.4.

* se calculează $\varepsilon_inchidere(s_0)$ și se consideră această mulțime ca fiind prima stare, nemarcată, a mulțimii "Dstări";

```

while *  $\exists$  o stare nemarcată  $T \in Dstări$  do begin
    * marchează  $T$ ;
    for * orice simbol de intrare  $a$  do begin
         $U := \varepsilon\_inchidere(f_t(T, a))$ ;
        if *  $U \notin Dstări$  then
            * adaugă  $U$  la  $Dstări$ , ca nemarcată;
        end if;
         $Dtranz[T, a] := U$ ;
    end for;
end while;

```

Fig. 1.4 Determinarea mulțimii stărilor și a tabelii de tranziții corespunzătoare AFD echivalent cu un AFN dat

O stare din D este o stare acceptoare dacă ea reprezintă o mulțime de stări AFN de care aparține cel puțin o stare acceptoare a lui N . Pentru calculul lui $\varepsilon_inchidere(T)$ se poate aplica algoritmul schițat în fig. 1.5.

```

* introdu toate stările din  $T$  în stivă;
* inițializează  $\varepsilon\_inchidere(T)$  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  $\varepsilon$  do
        if *  $u \notin \varepsilon\_inchidere(T)$  then begin
            * adaugă  $u$  la  $\varepsilon\_inchidere(T)$ ;
            * introdu  $u$  în stivă;
        end if;
    end for;
end while;

```

Fig. 1.5 Algoritmul pentru calculul mulțimii $\varepsilon_inchidere(T)$

Exemplu: Se consideră AFN din figura 1.6

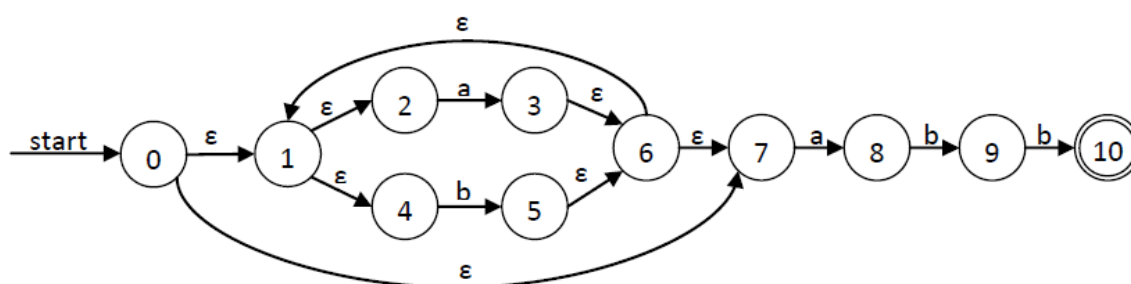


Fig. 1.6 AFN corespunzător expresiei $(a|b)^*abb$

Se aplică algoritmul din figura 1.4:

$\varepsilon_inchidere(0) = \{0, 1, 2, 4, 7\}$

$A = \{0, 1, 2, 4, 7\}$ – starea de start a AFD

$Dstări = \{A\}$ A, nemarcată

Ciclul 1

- marchează A => Dstări={A}

1) Pentru „A” și „a”:

$$f_t(A,a)=\{3,8\}$$

$U = \varepsilon_{\text{închidere}}(\{3,8\}) = \{1,2,3,4,6,7,8\}$ - nu există în Dstări

$$B = \{1,2,3,4,6,7,8\}$$

$$\text{Dstări} = \{\underline{A}, B\}$$

$$\text{Dtranz}[A,a]=B$$

2) Pentru „A” și „b”:

$$f_t(A,b) = \{5\}$$

$U = \varepsilon_{\text{închidere}}(\{5\}) = \{1,2,4,5,6,7\}$ - nu există în Dstări

$$C = \{1,2,4,5,6,7\}$$

$$\text{Dstări} = \{\underline{A}, B, C\}$$

$$\text{Dtranz}[A,b] = C$$

Ciclul 2

- marchează B => Dstări = {A, B, C}

1) Pentru „B” și „a”:

$$f_t(B,a)=\{3,8\} \Rightarrow \text{conduce la B, existentă în Dstări}$$

$$\text{Dtranz}[B,a]=B$$

2) Pentru „B” și „b”:

$$f_t(B,b)=\{5,9\}$$

$U = \varepsilon_{\text{închidere}}(\{5,9\})=\{1,2,4,5,6,7,9\}$ - nu exista în Dstări

$$D = \{1,2,4,5,6,7,9\}$$

$$\text{Dstări} = \{\underline{A}, \underline{B}, C, D\}$$

$$\text{Dtranz}[B,b]=D$$

Ciclul 3

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

1) Pentru „C” și „a”:

$$f_t(C,a) = \{3,8\} \Rightarrow \text{conduce la B, existentă în Dstări}$$

$$\text{Dtranz}[C,a]=B$$

2) Pentru „C” și „b”:

$$f_t(C,b)=\{5\} \Rightarrow \text{Conduce la C, existentă în Dstări}$$

$$\text{Dtranz}[C,b] = C$$

Ciclul 4

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

1) Pentru „D” și „a”:

$$f_t(D,a) = \{3,8\} \Rightarrow \text{conduce la B, existentă în Dstări}$$

$$\text{Dtranz}[D,a]=B$$

2) Pentru „D” și „b”:

$$f_t(D,b)=\{5,10\}$$

$U = \varepsilon_{\text{închidere}}(\{5,10\})=\{1,2,4,5,6,7,10\}$ - nu exista în Dstări

$$E = \{1,2,4,5,6,7,10\}$$

$$\text{Dstări} = \{\underline{A}, \underline{B}, \underline{C}, \underline{D}, E\}$$

$$\text{Dtranz}[D,b]=E$$

Ciclul 5

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

1) Pentru "E" și "a":

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

2) Pentru "E" și "b":

$f_t(E,b)=\{5\} \Rightarrow$ 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 tabelului de tranziții în 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 tranziții Dtranz pentru AFD echivalent cu AFN din fig 1.6

În Fig. 1.8 este prezentat AFD corespunzător tabelului Dtranz care este echivalent cu AFN inițial. Singura stare finală a AFN, starea 10 care aparține de starea E a AFD. Rezultă că E este singura stare finală a AFD.

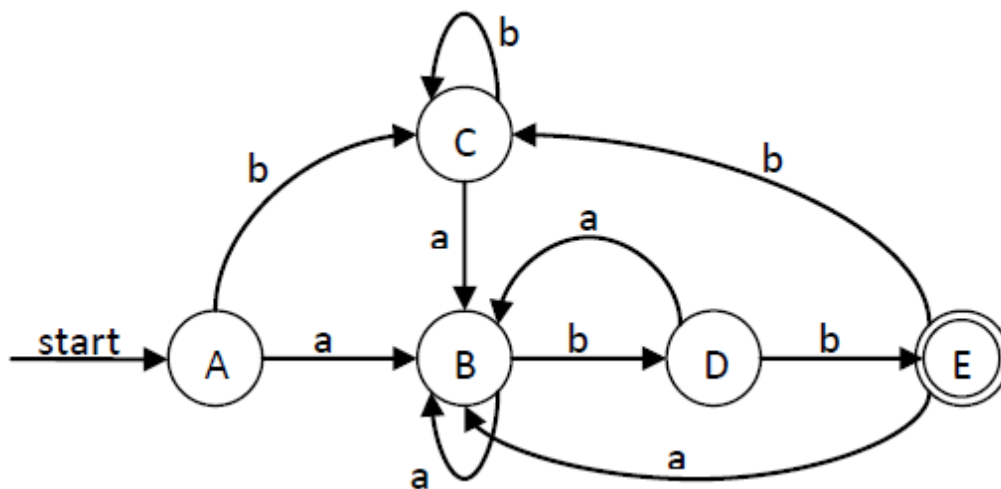


Fig. 1.8 AFD rezultat, echivalent cu AFN din fig. 1.6

1.4 Construirea unui AFN echivalent cu o expresie regulată

Există mai multe modalități de realizare a unui program de recunoaștere corespunzător unei expresii regulate. O primă modalitate, descrisă și analizată în acest paragraf, constă în construirea unui AFN echivalent cu expresia regulată urmată de simularea comportării AFN pentru un șir de intrare dat. Dacă viteza de execuție 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, fără a mai apela la construirea, în prealabil, a unui AFN echivalent.

Algoritmul prezentat în continuare urmărește cazurile parcurse la definiția unei expresii regulate. La început se construiesc AFN-uri care recunosc simbolul ϵ și orice simbol din alfabet. Pe baza acestor AFN elementare se realizează în continuare AFN-uri pentru expresii care conțin operații de reuniune (alternativă), de produs (concatenare) și de închidere Kleene (operația stea). Fiecare pas în construcție va avea un număr de stări cel mult egal cu dublul numărului de simboluri și de operatori din expresia regulată inițială.

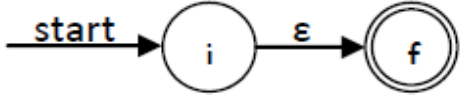
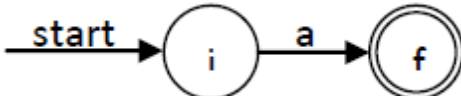
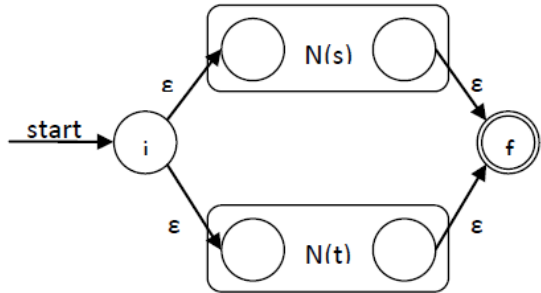
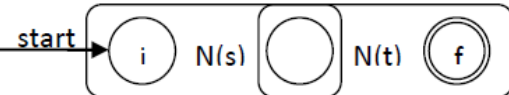
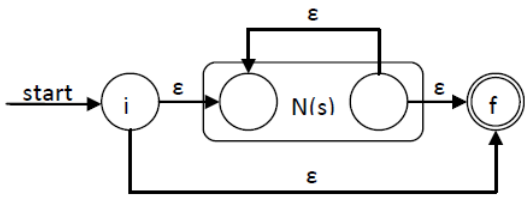
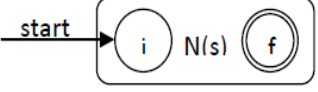
Regulile utilizate pentru construirea automatelor elementare, cât și cele corespunzătoare operațiilor de bază sunt prezentate sintetic în tab. 1.2. i și f reprezintă starea inițială respectiv starea finală a automatului construit. Notăția $N(r)$ desemnează automatul corespunzător expresiei regulate r .

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

- se descompune expresia dată, r , în subexpresiile constituente;
- se utilizează regulile 1 și 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 construcția 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 proprietăți:

- 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 și o stare finală (acceptoare); niciun arc nu intră în starea de start și nici un arc nu părăsește starea finală (starea finală nu are tranziții de ieșire). Această proprietate este valabilă atât pentru automatul final cât și pentru automatele componente.
- 3) Fiecare stare din $N(r)$ are cel mult o tranziție de ieșire etichetată cu un simbol din alfabet sau cel mult două tranziții de ieșire etichetate cu ϵ .

Regula nr:	Expresia regulată	AFN-rezultat (echivalent)	Observații
1	ϵ (simbolul vid)	 $N(\epsilon)$	
2	a (orice simbol din alfabet, $a \in A$)	 $N(a)$	Dacă același simbol apare de mai multe ori în expresia regulată, se construiește care un asemenea AFN, distinct, pentru fiecare apariție 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 inițială a lui $N(t)$.
5	s^* (închidere Kleene) limbajul acceptat $(L(s))^*$	 $N(s^*)$	„i” și „f” sunt stări noi
6	(s) Limbajul acceptat: $L(s)$	 $N((s))$	Același automat ca și 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.

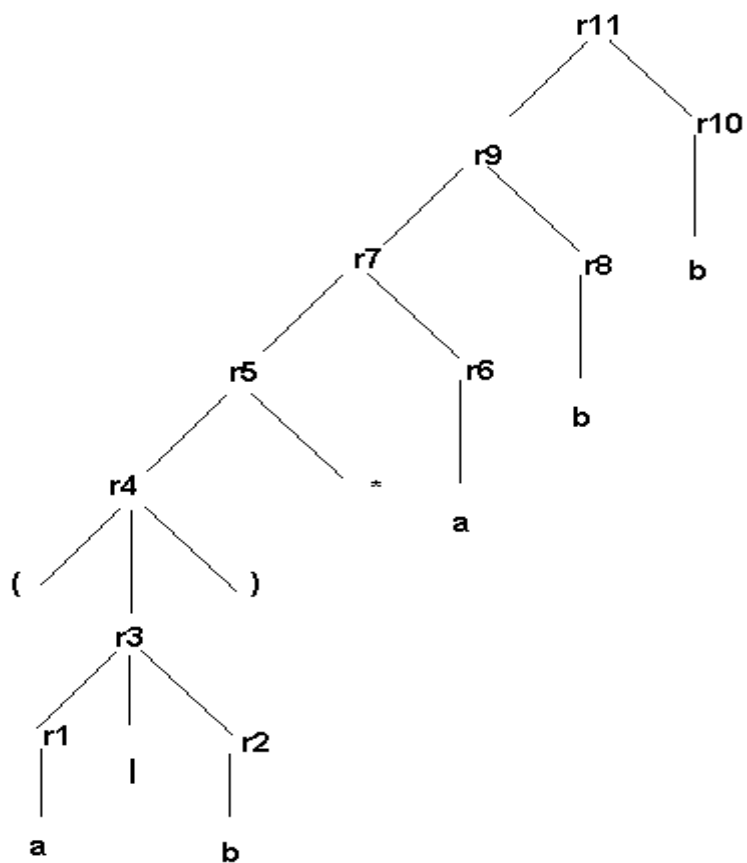


Fig 1.9 Arborele sintactic corespunzător expresiei $(a|b)^*abb$

În continuare, se contruiesc succesiv AFN corespunzător expresiilor constituate:
 r_1, r_2, \dots, r_{11} .

$r_1 = a$



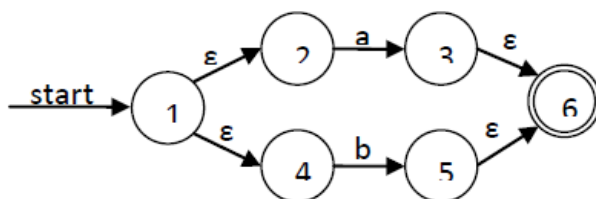
$N(r_1)$:

$r_2 = b$



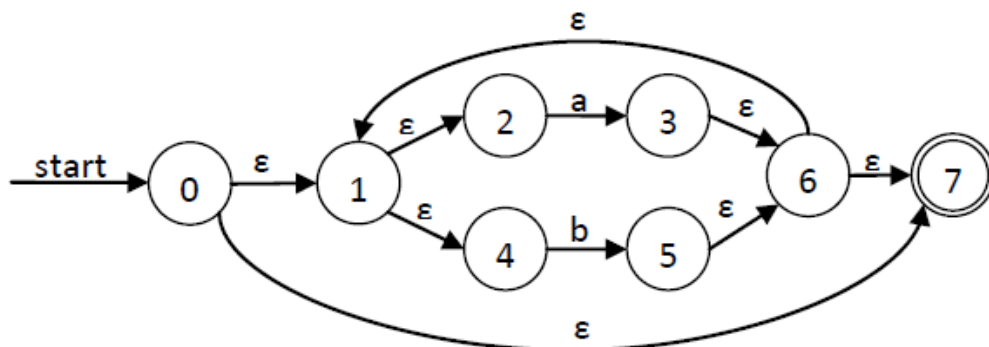
$N(r_2)$:

$r_3 = r_1 | r_2$



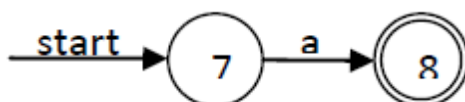
N(r3):
 $r4 = (r3)$
 $N(r4) \equiv N(r3)$

$r5 = r4^*$



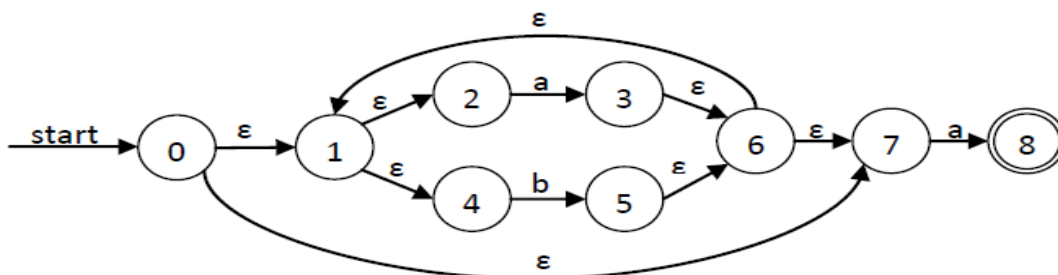
N(r5):

$r6 = a$



N(r6):

$r7 = r5r6$

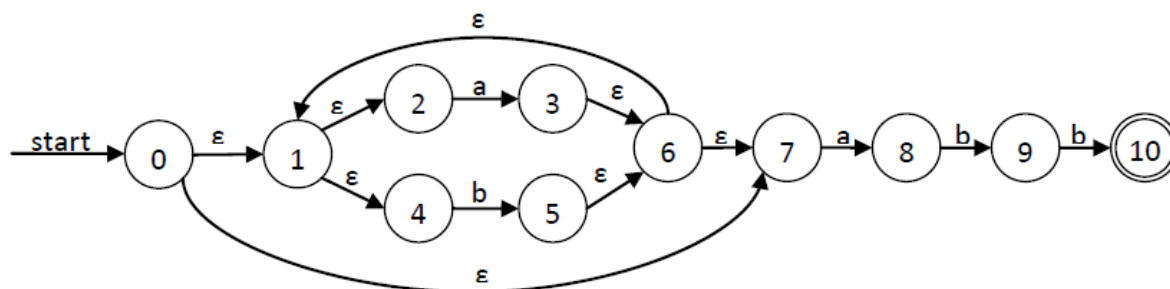


N(r7):

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

$r_{11} = r_9 r_{10} = (a|b)^* abb$

N(r11):



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

stări în comparație cu cele obținute prin alte metode. Acest dejavantaj este însă compensat de simplitatea și **naturalețea** mecanismului de construcție.

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 stabilește dacă N acceptă (recunoaște), sau nu, un șir de intrare dat, X . Pentru a măări eficiența acestui calcul, se utilizează proprietățile suplimentare ale unui AFN construit pentru o expresie regulată conform algoritmului anterior. Starea inițială a automatului este notată cu s_0 iar mulțimea stărilor finale este F . Se consideră de asemenea, că șirul de intrare se termină cu caracterul special **eof**. Algoritmul este prezentat în fig. 1.10.

```

S:=  $\varepsilon\_inchidere(s_0)$ ;
a:=carurm;
while (a $\neq$ eof) and ( $\exists$ tranziție de ieșire din  $s$  pt.  $c$ ) do
    begin
        s:=  $\varepsilon\_inchidere(f_t(S,a)$ ;
        a:= carurm;
    end;
if ( $S \cap F \neq \Phi$ ) 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 algoritm sunt două stive și un șir de cifre binare, indexat de stările lui N . Într-o stivă se ține evidența mulțimii curente a stărilor nedeterminate iar a doua, se utilizează pentru calculul mulțimii de stări următoare. Pentru calculul mulțimii $\varepsilon_inchidere$, 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. După calculul complet al stării următoare se poate schimba rolul stivelor. Deoarece fiecare stare din N are cel mult două tranziții de ieșire, rezultă că fiecare stare produce, după efectuarea unei tranziții, cel mult două stări noi. Notăm cu $|N|$ numărul de stări din automat și cu $|X|$ numărul de simboluri din șirul X (lungimea sa). Timpul necesar pentru calculul mulțimii de stări următoare este proporțional cu $|N|$ iar timpul total de simulare va fi proporțional cu $|N| \times |X|$.

Exemplu:

Se consideră AFN din fig 1.6, șirul X fiind format dintr-un singur caracter, a .

$S = \varepsilon_inchidere(0) = \{0, 1, 2, 4, 7\}$
 $f_t(S, a) = \{3, 8\} \Rightarrow S = \varepsilon_inchidere(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$

$F = \{10\} \Rightarrow S \cap F = \Phi \Rightarrow$ algoritmul generează („nu”).

1.6 Considerații de eficiență a algoritmilor prezentați

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

- 1) Aplicând algoritmul prezentat în § 1.4, se construiește AFN corespunzător lui r . Notând cu $|r|$ lungimea expresiei r , timpul necesar pentru construirea unui AFN este de ordinul lui $|r|$ și se notează $\theta(|r|)$. Tabela de tranziții pentru N poate fi înregistrată într-un spațiu de memorie, deasemenea proportional cu $|r|$. Utilizând algoritmul din fig. 1.10, se poate stabili dacă N acceptă șirul X , într-un timp $\theta(|r| \times |X|)$, cu un consum de spațiu de memorie $\theta(|r|)$.
- 2) AFN rezultat pentru o expresie regulată r se transformă într-un AFD echivalent, utilizând algoritmul din § 1.3. Implementând funcția de tranziție printr-o tabelă de tranziție, se poate determina dacă șirul de intrare X este recunoscut de automat, aplicând algoritmul din fig.1.3, într-un timp $\theta(|X|)$ – proporțional cu lungimea șirului și independent de numărul de stări din AFD. Rezultă că procedeul este foarte rapid și trebuie aplicat atunci când timpul de execuție este critic. Consumul de spațiu de memorie este $\theta(2^{|r|})$ și în anumite situații particulare, poate să fie foarte mare. Pentru exemplificare, se consideră expresia regulată:

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

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

O variantă a metodei 2), bazându-se tot pe AFD, constă în construirea parțială a tabelului de tranziții, utilizând tehnica numită „evaluarea leneșă a tranziției”. În acest caz tranzițiile sunt calculate în timpul execuției, analog cu simularea AFN dar o tranziție de stare pentru un anumit caracter de intrare nu se calculează decât atunci când este absolut necesar. Tranzițiile calculate se înregistrează într-o zonă care funcționează ca o **memorie cache**: de fiecare dată când urmează să se realizeze o tranziție, se consultă mai întâi memoria cache. Dacă tranziția nu este găsită, ea se calculează și se înregistrează. În momentul în care memoria cache se umple, se șterge o tranziție calculată anterior. În felul acesta, cerințele de spațiu sunt proporționale cu lungimea expresiei regulate, la care se adaugă dimensiunea memoriei cache, fiind de același ordin cu cele de la AFN. Performanțele de viteză sunt apropiate de cele ale AFD (nu se strică în mod spectaculos) pentru că, în majoritatea cazurilor, tranziția necesară va fi găsită în memoria cache.

1.7 Proiectarea analizatoarelor 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 parțiale se combină într-un automat unic, în care se introduce o unică stare de start, cu tranziții ε spre fiecare din cele n automate parțiale (fig. 1.11).

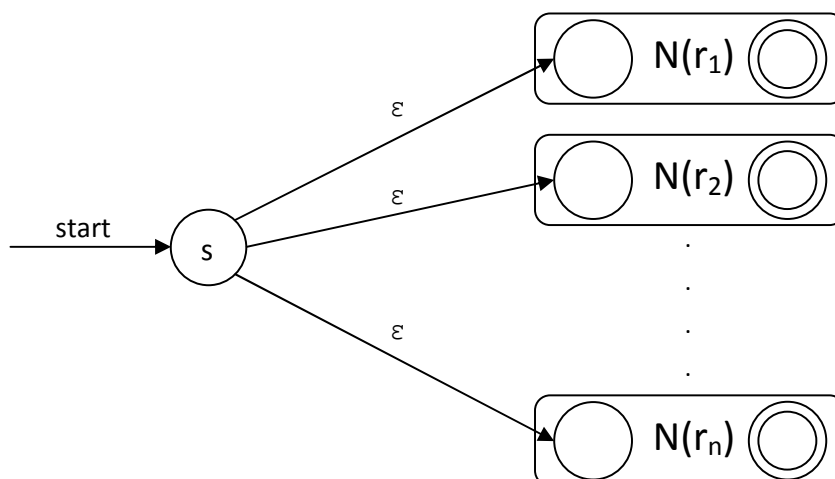


Fig. 1.11 Structura AFN combinat

Pentru a simula comportarea AFN combinat, algoritmul din fig. 1.10, elaborat pentru simularea unui singur AFN, se modifică astfel încât să se asigure recunoașterea celui mai lung prefix din fișierul de intrare care corespunde unui tipar. Pentru aceasta, atunci când se găsește o mulțime de stări ce include o stare acceptoare, se continuă simularea până se ajunge la “terminare”, adică la o mulțime de stări din care nu mai există tranziții de ieșire corespunzătoare simbolului de intrare curent. Dacă se adaugă o stare acceptoare la mulțimea curentă de stări, se înregistrează poziția curentă de intrare și tiparul x_i corespunzător acestei stări acceptoare (fiecare AFN parțial are o singură stare acceptoare). În cazul în care mulțimea curentă de stări conține o stare acceptoare, se păstrează ultima stare acceptoare întâlnită. La realizarea condiției de terminare, pointerul de anticipare este retras la ultima poziție de stare acceptoare înregistrată. Tiparul cu care s-a făcut corespondența pentru acea stare, identifică atomul găsit iar lexema este constituită de șirul 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 corespondență cu intrarea (eventual cel de eroare).

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

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

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

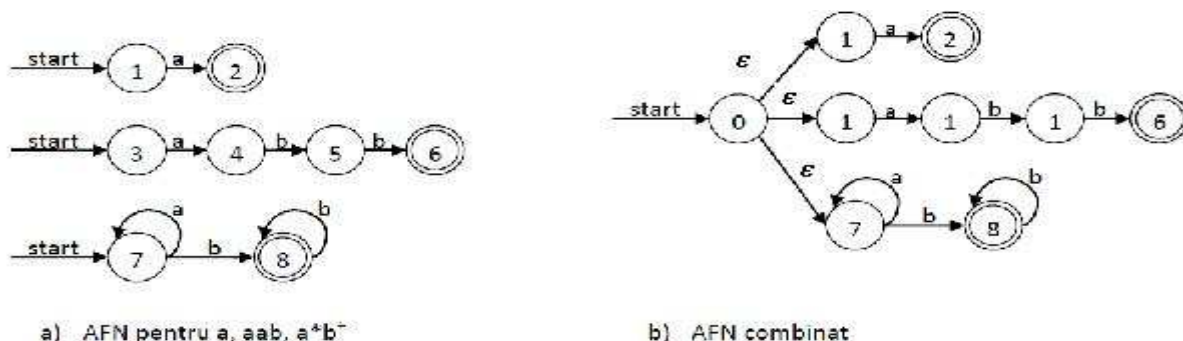


Fig. 1.12 AFN care recunoaște cele trei tipare diferite

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

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

În fig. 1.13 se prezintă corespondența mulțimilor de stări și tipare pe măsură ce se prelucerează caracterele din intrare.

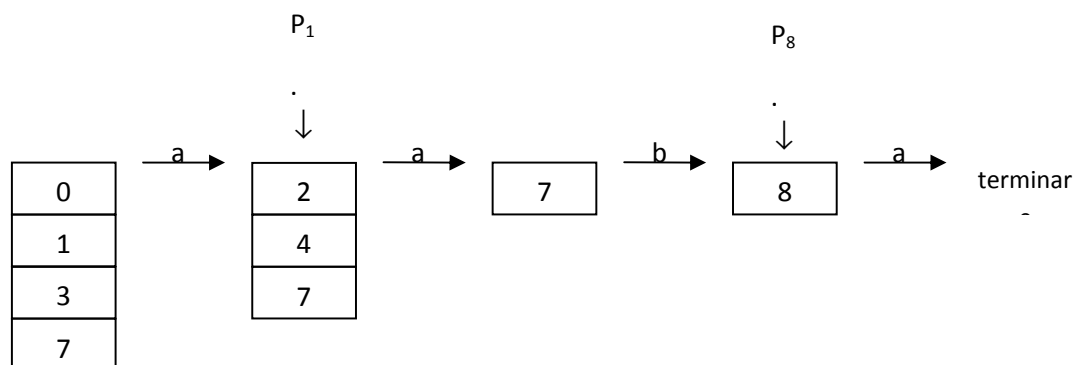


Fig. 1.13 Secvența de mulțimi de stări la prelucrarea intrării aaba

Se observă că mulțimea inițială de stări este $\{0, 1, 3, 7\}$. Stările 1, 3 și 7 au fiecare o tranziție la a în stările 2, 4 și 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 tranziție din starea 7 în starea 7, la al doilea caracter din intrare și o alta din starea 7 în starea 8 la al treilea caracter din intrare, b, se continuă efectuarea tranzițiilor. Starea 8 este stare acceptoare pentru al treilea tipar. Din starea 8 nu mai sunt posibile tranziții la următorul caracter din intrare, a, astfel că s-a ajuns la „terminare”. Deoarece ultima corespondență a apărut după ce s-a citit al treilea caracter din intrare, se va recunoaște al treilea tipar, corespunzător cu lexema aab.

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

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 corespondența de tipare, a unui AFD. Situația este complet analoagă cu simularea modificată a AFN din paragraful precedent. La conversia unui AFN în AFD, într-o submulțime dată de stări nedeterminate, pot exista mai multe stări acceptoare. Într-o astfel de situație, are prioritate starea acceptoare corespunzătoare tiparului listat primul în specificarea Lex. Ca și în cazul simulării AFN, singura modificare ce trebuie realizată este efectuarea în continuare a tranzițiilor 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 poziții din intrare pentru care AFD a intrat într-o stare acceptoare.

Exemplu: Prin conversia AFN din fig. 1.12 în AFD se obține tabela de tranziții 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 și 7, numai 2 este acceptoare și anume este starea acceptoare a automatului pentru ER **a**. Deci starea AFD 247 recunoaște tiparul **a**.

STARE	SIMBOL INTRARE		Tipar anunțat
	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 tranziții pentru AFD

Se observă că șirul **abb** corespunde cu 2 tipare, **abb** și $a*b^+$, recunoscute în stările AFN 6 și respectiv 8. Din acest motiv, starea 68 a AFD din ultima linie a tabelului de tranziții, include 2 stări acceptoare ale AFN. Deoarece, în regulile de traducere pentru specificarea Lex, **abb** apare înainte de $a*b^+$, în starea AFD 68 se va recunoaște tiparul **abb**.

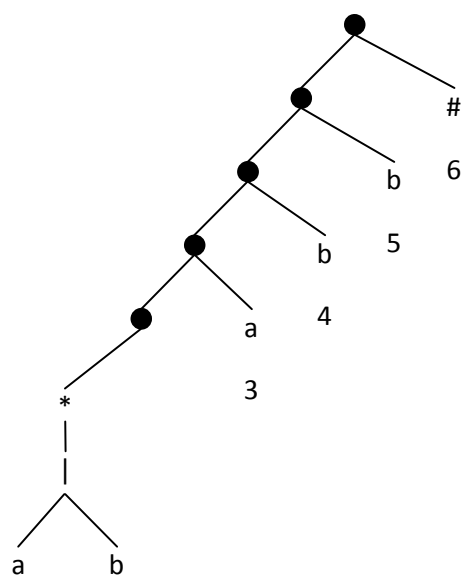
Pentru șirul de intrare aaba, AFD intră în stările sugerate de simularea AFN prezentată în fig. 1.13. Pentru un al doilea exemplu (șirul de intrare **aba**), AFD din fig. 1.14 pornește din starea 0137. La intrarea a trece în starea 247, apoi, pentru b, trece în 58, iar la intrarea a, nu are stare următoare. Astfel se ajunge la terminare, trecând prin stările AFD 0137, 247 și 58. Ultima dintre acestea include starea AFN 8, care este acceptoare. Deci, în starea 58, AFD anunță că s-a recunoscut tiparul $a*b^+$ și 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 tranziție de ieșire diferită de ε . De exemplu, la conversia unui AFN în AFD, se utilizează numai stările importante din submulțimea T când se determină $\varepsilon_închidere(f_t(T, a))$ (mulțimea de stări accesibile din T , la intrarea a). Mulțimea $f_t(s, a)$ este nevidă numai dacă starea s este importantă. Pe parcursul construcției, două submulțimi pot fi identificate (se pot confunda) dacă au aceleași stări importante și dacă, fie amândouă includ, fie nici una nu include stări acceptoare ale AFN.

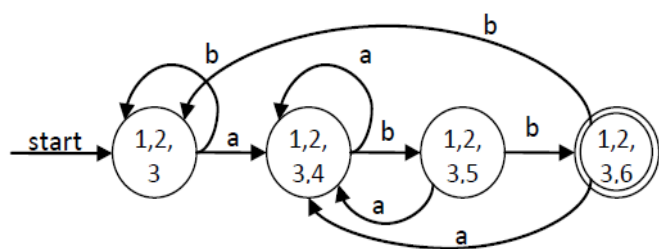
Când construcția submulțimilor este aplicată unui AFN obținut din ER prin *algoritmul lui Thompson*, se pot exploata proprietățile speciale ale AFN. În acest algoritm se construiește o stare importantă atunci când în ER apare un simbol din alfabet. De exemplu, pentru expresia $(a|b)*abb$ se construiesc stări importante pentru fiecare a și pentru fiecare b .

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

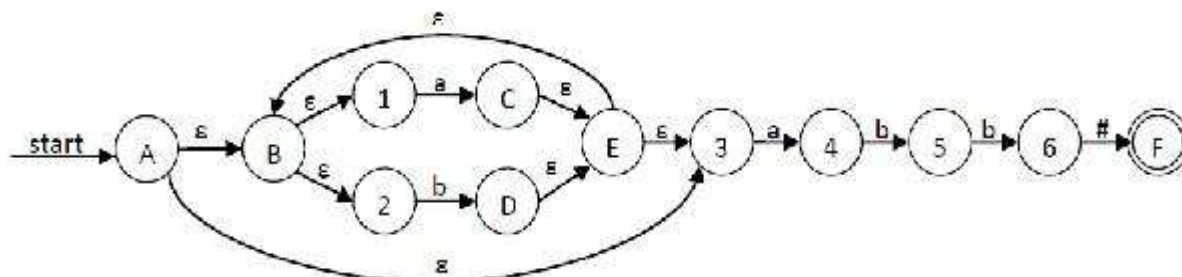


$(a|b)^* a b b$
1 2 3 4 5

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



b) AFD rezultat



c) AFN subintelect

Fig. 1.15 AFD și 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 și 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 atașăm un număr întreg unic care va reprezenta poziția frunzei precum și poziția simbolului în expresie. Un simbol repetat va avea mai multe astfel de poziții.

Stările numerotate în AFN din fig. 1.15(c) corespund poziției 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** \Rightarrow **AFD** în aceste condiții, rezultă AFD din fig. 1.15(b), cu o stare mai puțin decât anterior.

AFD din fig. 1.15(b) poate fi obținut din AFN din fig. 1.15(c) construind submulțimile de stări și identificând submulțimile ce conțin aceleași stări importante. Față de AFD obținut anterior (§ 1.3) din același AFN, rezultă o stare mai puțin.

1.10 Construirea unui AFD echivalent cu o expresie regulată

1.10.1 Funcții utilizate în procesul de construcție

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

Pe baza echivalenței între stările importante ale AFN și pozițiile frunzelor din arborele de sintaxă al ER, se poate scurtcircuita construirea AFN, construind direct AFD ale cărui stări corespund mulțimilor de poziții din arbore.

Tranzițiile ϵ conțin informații referitoare la situațiile posibile ale simbolului, în sensul că fiecare simbol din șirul de intrare la AFD poate fi pus în corespondență cu anumite poziții. Mai concret, un simbol de intrare \underline{c} poate fi pus în corespondență numai cu poziții la care există un \underline{c} , dar nu orice poziție având un \underline{c} poate fi pusă în corespondență cu o anumită apariție a lui \underline{c} în șirul de intrare.

Noțiunea de poziție pusă în corespondență cu un simbol de intrare va fi definită prin funcția **pozurm**, pe pozițiile arborelui de sintaxă, și anume: dacă **i** este o poziție, atunci **pozurm(i)** este mulțimea pozițiilor **j** pentru care există un șir de intrare de forma $\dots c d \dots$ astfel încât **i** corespunde acestei apariții a lui **c**, iar **j**, acestei apariții a lui **d**.

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

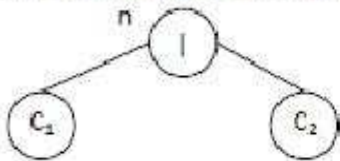
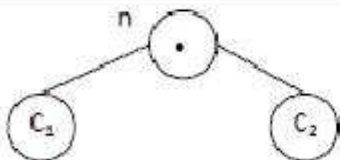
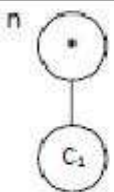
Pentru a calcula funcția **pozurm**, trebuie să se cunoască ce poziții pot fi puse în corespondență cu primul sau cu ultimul simbol al unui șir generat de o subexpresie dată a unei ER. De exemplu, dacă subexpresia este de forma r^* , atunci fiecare poziție care poate fi

prima în \underline{r} , va urma fiecărei poziții care poate fi ultima în \underline{r} . Pentru o subexpresie de forma rs , fiecare primă poziție din \underline{s} urmează fiecărei ultime poziții din \underline{r} .

La fiecare nod n al arborelui sintactic corespunzător ER, se definește o funcție $\text{primapoz}(n)$ care dă submulțimea pozițiilor corespunzătoare cu primul simbol al unui șir generat de subexpresia cu rădăcina în n . Analog, se definește o funcție $\text{ultimapoz}(n)$, care furnizează mulțimea pozițiilor corespunzătoare ultimului simbol dintr-un astfel de șir. De exemplu, dacă n este rădăcina întregului arbore (fig. 1.15(a)), atunci $\text{primapoz}(n) = \{1, 2, 3\}$ și $\text{ultimapoz}(n) = \{6\}$. În continuare se prezintă algoritmi de calcul pentru aceste funcții.

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

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

Nod n	$\text{anulabil}(n)$	$\text{primapoz}(n)$
n este o frunză cu eticheta ε	TRUE	Φ
n este o frunză cu eticheta cu poziția i	FALSE	$\{i\}$
	$\text{anulabil}(c_1) \text{ OR } \text{anulabil}(c_2)$	$\text{primapoz}(c_1) \cup \text{primapoz}(c_2)$
	$\text{anulabil}(c_1) \text{ AND } \text{anulabil}(c_2)$	IF $\text{anulabil}(c_1)$ THEN $\text{primapoz}(c_1) \cup \text{primapoz}(c_2)$ ELSE $\text{primapoz}(c_1)$
	TRUE	$\text{primapoz}(c_1)$

Tab. 1.3 Regulile pentru calculul funcțiilor **anulabil** și **primapoz**

Regulile pentru **ultimapoz** sunt aceleași ca și cele pentru **primapoz**, dar cu C_1 și C_2 inversate.

Ultima regulă pentru **anulabil** arată că dacă n este un *nod-stea* cu fiul C_1 , atunci $\text{anulabil}(n)$ 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 ($\text{anulabil}(c_1)$ este *true*), atunci primele poziții ale lui s „se văd prin” r , fiind, de asemenea, prime poziții și pentru rs ; în caz contrar numai primele poziții ale lui r ($\text{primapoz}(c_1)$) sunt și primepoziții pentru rs . Celelalte

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

Funcția **pozurm(i)** arată ce poziții pot urma poziției **i** în arborele de sintaxă. Toate modalitățile în care o poziție o poate urma pe alta se definesc cu următoarele două reguli:

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

Dacă s-au calculat **primapoz** și **ultimapoz** pentru fiecare nod dintr-un arbore; funcția **pozurm** pentru fiecare poziție se poate calcula făcând o traversare spre adâncime a arborelui sintactic.

Exemplu: În fig. 1.16 se arată valorile funcțiilor **primapoz** și **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 poziția 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 condiția 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 și stâng. Pe de altă parte, condiția **ELSE** se aplică pentru **ultimapoz** a acestui nod, deoarece frunza din poziția 3 nu este “anulabilă”. Deci **ultimapoz** pentru parintele *nodului-stea* este doar {3}.

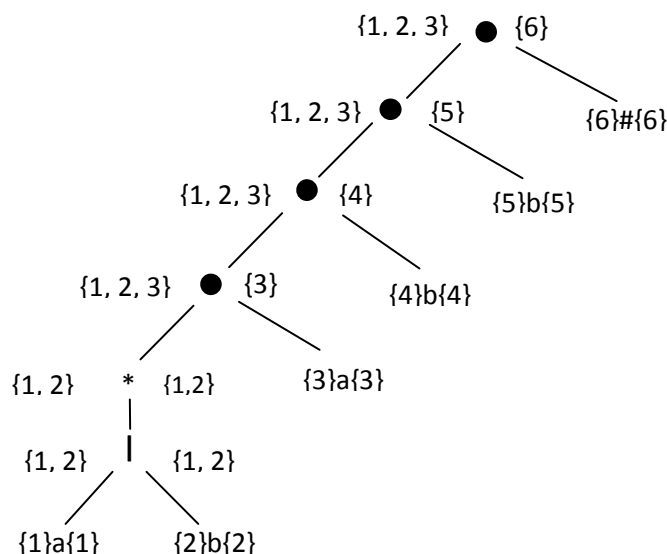


Fig. 1.16 **primapoz** și **ultimapoz** pe nodurile din arborele de sintaxă pentru $(a|b)^*abb\#$

Funcția **pozurm** se calculează de jos în sus, pentru fiecare nod al arborelui sintactic. La *nodul-stea* se adaugă 1 și 2 atât la **pozurm(1)** cât și la **pozurm(2)**, pe baza regulii 2. La nodul părinte al *nodului-stea*, se adaugă 3 atât la **pozurm(1)** cât și la **pozurm(2)**, utilizând regula 1. La următoarele 2 *noduri-cat* se adaugă 5 la **pozurm(4)** și respectiv 6 la **pozurm(5)**, utilizând aceeași regulă. Construcția 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 Funcția **pozurm**

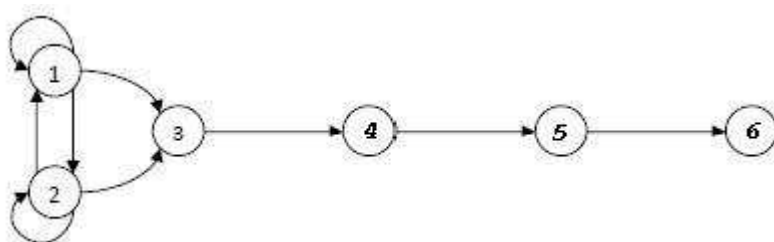


Fig. 1.17 Graf orientat ilustrând funcția **pozurm**

Funcția **pozurm** se poate reprezenta ca un graf orientat având câte un nod pentru fiecare poziție și 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 tranziții ϵ pentru ER respectivă, dacă:

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

1.10.2 Algoritm pentru construcția unui AFD dintr-o expresie regulată r

Intrare: O expresie regulată r .

Ieșire: Un AFD notat cu D care recunoaște $L(r)$

Metoda:

1. Se construiește arborele de sintaxă T , pentru expresia regulată augmentată $(r)\#$.
2. Se construiesc funcțiile **anulabil**, **primapoz**, **ultimapoz** și **pozurm** prin traversări în adâncime ale arborelui T .
3. Se construiește $D_{stări}$ (mulțimea de stări ale lui D) și D_{tranz} (tabela de tranziții pentru D) conform procedurii din fig. 1.18.

Stările din $D_{stări}$ sunt mulțimi de poziții, fiecare fiind inițial nemarcată. O stare devine marcată la tratarea tranzițiilor sale de ieșire. Starea de start a lui D este **primapoz**($r_{\text{ăd}}$), iar stările acceptoare sunt toate cele care conțin poziții asociate cu marcajul de sfârșit, **#**.

```

* inițial, singura stare nemarcată în  $D_{stări}$  este  $primapoz(r\ddot{a}d)$ , unde
   $r\ddot{a}d$  este rădăcina arborelui sintactic pentru(r)#;
while * există stare nemarcată  $T$  în  $D_{stări}$  do begin
  * marchează  $T$ ;
  for * orice simbol de intrare  $a$  do begin
    * fie  $U$  mulțimea pozițiilor care sunt în  $pozurm(p)$  pentru
      orice poziție  $p \in T$ , astfel că simbolul din poziția  $p$  este  $a$ ;
    if *  $(U \neq \emptyset)$  and  $(U \notin D_{stări})$  then
      *adaugă  $U$  ca stare nemarcată la  $D_{stări}$ ;
       $Dtranz[T, a] := U$ 
    end
  end
end
end

```

Fig. 1.18 Construcția 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). Funcția **anulabil** este adevărată numai pentru nodul etichetat *. Funcțiile **primapoz** și **ultimapoz** sunt prezentate în fig. 1.16 iar **pozurm**, în tab. 1.4.

Din fig. 1.16 \Rightarrow **primapoz**($r\ddot{a}d$) = {1, 2, 3}. Se notează această mulțime cu A și se introduce ca primă stare nemarcată în $D_{stări}$.

Se consideră simbolul de intrare a . Pozițiile corespunzătoare lui a din A sunt 1 și 3; \Rightarrow Starea $B = \text{pozurm}(1) \cup \text{pozurm}(3) = \{1, 2, 3, 4\}$ care nu este în $D_{stări}$ și, ca atare, se adaugă. De asemenea, $Dtranz[A, a] = B$.

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

Se continuă apoi cu $B = \{1, 2, 3, 4\}$. Stările și tranzițiile care se obțin 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 mulțime 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 mulțimea de stări S , alfabetul de intrare A și include tranziții de ieșire din fiecare stare pentru toate simbolurile din alfabet. Dacă această ultimă condiție nu este îndeplinită, se introduce o stare suplimentară notată **m**, numită **stare moartă**, prevăzută cu tranziții de la **m** la **m** pentru oricare simbol de intrare și adăugând tranziții de la **s** la **m** pentru simbolul de intrare a dacă nu există tranziție de ieșire din **s** pentru acest simbol de intrare ($a \in S$).

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

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

fi distinse de nici un șir de intrare, va fi reprezentat în automatul minimizat printr-o singură stare. Algoritmul lucrează prin rafinarea unei partiții a mulțimii de stări. Fiecare grup de stări din partiție 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 șir de intrare avut în vedere anterior.

Inițial, partiția constă din două grupuri: stările acceptoare (F) și stările neacceptoare (S-F). Pasul fundamental al algoritmului constă în considerarea unui grup oarecare de stări, de exemplu $A = \{s_1, s_2, \dots, s_k\}$ și a unui simbol de intrare a și verificarea tranzițiilor din stările s_1, s_2, \dots, s_k pentru intrarea a . Dacă aceste tranziții se efectuează la stări care cad în două sau mai multe grupe diferite ale partiției curente, atunci A trebuie divizată astfel încât tranzițiile din submulțimile obținute, pentru simbolul de intrare a , să se limiteze la un singur grup al partiției curente. *Exemplu:* s_1 și s_2 merg, pentru simbolul de intrare a , la t_1 și t_2 care sunt în grupe diferite ale partiției curente $\Rightarrow A$ trebuie divizat în cel puțin 2 submulțimi astfel încât o submulțime va conține pe s_1 , iar cealaltă pe s_2 . Se remarcă faptul că t_1 și t_2 au fost distinse anterior, de un șir notat de exemplu w , iar s_1 și s_2 sunt distinse de șirul aw .

Acest proces de divizare a grupurilor din partiția curentă se repetă până când nici un grup nu mai trebuie împărțit (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 șir de intrare. În acest moment s-a ajuns la partiția finală, fiecare grup de stări din această partiție reprezentând câte o stare din automatul minimizat. Se renunță, de asemenea, la *starea moartă* și la stările care nu pot fi atinse din starea de start. Se poate arăta că **AFD obținut prin acest procedeu, acceptă același limbaj de intrare și 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 mulțimea de stări S , alfabetul de intrare A , cu tranziții definite pentru toate stările și toate elementele alfabetului, cu starea de start s_0 și mulțimea de stări acceptoare F .

Ieșire: Un AFD notat D_{min} , care acceptă același limbaj ca și D și are un număr minim posibil de stări.

Metoda:

- 1) Se construiește o partiție inițială Π a mulțimii de stări, cu 2 grupuri: stările acceptoare F și stările neacceptoare $S-F$.
- 2) Din partiția Π , pe baza procedurii din fig. 1.19, se construiește o nouă partiție, Π_{nou} .
- 3) **Dacă** $\Pi_{nou} = \Pi$, se consideră $\Pi_{final} := \Pi$ și se trece la pasul 4; **altfel**, se consideră $\Pi := \Pi_{nou}$ și se repetă pasul 2.
- 4) Se alege câte o stare din fiecare grup al partiției Π_{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 tranziție în D , pentru intrarea a , de la s la t . Fie r reprezentantă grupului t (în particular r poate fi chiar t). În aceste condiții, D_{min} va avea o tranziție de la s la r pentru simbolul de intrare a . Se alege ca stare de start pentru D_{min} reprezentanta grupului ce conține starea de start s_0 a lui D , iar stările acceptoare ale lui D_{min} sunt acelea care conțin stări din F . De remarcat faptul că, datorită modului în care s-a alcătuit partiția inițială, fiecare grup din Π_{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 și are tranziții de ieșire spre ea însăși 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 tranzițiile spre m , din alte stări, devin nedefinite.

```

for * oricare grup G din  $\Pi$  begin
  * partiționată G în subgrupe astfel încât 2 stări s și t din G
    sunt în același subgrup doar dacă, pentru oricare simbol de
    intrare a, stările s și t au traziții de ieșire etichetate a
    spre stări din același grup al lui  $\Pi$ ;
  * înlocuiește G, în  $\Pi_{nou}$ , prin mulțimea subgrupelor obținute
end

```

Fig. 1.19 Construirea lui Π_{nou}

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

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 și C au tranziții la C iar B are tranziție la D, membru al altui grup din partiție, (A, B, C) trebuie divizat în 2 noi grupuri (A, C) și (B) \Rightarrow următorul $\Pi_{nou} = \{(E), (A, C), (B), (D)\}$

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

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

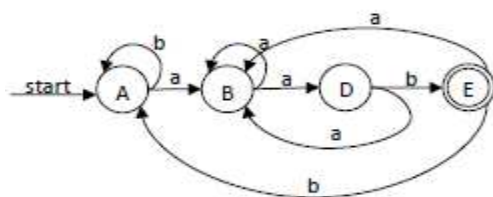


Fig. 1.21 AFD redus

2

ANALIZA SINTACTICĂ ASCENDENTĂ

Se va prezenta un tip general de analiză numit **cu deplasare și reducere (ASDR)**. Aceasta încearcă să construiască arborele sintactic pentru șirul de intrare începând de la frunze spre rădăcină (de jos în sus—ascendent). Procesul se poate privi ca reducerea unui șir de terminale, “w”, la simbolul de start al gramaticii. În fiecare pas al reducerii un subșir particular, corespunzător părții dreapta a unei producții este înlocuit cu simbolul din partea stângă a producției respective. Dacă subșirul este corect ales, la fiecare pas se va parcurge, în sens invers, o derivare dreapta.

Exemplu: Se consideră gramatica:

$$S \rightarrow aABe \qquad A \rightarrow Abc \mid b \qquad B \rightarrow d$$

Propoziția “abcde” poate fi redusă la S în următorii pași:

$$abcde \qquad aAbcde \qquad aAde \qquad aABe \qquad S$$

Înlocuirile s-au realizat căutând subșiruri care să corespundă părții dreapta a unei producții și înlocuind subșirurile respective cu partea stângă a acelei producții. Șirul de reduceri urmărește, în sens invers, următoarea derivare dreapta:

$$S \xRightarrow{d} aABe \xRightarrow{d} aAde \xRightarrow{d} aAbcde \xRightarrow{d} abcde$$

2.1 Noțiuni de bază

Un **capăt al unui șir** este un subșir care corespunde părții drepte a unei producții și a cărui reducere la neterminatul din partea stângă a producției reprezintă un pas în parcurgerea în sens invers a unei derivări dreapta. În multe cazuri, subșirul stâng β care corespunde părții drepte a unei producții $A \rightarrow \beta$ nu este un **capăt** deoarece o reducere prin producția $A \rightarrow \beta$ produce un șir care nu poate fi redus în continuare, până la simbolul de start al gramaticii.

Exemplu: înlocuirea lui b cu A în pasul 2 al exemplului de mai sus
 $aAbcde \rightarrow aAAcde$ (care nu poate fi redus la S)
 \Rightarrow Trebuie dată o definiție mai precisă pentru noțiunea de “capăt”.

Un **capăt** al unei forme propoziționale drepte γ constă, formal dintr-o **producție** $A \rightarrow \beta$ și o **poziție** în γ în care șirul β poate fi găsit și înlocuit prin A pentru a produce forma propozițională anterioară a unei derivări dreapta a lui γ .

Dacă $S \Rightarrow \alpha Aw \Rightarrow \alpha \beta w$, atunci $A \rightarrow \beta$ și poziția care urmează lui α , definește un capăt al lui $\alpha \beta w$. Subșirul w , situat în dreapta unui capăt, conține numai terminale.

În cazul în care gramatica este ambiguă, este posibil să existe mai multe derivări dreapta ale șirului $\alpha \beta w$ ca atare pot exista mai multe capete. Dacă gramatica este neambiguă, capătul corespunzător unei forme propoziționale este unic.

Grafic, un capăt β , al unei forme propoziționale $\alpha \beta w$, se poate reprezenta astfel:

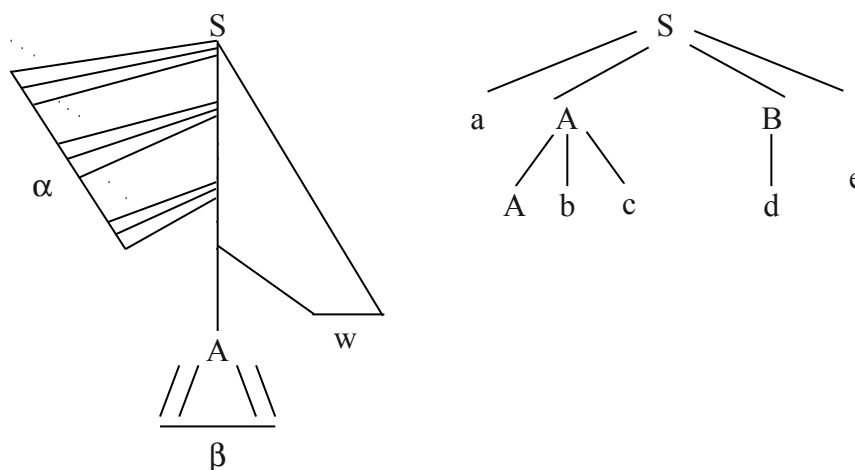


Fig.5.1 Capătul $A \rightarrow \beta$ în Arborele Sintactic pentru $\alpha \beta w$

Capătul reprezintă cel mai din stânga subarbor complete format dintr-un nod și toți fii săi. A este nodul interior cel mai de jos și cel mai din stânga având toți fiii prezenți în arbore. Reducerea lui β la A în $\alpha \beta w$, numită **fasonarea** capătului β , constă în îndepărtarea fiilor lui A din arborele sintactic.

Exemplu. Se consideră gramatica:

- (1) $E \rightarrow E + E$ (2) $E \rightarrow E * E$ (3) $E \rightarrow (E)$ (4) $E \rightarrow id$

și derivarea dreaptă:

$$E \Rightarrow_d E + E \Rightarrow_d E + E * E \Rightarrow_d E + E * id_3 \Rightarrow_d E + id_2 * id_3 \Rightarrow_d id_1 + id_2 * id_3$$

Sublinierile marchează capetele din fiecare formă propozițională. Se remarcă și aici faptul că șirurile care apar la dreapta unui capăt conțin numai simboluri terminale.

Gramatica dată fiind ambiguă, se poate obține același șir și prin următoarea derivare dreapta:

$$E \Rightarrow_d E * E \Rightarrow_d E * id_3 \Rightarrow_d E + E * id_3 \Rightarrow_d E + id_2 * id_3 \Rightarrow_d id_1 + id_2 * id_3$$

În forma propozițională $E+E*id_3$ sunt două capete, $E+E$ și id_3 , corespunzând primei și respectiv celei de-a doua derivări prezentate. Ca și în exemplul prezentat la gramatici ambigue, existența celor două derivări semnifică priorități relative diferite ale operațiilor de $*$ și $+$ (în primul caz, $*$ mai prioritar ca $+$).

Fasonarea capetelor

Procesul de parcurgere în sens invers a unei derivări dreapta se numește **fasonarea capetelor**. Se pornește de la un șir de terminale, notat cu w , care urmează să fie analizat. Dacă w aparține limbajului descris prin gramatica dată atunci el se poate obține după un pas oarecare n într-o derivare dreapta încă necunoscută:

$$S \Rightarrow_d \gamma_1 \Rightarrow_d \gamma_2 \Rightarrow_d \dots \Rightarrow_d \gamma_{n-1} \Rightarrow_d \gamma_n = w$$

Pentru a reconstitui această derivare în ordine inversă, se localizează capătul β_n în forma propoziției γ_n , după care se înlocuiește β_n cu partea stângă a unei producții $A_n \rightarrow \beta_n$, obținându-se forma propozițională dreapta γ_{n-1} . Apoi se continuă acest proces pentru γ_{n-1} , ș.a.m.d. Procesul se termină cu succes dacă, în final, se obține ca formă propozițională simbolul de start S .

Exemplu: se consideră gramatica din exemplul anterior și șirul de intrare $id_1+id_2*id_3$. Secvența de reduceri prezentată în figură este inversul secvenței din prima derivare a exemplului anterior (cu $*$ prioritar față de $+$).

Forma propozițională dreapta	Capăt	Producția pentru reducere
$id_1+id_2*id_3$	id_1	$E \rightarrow id$
$E+id_2*id_3$	id_2	$E \rightarrow id$
$E+E*id_3$	id_3	$E \rightarrow id$
$E+E*E$	$E*E$	$E \rightarrow E*E$
$E+E$	$E+E$	$E \rightarrow E+E$
E		

Fig. 5.2. Reduceri efectuate de un analizor sintactic cu deplasare și reducere

2.2 Implementarea cu stivă a analizei sintactice de tip deplasare – reducere (ASDR)

Pentru implementarea AS prin fasonarea capetelor, trebuie rezolvate două probleme:

- localizarea subșirului care urmează să fie redus, într-o formă propozițională dreaptă;
- alegerea producției care se aplică, dacă gramatica are mai multe producții cu aceeași parte dreaptă.

Ca structură de date de bază se poate utiliza pentru implementare o stivă în care se păstrează simbolurile gramaticale și un tampon de intrare care conține șirul de analizat, **w**. Baza stivei și extremitatea dreaptă a șirului de intrare va fi marcată prin simbolul \$. Inițial, stiva este goală (conține doar \$) iar la intrare este șirul **w** întreg:

STIVA ... ab	INTRARE cd ...\$
STIVA \$	INTRARE w\$

ANSIN deplasează în stivă zero sau mai multe simboluri de intrare până când în vârful stive apare un capăt, β . Apoi se reduce β la partea stângă a producției corespunzătoare și se continuă în mod ciclic aceste operații până când se detectează o eroare sau până când stiva conține doar simbolul de start iar intrarea este vidă:

STIVA \$\$	INTRARE w\$
----------------------	-----------------------

În acest caz AS se oprește și semnalează terminarea cu succes a analizei.

Exemplu. Se urmăresc pas cu pas acțiunile ANSIN deplasare – reducere la analiza șirului de intrare $id_1 + id_2 * id_3$ (gramatica expresiilor din exemplul anterior).

STIVA	INTRARE	ACȚIUNE
(1) \$	$id_1 + id_2 * id_3 \$$	deplasare
(2) \$ id_1	$+ id_2 * id_3 \$$	reducere $E \rightarrow id$
(3) \$E	$+ id_2 * id_3 \$$	deplasare
(4) \$E+	$id_2 * id_3 \$$	deplasare
(5) \$E+ id_2	$* id_3 \$$	reducere $E \rightarrow id$
(6) \$E+E	$* id_3 \$$	deplasare
(7) \$E+E*	$id_3 \$$	deplasare
(8) \$E+E* id_3	\$	reducere $E \rightarrow id$
(9) \$E+E*E	\$	reducere $E \rightarrow E * E$
(10) \$E+E	\$	reducere $E \rightarrow E + E$
(11) \$E	\$	acceptă

Fig. 5.3. Configurațiile ANSIN deplasare – reducere pentru intrarea $id_1 + id_2 * id_3$

ANSIN cu deplasare și reducere execută următoarele patru acțiuni:

- 1) Deplasare:** Simbolul de intrare următor este introdus în vârful stivei
- 2) Reducere:** Se realizează în situația în care extremitatea dreaptă a unui capăt se află în vârful stivei. Analizorul trebuie să localizeze extremitatea sa stângă, să stabilească neterminatul cu care se înlocuiește capătul și să realizeze efectiv reducerea lui.
- 3) Acceptare:** Analizorul semnalează terminarea cu succes a analizei.
- 4) Eroare:** Analizorul descoperă apariția unei erori și se apelează o rutină de revenire.

Există o motivație importantă care justifică utilizarea stivei la acest tip de analiză sintactică: capătul va fi întotdeauna în vârful stivei și nu în interior. Modalitatea de alegere a acțiunii, astfel încât analizorul să lucreze corect, depinde de tipul concret de analiză. În continuare, pe parcursul acestui capitol, vor fi abordate două astfel de tehnici: precedența operatorilor și analiza sintactică LR.

Prefixe viabile

Mulțimea prefixelor formelor propoziționale dreapta care pot apărea în stiva unui ASDR se numesc **prefixe viabile**. Altfel spus, un prefix viabil este un prefix al unei forme propoziționale dreapta care nu continuă dincolo de extremitatea dreaptă a celui mai din dreapta capăt al acelei forme propoziționale. Conform acestei definiții este întotdeauna posibil să se adauge simboluri terminale la extremitatea unui prefix viabil, pentru a obține o formă propozițională dreapta. De aceea, pe parcursul analizei, nu va apărea nici o eroare atâta timp cât porțiunea din intrare văzută până la un anumit punct poate fi redusă la un prefix viabil.

2.3 Analizoare sintactice LR

Tehnica LR reprezintă o metodă eficientă de AS ascendentă care poate fi aplicată pentru o clasă largă de GIC. Tehnica este numită ASLR(k). Pentru $k=1$, el poate fi omis.

Principalele *avantaje* ale analizei LR sunt:

- se pot elabora analizoare LR practic pentru toate construcțiile de limbaje exprimabile prin GIC;
- deși este cea mai generală metodă fără reveniri, dintre cele cunoscute, ASLR poate fi implementată la fel de eficient ca și alte metode de tipul deplasare – reducere;
- un ASLR poate detecta o eroare imediat ce este posibil, într-o baleiere de la stânga la dreapta a șirului de intrare (mai repede și mai exact decât alte metode).

Principalele *dezavantaje* ale metodei sunt:

- cantitatea relativ mare de memorie necesară;
- volumul mare de muncă pentru construcția manuală a unui analizor, pentru gramatica unui limbaj de programare tipic.

Există însă în prezent generatoare de ASLR care primesc la intrare o GIC și produc la ieșire tabelele de analiză necesare analizorului. Dacă gramatica conține ambiguități sau alte construcții dificil de analizat într-o baleiere stânga – dreapta a intrării, generatorul poate localiza aceste construcții și poate informa proiectantul compilatorului.

În prezent sunt utilizate trei variante de construire a tabeli de ASLR pentru o gramatică:

- 1) LR simplu (SLR): cea mai ușor de implementat dar și cea mai slabă (pot fi analizate doar o clasă mai redusă de gramatici);
- 2) LR canonic (LR): cea mai puternică dar și cea mai costisitoare (efort + memorie);
- 3) LR cu anticipare (LALR): varianta de compromis, intermediară; corespunde cel mai bine LP uzuale.

2.3.1 Algoritmul de analiză sintactică LR

Schema de principiu a unui ASLR este următoarea:

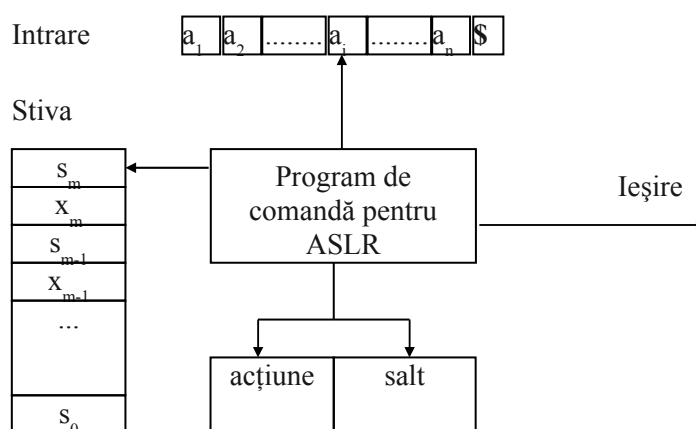


Fig. 5.4. Modelul unui ASLR

Un ASLR constă din: - tampon de intrare

- stivă
- un program de comandă
- o tabelă de analiză cu două părți
 $\left\{ \begin{array}{l} \text{acțiune} \\ \text{salt} \end{array} \right.$
- ieșire

Programul de comandă este același pentru toate ASLR; ceea ce diferă de la o variantă de analizor la alta sunt doar tabelele de analiză. Programul de analiză citește, unul câte unul, caracterul din tamponul de intrare. În stivă se memorează șiruri de forma:

s_0	x_1	s_1	x_2	s_2	\dots	x_m	s_m
-------	-------	-------	-------	-------	---------	-------	-------

cu sau s_m în vârf

Fiecare x_i este un simbol gramatical și fiecare s_i este un simbol numit **stare**. Se consideră că fiecare stare conține, în rezumat, informația prezentă în stivă sub acea stare. Combinația dintre simbolul de stare din vârful stivei (s_m) și simbolul de intrare curent (a_i), utilizată ca indici în tabela de analiză, determină comportarea analizorului (decizia deplasare – reducere). Într-o implementare concretă, simbolurile gramaticale nu trebuie să apară în stivă (sunt suficiente stările). Prezența lor contribuie doar la înțelegerea comportării analizorului.

Programul de comandă lucrează astfel: La început se consultă *acțiune* $[s_m, a_i]$, care poate să conțină una din următoarele valori:

- 1) **deplasează** și acoperă cu starea s ;
- 2) **reduce** prin producția $A \rightarrow \beta$;
- 3) **acceptă** șirul;
- 4) **eroare**.

Tabela "salt" se comportă ca o funcție care primește ca argument o stare și un simbol gramatical și produce o stare nouă (valorile din această tabelă sunt stări). Această tabelă este funcția de tranziție a unui AFD care recunoaște prefixele viabile ale gramaticii date, G . Starea inițială a acestui AFD este starea pusă inițial în vârful stivei (s_0).

Se numește **configurație** a unui ASLR o pereche în care prima componentă este conținutul momentan al stivei iar a doua componentă este partea rămasă neparcursă din intrare:

$$(s_0 \ x_1 \ s_1 \ \dots \ x_m \ s_m, a_i \ a_{i+1} \ \dots \ a_n \ \$)$$

Următoarea operație efectuată de analizor este determinată de citirea următorului simbol de intrare a_i , și de starea din vârful stivei, s_m , care conduce la consultarea tabelii "acțiune". În funcție de conținutul elementului *acțiune* $[s_m, a_i]$, rezultă în final una din următoarele configurații:

- 1) Dacă *acțiune* $[s_m, a_i]$ = "deplasare s ", se realizează, deplasarea lui a_i din intrare în stivă, urmată de acoperirea lui a_i în stivă, cu starea $s = salt[s_m, a_i]$; a_{i+1} devine noul simbol de intrare. Prin aceste mișcări se obține configurația:

$$(s_0 \ x_1 \ s_1 \ \dots \ x_m \ s_m a_i \ s, a_{i+1} \ a_{i+2} \ \dots \ a_n \ \$)$$

- 2) Dacă *acțiune* $[s_m, a_i]$ = "reducere $A \rightarrow \beta$ " atunci se obține configurația:

$$(s_0 \ x_1 \ s_1 \ \dots \ x_{m-r} \ s_{m-r} \ A s, a_i \ a_{i+1} \ \dots \ a_n \ \$)$$

unde $s = salt[s_{m-r}, A]$ iar r este $|\beta|$ (lungimea lui β).

S-au extras $2r$ simboluri din stivă (r simboluri de stare + r simboluri gramaticale), expunând starea s_{m-r} , completând stiva cu A și s . Simbolul de intrare curent nu este modificat. Ieșirea constă în executarea acțiunii semantice asociate cu producția de reducere (în exemplele de mai jos – o simplă tipărire a producției utilizate).

- 3) Dacă *acțiune* $[s_m, a_i]$ = "**acceptare**", analiza sintactică se termină.
- 4) Dacă *acțiune* $[s_m, a_i]$ = "**eroare**", se va apela o rutină de revenire din erori, iar configurația rezultată depinde de activitatea acestei rutine.

Algoritmul de analiză este prezentat schematic mai jos:

Intrare: un șir de intrare w și o tabelă de ASLR cu funcțiile **acțiune** și **salt** corespunzătoare gramaticii G .

Ieșire: Dacă $w \in L(G) \Rightarrow$ o analiză ascendentă pentru w altfel \Rightarrow o indicație de eroare.

Metoda: Se pornește de la configurația:

$(s_0, w\$)$



 în stivă în tamponul de intrare

după care se execută programul de mai jos, până se întâlnește o acțiune de **acceptare** sau **eroare**.

* poziționează ip la primul simbol de intrare;

repeat forever begin

* fie s starea din vârful stivei și a simbolul indicat de ip ;

if acțiune $[s,a]=\text{deplasează } s'$ **then begin**

* introdu a apoi s' în vârful stivei; {deplasează și acoperă cu s' }

* avansează ip la următorul simbol de intrare

end

else if acțiune $[s,a]=\text{reduce } A \rightarrow \beta$ **then begin**

* extrage $2 * |\beta|$ simboluri din stivă;

* fie s' starea rămasă, după extragere, în vârful stivei;

* introdu A , apoi $\text{salt } [s', A]$ în stivă;

* tipărește producția $A \rightarrow \beta$

end

else if acțiune $[s,a]=\text{acceptă}$ **then return**

else eroare()

end

Exemplu: Se consideră gramatica pentru expresii:

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow id$

Tabelele (funcțiile) acțiune și salt corespunzătoare sunt:

stare	acțiune						salt		
	id	+	*	()	\$	E	T	F
0	d_5			d_4			1	2	3
1		d_6				acc			
2		r_2	d_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	d_5			d_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	d_5			d_4				9	3
7	d_5			d_4					10

8		d_6			d_{11}				
9		r_1	d_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Codurile pentru acțiuni au următoarele semnificații:

- 1) d_i – deplasează și introduce în stivă starea i ;
- 2) r_j – reduce prin producția cu numărul j ;
- 3) acc – acceptă;
- 4) spațiu – eroare

Observație: Valoarea lui $\text{salt}[s,a]$, pentru a-terminal, se găsește în $\text{acțiune}[s,a]$, legată cu acțiunea de deplasare. În consecință, tabela salt are prevăzute coloane doar pentru neterminale.

În continuare se prezintă conținutul stivei și al intrării precum și acțiunile efectuate de analizor la analiza șirului de intrare: **id*id+id**.

STIVA	INTRARE	ACȚIUNE
(1) 0	id*id+id\$	depl (5)
(2) 0 id 5	*id+id\$	red (6) : $F \rightarrow \text{id}$
(3) 0 F 3	*id+id\$	red (4) . $T \rightarrow F$
(4) 0 T 2	*id+id\$	depl (7)
(5) 0 T 2 * 7	id+id\$	depl (5)
(6) 0 T 2 * 7 id 5	+id\$	red (6) $F \rightarrow \text{id}$
(7) 0 T 2 * 7 F 10	+id\$	red (3) $T \rightarrow T * F$
(8) 0 T 2	+id\$	red (2) $E \rightarrow T$
(9) 0 E 1	+id\$	depl (6) :
(10) 0 E 1 + 6	id\$	depl (5)
(11) 0 E 1 + 6 id 5	\$	red (6) : $F \rightarrow \text{id}$
(12) 0 E 1 + 6 F 3	\$	red (4) . $T \rightarrow F$
(13) 0 E 1 + 6 T 9	\$	red (1) . $E \rightarrow E+T$
(14) 0 E 1	\$	acceptă

2.3.2 Gramatici LR

O gramatică pentru care se poate construi o tabelă de analiză sintactică pentru ASDR se numește gramatică LR. Există GIC care nu sunt LR dar, pentru construcțiile tipice ale limbajelor de programare uzuale, acestea pot fi în general evitate sau înlocuite prin producții echivalente care se încadrează în clasa LR. Intuitiv, pentru ca o gramatică să fie LR este suficient ca un ASDR, parcurgând șirul de intrare de la stânga la dreapta, să fie capabil să recunoască toate capetele, atunci când ele apar în vârful stivei. Pentru aceasta, un ASLR nu trebuie să baleieze întreaga stivă întrucât simbolul de stare din vârful stivei conținând toată informația necesară.

Posibilitatea de recunoaștere a unui capăt cunoscând numai simbolurile gramaticale din stivă este echivalentă cu existența unui automat finit care, prin citirea simbolurilor gramaticale din stivă, de la vârful spre bază, poate localiza capătul în vârful stivei. Acest automat finit este în esență funcția salt a tabelii ASLR. Automatul nu trebuie însă să parcurgă stiva la fiecare mișcare. Simbolul de stare memorat în vârful stivei corespunde stării în care ar ajunge automatul finit de recunoaștere a capetelor în urma citirii simbolurilor gramaticale din stivă, de la bază la vârf. Din această cauză

ASLR nu are nevoie decât de starea din vârful stivei pentru a “cunoaște” conținutul stivei și pentru a lua decizia corectă de continuare a analizei.

Alte surse de informații utilizate de ASLR în luarea deciziilor deplasare – reducere sunt următoarele k simboluri din intrare. O gramatică ce poate fi analizată de un ASLR ce examinează până la k simboluri de intrare la fiecare mișcare se numește gramatică LR(k). de interes practic sunt cazurile $k=0$ și $k=1$ (Exemplu: tabela de acțiuni din exemplul anterior utilizează un simbol de anticipare, deci $k=1$).

Diferența de principiu între gramaticile LL și LR este următoarea: pentru ca o gramatică să fie LR(k) analizorul trebuie să fie capabil să recunoască apariția în stivă a părții drepte a unei producții, în stivă fiind chiar acea parte dreaptă sau tot ce derivă din ea, plus k simboluri de anticipare în șirul de intrare; în cazul unei gramatici LL(k), analizorul trebuie să recunoască utilizarea unei producții doar pe baza primelor k simboluri din subșirul derivat în partea dreaptă a producției respective. Rezultă că cerința ca o gramatică să fie LR(k) este mult mai puțin restrictivă decât în cazul LL(k) iar clasa gramaticilor LR este mai cuprinzătoare decât a gramaticilor LL.

2.4 Întrebări și probleme

1. Definiți analiza sintactică ascendentă.
2. Ce se înțelege prin *capăt*? Dar prin *fasonarea capetelor*?
3. Descrieți operațiile pe care le efectuează un analizor sintactic cu deplasare și reducere.
4. Ce avantaj are implementarea cu stivă a analizei sintactice cu deplasare și reducere?
5. Descrieți structura și funcționarea unui analizor sintactic LR.
6. Ce este o gramatică LR (k)?

Bibliografie

1. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers. Principles, Techniques and Tools, Addison-Wesley Publishing Company, 1986
2. Dick Grune, Henri E. Bal, Criel J.H. Jacobs, Koen Langendoen: Modern Compiler Design, John Wiley, 2003
3. Horia Ciocârlie: Limbaje de programare. Concepte fundamentale, Editura de Vest, 2007
4. Irina Athanasiu: Limbaje formale și compilatoare, Universitatea Politehnica București, 1992
5. Luca-Dan Șerbănați: Limbaje de programare și compilatoare, Editura Academiei, București, 1987
6. Teodor Rus: Mecanisme formale pentru specificarea limbajelor, Editura Academiei, 1983
7. David A. Watt, Deryck F. Brown: Programming Language Processors in Java - Compilers and Interpreters, Prentice Hall, 2000
8. Gabriel V. Orman: Limbaje formale, Editura Tehnică, București, 1982
9. Alexandru Mateescu, Dragoș Vaida: Structuri matematice discrete. Aplicații, Editura Academiei, București, 1989
10. Carmen De Sabata: Limbaje formale și translaatoare - îndrumător de laborator, Casa Cărții de Știință, Timișoara, 1999
11. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Șabloane de proiectare, Editura Teora, 2002

Analiza sintactică bazată pe precedența operatorilor (ASPO)

ASPO se poate aplica doar la o clasă redusă dar importantă de gramatici iar analizorul se poate construi ușor manual, pe principiul de lucru al ASDR. Gramatica trebuie să îndeplinească următoarele două cerințe:

- 1) Să nu aibă producții vide (cu ϵ în partea dreaptă).
- 2) În nici o parte dreaptă să nu fie două neterminale adiacente.

Gramaticile care satisfac ultima proprietate se numesc **gramatici de operatori**.

Exemplu: Următoarea gramatică pentru expresii nu este gramatică de operatori:

$$E \rightarrow EAE \mid (E) \mid -E \mid \text{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

Ea poate fi transformată într-o gramatică de operatori prin substituirea lui A cu fiecare din alternativele sale:

$$E \rightarrow E+E \mid E-E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid \text{id}$$

Ca tehnică generală de analiză sintactică, ASPO are o serie de dezavantaje:

- Sunt dificil de prelucrat acei atomi care au două precedențe diferite (exemplu: semnul minus, care este atât operator unar cât și binar).
- Relația între analizor și gramatică nu este întotdeauna biunivocă ceea ce face ca analizorul să accepte șiruri care nu fac parte din limbajul definit de gramatică.
- Clasa de gramatici analizabile prin tehnica ASPO este redusă.

Totuși, datorită simplității, în numeroase compilatoare s-a utilizat această tehnică exclusiv pentru expresii, instrucțiunile și construcțiile de nivel mai înalt fiind analizate prin alte metode (de exemplu, cu descendenți recursivi). S-au construit însă și ASPO pentru limbaje întregi.

Definirea relațiilor de precedență

În ASPO se definesc trei relații de precedență disjuncte, notate \langle , $=$ și \rangle , care se stabilesc între anumite perechi de terminale. Pe baza acestor relații se selectează capătul formei propoziționale, în stiva analizorului. Semnificația lor este următoarea:

Relația	Semnificația
$a \langle b$	a “cedează precedența” lui b
$a = b$	a “are aceeași precedență” ca b
$a \rangle b$	a “are precedență” față de b

Aceste relații se deosebesc în esență de relațiile algebrice $<$, $=$ și $>$ din următoarele motive:

- au cu totul altă semantică;
- este posibil ca între două terminale să nu existe nici o relație de precedență;
- este posibil ca între două terminale să existe simultan două relații de precedență (exemplu: $a \langle b$ și $a \rangle b$).

Relațiile de precedență între perechile de terminale se pot stabili pe două căi:

1) **Intuitiv:** pornind de la semnificația algebrică a operatorilor și ținând cont de prioritatea și de asociativitatea lor. De exemplu, deoarece “*” trebuie să aibă prioritate mai mare decât “+” se face $a * \rangle +$ și $+ \langle * a$. Astfel se rezolvă și ambiguitățile gramaticii expresiilor și se poate scrie și programul de ASPO.

2) **În mod automat:** aplicând un algoritm adecvat. În prealabil, trebuie eliminată ambiguitatea din gramatica inițială a limbajului, pentru ca ea să reflecte corect asociativitatea și prioritatea operatorilor. Pentru gramatici mai complicate decât gramatica expresiilor este posibil ca relațiile generate automat să nu fie disjuncte și limbajul să fie mai cuprinzător decât cel inițial.

Utilizarea relațiilor de precedență a operatorilor

Scopul introducerii relațiilor de precedență este delimitarea capetelor într-o formă propozițională dreaptă. Astfel, < marchează extremitatea stângă a unui capăt, = apare în interiorul capătului și > marchează extremitatea sa dreaptă.

Capătul poate fi găsit prin următorul procedeu:

1) Se baleiază șirul, cu relațiile de precedență introduse, de la extremitatea stângă și până la întâlnirea primului marcaj >. Acesta reprezintă extremitatea dreaptă a capătului.

2) Se baleiază șirul înapoi, omițând eventualele simboluri =, până la întâlnirea primului marcaj <. Acesta reprezintă extremitatea stângă a capătului.

3) Se consideră ca fiind capăt întreg șirul de simboluri gramaticale situat la stânga marcajului > și la dreapta marcajului <, inclusiv toate neterminalele care apar ca incluse sau înconjurând terminalele dintre cele două marcaje. Considerarea neterminalelor înconjurătoare este necesară pentru a asigura că nu vor apărea două neterminale adiacente în forma propozițională următoare.

Exemplu: Se consideră șirul de intrare: \$id+id* id \$ și matricea relațiilor de precedență de mai jos:

	id	+	*	\$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
\$	<·	<·	<·	

Șirul cu relațiile de precedență introduse, este:

\$<id>+<id>*<id>\$	capătul : id
\$E+id*id\$	red : E→ id
\$<+<id>*<id>\$	capătul : id
\$E+E*id\$	red : E→ id
\$<id+<·<id>\$	capătul : id
\$E+E*\$	red : E→ id
\$<+<·>\$	capătul : E*E
\$E+E\$	red : E→E*E
\$<+>\$	capătul : E+E
\$E\$	red : E→E+E

Deoarece neterminalele nu influențează analiza sintactică, nu trebuie făcută distincție între ele. În stiva analizorului este suficient să se țină un singur fel de marcaj, “neterminal”, pentru a indica locurile respective (utile doar pentru înregistrarea atributelor semantice).

Acțiunile analizorului sunt cele cunoscute:

1) **Deplasare:** cât timp nu s-a găsit extremitatea dreaptă a capătului adică, între simbolul terminal cel mai apropiat de vârful stivei și simbolul curent de intrare, este valabilă una din relațiile < sau =.

2) **Reducere:** S-a găsit extremitatea dreaptă a capătului adică, între simbolul terminal cel mai apropiat de vârful stivei și simbolul curent de intrare, este relația >. Prin baleierea în sens invers a conținutului stivei și verificarea relațiilor de precedență se caută extremitatea stângă după care se efectuează reducerea.

3) **Acceptare:** în situația în care ambele simboluri care se compară (vârful stivei și simbolul curent de intrare) sunt \$.

4) **Eroare:** dacă se ajunge în situația să se compare o pereche de terminale între care nu există nici o relație de precedență. Se va apela la o rutină specială de tratare și revenire.

Ideile de mai jos pot fi sintetizate în următorul algoritm:

Algoritmul de ASPO

Intrare: șirul w și tabela relațiilor de precedență.

Ieșire: dacă șirul w este corect se va obține un schelet al arborelui său sintactic (datorită substituirii neterminalelor pe parcursul analizei, nodurile interioare, corespunzătoare acestora, vor fi uniformizate); în caz contrar, se va da un mesaj de eroare.

Situația inițială a structurii de date:

STIVA
\$

INTRARE
 w \$

Algoritmul propriu-zis:

```

(1) poziționează pointerul de intrare pe primul simbol din  $w$ ;
(2) repeat forever
(3)   if * atât simbolul din vârful stivei cât și simbolul curent de intrare
(4)     sunt $ then return {acceptare}
(5)   else begin
(6)     * fie  $a$  simbolul terminal cel mai apropiat de vârful stivei
(7)     și  $b$  simbolul curent de intrare;
(8)     if ( $a < b$ ) or ( $a = b$ ) then begin {deplasare}
(9)       * introdu  $b$  în stivă;
(10)      * avansează cu o poziție pointerul de intrare
(11)    end else if  $a > b$  then {reducere}
(12)      repeat
(13)        * extrage din stivă
(14)      until * terminalul din vârful stivei este în relația <
(15)        cu terminalul cel mai recent extras
(16)    else eroare ()
(17)  end

```

Deducerea intuitivă a relațiilor de precedență din asociativitatea și prioritatea algebrică a operatorilor

Singura cerință care trebuie avută în vedere la stabilirea relațiilor de precedență este aceea ca ele să conducă la analiza corectă a limbajului definit de gramatică. Ținând cont de faptul că ASPO se aplică în primul rând la gramatici pentru expresii sau similare cu acestea, iar între operatorii din expresii există reguli de asociativitate și prioritate riguroase care rezolvă eventualele ambiguități, aceste reguli pot reprezenta baza stabilirii relațiilor de precedență.

Pentru cazul operatorilor binari, notați cu θ , θ_1 și θ_2 , relațiile de precedență pot fi deduse astfel:

1) Dacă θ_1 are prioritate algebrică mai mare ca θ_2 , se crează relațiile de precedență: $\theta_1 > \theta_2$ și $\theta_2 < \theta_1$

Exemplu: $* > +$, $+ < *$ și din $E + E * E + E$, $E * E$ se va reduce primul.

2) Dacă θ_1 și θ_2 sunt de prioritate egală (inclusiv cazul când ambele reprezintă același operator) apar următoarele situații:

a) θ_1 și θ_2 sunt asociativi la stânga $\Rightarrow \theta_1 > \theta_2$ și $\theta_2 > \theta_1$

b) θ_1 și θ_2 sunt asociativi la dreapta $\Rightarrow \theta_1 < \theta_2$ și $\theta_2 < \theta_1$

Exemplu: $+ > +$, $+ > -$, $- > -$, $- > +$ și din $E - E + E$ se va selecta la început capătul $E - E \uparrow ($ (asociativ la dreapta) și din $E \uparrow E \uparrow E$ se va selecta la început ultimul $E \uparrow E$

3) Oricare ar fi θ , există următoarele relații de precedență cu celelalte terminale: id, (,), \$

$\theta < id$ $\theta < ($ $\theta >)$ $\theta > \$$

$id > \theta$ $(< \theta$ $) > \theta$ $\$ < \theta$

Pentru a asigura reducerea la E a lui **id** și a (E), între terminalele care nu sunt operatori se introduc următoarele relații:

(\neq) $\$ < ($ $\$ < id$

$(< ($ $id > \$$ $) > \$$

$(< id$ $id >)$ $) >)$

Exemplu: gramatica $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Operatorii au următoarele priorități:

- 1) \uparrow este cel mai prioritar și este asociativ la dreapta;
- 2) $*$ și $/$ au prioritatea următoare și sunt asociativi la stânga;
- 3) $+$ și $-$ au cea mai mică prioritate și sunt asociativi la stânga

Rezultă următoarea tabelă a relațiilor de precedență (absența relației înseamnă caz de eroare):

	+	-	*	/	\uparrow	id	()	\$
+	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$
-	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$
*	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$
/	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$
\uparrow	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$
id	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$			$\cdot >$	$\cdot >$
($\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	=	
)	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$			$\cdot >$	$\cdot >$
\$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$		

Exemplu: $id * (id \uparrow id) - id / id$

$\$ \cdot id \cdot * \cdot (\cdot id \cdot \uparrow \cdot id \cdot) \cdot - \cdot id \cdot / \cdot id \cdot \$$

$E \rightarrow id$

$\$ \cdot * \cdot (\cdot id \cdot \uparrow \dots$

$E \rightarrow id$

$\$ \cdot * \cdot (\cdot \uparrow \cdot id \cdot) \dots$

$E \rightarrow id$

$\$ \cdot * \cdot (\cdot \uparrow \cdot) \dots$

$E \rightarrow E \uparrow E$

$\$ \cdot * \cdot (\cdot \uparrow \cdot) - \dots$

$E \rightarrow (E)$

$\$ \cdot * \cdot) - \dots$

$E \rightarrow E * E$

$\$ \cdot - \cdot id \cdot / \dots$

$E \rightarrow id$

$\$ \cdot - \cdot / \cdot id \cdot \$$

$E \rightarrow id$

$\$ \cdot - \cdot / \cdot \$$

$E \rightarrow E / E$

$\$ \cdot - \cdot \$$

$E \rightarrow E - E$

SE\$

Acceptare

Tratarea operatorilor unari

Pentru acei operatori unari (prefix sau postfix) care nu sunt în același timp și binari, se pot ușor extinde regulile din paragraful precedent. Pentru exemplificare să considerăm θ ca fiind un operator oarecare, binar sau unar, și să definim relațiile sale de precedență cu operatorul unar \neg (negare logică):

$\theta < \neg$	și
$\neg > \theta$	dacă θ are o prioritate mai mică decât \neg
$\neg < \theta$	dacă θ are o prioritate mai mare decât \neg

Exemplu: \neg (operator unar) mai prioritar decât $\&$ (operator binar). Conform acestor reguli, expresia $E \& \neg E \& E$ va fi evaluată astfel: $(E \& (\neg E)) \& E$.

Dacă operatorul unar este în același timp și binar (cazul semnelui $-$) situația se complică foarte mult, chiar dacă cei doi operatori, identici ca formă, ar avea aceeași precedență. Pentru exemplificare se poate studia tratarea șirului **id*-id**, conform tabelului anterior. O soluție posibilă este modificarea ANLEX astfel încât să returneze coduri lexicale diferite pentru cele două situații de utilizare a semnelui $-$. Dacă se realizează acest lucru, operatorul unar minus devine un operator distinct și poate fi tratat cu regulile de mai sus. Trebuie totuși menționat că nu este simplu pentru ANLEX să distingă între cele două situații, deoarece este nevoie să privească operatorul în contextul în care apare.

Funcții de precedență

Un dezavantaj al ASPO poate să fie dimensiunea relativ mare a tabelului relațiilor de precedență. În majoritatea cazurilor însă, nu este obligatoriu ca analizorul să păstreze în memorie această tabelă. Ea poate fi substituită prin două **funcții de precedență** notate f și g , care pun în corespondență simbolurile gramaticale cu numere întregi. Funcțiile f și g vor fi astfel alese încât să satisfacă următoarele condiții:

1) $f(a) < g(b)$	dacă	$a < b$
2) $f(a) = g(b)$	dacă	$a = b$
3) $f(a) > g(b)$	dacă	$a > b$

În acest fel, determinarea relației de precedență între a și b se reduce la simpla comparare numerică între $f(a)$ și $g(b)$. Un dezavantaj al utilizării funcțiilor de precedență este acela că nu mai pot fi sesizate acele cazuri de eroare marcate în tabelă prin intrări vide (lipsa relației de precedență). Cauza este aceea că între numerele $f(a)$ și $g(b)$ există întotdeauna una dintre relațiile 1), 2) sau 3). Dezavantajul nu este însă foarte mare pentru că erorile nu se pierd ci doar se amână detectarea lor până în momentul încercării de reducere, când nu va putea fi găsit un capăt corespunzător.

Nu orice tabelă a relațiilor de precedență poate fi codificată prin funcții de precedență pentru că nu pot fi construite funcțiile astfel încât să respecte cerințele de mai sus pentru toate relațiile. Totuși, pentru multe cazuri practice, funcțiile există și se pot construi.

Exemplu: Tabela de precedență anterioară are următoarea pereche de funcții de precedență:

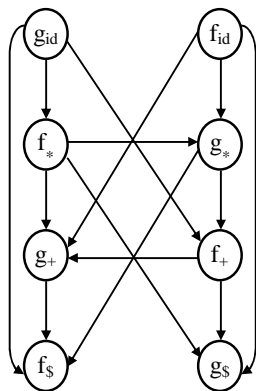
	+	-	*	/	↑	()	id	\$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

Se observă că $f(\text{id}) > g(\text{id}) \Rightarrow \text{id} > \text{id}$, deși, în realitate, între **id** și **id** nu există relație de precedență – caz de eroare.

Algoritmi de construire a funcțiilor de precedență:

- Metoda**
- 1) Crează simbolurile f_a și g_a pentru fiecare terminal a , inclusiv $\$$.
 - 2) Se partiționează simbolurile create în grupuri, astfel încât dacă $a = b$ atunci f_a și g_b sunt în același grup.
 - 3) Se crează un graf orientat ale cărui noduri sunt constituite din grupurile găsite anterior iar arcele se stabilesc astfel:
 - a) pentru toate perechile a și b pentru care $a < b$ se duce un arc de la grupul lui g_b la grupul lui f_a ;
 - b) dacă $a > b$, se duce un arc de la grupul lui f_a la grupul lui g_b .
 Existența unui arc sau a unui drum de la f_a la g_b înseamnă că valoarea lui $f(a)$ trebuie să depășească pe cea a lui $g(b)$. Invers, un drum de la g_b la f_a înseamnă că trebuie ca $g(b) > f(a)$.
 - 4) Dacă graful construit la punctul 3) are un ciclu, atunci nu există funcții de precedență. Dacă nu există cicluri atunci, pentru orice terminal a , inclusiv $\$$, $f(a)$ va primi ca valoare lungimea celui mai lung drum care începe în grupul lui f_a iar $g(a)$ va fi egală cu lungimea celui mai lung drum care începe în grupul lui g_a .

Exemplu: Se consideră matricea de precedență redusă (pentru simbolurile **id**, **+**, ***** și **\$**). Nu există relația $=$ așa că fiecare simbol constituie el însuși un grup conform algoritmului, rezultă următorul graf:



Nu există cicluri \Rightarrow există funcții de precedență

	+	*	id	\$
f	2	4	4	0
g	1	3	5	0

$f(\$)=g(\$)=0$ deoarece $f_\$$ și $g_\$$ nu au arce de ieșire. $g(+)=1$ pentru că cel mai lung drum este format dintr-un singur arc. $\$$. a. m. d.

Revenirea din erori în analiza sintactică bazată pe precedența operațiilor.

În procesul de ASPO există două situații în care se pot detecta erori sintactice:

- 1) Dacă s-a găsit un capăt (conform relațiilor de precedență) dat nu există nici o producție având acel capăt ca parte dreaptă.
- 2) Dacă între terminalul din vârful stivei și simbolul curent de intrare nu există relație de precedență.

Într-un capăt detectat în stivă apar numai simboluri terminale însă trebuie ținut cont și de pozițiile în care există neterminale care, deși nu sunt nominalizate, au totuși locuri rezervate în stiva de analiză. Ca atare, erorile de tipul 1) constau în negăsirea unei producții care să aibă în partea dreaptă aceleași terminale și în aceeași ordine ca și cele din capăt și aceleași poziții pentru neterminale. Pentru aceasta, algoritmul de ASPO prezentat anterior trebuie completat, în porțiunea de tratare a reducerilor și cu verificarea corespondenței între capătul extras din stivă și partea dreaptă a unei producții. Această operație are ca efect și localizarea producției pe baza căreia se face reducerea, operație esențială la analiza semantică, deoarece regulile semantice sunt asociate producțiilor și ele se execută în timpul reducerilor.

Alte situații de eroare, în afara celor semnalate la punctele 1) și 2) de mai sus nu pot să apară. De exemplu, la extragerea din stivă, există certitudinea că se va localiza extremitatea stângă a capătului (aparitia unei relații $<$), cel târziu între $\$$ -ul de la baza stivei și primul simbol situat deasupra lui, deoarece $\$ < \forall$ simbol. Ca atare, extragerea din stivă, operație care reprezintă efectiv reducerea, se va putea finaliza cu certitudine.

Tratarea erorilor în timpul reducerilor (tipul 1)

În momentul detectării unei erori de tipul 1), pentru a emite un mesaj potrivit, trebuie căutată o parte dreaptă a unei producții, asemănătoare cu capătul extras.

Exemple:

- Dacă s-a extras un capăt ce conține terminalele **abc** și există o parte dreaptă de forma **aEcE** în care apar terminalele **ac** dar nu și **b**, se poate emite mesajul: “**b** ilegal în linia ...”
- În altă situație se poate cere să inserăm un terminal pentru a găsi corespondența cu partea dreaptă:
abEdc “lipsește **d** în linia ...”
- În situația aEbc iar capătul extras nu solicită prezența unui neterminat între a și b, se poate emite mesajul “lipsește E în linia ...”

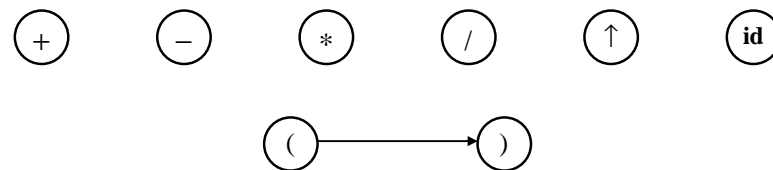
Posibilitatea de a emite un mesaj de eroare exact în situația în care capătul nu corespunde cu partea dreaptă a nici unei producții din gramatică este direct legată de existența unui număr finit sau infinit de șiruri care pot fi extrase ca și capete. Considerând un astfel de șir, notat $b_1 b_2 \dots b_k$, dacă el este capăt, atunci între simbolurile adiacente care îl compun este valabilă doar relația de precedență \doteq , adică $b_1 \doteq b_2 \doteq \dots \doteq b_k$. Dacă din analiza tabelii de precedență a operatorilor rezultă existența unui număr finit de secvențe de terminale legate prin \doteq , atunci fiecare astfel de caz se poate trata distinct și, având ca element de referință partea dreaptă cea mai apropiată, se poate elabora un mesaj de eroare potrivit.

Pentru a evidenția șirurile care pot fi extrase din stivă ca și capete se va construi un graf orientat ale cărui noduri sunt terminalele gramaticii iar arcele leagă nodurile între care există relația \doteq . Capetele posibile sunt toate șirurile date de etichetele nodurilor care formează drumuri în acest graf. În plus, pentru ca un drum $b_1 b_2 \dots b_k$ să poată fi capăt, trebuie să existe două simboluri notate a și c (terminale sau \$) astfel ca $a < b_1$ și respectiv $b_k > c$. Dacă respectă și această condiție, nodurile corespunzătoare lui b_1 și b_k se vor numi **nod inițial** și respectiv **nod final**. Cu aceste notații, concluzia va fi următoarea: dacă în graful construit există cel puțin un drum care să conțină un ciclu, atunci numărul posibil de capete este infinit, astfel este finit.

Exemplu: Se consideră gramatica:

$$E \rightarrow E+E \mid E-E \mid E * E \mid E/E \mid E \uparrow E \mid (E) \mid \text{id}$$

a cărei tabelă de precedență a fost dată anterior. Graful corespunzător este următorul:



Singura pereche de noduri legate este cea corespunzătoare parantezelor (singurul \doteq din tabelă). Toate nodurile, mai puțin “(”, pot fi noduri inițiale și toate nodurile, mai puțin “)”, pot fi noduri finale. Deci există următoarele drumuri (în număr finit) de la un nod inițial la unul final: +, −, *, /, ↑, **id** de lungime 1 și () de lungime 2 și toate corespund terminalelor din partea dreaptă a unei producții. Ca atare, rutina de tratare a erorilor în timpul reducerilor trebuie să verifice doar existența marcajelor de neterminale, pe pozițiile corecte, între șirurile de terminale care se reduc. Pot exista următoarele situații:

- Dacă se reduce un capăt care conține unul din terminalele +, −, *, /, ↑, atunci trebuie verificat că apare câte un neterminat în ambele părți ale terminalului. Altfel, se va emite mesajul: “Lipsește operand în stânga/dreapta operatorului ...”.
- Dacă se reduce **id**, atunci trebuie verificat să nu apară neterminat nici la stânga și nici la dreapta sa. Dacă apare un neterminat, se emite mesajul “Lipsește operator la stânga/dreapta identificatorului ...”.
- Dacă se reduce capătul (), atunci pot apărea următoarele situații de eroare:

- Nu există neterminat între paranteze. Se poate emite mesajul: “Lipsește expresie între ()”
- Există neterminale adiacente în exterior cu parantezele, de o parte sau de alta. Se pot da mesaje similare cu cel de la punctul 2): “Lipsește operator la stânga (sau la dreapta)”

Dacă numărul de șiruri ce pot fi extrase ca și capete este infinit, mesajele de eroare nu se pot stabili cu exactitate. În aceste cazuri, rutina de tratare a erorilor ar putea determina dacă există o parte dreaptă apropiată de șirul extras (de exemplu, la distanța de 1 sau 2, unde distanța înseamnă numărul de atomi inserat, șters sau schimbat). Dacă există o astfel de producție, se poate emite un mesaj concret, în sensul că se așteaptă să apară acea parte dreaptă. Dacă nici o parte dreaptă nu este suficient de aproape de capăt, nu se poate emite decât un mesaj general, de genul: “Eroare de sintaxă în linia ...”.

Tratarea erorilor deplasare – reducere (tipul 2)

Dacă la consultarea matricei de precedență se constată că, între simbolul din stivă și cel din intrare, nu există relație de precedență atunci, pentru a continua analiza, trebuie să se modifice fie stiva, fie intrarea, fie ambele structuri. Modificarea va consta în inserarea, ștergerea sau înlocuirea unui simbol. La inserări sau ștergeri există pericolul de a se provoca un ciclu infinit. De exemplu, inserarea repetată de simboluri în intrare, fără a putea reduce sau deplasa simbolurile inserate.

Pentru a preveni ciclurile infinite, rutina de tratare a erorii trebuie să garanteze că, după modificare și revenire, simbolul curent de intrare poate fi deplasat în stivă. Pentru cazul particular când, inițial, simbolul curent de intrare este \$, se evită inserarea de simboluri în intrare și se reduce stiva. Să considerăm configurația:

STIVA
... ab

INTRARE
cd ...\$

Configurația conduce la eroare întrucât între b și c nu există relație de precedență. Pot exista următoarele variante de tratare a erorii:

- Dacă $a \leq c$ (înseamnă $<$ sau $=$), se poate extrage **b** din stivă.
- Dacă $b \leq d$, se șterge **c** din intrare.
- Se caută un simbol **e**, astfel încât $b \leq e \leq c$ care să se insereze în intrare, în fața lui **c**. Dacă nu există un astfel de simbol, se poate căuta pentru inserare un șir de simboluri e_1, e_2, \dots, e_n astfel ca $b \leq e_1 \leq e_2 \leq \dots \leq e_n \leq c$.

Ațiunea exactă ce se alege reflectă, în general, intuiția proiectantului compilatorului relativ la cea mai probabilă eroare din fiecare caz. Pentru fiecare intrare necompletată din matricea de precedență trebuie să se specifice o rutină de revenire din erori (nu neapărat distinctă de cea corespunzătoare altor intrări) în care se execută modificarea adecvată a stivei și/sau a intrării și se emite mesajul de eroare.

Exemplu: Se consideră din nou matricea de precedență din § 5. 2. 3. Se rețin din această matrice doar liniile și coloanele pe care apar intrări vide, completate cu numele rutinelor de tratare a erorilor:

	id	()	\$
id	e_3	e_3	$>$	$>$
($<$	$<$	$=$	e_4
)	e_3	e_3	$>$	$>$
\$	$<$	$<$	e_2	e_1

În esență, cele patru rutine cuprind următoarele operații:

- Se apelează dacă lipsește întreaga expresie.
Se tratează prin inserarea unui **id** în intrare.
Mesajul: “Lipsește operand”.
- Se apelează când expresia începe cu o “(”.
Se tratează prin ștergerea “)” din intrare.
Mesajul: “)” fără pereche – în plus”.
- Se apelează când **id** sau “(” urmează după **id** sau “)”.
Se tratează prin inserarea unui operator, de ex. “+”, în intrare.
Mesajul: “Lipsește operator”.
- Se apelează când expresia se termină cu “(”.
Se tratează prin extragerea “(” din stivă.
Mesajul: “Lipsește paranteză dreaptă”.

Pentru a analiza modul de funcționare al acestui mecanism, să considerăm șirul de intrare eronat **id +**). După primele acțiuni (deplasarea lui **id**, reducerea la E și deplasarea lui “+”) se ajunge în următoarea configurație:

STIVA
\$E

INTRARE
)\$

Deoarece + >) urmează o reducere a capătului “+” iar, în timpul reducerii, se observă absența lui E din dreapta operatorului. Mesaj: “Lipsește operand în dreapta operatorului +”. După efectuarea reducerii, se ajunge la configurația:

STIVA
\$E

INTRARE
)\$

Pentru \$ și) nu există relație de precedență și se apelează rutina **e₂**. Mesaj: “)” fără pereche – în plus” Prin eliminarea “)” din intrare, se ajunge la configurația finală a analizorului:

STIVA
\$E

INTRARE
\$

Construirea tabelelor de analiză sintactică în varianta SLR

O gramatică pentru care se pot construi tabele de ASLR prin tehnica SLR se numește **gramatică SLR**.

Se numește **element LR(0)** al unei gramatici G (sau, pe scurt, **element**), o producție a acelei gramatici în care s-a plasat un punct într-o anumită poziție a părții drepte. De exemplu, producției $A \rightarrow XYZ$ i se pot atașa următoarele patru elemente.

$A \rightarrow .XYZ$ $A \rightarrow X.YZ$ $A \rightarrow XY.Z$ $A \rightarrow XYZ.$

Producției $A \rightarrow \epsilon$ îi corespunde un singur element: $A \rightarrow .$

Elementele pot fi reprezentate printr-o pereche de numere întregi corespunzătoare numărului producției și respectiv poziției punctului. Elementele se utilizează în procesul de analiză sintactică pentru a arăta cât s-a văzut, la un moment dat, dintr-o anumită producție. De exemplu, $A \rightarrow .XYZ$ arată că se așteaptă ca în intrare să existe un șir derivat din XYZ iar $A \rightarrow X.YZ$ arată că tocmai s-a văzut în intrare un șir derivabil din X și se așteaptă, în continuare, un șir derivabil din YZ.

Metoda SLR de construire a tabeli de analiză se bazează pe construirea prealabilă, din gramatica dată, a unui AFD care să recunoască prefixele viabile. Stările acestui AFD sunt mulțimi de elemente. Elementele individuale pot fi considerate ca stările AFN ce recunoaște, de asemenea, prefixele viabile iar gruparea elementelor în mulțimi, care trebuie efectuată aici, este similară construirii submulțimilor de la transformarea AFN în AFD.

La baza construirii tabelelor de analiză prin metoda SLR stă o colecție de mulțimi de elemente LR(0) numită **colecția LR(0) canonică**. Aceasta se obține cu ajutorul a două funcții, numite **închidere** și **salt** aplicate asupra **gramaticii augmentate**, corespunzătoare gramaticii inițiale.

Dacă G este o gramatică având simbolul de start S, atunci gramatica sa augmentată G' constă din gramatica G, completată cu un nou simbol de start S' și cu producția $S' \rightarrow S$. Scopul introducerii acestei producții suplimentare este de a permite oprirea analizei sintactice și anunțarea acceptării șirului de intrare, atunci când se efectuează reducerea cu producția $S' \rightarrow S$.

Operația de închidere

Dacă I este o mulțime de elemente pentru o gramatică G, atunci *închidere* (I) este, de asemenea, o mulțime de elemente, construită din I prin următoarele două reguli:

- 1) Inițial, toate elementele din I se introduc în *închidere* (I).
- 2) Dacă $A \rightarrow \alpha.B\beta$ este un element din *închidere* (I) și $B \rightarrow \gamma$ este o producție a gramaticii G, atunci se adaugă la *închidere* (I) elementul $B \rightarrow \gamma$ (dacă nu este deja prezent). Această regulă se aplică în mod repetat până nu se mai adaugă elemente noi la *închidere* (I).

Intuitiv, prezența elementului $A \rightarrow \alpha.B\beta$ în *închidere* (I) arată că, la un moment dat în procesul de analiză ne așteptăm să vedem în continuare, în intrare, un subșir derivabil din $B\beta$. Dacă în plus, $B \rightarrow \gamma$ este o producție a gramaticii, înseamnă că s-ar putea să vedem mai întâi, în acest moment, un subșir derivabil din γ , motiv pentru care se va include în *închidere* (I) și elementul $B \rightarrow \gamma$.

Exemplu: Se consideră gramatica augmentată pentru expresii:

$E' \rightarrow E$
 $E \rightarrow E+T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

și se ia $I = \{ [E' \rightarrow .E] \}$

\Rightarrow închidere (I) = $\{ [E' \rightarrow .E], [E \rightarrow .E+T], [E \rightarrow .T], [T \rightarrow .T * F], [T \rightarrow .F],$

$[F \rightarrow .(E)], [F \rightarrow .id] \}$

Cele două reguli de construire a funcției închidere (I) pot fi cuprinse în următorul algoritm:

function închidere (I);

begin

 J := I;

repeat

for * fiecare element $[A \rightarrow \alpha.B\beta]$ din J și fiecare producție $B \rightarrow \gamma$ din G
 astfel încât $[B \rightarrow \gamma] \notin J$ **do**

 * adaugă $[B \rightarrow \gamma]$ la J

until * nu se mai pot adăuga elemente noi la J;

return J

end;

În vederea implementării funcției închidere se recomandă crearea unui tablou cu elemente logice numit *adăugat*, indexat de neterminalele gramaticii G, astfel încât *adăugat* [B] primește valoarea *true* atunci când se adaugă elementele $B \rightarrow \gamma$ pentru acel neterminal B.

Din modul de calcul al funcției *închidere* rezultă că dacă se adaugă o producție B cu punct la extremitatea stângă, atunci vor fi adăugate toate producțiile pentru B. În concluzie, nu este necesar ca elementele de forma $B \rightarrow \gamma$ să se rețină în totalitate în *închidere* (I), fiind suficientă o listă de neterminale B ale căror producții trebuie adăugate în acest mod. Rezultă că elementele dintr-o mulțime pot fi împărțite în două clase astfel:

- 1) **Elemente nucleu:** care includ elementul inițial $S' \rightarrow S$ și toate elementele ale căror puncte nu sunt la extremitatea stângă.
- 2) **Elemente nenucleu:** cele care au punctul la extremitatea stângă.

Mulțimile de elemente care apar în procesul de calcul prin metoda SLR se obțin întotdeauna prin închiderea unei mulțimi de *elemente nucleu* iar elementele care se adaugă prin închidere sunt *elemente nenucleu*. În concluzie, mulțimile de elemente care interesează în această metodă pot fi reprezentate în memorie doar prin elemente nucleu, cele nenucleu urmând să fie regenerate ori de câte ori ar fi necesar.

Operația salt

Argumentele funcției *salt* (I,X) au următoarea semnificație:

I – o mulțime de elemente LR(0);

X – un simbol al gramaticii G(neterminal sau terminal).

Rezultatul funcției *salt* (I,X) este reprezentat de închiderea mulțimii tuturor elementelor $[A \rightarrow \alpha X \beta]$ pentru care $[A \rightarrow \alpha X \beta]$ este în I. Intuitiv, aceasta înseamnă că, dacă I este mulțimea de elemente valide pentru un anumit prefix viabil γ , atunci *salt* (I,X) este mulțimea de elemente valide pentru prefixul viabil γX .

Exemplu: $I = \{[E' \rightarrow E.], [E \rightarrow E.+T]\}$

$\text{salt}(I,+) = \{[E \rightarrow E.+T], [T \rightarrow T.*F], [T \rightarrow T.F], [F \rightarrow (.(E)], [F \rightarrow (.id)]\}$

Procedeul de calcul este următorul: se caută în mulțimea I acele elemente care au punctul în stânga simbolului +. Din aceste elemente se obțin elemente noi mutând punctul în dreapta simbolului +, după care se determină închiderea acestor din urmă mulțimi.

Construirea colecției LR(0) canonice

Algoritmul pentru construirea colecției LR(0) canonice, notată cu C, este următorul: **procedure** colecție (G');

begin

$C := \{\text{închidere}(\{[S' \rightarrow S]\})\};$

repeat

for * fiecare mulțime de elemente I din C și fiecare simbol gramatical X astfel încât *salt* (I,X) nu e vidă și nu e prezentă în C **do**
* adaugă *salt* (I,X) la C

until * nu se mai pot adăuga la C mulțimi noi de elemente

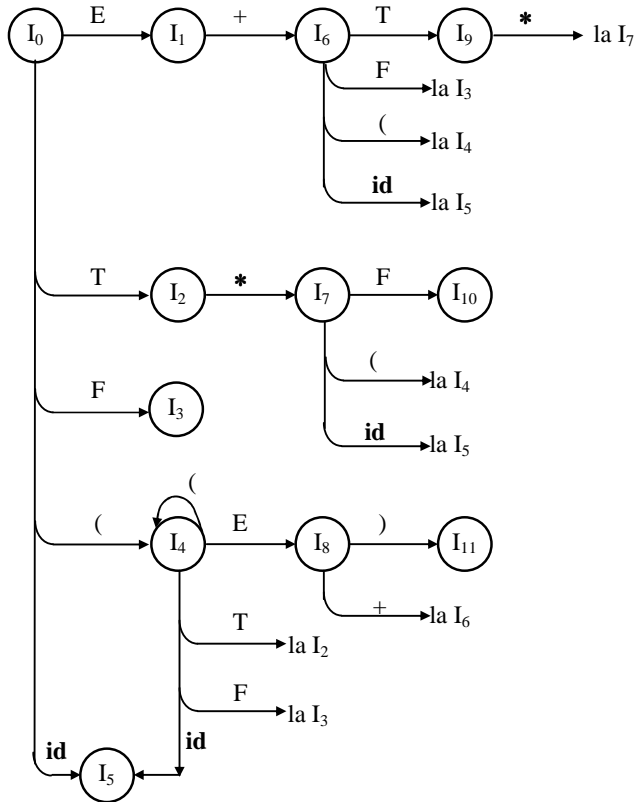
end;

Exemplu: Se consideră gramatica pentru expresii din exemplele anterioare.

Colecția canonică de mulțimi de elemente LR(0) este următoarea:

$I_0 = \text{închidere}(\{[E' \rightarrow E.]\})$	$I_1 = \text{salt}(I_0, E)$	$I_4 = \text{salt}(I_0, ($	} element nucleu
$E' \rightarrow E.$	$E' \rightarrow E.$	$F \rightarrow (.E)$	
$E \rightarrow E.+T$	$E \rightarrow E.+T$	$E \rightarrow E.+T$	} elemente nenucleu
$E \rightarrow T.$		$E \rightarrow T.$	
$T \rightarrow T.*F$	$I_2 = \text{salt}(I_0, T)$	$T \rightarrow T.*F$	
$T \rightarrow F.$	$E \rightarrow T.$	$T \rightarrow F.$	
$F \rightarrow (.E)$	$T \rightarrow T.*F$	$F \rightarrow (.E)$	
$F \rightarrow .id$		$F \rightarrow .id$	
	$I_3 = \text{salt}(I_0, F)$		
	$T \rightarrow F.$	$I_{10} = \text{salt}(I_7, F)$	
$I_5 = \text{salt}(I_0, id)$		$T \rightarrow T.*F.$	
$F \rightarrow id.$			
	$I_8 = \text{salt}(I_4, E)$		
$I_6 = \text{salt}(I_1, +)$	$F \rightarrow (.E)$	$\text{salt}(I_7, () \equiv I_4$	
$E \rightarrow E.+T$	$E \rightarrow E.+T$	$\text{salt}(I_7, id) \equiv I_5$	
$T \rightarrow T.*F$			
$T \rightarrow F.$	$\text{salt}(I_4, T) \equiv I_2$	$I_{11} = \text{salt}(I_8,)$	
$F \rightarrow (.E)$	$\text{salt}(I_4, F) \equiv I_3$	$F \rightarrow (.E).$	
$F \rightarrow .id$	$\text{salt}(I_4, () \equiv I_4$		
	$\text{salt}(I_4, id) \equiv I_5$	$\text{salt}(I_8, +) \equiv I_6$	
		$\text{salt}(I_9, *) \equiv I_7$	
$I_7 = \text{salt}(I_2, *)$	$I_9 = \text{salt}(I_6, T)$		
$T \rightarrow T.*F$	$E \rightarrow E.+T.$		
$F \rightarrow (.E)$	$T \rightarrow T.*F.$		
$F \rightarrow .id$			
		$\text{salt}(I_6, F) \equiv I_3$	
		$\text{salt}(I_6, () \equiv I_4$	
		$\text{salt}(I_6, id) \equiv I_5$	

Funcția *salt* pentru întreaga colecție canonică se poate reprezenta sub forma unui AFD ca în **fig.** Dacă I_0 este stare inițială și orice stare a AFD este considerată stare finală, atunci automatul recunoaște exact prefixele viabile ale gramaticii G' . Explicația acestui fapt este următoarea: se consideră un AFN ale cărui stări sunt chiar elementele individuale, înzestrat cu tranziții de la $[A \rightarrow \alpha.X\beta]$ la $[A \rightarrow \alpha X.\beta]$ etichetate cu X și cu tranziții de la $[A \rightarrow \alpha.B\beta]$ la $[B \rightarrow \gamma]$ etichetate cu ϵ . *Închidere* (I) este exact funcția ϵ -închidere aplicată mulțimii de stări corespunzătoare a AFN în procedeul de transformare a AFN în AFD echivalent. În concluzie, *salt* (I,X) va furniza tranziția de ieșire din starea I pentru simbolul X în AFD construit din AFN prin aplicarea algoritmului de conversie bazat pe construirea de submulțimi. În acest context, procedura *colecție* (G') corespunde exact construirii funcției de tranziție pentru AFD rezultat și echivalent cu AFN inițial, definit ca mai sus.



Se spune că **elementul** $A \rightarrow \beta_1.\beta_2$ **este valid** pentru un prefix viabil $\alpha\beta_1$ dacă există o derivare $S' \xRightarrow{*} \alpha A w \xRightarrow{*} \alpha\beta_1\beta_2 w$. În general, un element va fi valid pentru mai multe prefixe viabile. Faptul că $A \rightarrow \beta_1.\beta_2$ este valid pentru $\alpha\beta_1$ indică dacă trebuie făcută *deplasare* sau *reducere* în situația în care în stivă apare prefixul viabil $\alpha\beta_1$, și anume:

- a) dacă $\beta_2 \neq \epsilon \Rightarrow$ capătul nu este complet deplasat în stivă \Rightarrow **deplasare**
- b) dacă $\beta_2 = \epsilon \Rightarrow A \rightarrow \beta_1$ ar trebui să fie capăt \Rightarrow **reducere**

Dacă există două sau mai multe elemente valide pentru același prefix viabil, examinarea lor ar putea să ne ducă la concluzii diferite privind operația (deplasare sau reducere) ce trebuie efectuată. Aceste situații sunt **conflicte de analiză sintactică**. Unele dintre aceste conflicte se pot rezolva luând în considerare următorul simbol de intrare iar altele aplicând metode de analiză mai puternice.

Pentru fiecare prefix viabil care poate apărea în stiva unui ASLR se poate calcula mulțimea de elemente valide corespunzătoare. O teoremă importantă a teoriei ASLR, care se poate și demonstra, arată că mulțimea de elemente valide pentru un prefix viabil γ este chiar mulțimea de elemente la care se ajunge din starea inițială, pe un drum etichetat γ în AFD construit din colecția canonică de mulțimi de elemente cu tranzițiile date de funcția *salt*. În esență, mulțimea de elemente valide cuprinde toată informația utilă care poate fi culeasă din stivă.

Exemplu: Se consideră din nou gramatica pentru expresii, ale cărei mulțimi de elemente și funcție *salt* au fost deja calculate. Pentru această gramatică, șirul "E+T*" este un prefix viabil. După citirea elementelor șirului, AFD ajunge din starea I_0 în starea I_7 . Starea I_7 conține elementele $T \rightarrow T*.F$ $F \rightarrow \cdot(E)$ $F \rightarrow \cdot id$ care sunt toate elementele valide pentru prefixul E+T*, așa cum rezultă din următoarele trei derivări dreapta:

$E' \Rightarrow E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+T*F*id$
 $E' \Rightarrow E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*(E)$
 $E' \Rightarrow E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id$

Prima derivare confirmă validitatea lui $T \rightarrow T*.F$, cea de a doua confirmă validitatea elementului $F \rightarrow \cdot(E)$ iar a treia, validitatea lui $F \rightarrow \cdot id$ pentru prefixul viabil E+T*. Se poate și arăta că pentru acest prefix viabil nu există alte elemente valide.

Algoritmul de construire a tabelii de analiză

Din AFD care recunoaște prefixele viabile se pot construi cele două părți, *acțiune* și *salt*, ale tabelii de analiză sintactică prin metoda SLR. Algoritmul prezentat în continuare se poate aplica unei clase destul de largă de gramatici. Sunt însă și situații în care el eșuează: produce tabele cu intrări multiplu definite.

Se pornește de la gramatica G , pentru care se obține gramatica augmentată G' . Din G' se construiește colecția LR(0) canonică, notată C . Din colecția C , pe baza algoritmului următor, se construiesc tabelele *acțiune* și *salt*:

1. Se construiește colecția de mulțimi de elemente LR(0) pentru G' : $C = \{I_0, I_1, \dots, I_n\}$
2. Din mulțimea I_i , a colecției se construiește starea i a analizorului sintactic. Acțiunile de analiză pentru starea i se determină astfel:
 - a) Dacă $[A \rightarrow \alpha \cdot a \beta] \in I_i$ și $\text{salt}(I_i, a) = I_j$ atunci *acțiune* $[i, a]$ se pune pe valoare “depl. j ” (prescurtat d_j); a —terminal.
 - b) Dacă $[A \rightarrow \alpha \cdot] \in I_i$ atunci *acțiune* $[i, a]$ ia valoarea “reduce cu $A \rightarrow \alpha$ ” (prescurtat r_k), unde $a \in \text{URM}(A)$; $A \neq S'$.
 - c) Dacă $[S' \rightarrow S \cdot] \in I_i$ atunci *acțiune* $[i, \$] = \text{acceptă}$.

Dacă prin regulile de mai sus se generează acțiuni în conflict (intrări multiplu definite ale tabelii *acțiune*) se spune că gramatica nu este SLR(1) iar algoritmul nu poate construi tabela de analiză sintactică.

3. Tranzițiile *salt* pentru starea i se vor construi pentru toate neterminalele A , utilizând următoarea regulă:
Dacă $\text{salt}(I_i, A) = I_j$ atunci *salt* $[i, A] = j$
(funcția) (tabela)
4. Toate intrările (elementele din tabele) nedefinite prin **regulile 2) și 3)** se consideră intrări de eroare.
5. Starea inițială a analizorului sintactic este construită din mulțimea de elemente care conține elementul $[S' \rightarrow S]$.

Exemplu: Construirea tabelii SLR pentru gramatica expresiilor din exemplele anterioare.

Colecția canonică de mulțimi de elemente LR(0) a fost determinată anterior. Mai întâi considerăm mulțimea de elemente I_0 : $E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot id$

Elementul $F \rightarrow \cdot (E)$ dă naștere intrării *acțiune* $[0, (] = d_4$ (**regula 2**) iar elementul $F \rightarrow \cdot id$ conduce la acțiune *acțiune* $[0, id] = d_5$ (**regula 2**). Celelalte elemente din I_0 nu produc acțiuni.

Considerăm apoi I_1 : $E' \rightarrow E \cdot$
 $E \rightarrow E \cdot + T$

Primul element produce *acțiune* $[1, \$] = \text{acceptă}$ (**regula 2.c**), prezentată mai sus) iar al doilea determină acțiune $[1, +] = d_6$.

În continuare se consideră I_2 : $E \rightarrow T \cdot$
 $T \rightarrow T \cdot * F$

Deoarece $\text{URM}(E) = \{ \$, +,) \}$, din primul element

\Rightarrow *acțiune* $[2, \$] = \text{acțiune}$ $[2, +] = \text{acțiune}$ $[2,)] = \text{reduce}$ $E \rightarrow T$.

Din al doilea element \Rightarrow *acțiune* $[2, *] = d_7$.

Continuând în acest mod se vor obține tabelele *acțiune* și *salt*.

Observație: Orice gramatică SLR(1) este neambiguă. Există însă multe gramatici neambigue care nu sunt SLR(1).

Ambianțe de execuție

În etapa premergătoare generării de cod trebuie realizate o serie de activități care implementează legăturile între textul static al programului sursă cu acțiunile care urmează să aibă loc la execuție. De exemplu, același nume din textul sursă poate reprezenta în execuție obiecte de date diferite iar codul obiect generat trebuie să le diferențieze fără echivoc.

Alocarea și relocarea obiectelor de date este realizată de **suportul de execuție**. Acesta constă într-o colecție de rutine încărcate împreună cu codul obiect generat. Tehnicile prezentate în continuare sunt utile la proiectarea unui astfel de suport pentru limbaje de programare ca Fortran, Pascal, C sau Lisp.

Fiecare execuție a unei proceduri se numește **activare** a procedurii. Dacă o procedură este recursivă, pot fi în execuție, în același timp, mai multe activări ale sale; fiecare activare în parte manipulează obiecte de date alocate special pentru ea.

Reprezentarea unui obiect de date în timpul execuției este conformă tipului său. Astfel, datele elementare (caractere, valori întregi sau reale) au reprezentări echivalente în mașina de calcul și se reprezintă direct prin acestea. Ansambluri de date cum sunt tablourile, șirurile sau structurile sunt reprezentate prin colecții de obiecte primitive cu o anumită dispunere în memorie de care trebuie să se țină cont la selectarea componentelor.

Probleme specifice derivate din structura și conținutul limbajului sursă

Se presupune că limbajul sursă include proceduri în forma din Pascal. Trebuie făcută distincție între textul sursă al unei proceduri și activările sale la execuție.

Proceduri

O **definiție de procedură** este o declarație care, în forma sa cea mai simplă, asociază un identificator cu o instrucțiune. Identificatorul este **numele procedurii** iar instrucțiunea este **corpul procedurii**.

Să considerăm următorul exemplu de program Pascal care citește și sortează un șir de numere întregi:

```
(1) program Sort;
(2)   var a: array [0 .. 10] of integer;
(3)   procedure CiteșteSir;
(4)     var i: integer;
(5)     begin
(6)       for i:=1 to 9 do read (a[i])
(7)     end;
(8)   function Partitionare (y, z: integer) : integer;
(9)     var i, j, x, v: integer;
(10)    begin ...
(11)  end;
(12)  procedure QuickSort (m, n: integer);
(13)    var i: integer;
(14)    begin
(15)      if (n>m) then begin
(16)        i:=Partitionare (n, m);
(17)        QuickSort (m, i-1);
(18)        QuickSort (i+1, m);
(19)      end;
(20)    end;
(21)  begin
(22)    a[0]:=-9999; a[10]:=9999;
(23)    CiteșteSir;
(24)    QuickSort (1, 9)
(25)  end.
```

Subprogramele care returnează valori prin numele lor sunt numite în mod uzual **funcții**. În cele ce urmează ele vor fi referite într-un mod similar procedurilor cu deosebirea că apelurile de funcții pot apărea în expresii, ca în linia 16. De asemenea și programul complet va fi tratat ca procedură. În programul exemplu sunt ilustrate principalele noțiuni legate de conceptul de procedură:

- definiția procedurii;
- apelul său;
- parametrii formali (în C – argumente formale iar, în Fortran – argumente fictive);
- argumentele (parametrii actuali).

Metodele de stabilire a corespondenței între parametrii formali și argumente sunt specifice limbajului de programare și vor fi discutate ulterior, în mod distinct.

Arbori de activare

În legătură cu fluxul comenzilor între proceduri pe parcursul execuției programului, se fac următoarele presupuneri:

1) Comenzile se execută secvențial: execuția unui program constă dintr-o secvență de pași, la fiecare pas controlul fiind într-un punct anume din program.

2) Fiecare execuție a unei proceduri debutează cu începutul corpului procedurii și, la sfârșit, returnează controlul la punctul imediat următor locului în care procedura a fost apelată \Rightarrow Fluxul controlului între proceduri poate fi ilustrat prin arbori.

Așa cum s-a arătat, fiecare execuție a unui corp de procedură se numește activare a procedurii. **Durata de viață** a unei activări a procedurii **p** este intervalul de timp consumat pentru execuția corpului procedurii, inclusiv timpul petrecut pentru executarea procedurilor apelate de **p**, a procedurilor apelate de acestea ș.a.m.d. În general, termenul de “durată de viață” se referă la o secvență consecutivă de pași în timpul execuției unui program.

În limbajele algoritmice uzuale, ca Pascal de exemplu, de fiecare dată când controlul intră într-o procedură “q”, apelată din procedura “p”, el se va întoarce în final la “p” (în absența unei erori fatale). Mai exact, de fiecare dată când comanda trece de la o activare a procedurii “p” la o activare a procedurii “q”, ea se va întoarce la aceeași activare a procedurii “p”.

Dacă **a** și **b** sunt activări de proceduri atunci duratele lor de viață sunt fie disjuncte fie suprapuse (încuibate): dacă se intră în **b** înainte de a părăsi **a**, atunci controlul trebuie să părăsească **b** înainte de a părăsi **a**. Această proprietate de încuibare a duratei de viață a activărilor poate fi ilustrată prin inserarea a două instrucțiuni de tipărire în fiecare procedură, una înainte de prima instrucțiune a corpului procedurii iar cealaltă după ultima instrucțiune. Prima tipărește **intrare** “nume procedură (argumente)” iar ultima tipărește **iesire** “nume procedură (argumente)”. Execuția programului **Sort** din exemplul anterior, cu aceste tipăriri incluse, determină afișarea următoarelor informații:

Începutul execuției

intrare CitesteSir

iesire CitesteSir

intrare QuickSort (1, 9)

intrare Partitionare (1, 9)

iesire Partitionare (1, 9)

intrare QuickSort (1, 3)

.....
iesire QuickSort (1, 3)

intrare QuickSort (5, 9)

.....
iesire QuickSort (5, 9)

iesire QuickSort (1, 9)

.....
iesire CitesteSir

.....
iesire CitesteSir

De exemplu, durata de viață a activării “QuickSort (1, 9)” este secvența de pași executați între tipărirea mesajului “**intrare** QuickSort (1, 9)” și cea a mesajului “**iesire** QuickSort (1, 9)”. În ceea ce privește procedura (funcția) **Partitionare**, s-a presupus că valoarea returnată de apelul “Partitionare (1,9)” este **4**.

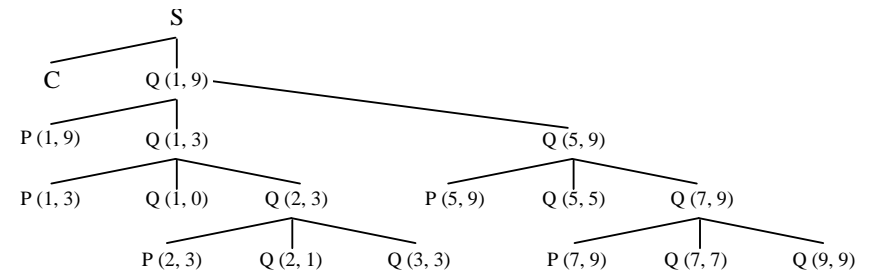
În termenii de mai sus, o procedură este **recursivă** dacă o nouă activare a ei poate începe înainte de încheierea unei activări anterioare. Exemplu: **QuickSort**, de mai sus. Apelul unei proceduri recursive poate fi direct (**p** apelează **p**) sau poate fi indirect (**p** apelează **q** care apelează, apoi, **p**).

Modul în care fluxul de control intră și iese din activările procedurilor poate fi ilustrat printr-un arbore numit **arbore de activare**, având următoarele proprietăți:

- 1) Fiecare nod reprezintă o activare a unei proceduri;
- 2) Rădăcina reprezintă activarea programului principal;
- 3) Nodul pentru **a** este părintele nodului pentru **b** dacă și numai dacă fluxul de control trece de la activarea lui **a** la **b**;
- 4) Nodul pentru **a** este la stânga nodului pentru **b** dacă și numai dacă durata de viață a lui **a** începe înaintea duratei de viață a lui **b**.

Deoarece fiecare nod reprezintă o unică activare și invers \Rightarrow controlul se află la un nod atunci când el este în activarea reprezentată de acel nod.

Exemplu: Arborele de activare corespunzător execuției programului **Sort** este următorul:



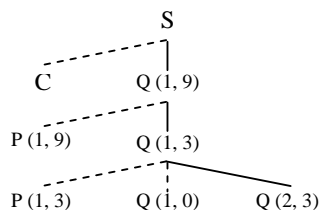
N-a fost ilustrată decât partea superioară a arborelui. Rădăcina corespunde întregului program **Sort**. În timpul execuției lui **Sort** există o activare a lui CitesteSir reprezentată de fiul rădăcinii cu eticheta C. Următoarea activare, reprezentată de al doilea fiu al rădăcinii, este pentru QuickSort cu argumentele 1 și 9. În timpul acestei activări, apelurile la Partitionare și QuickSort conduc la activările P(1, 9), Q(1, 3) și Q(5, 9). Se observă că activările Q(1, 3) și Q(5, 9) sunt recursive: ele încep înainte ca activarea Q(1, 9) să se încheie.

Stive de control

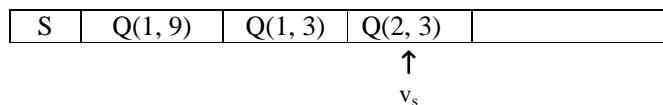
Fluxul controlului într-un program corespunde unei traversări spre adâncime a arborelui de activare corespunzător: începe de la rădăcină, vizitează un nod părinte înaintea fiilor săi iar fiii fiecărui nod sunt vizitați recursiv, de la stânga la dreapta. Prin traversarea în acest fel a arborelui de activare, tipărind **intrare ...** când nodul pentru o activare a fost atins prima dată și tipărind **iesire ...** la revenire, după ce a fost vizitat întregul subarboare al nodului, se poate reconstitui fișierul de afișare pentru execuția programului **Sort**.

Pentru a ține evidența activărilor **viabile** de proceduri, se poate utiliza o stivă, numită **stivă de control**. Principiul de lucru cu stiva este următorul: nodul pentru o activare se introduce în stivă când începe activarea și se extrage la terminarea ei. \Rightarrow Conținutul stivei de control este legat de căile la rădăcina arborelui de activare: când un nod **n** este în vârful stivei de control, stiva conține nodurile pe calea de la **n** la rădăcină.

Exemplu: Referindu-ne tot la programul **Sort** și la arborele de activare corespunzător, în momentul în care controlul intră în activarea reprezentată de nodul **Q(2, 3)**, nodurile din arborele de activare care au fost deja atinse sunt:



Activările cu etichetele C, P(1, 9), P(1, 3), Q(1, 0) au fost executate și terminate, legăturile lor în arbore fiind marcate cu o linie întreruptă. Liniile continue marchează drumul de la Q(2, 3) la rădăcină. În acest punct, stiva de control conține următoarele noduri (vârful stivei este la dreapta):



Stivele de control sunt utile la implementarea unor limbaje de programare ca Pascal sau C, pentru tehnicile de alocare a memoriei prin stivă.

Domeniul unei declarații

O declarație este o construcție sintactică a unui limbaj prin care unui nume **i** se asociază alte informații. Declarațiile pot fi explicite, ca în exemplul de mai jos (Pascal):

var i: integer;

sau pot fi implicite, ca în Fortran, unde **I** va desemna o variabilă întreagă, dacă nu se declară altfel în mod explicit.

În diferite părți ale unui program pot exista mai multe declarații, independente, ale aceluiași nume. Stabilirea declarației valabile într-un anumit punct din textul programului depinde de **regulile de domeniu** ale limbajului respectiv. De exemplu, în programul Pascal **Sort**, **i** este declarat de trei ori, în liniile 4, 9 și 13 iar utilizările numelui **i** în procedurile **CitesteSir**, **Partitionare** și **QuickSort** sunt independente una de alta \Rightarrow cele două apariții ale lui **i** în linia 6 sunt în domeniul declarației din linia 4 ș. a. m. d.

Porțiunea programului la care se aplică o declarație se numește **domeniul** acelei declarații. O apariție a unui nume într-o procedură se spune că este **locală** procedurii dacă este în domeniul unei declarații din cadrul procedurii; altfel, se spune că apariția este **nelocală**. Distincția între numele locale și nelocale se aplică la orice construcție sintactică ce poate conține în ea declarații.

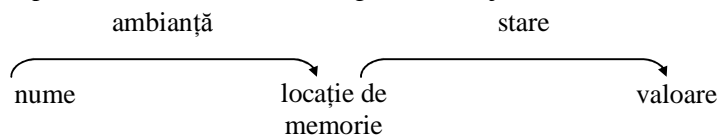
Deși domeniul este o proprietate a declarației unui nume, **x**, frecvent se utilizează abrevierea “domeniul numelui **x**”. De exemplu, domeniul lui **i** din linia 17 a programului **Sort** este corpul procedurii **QuickSort**.

La compilare, pentru a găsi declarația care se aplică unei apariții a unui nume, se va utiliza Tabela de simboluri (TS). Când se întâlnește o declarație, se creează pentru ea o intrare în TS. Cât timp, pe parcursul compilării, suntem în domeniul declarației, la căutarea, în tabelă, a numelui asociat declarației, se va returna această intrare.

Legarea numelor

Un nume poate denumi obiecte de date diferite în timpul execuției, chiar dacă fiecare nume este declarat numai o dată într-un program. Termenul informal de “obiect de date” corespunde unei locații de memorie care poate conține valori.

În semantica limbajelor de programare, termenul de **ambianță** se referă la o funcție care pune în corespondență un nume cu o locație de memorie iar termenul **stare** se referă la o funcție care pune în corespondență o locație de memorie cu valoarea înmagazinată în ea. Aceste corespondențe în două etape ale numelui cu valoarea corespunzătoare sunt ilustrate în figura de mai jos:



Utilizând termenii cunoscuți de **valoare-stânga** și **valoare-dreapta**, o ambianță pune în corespondență un nume cu o valoare-stânga iar o stare pune în corespondență valoarea-stânga cu o valoare-dreapta.

Ambianțele și stările sunt diferite; o asignare schimbă starea dar nu și ambianța.

Când o ambianță asociază o locație de memorie **s** cu un nume **x**, se spune că **x** este **legat** la **s**; asocierea însăși se numește **legarea** lui **x**. Termenul de “locație” de memorie trebuie luat în sens figurat: dacă tipul lui **x** nu este un tip de bază, memoria **s** pentru **x** poate fi o colecție de cuvinte de memorie.

O legare este corespondentul dinamic al unei declarații, ca în exemplele din tabelul de mai jos:

Noțiunea Statică	Corespondența Dinamică
definiția unei proceduri	activările procedurii
declararea unui nume	legările numelui
domeniul unei declarații	durata de viață a legăturii

Exemple: Mai multe activări ale unei proceduri recursive pot fi viabile în același timp. De asemenea, în Pascal, un nume de variabilă locală este legat la diferite locații de memorie, în fiecare activare a procedurii.

Rezumat al problemelor ce țin de limbajul de programare și influențează implementarea ambianței de execuție

Modul în care un compilator pentru un limbaj trebuie să-și organizeze memoria și să lege numele este determinat, în mare măsură, de răspunsurile la următoarele întrebări:

- 1) Procedurile se pot apela recursiv?
- 2) Ce se întâmplă cu valorile numelor locale când controlul revine dintr-o activare a unei proceduri?
- 3) Un nume nelocal poate fi referit într-o procedură?
- 4) Cum se transferă parametrii la apelul unei proceduri?
- 5) Procedurile pot fi transmise ca parametri?
- 6) Numele de proceduri pot fi returnate ca rezultate?
- 7) Memoria poate fi alocată dinamic, sub controlul programului?
- 8) Se poate face relocarea explicită a memoriei?

În continuarea acestui capitol se va prezenta efectul acestor probleme asupra suportului de execuție necesar pentru implementarea unui limbaj de programare dat.

Organizarea memoriei

Problemele privind organizarea memoriei la execuție au o legătură strânsă cu unele facilități ale limbajului de programare. Soluțiile tratate în acest subcapitol sunt potrivite pentru limbaje algoritmice de genul Fortran, Pascal sau C.

Subîmpărțirea memoriei de execuție

În principiu, zona de memorie necesară execuției unui program compilat poate fi împărțită, în funcție de destinația părților sale, astfel:

- 1) codul obiect generat de compilator;
- 2) obiectele de date ale programului;
- 3) parte corespunzătoare a *stivei de control*, necesară pentru evidența activărilor procedurilor.

Dimensiunea codului obiect generat este fixată la compilare \Rightarrow zona respectivă poate fi determinată static și plasată, de exemplu, la începutul memoriei disponibile pentru execuție. Similar, și dimensiunile unora dintre obiectele de date pot fi calculate la compilare \Rightarrow și acestea pot fi plasate într-o zonă determinată static, ca în figura de mai jos:

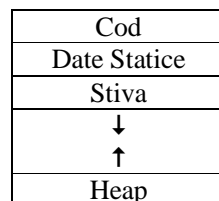


Fig. Împărțire tipică a memoriei de execuție în zone pentru cod obiect și pentru date (statistice și dinamice)

Deoarece adresele obiectelor statice pot fi completate direct în codul destinație, simplificând accesul la aceste obiecte în timpul execuției, există chiar un “interes” în favoarea alocării statice a cât mai multor obiecte ale programului. Unele limbaje (ex. Fortran) permit ca toate obiectele de date să fie alocate static.

Implementarea apelurilor recursive determină ca în alte limbaje (ex. Pascal, C) să fie necesară gestionarea explicită a apelurilor procedurilor sub forma unei extensii a stivei de control (stiva din figura anterioară). La apariția unui apel, se suspendă execuția activării curente și se salvează în stivă informații despre starea mașinii: valorile număratorului de instrucțiuni și ale registrelor. La revenirea din apel, se restaurează valorile registrelor, se poziționează număratorului de instrucțiuni la valoarea imediat următoare apelului și se continuă activarea suspendată. Obiectele de date ale căror durate de viață sunt conținute în cea a unei activări, pot fi alocate în stivă, împreună cu celelalte informații asociate cu activarea.

Memoria pentru execuție poate să includă și o zonă distinctă numită **heap** în care, printre altele, se înmagazinează datele dinamice: datele a căror alocare și eventual relocare se efectuează în mod explicit, prin programul utilizator. Tot în heap se pot păstra informații despre alocările procedurilor în implementările acelor limbaje în care duratele de viață ale activărilor nu pot fi reprezentate sub formă arborescentă. Modul controlat și simplu în care datele sunt alocate și relocate în stivă, face ca gestiunea datelor pe principiul stivei să fie mai eficientă decât gestionarea heap-ului.

Reprezentarea creșterii stivei și a heap-ului ca în figura anterioară (una spre cealaltă) permite ca dimensiunile lor să se poată modifica pe parcursul execuției programului, după necesități. Unele limbaje, cum sunt Pascal sau C, au nevoie la execuție atât de stivă cât și de heap. Prin convenție, stivele cresc în jos, adică spre adrese mari. Din motive de eficiență a implementării, vârful stivei, notat de obicei cu **top**, se păstrează într-un registru iar adresele în stivă se reprezintă ca deplasamente față de top.

Înregistrări de activare

Informațiile necesare pentru o singură execuție a unei proceduri sunt organizate într-un bloc contiguu de memorie numit **înregistrare de activare** sau **cadru**, având conținutul prezentat în figura de mai jos:

valoarea returnată
argumente
legătura opțională de control
legătura opțională de acces
starea salvată a mașinii
date locale
date temporare

Fig. Structura generală a unei înregistrări de activare

Nu orice implementare de limbaj utilizează toate aceste câmpuri. Unele dintre ele se pot înmagazina în registre. Pentru limbaje ca Pascal sau C, înregistrarea de activare se creează și se introduce în stiva de execuție la apelul procedurii iar extragerea din stivă și desființarea ei se realizează la încheierea apelului, atunci când controlul revine la apelant.

Rolul câmpurilor unei înregistrări de activare este următorul (de jos în sus):

1) **Date temporare**: locații pentru valori temporare care apar în procesul de calcul din procedură (de exemplu la evaluarea expresiilor).

2) **Date locale**: datele care sunt locale unei execuții a procedurii.

3) **Starea salvată a mașinii**: informații despre starea mașinii imediat înainte de apelul procedurii; include valorile contorului de program și ale registrelor, valori care trebuie restaurate atunci când controlul revine din alt apel de procedură.

4) **Legătura opțională de acces**: se utilizează pentru referirea datelor nelocale, conținute în alte înregistrări de activare. Această legătură nu este necesară în limbaje ca Fortran deoarece datele nelocale se păstrează într-un loc fix.

5) **Legătura opțională de control**: indică spre înregistrarea de activare a apelantului.

6) **Argumentele**: sunt utilizate de apelant pentru a furniza parametrii procedurii apelate. Pentru eficiența accesului, în practică, parametrii pot fi înmagazinați și în registre.

7) **Valoarea returnată**: este utilizată de procedura apelată pentru a returna procedurii apelante o valoare (apel de funcție). Pentru creșterea eficienței și această valoare poate fi returnată într-un registru.

În mod normal, dimensiunile acestor câmpuri se pot determina în timpul compilării. Ele sunt necesare în momentul execuției, atunci când este apelată procedura. Un caz mai deosebit apare dacă procedura are un tablou local a cărui dimensiune este determinată de valoarea unui argument disponibil, la execuție, abia în momentul apelului.

Organizarea datelor locale la compilare

Presupunem că memoria de execuție este disponibilă în blocuri contigue de octeți. Un anumit număr de octeți formează un cuvânt-calculator. Obiectele multiocet sunt memorate în octeți consecutivi și primesc adresa primului octet.

Cantitatea de memorie necesară pentru un **nume** este determinată de tipul său. Un tip de date elementar (caracter, întreg, real) poate fi memorat într-un număr întreg de octeți. Memoria pentru o structură de date (tablou, înregistrare) trebuie să înmagazineze toate componentele sale. Pentru acces ușor la componente, este de dorit ca și zona de memorie pentru o structură să fie alocată într-un bloc contiguu de octeți.

Zona pentru datele locale este organizată la compilare, pe măsură ce se examinează declarațiile dintr-o procedură. Datele de lungime variabilă sunt păstrate în afara acestei zone. Se păstrează un contor al locațiilor de memorie care au fost alocate pentru declarațiile precedente. Din contor se determină adresa relativă a memoriei pentru un obiect local în raport cu începutul zonei sau cu începutul înregistrării de activare.

Organizarea memoriei pentru obiectele date este puternic influențată de limitările de adresare ale calculatorului destinație. De exemplu, instrucțiunile pentru adunarea întregilor ar putea impune ca întregii să fie **aliniați**, adică să fie plasați la anumite poziții în memorie astfel încât adresele lor să fie divizibile cu 4. Din considerente de aliniere, pentru un tablou de 10 caractere s-ar putea alocă 12 octeți (în loc de 10 octeți necesari), lăsând doi octeți neutilizați. Spațiul rămas neutilizat din considerente de aliniere se numește “umplură”. Când memoria disponibilă pentru execuție este resursă critică, compilatorul poate **împacheta** datele astfel încât să nu rămână “umplură”. Împachetarea ar putea necesita executarea unor instrucțiuni suplimentare care să poziționeze datele împachetate astfel încât să se poată opera asupra lor ca și când ar fi aliniate corespunzător.

Exemplu: În tabelul de mai jos se prezintă în mod simplificat organizarea datelor utilizată de compilatoare C pentru două calculatoare numite **Calc.1** și **Calc.2**. C permite întregi de trei dimensiuni, declarați cu cuvintele cheie **short**, **int** și **long**. Setul de instrucțiuni pentru cele două calculatoare impune ca în Calc.1 să se aloce 16, 32 și 32 de biți pentru cele trei tipuri de întregi iar în Calc.2 se vor aloca 24, 48 și respectiv 64 de biți. Cu toate că nici un calculator nu permite adresarea directă a biților, pentru compararea între calculatoare, dimensiunile din tabel se dau în biți.

TIP	DIMENSIUNE (biți)		ALINIERE (biți)	
	Calc.1	Calc.2	Calc.1	Calc.2
char	8	8	8	64
short	16	24	16	64
int	32	48	32	64
long	32	64	32	64
float	32	64	32	64
double	64	128	32	64
pointer la char	32	30	32	64
alți pointer	32	24	32	64
structuri	≥8	≥64	32	64

Fig. Organizarea datelor utilizată în două compilatoare C

Memoria Calc.1 este organizată pe octeți. Deși fiecare octet are o adresă, setul de instrucțiuni mașină cere ca întregii scurți să fie poziționați la adrese pare iar întregii la adrese multiplu de 4. Compilatorul va respecta aceste reguli chiar dacă trebuie să sară peste 1÷3 octeți. De exemplu, pentru un caracter urmat de un întreg scurt se vor alocă 4 octeți deși este necesar doar unul.

În Calc.2 fiecare cuvânt constă din 64 de biți, iar pentru adresarea unui cuvânt sunt permisi 24 biți. Pentru biții individuali în interiorul unui cuvânt există 64 de posibilități de plasare \Rightarrow sunt necesari 6 biți suplimentari ($2^6=64$) pentru a distinge între ele. \Rightarrow Un pointer la un caracter, plasat oriunde în interiorul cuvântului, necesită 30 biți: 24 pentru a găsi cuvântul și 6 pentru poziția caracterului în cuvânt, ceea ce ar fi destul de complicat de implementat.

Puternica orientare pe cuvinte a setului de instrucțiuni al Calc.2 determină compilatorul să aloce un cuvânt complet chiar și în situațiile în care, pentru reprezentarea tuturor valorilor aceluși tip, ar fi suficienți mai puțini biți. De exemplu, pentru un caracter sunt necesari 8 biți, din motive de aliniere se vor alocă 64 biți. În interiorul unui cuvânt biții pentru fiecare tip de bază sunt în poziții specificate (de exemplu la început sau la sfârșit). Alt exemplu: pentru un caracter urmat de un întreg scurt se vor alocă două cuvinte, cumulând 128 biți; pentru un caracter se utilizează efectiv 8 biți din primul cuvânt iar pentru întregul scurt se utilizează 24 biți din al doilea cuvânt.

Strategii de alocare a memoriei

Fiecare din cele trei zone de date necesită o strategie de alocare proprie. Astfel:

- 1) **Alocarea statică** stabilește, la compilare, memoria pentru toate obiectele de date statice.
- 2) **Alocarea din stivă** administrează memoria de execuție ca o stivă.
- 3) **Alocarea din heap** alocă și relocă memoria când este necesar, în timpul execuției, din zona de date numită **heap**.

Aceste strategii de alocare sunt aplicate simultan, pentru aceeași înregistrare de activare. Interesează, de asemenea, felul în care se realizează accesul la un nume local din codul destinație al unei proceduri.

Alocarea statică

La alocarea statică numele sunt legate de locațiile de memorie asociate la compilarea programului \Rightarrow nu este necesar un pachet suport de execuție. Pe parcursul execuției, legăturile nu se schimbă la fiecare activare a procedurii \Rightarrow numele sunt legate la aceleași locații de memorie. Această proprietate permite ca valorile numelor să se păstreze de la o activare la alta: când controlul revine la o procedură, valorile sunt aceleași care au fost când controlul a părăsit-o ultima dată.

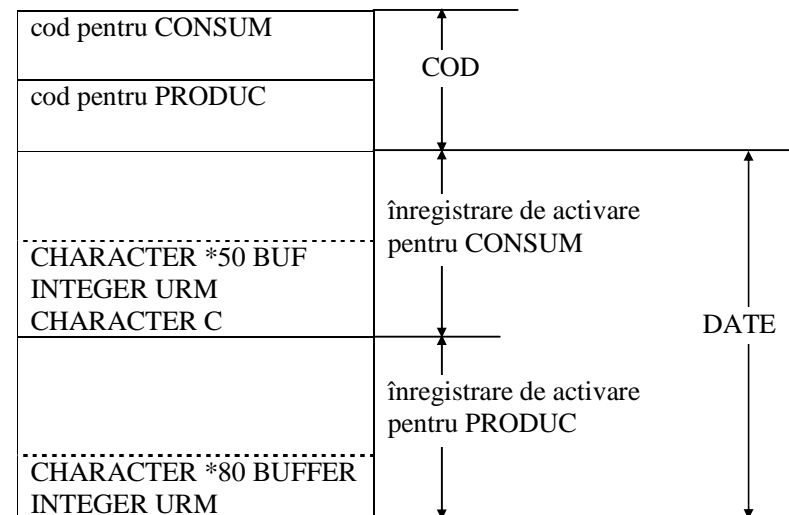
Din tipul unui nume, compilatorul determină dimensiunea locației de memorie pentru acel nume. Adresa acestei locații constă dintr-un deplasament față de un capăt al înregistrării de activare pentru procedură. Compilatorul trebuie, în final, să decidă unde se pun înregistrările de activare relativ la codul destinație precum și una față de alta. Odată ce a fost luată această decizie, poziția fiecărei înregistrări de activare și, implicit, a locației fiecărui nume local, a fost fixată. \Rightarrow La compilare, se pot completa în codul destinație adresele datelor locale. Similar, sunt de asemenea cunoscute la compilare, adresele la care trebuie salvate informații când apare un apel de procedură.

Alocare statică introduce însă și unele limitări:

- 1) Dimensiunea unui obiect de date și restricțiile asupra poziției sale în memorie, trebuie să fie cunoscute la compilare.
- 2) Deoarece toate activările unei proceduri utilizează aceleași legări pentru numele locale, apar restricții privind apelurile recursive ale procedurilor.
- 3) Neexistând nici un mecanism pentru alocarea memoriei la execuție, structurile de date nu pot fi create dinamic.

Un limbaj proiectat pentru a permite alocarea statică a memoriei este Fortran. Un program Fortran constă într-un program principal, subrutine și funcții (le vom numi pe toate proceduri) ca în exemplul de mai jos (Fortran 77):

```
(1) PROGRAM CONSUM
(2) CHARACTER *50 BUF
(3) INTEGER URM
(4) CHARACTER C, PRODUC
(5) DATA URM /1/, BUF /' '/
(6) C=PRODUC ( )
(7) BUF (URM : URM)=C
(8) URM=URM+1
(9) IF (C.NE. ' ') GOTO 6
(10) WRITE (*,'(A)') BUF
(11) END
(12) CHARACTER FUNCTION PRODUC ( )
(13) CHARACTER *80 BUFFER
(14) INTEGER URM
(15) SAVE BUFFER, URM
(16) DATA URM /81/
(17) IF (URM.GT.80) THEN
(18) READ (*,'(A)') BUFFER
(19) URM=1
(20) END IF
(21) PRODUC=BUFFER (URM : URM)
(22) URM=URM+1
(23) END
```



În cadrul înregistrării de activare pentru CONSUM, există spațiu pentru variabilele locale BUF, URM și C. Locația de memorie legată la BUF conține un șir de 50 caractere. Ea este urmată de locația pentru URM în care se poate înregistra o valoare întreagă și o locație, C, pentru un caracter.

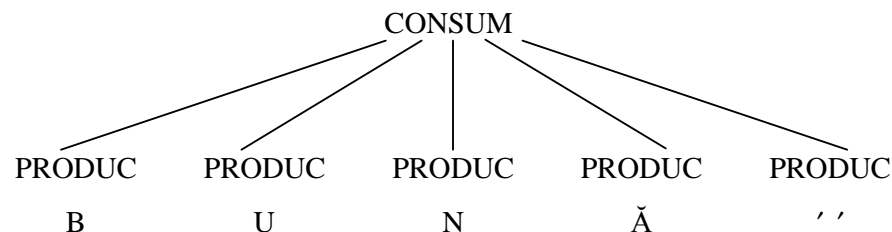
Faptul că URM este declarat și în PRODUC nu prezintă nici o problemă deoarece cele două variabile locale fac parte din înregistrări de activare diferite.

Dimensiunile codului executabil și ale înregistrărilor de activare sunt cunoscute la compilare \Rightarrow sunt posibile și alte organizări ale memoriei decât cea de mai sus. De exemplu, compilatorul ar putea plasa înregistrarea de activare pentru o procedură împreună cu codul pentru acea procedură. În alt caz, s-ar putea lăsa nespecificată poziția relativă a înregistrării de activare, permițând editorului de legături să lege înregistrarea de activare și codul executabil.

Exemplu: Logica programului anterior se bazează pe faptul că valorile variabilelor locale se păstrează de la o activare la alta. Acest lucru este exprimat de instrucțiunea SAVE care specifică faptul că valoarea unei variabile locale la începutul unei activări trebuie să fie aceeași ca și la sfârșitul activării precedente. Valorile inițiale ale variabilelor locale pot fi specificate utilizând instrucțiunea DATA.

Instrucțiunea READ din linia 18 a procedurii PRODUC citește o linie de text într-un tampon. De fiecare dată când este activată, procedura furnizează caractere succesive din acest tampon. Programul principal CONSUM are, de asemenea, un tampon în care acumulează caractere până la apariția unui spațiu. Graful de mai jos reprezintă caracterele returnate de activările lui PRODUC pentru șirul de intrare.

BUNĂ ZIUA



Ieșirea programului, tipărită în linia 10, este: BUNĂ.

Tamponul în care PRODUC citește linii trebuie să-și păstreze conținutul între activări. Acest lucru este asigurat de instrucțiunea SAVE din linia 15, atât pentru BUFFER cât și pentru URM. La primul apel al lui PRODUC, valoarea variabilei locale URM este precizată de instrucțiunea DATA din linia 16 \Rightarrow URM este inițializat pe 81, ceea ce asigură citirea primei linii.

Alocarea din stivă

În acest caz memoria este organizată ca o stivă iar înregistrările de activare sunt introduse și extrase pe măsură ce încep și respectiv se termină activările procedurilor. La fiecare apel al procedurii, memoria pentru variabilele locale este conținută în înregistrarea de activare pentru acel apel. În acest fel, variabilele locale le corespund, la fiecare apel, locații de memorie noi. În plus, valorile variabilelor locale sunt **șterse** la încheierea activării, când înregistrarea de activare, care include locațiile de memorie, se extrage din stivă.

Se va descrie, la început, o formă de alocare din stivă în care dimensiunile tuturor înregistrărilor de activare se presupun cunoscute la compilare. Vârful stivei va fi marcat de registrul **top**. La execuție, alocarea și relocarea unei înregistrări de activare se realizează incrementând și respectiv decrementând **top** cu dimensiunea înregistrării: dacă procedura “q” are o înregistrare de activare de dimensiune **a**, atunci, înainte de a se executa codul lui “q”, **top** este incrementat cu **a**; la închiderea apelului, când controlul revine din “q”, **top** este decrementat cu **a**.

Exemplu: Să considerăm, din nou, unul din programele anterioare și arborele său de activare. Pe măsură ce controlul parcurge acest arbore de activare, în stiva de execuție se introduc și se extrag înregistrările de activare.

Poziția în Arborele de activare	Înregistrarea de activare în stivă	Observații
S	<div>S</div> <div>a : array</div>	a fost creată înregistrarea de activare pentru S
<div>S</div> <div>C</div>	<div>S</div> <div>a : array</div> <div>C</div> <div>i : integer</div>	a fost activat C
<div>S</div> <div>C</div> <div>Q(1,9)</div>	<div>S</div> <div>a : array</div> <div>Q(1,9)</div> <div>i : integer</div>	a fost extrasă înregistrarea de activare pentru C și s-a introdus în stivă cea pentru Q(1,9)
<div>S</div> <div>C</div> <div>Q(1,9)</div> <div>P(1,9)</div> <div>Q(1,3)</div> <div>P(1,3)</div> <div>Q(1,0)</div>	<div>S</div> <div>a : array</div> <div>Q(1,9)</div> <div>i : integer</div> <div>Q(1,3)</div> <div>i : integer</div>	stiva în momentul în care controlul a revenit de la Q(1,0) la Q(1,3)

Liniile întrerupte din arbore merg la activări care s-au încheiat. Execuția începe cu activarea procedurii S (programul principal **sort**). Când controlul ajunge la primul apel din corpul lui S, este activată procedura C și se introduce în stivă înregistrarea de activare pentru ea. Când controlul revine din această activare, înregistrarea este extrasă, rămânând în stivă doar înregistrarea pentru S. Apoi controlul ajunge la apelul lui Q(1,9) și se introduce în vârful stivei înregistrarea de activare corespunzătoare. Ori de câte ori controlul este într-o anumită activare, înregistrarea de activare pentru ea este în vârful stivei. Între penultimul și ultimul rând din tabel s-au derulat mai multe momente, timp în care au început și s-au încheiat activările P(1,9), P(1,3) și Q(1,0). Înregistrările de activare corespunzătoare lor au fost introduse și, ulterior, au fost extrase din stivă, lăsând în vârful stivei activarea pentru Q(1,3) de unde s-au declanșat apelurile P(1,3) și Q(1,0).

În ceea ce privește accesul la datele locale într-o înregistrare de activare, acesta se poate realiza printr-un deplasament specific fiecărei date în parte raportat, de exemplu, la sfârșitul înregistrării de activare, marcată de registrul **top**. Adresele variabilelor locale pot fi însă calculate ca deplasament față de orice alt registru care indică spre o poziție fixă din înregistrarea de activare.

Secvențele de apel și secvențele de revenire

Secvențele de apel (SA) sunt generate în codul destinație pentru a implementa apelurile de proceduri. O SA alocă înregistrarea de activare și completează cu informații câmpurile sale. Restaurarea stării calculatorului la încheierea apelului, astfel ca să poată continua execuția procedurii apelante, este realizată de **secvența de revenire** (SR).

Secvențele de apel și înregistrările de activare diferă chiar pentru implementările aceleiași limbaj. Codul corespunzător SA este deseori împărțit între procedura apelantă și cea apelată. Nu există o împărțire exactă a sarcinilor la execuție între apelant și apelat. Limbajul sursă, mașina destinație și sistemul de operare impun cerințe care pot favoriza o soluție sau alta.

Un principiu care ajută atât în proiectarea SA cât și a înregistrării de activare este plasarea la mijlocul înregistrării a câmpurilor a căror dimensiune este cunoscută devreme. De exemplu, în înregistrarea de activare generală sunt plasate la mijloc legăturile opționale de control și acces și starea calculatorului. Decizia dacă să se utilizeze sau nu legăturile de control și acces este luată de proiectantul compilatorului ⇒ aceste câmpuri pot fi fixate în timpul construirii compilatorului. De asemenea, în măsura în care cantitatea de informații de stare a calculatorului este constantă, salvarea și restaurarea stării calculatorului este aceeași pentru toate activările. Această uniformizare poate să fie utilizată și de un eventual depanator de programe, pentru a descifra și afișa conținutul stivei, în cazul apariției unei erori.

Deși dimensiunea câmpului pentru variabilele temporare este, în final, fixată în timpul compilării, această dimensiune poate să nu fie cunoscută în partea de front-end. Generarea sau optimizarea de cod poate să reducă numărul de variabile temporare ⇒ În structura generală a înregistrării de activare acest câmp a fost plasat la sfârșit, după cel pentru date locale, astfel ca dimensiunea lui să nu afecteze deplasamentele corespunzătoare obiectelor de date locale.

Fiecare apel are propriile sale argumente care sunt evaluate de apelant și sunt comunicate apelatului. În stiva de execuție, înregistrarea de activare a apelantului este chiar sub cea a apelatului, ca în figura de mai jos ⇒ Este avantajos să se plaseze câmpurile pentru argumente și pentru eventuala valoare returnată imediat după înregistrarea de activare a apelantului și în debutul celei a apelatului. În acest fel apelantul poate avea acces la aceste câmpuri utilizând deplasamentul din stivă al sfârșitului propriei înregistrări fără a cunoaște organizarea completă a înregistrării apelatului. De altfel, nu există nici un motiv ca apelantul să cunoască datele locale sau temporare ale apelatului. O consecință benefică a acestei ascunderi a informațiilor este prelucrarea procedurilor cu un număr variabil de parametri (read și write din Pascal, printf din C etc).

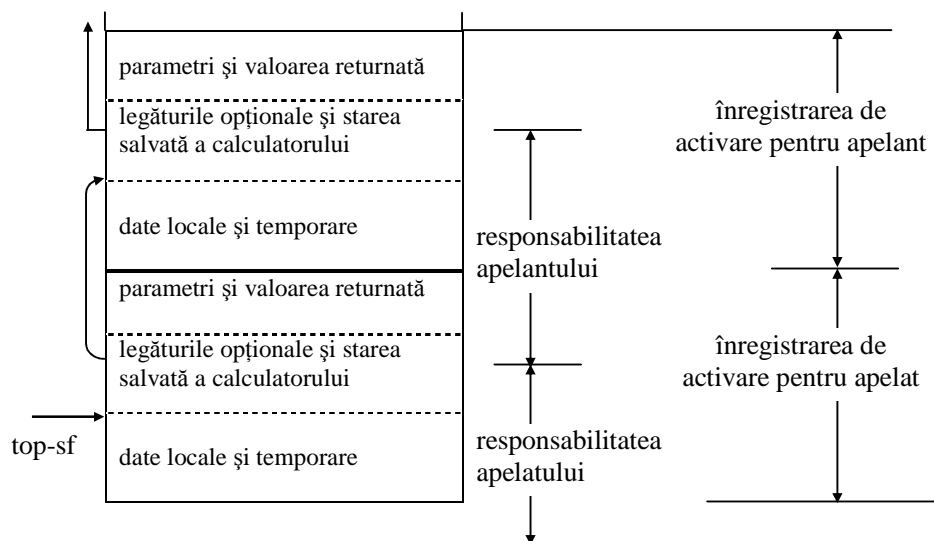


Fig. Divizarea sarcinilor între apelant și apelat

Pascal este unul dintre puținele limbaje în care se cere ca tablourile locale unei proceduri să aibă o lungime fixă, determinabilă la compilare. În alte limbaje, dimensiunea unui tablou local poate depinde de valoarea unui argument. În acest din urmă caz, dimensiunea zonei datelor locale procedurii nu poate fi determinată până în momentul apelului (în execuție). Tehnicile pentru tratarea datelor de lungime variabilă vor fi prezentate ulterior.

În figura de mai sus, registrul **top-sf** indică la sfârșitul câmpului de stare calculatorului din înregistrarea de activare. Această poziție este cunoscută apelantului care, astfel, poate și realiza poziționarea **top-sf**, înainte de a transfera controlul procedurii apelate. Codul pentru apelat poate realiza accesul la datele sale locale și temporare utilizând deplasamentul față de **top-sf**.

Cu aceste precizări, SA poate consta în următoarele operații:

- 1) Apelantul evaluează argumentele.
- 2) Apelantul memorează adresa de revenire și vechea valoare a lui **top-sf** în înregistrarea de activare a apelatului. Apelantul incrementează apoi **top-sf** la noua poziție, ca în figura de mai sus. Astfel **top-sf** trece peste datele locale și temporare ale apelantului și peste argumente și starea salvată a calculatorului, corespunzătoare apelatului.
- 3) Apelatul salvează valoarea registrelor și alte informații de stare.
- 4) Apelatul inițializează datele sale locale și începe execuția.

O posibilă SR poate fi următoarea:

- 1) Apelatul plasează o eventuală valoare returnată după înregistrarea de activare a apelantului.
- 2) Utilizând informațiile din starea salvată, apelatul restaurează **top-sf** și alte registre și efectuează saltul în codul apelantului, conform adresei de revenire.
- 3) Deși **top-sf** a fost decrementat, apelantul are acces și poate utiliza valoarea returnată în evaluarea unei expresii proprii.

SA prezentată mai sus permite ca numărul de argumente pentru o procedură să depindă de apel (să fie variabil). La compilare, codul destinație al apelantului cunoaște numărul argumentelor pe care le furnizează apelatului. În schimb, codul destinație al apelatului trebuie să fie pregătit să trateze și alte apeluri \Rightarrow așteaptă să fie apelat și apoi examinează câmpul parametrilor. Utilizând organizarea din figura de mai sus, informațiile ce descriu parametrii trebuie plasate lângă câmpul de stare, astfel ca apelatul să le poată găsi și identifica. De exemplu, considerând funcția standard printf din C, primul argument al lui printf specifică natura restului parametrilor \Rightarrow odată ce printf a localizat primul argument le poate găsi și pe celelalte.

Date de lungime variabilă

O strategie uzuală de tratare a datelor de lungime variabilă este sugerată în figura următoare, considerând o procedură “p” care are trei tablouri locale. Locațiile de memorie pentru aceste tablouri nu sunt incluse în înregistrarea de activare pentru “p”. Aceasta conține doar câte un pointer la începutul fiecărui tablou. Adresele relative ale locațiilor acestor pointeri sunt cunoscute la compilare ⇒ codul destinație poate avea acces la tablouri prin acești pointeri

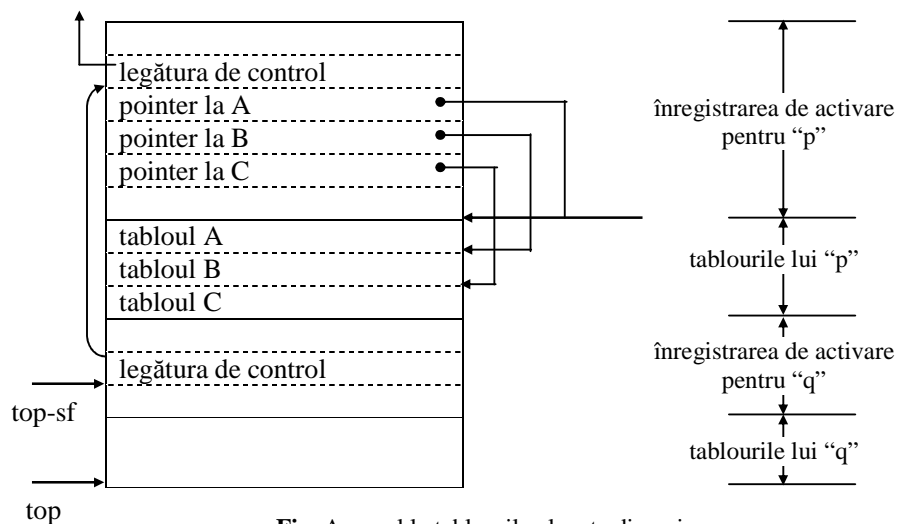


Fig. Accesul la tablourile alocate dinamic

În figura de mai sus apare și o procedură “q”, apelată din “p”. Înregistrarea de activare pentru “q” începe după tablourile lui “p” iar eventualele tablouri de lungime variabilă ale lui “q” sunt plasate după aceasta.

Accesul la datele din stivă se realizează prin doi pointeri:

- **top**: marchează vârful efectiv al stivei; indică spre poziția la care va începe următoarea înregistrare de activare;
- **top-sf**: punctează datele locale.

Pentru consistență între ultima figură și cea anterioară, presupunem că **top-sf** indică sfârșitul câmpului de stare a calculatorului din înregistrarea de activare a procedurii active (q), iar legătura opțională de control din această înregistrare marchează valoarea anterioară a lui **top-sf**: corespunzătoare situației când controlul este în procedura “p”.

Pe baza dimensiunilor câmpurilor din înregistrarea de activare, cunoscute la compilare, codul pentru repoziționarea lui **top** și **top-sf** se poate genera în timpul compilării. Astfel, la revenirea din “q”, noua valoare a lui **top** este **top-sf** minus lungimea câmpurilor pentru salvarea stării calculatorului și cel pentru parametrii lui “q”. Această lungime este cunoscută la compilare, cel puțin apelantului (p). După actualizarea lui **top**, noua valoare a lui **top-sf** poate fi copiată din legătura de control a lui “q”.

Referiri suspendate

Ori de câte ori se poate reloca memorie, apare problema referirilor suspendate. O **referire suspendată** este o referire din program la o zonă de memorie care a fost între timp relocată. Deoarece valoarea memoriei relocate este nedefinită, utilizarea unei referiri suspendate este o eroare logică în majoritatea limbajelor de programare. Deoarece memoria relocate poate fi ulterior alocată unei alte date, în programele cu referiri suspendate pot apărea erori dificil de depistat, cu manifestări aleatoare.

Exemplu: Să considerăm următorul program C:

```
main ()
{
    int *p
    p=eroare ();
}
int *eroare ()
{
    int i=23;
    return & i;
}
```

Procedura **eroare** returnează un pointer la memoria legată de numele local **i**. Pointerul este creat de operatorul **&** aplicat lui **i**. Când controlul revine din **eroare** la **main** memoria pentru datele locale este eliberată și poate fi utilizată în alte scopuri. Deoarece **p** din **main** se referă la această memorie, utilizarea lui **p** este o referință suspendată.

Referințele suspendate se pot crea, evident, și prin relocarea dinamică a memoriei, sub controlul programului.

Alocarea din heap

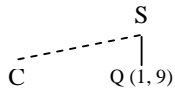
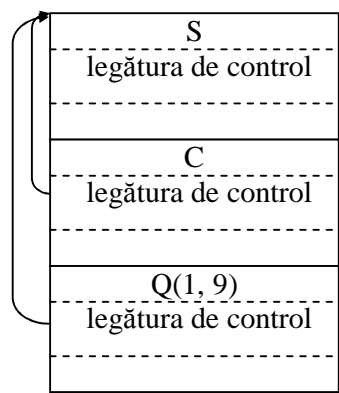
Strategia de alocare din stivă prezentată anterior nu poate fi utilizată dacă apare una din următoarele situații:

- 1) Valorile numelor locale trebuie păstrate când se termină activarea procedurii.
- 2) o activare apelată trăiește mai mult decât apelantul. Această posibilitate nu poate să apară pentru acele limbaje în care fluxul controlului între proceduri este descris prin arbori.

În fiecare din cazurile de mai sus, relocarea memoriei nu respectă ordinea LIFO \Rightarrow memoria nu poate fi organizată ca stivă.

Alocarea din **heap** partiționează porțiuni de memorie contiguă, necesare atât pentru înregistrări de activare cât și pentru alte obiecte. Porțiunile pot fi relocate în orice ordine \Rightarrow în timp, memoria **heap** va consta în zone alternative, libere și utilizate(ocupate).

Diferența între alocarea din stivă și din **heap** a înregistrărilor de activare poate fi urmărită prin comparație între tabelul anterior și următorul:

Poziția în Arborele de activare	Înregistrări de activare în heap	Observații
		

În cazul alocării din **heap**, înregistrarea pentru o activare a procedurii C este păstrată și după terminarea activării. \Rightarrow Înregistrarea pentru noua activare Q(1, 9) nu o va urma, fizic, pe cea pentru S, așa cum s-a întâmplat în cazul anterior. Dacă, ulterior, înregistrarea lui C va fi relocată, va apărea un spațiu liber, în **heap**, între înregistrările pentru S și Q(1, 9). Rămâne ca administratorul **heap**-ului să decidă cum să utilizeze acest spațiu.

Problema administrării eficiente a **heap**-ului este un subiect specializat în teoria structurilor de date. În general, utilizarea unui administrator de **heap** presupune regie de timp și spațiu. Din motive de eficiență poate fi util să se trateze ca și cazuri speciale înregistrările de dimensiuni mici sau cele de dimensiuni predictibile, astfel:

- 1) Pentru fiecare dimensiune de interes, se păstrează o listă de blocuri libere de acea dimensiune.
- 2) Dacă este posibil, se va satisface o cerere de dimensiune s , cu un bloc de dimensiune s' , unde s' este cea mai mică dimensiune de blocuri disponibile $\geq s$. În cazul relocării, blocul va fi returnat la lista din care a provenit.
- 3) Pentru blocuri mari de memorie, se utilizează administratorul general de **heap**.

Această abordare conduce la alocarea și relocarea rapidă a cantităților mici de memorie, deoarece extragerea și respectiv introducerea unui bloc dintr-o listă înlănțuită sunt operații eficiente. Pentru cantități mari de memorie, este probabil ca timpul de calcul în care se utilizează acea memorie să fie suficient de mare, astfel ca timpul necesar administratorului să poată fi neglijat în comparație cu acesta.

Bibliografie

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: Compilers. Principles, Techniques and Tools, Second edition, Addison-Wesley Publishing Company, 2007
2. Dick Grune, Henri E. Bal, Criel J.H. Jacobs, Koen Langendoen: Modern Compiler Design, John Wiley, 2003
3. Horia Ciocârlie: Limbaje de programare. Concepte fundamentale, Editura de Vest, 2007
4. Irina Athanasiu: Limbaje formale și compilatoare, Universitatea Politehnica București, 1992
5. Luca-Dan Șerbănați: Limbaje de programare și compilatoare, Editura Academiei, București, 1987
6. Teodor Rus: Mecanisme formale pentru specificarea limbajelor, Editura Academiei, 1983
7. David A. Watt, Deryck F. Brown: Programming Language Processors in Java - Compilers and Interpreters, Prentice Hall, 2000
8. Gabriel V. Orman: Limbaje formale, Editura Tehnică, București, 1982
9. Alexandru Mateescu, Dragoș Vaida: Structuri matematice discrete. Aplicații, Editura Academiei, București, 1989
10. Carmen De Sabata: Limbaje formale și translaatoare - îndrumător de laborator, Casa Cărții de Știință, Timișoara, 1999
11. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Șabloane de proiectare, Editura Teora, 2002