

# Analiza lexicala

---

Fie urmatoarea specificatie lexicala:

- (1)  $(ab|ba)^*aa$
- (2)  $abba$
- (3)  $bab$

**Care este secventa de reguli lexicales (1-3) utilizate in analiza urmatorului sir  $abbaabbabababbaaababaa$ ?**

Tinem mereu cont de cele 2 reguli de baza:

- maximal munch - consumam cel mai lung subsir care este cuprins de o regula;
- daca doua reguli cuprind un subsir, se utilizeaza prima regula definita.

Astfel, parcurgem sirul de la intrare token cu token cat timp exista cel putin o regula care (mai poate) cuprinde subsirul obtinut pana in acel punct. Daca la final ramane o singura regula, ea cuprinde subsirul obtinut. Daca sunt mai multe reguli, aplicam cele doua reguli de baza de mai sus. Apoi reluam procesul din punctul ramas spre sfarsitul sirului.

In cazul in care toate regulile au esuat la un moment dat, intreaga analiza a sirului esueaza.

In continuare sunt reprezentati in prima coloana sirul, cursorul care parcurge sirul alaturi de cursoarele regulilor in cadrul sirului, iar in a doua coloana regulile lexicales care au recunoscut un subsir. Cand o regula nu mai poate recunoaste un subsir, cursorul acesteia dispare. Subsirurile recunoscute au regula corespunzatoare ca indice. Pentru a evidentia intalnirea sfarsitului sirului, s-a folosit caracterul \$.

$ ^{123} abbaabbabababbaaababaa$	-
$a ^{12} bbaabbabababbaaababaa$	-
$ab ^{12} baabbabababbaaababaa$	-
$abb ^{12} aabbabababbaaababaa$	-
$abba ^{12} abbabababbaaababaa$	2
$abba ^{2} a ^{1} bbabababbaaababaa$	2
$abba ^{2} ab ^{1} babababbaaababaa$	2
$abba ^{2} abb ^{1} abababbaaababaa$	2
$abba ^{2} abba ^{1} bababbaaababaa$	2
$abba ^{2} abbab ^{1} ababbbaaababaa$	2
$abba ^{2} abbaba ^{1} babbbaaababaa$	2
$abba ^{2} abbabab ^{1} abbaaababaa$	2
$abba ^{2} abbababa ^{1} bbaaababaa$	2
$abba ^{2} abbababab ^{1} baaababaa$	2
$abba ^{2} abbabababb ^{1} aaababaa$	2

Singura regula care a recunoscut o portiune din sir este regula **2**. Chiar daca regula **1** a consumat mult mai mult din sir decat regula **2**, aceasta ajunge la esec, intrucat regula incepuse sa consume din sir cu

subregula *ba*, dar token-ii de la intrare erau *bb*, astfel ca nu poate fi recunoscut subsirul.

Se utilizeaza, deci, regula **2** care recunoaste **abba** din sirul de la intrare si analiza continua din pozitia cursorului regulii.

abba <sub>2</sub>	<sup>123</sup> abbabababbaaababaa	-
abba <sub>2</sub>	a   <sup>12</sup> bbabababbaaababaa	-
abba <sub>2</sub>	ab   <sup>12</sup> babababbaaababaa	-
abba <sub>2</sub>	abb   <sup>12</sup> abababbaaababaa	-
abba <sub>2</sub>	abba   <sup>12</sup> bababbaaababaa	2
abba <sub>2</sub>	abba   <sup>2</sup> b   <sup>1</sup> ababbaaababaa	2
abba <sub>2</sub>	abba   <sup>2</sup> ba   <sup>1</sup> babbaaababaa	2
abba <sub>2</sub>	abba   <sup>2</sup> bab   <sup>1</sup> abbbaaababaa	2
abba <sub>2</sub>	abba   <sup>2</sup> baba   <sup>1</sup> bbbaaababaa	2
abba <sub>2</sub>	abba   <sup>2</sup> babab   <sup>1</sup> baaababaa	2
abba <sub>2</sub>	abba   <sup>2</sup> bababb   aaababaa	<b>2</b>

Situatia este identica cu cea dinainte, regula **1** esueaza asteptand sa regaseasca *ba* in sir, dar gaseste *bb*, iar regula **2** este singura care reuseste sa recunoasca un subsir.

Se utilizeaza tot regula **2** pentru subsirul **abba** si se continua analiza.

abba <sub>2</sub> abba <sub>2</sub>	<sup>123</sup> bababbaaababaa	-
abba <sub>2</sub> abba <sub>2</sub>	b   <sup>13</sup> ababbaaababaa	-
abba <sub>2</sub> abba <sub>2</sub>	ba   <sup>13</sup> babbaaababaa	-
abba <sub>2</sub> abba <sub>2</sub>	bab   <sup>13</sup> abbbaaababaa	3
abba <sub>2</sub> abba <sub>2</sub>	bab   <sup>3</sup> a   <sup>1</sup> bbbaaababaa	3
abba <sub>2</sub> abba <sub>2</sub>	bab   <sup>3</sup> ab   <sup>1</sup> baaababaa	3
abba <sub>2</sub> abba <sub>2</sub>	bab   <sup>3</sup> abb   aaababaa	<b>3</b>

Singura regula care a recunoscut o portiune din sir este regula **3**. Regula **1** a intampinat aceeaasi problema ca si in parcurgerile anterioare.

Se utilizeaza regula **3** pentru subsirul **bab** si se continua analiza.

abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub>	<sup>123</sup> abbbaaababaa	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub>	a   <sup>12</sup> bbbaaababaa	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub>	ab   <sup>12</sup> baaababaa	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub>	abb   <sup>12</sup> aaababaa	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub>	abba   <sup>12</sup> aababaa	2
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub>	abba   <sup>2</sup> a   <sup>1</sup> ababaa	2
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub>	abba   <sup>2</sup> aa   <sup>1</sup> babaa	<b>2 &amp; 1</b>

De data aceasta, atât regula **1**, cât și regula **2** au recunoscut cu succes subsiruri, însă conform regulii de bază *maximal munch*, se va utiliza regula care a consumat cel mai mult din sir.

Se utilizează, deci, regula **1** care recunoaște **abbaaa** din sirul de la intrare și analiza continuă.

abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub>	<sup>123</sup> babaa	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub>	b   <sup>13</sup> abaa	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub>	ba   <sup>13</sup> baa	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub>	bab   <sup>13</sup> aa	3
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub>	bab   <sup>3</sup> a   <sup>1</sup> a	3
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub>	bab   <sup>3</sup> aa   <sup>1</sup>	3
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub>	bab   <sup>3</sup> aa   \$	<b>3</b>

Situația actuală este una nouă. Regula **1** esuează intrucât sirul de la intrare s-a consumat înainte ca regula să recunoască un subsir. Regula așteaptă să regasească *ab* sau *aa* în sir, dar a găsit doar *a*.

Intrucât este singura regulă care a recunoscut un subsir, se utilizează regula **3** pentru subsirul **bab** și se continuă analiza.

abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub> bab <sub>3</sub>	<sup>123</sup> aa	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub> bab <sub>3</sub>	a   <sup>12</sup> a	-
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub> bab <sub>3</sub>	aa   <sup>1</sup>	1
abba <sub>2</sub> abba <sub>2</sub> bab <sub>3</sub> abbaaa <sub>1</sub> bab <sub>3</sub>	aa   <sup>1</sup> \$	<b>1</b>

Regula **1** este singura care recunoaște o porțiune din sirul rămas, mai exact tot ce a rămas din sir. Se utilizează regula **1** care recunoaște **aa** din sirul de la intrare și analiza se oprește intrucât s-a consumat tot sirul.

Asadar, secvența de reguli lexicale utilizate pentru analiza sirului este

**2 2 3 1 3 1**

abba<sub>2</sub> abba<sub>2</sub> bab<sub>3</sub> abbaaa<sub>1</sub> bab<sub>3</sub> aa<sub>1</sub>

***Dati un exemplu de sir care nu poate fi analizat de aceasta specificatie.***

Orice sir care incepe sau se termina cu **bb**.

# Recursivitate la stanga

---

Fie urmatoarea gramatica:

$$S \rightarrow A\alpha \mid \delta$$

$$A \rightarrow S\beta$$

\*  $\alpha$  si  $\beta$  contin doar terminali

***Este gramatica recursiva la stanga? Daca da, direct sau indirect? De ce?***

Da, indirect, pentru ca S se poate expanda in A urmat de ceva, iar A se poate expanda in S urmat de ceva. Formal, acest lucru s-ar scrie:  $S \rightarrow^* S...$  (S "trece" in 0 sau mai multi pasi tot in S urmat de ceva).

***Ce strategie avea probleme cu gramaticile recursive la stanga?***

Strategia top-down.

***Cum putem elimina recursivitatea la stanga?***

Incercam sa deducem forma sirurilor generate de gramatica, apoi rescriem regulile evitand recursivitatea la stanga.

Putem inlocui A-ul in regula lui S:  $S \rightarrow S\beta\alpha \mid \delta$ .

Construim cativa pasi de derivare:  $S \rightarrow S\beta\alpha \rightarrow S\beta\alpha\beta\alpha \rightarrow S\beta\alpha\beta\alpha\beta\alpha \rightarrow \delta\beta\alpha\beta\alpha\beta\alpha$ .

Observam ca sirurile care se pot genera incep cu  $\delta$  si apoi au perechi  $\beta\alpha$  (sau niciuna, cand  $S \rightarrow \delta$  inca din primul pas).

Gramatica se poate rescrie astfel (inlocuind recursivitatea la stanga cu recursivitate la dreapta):

$$S \rightarrow \delta A$$

$$A \rightarrow \beta\alpha A \mid \epsilon$$

# Arbori de derivare

Fie urmatoarea gramatica:

$$E \rightarrow E * E \mid E + E \mid (E) \mid \text{int}$$

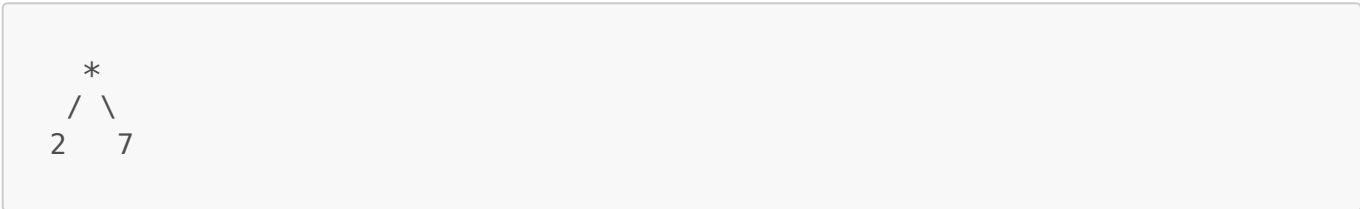
si urmatorul sir:

$$5 * 3 + (2 * 7) + 4$$

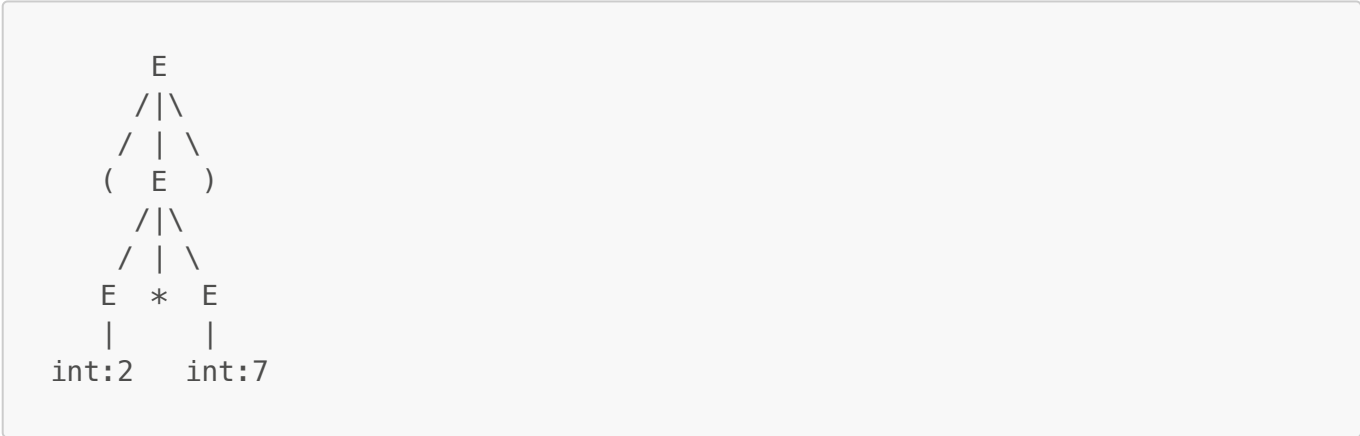
**Cati arbori de derivare distincti exista pentru aceasta gramatica si acest sir?**

Numarul arborilor de derivare este egal cu numarul de moduri in care poate fi parantezata suplimentar expresia ca sa grupam operatiile.

**(2 \* 7)** este atomic, nu poate fi spart. Arborele (subarborele) aferent acestei expresii este:

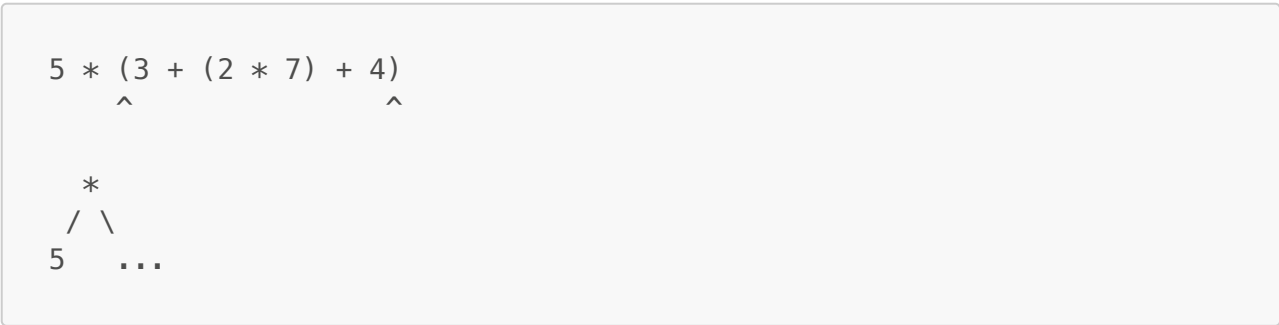


Pentru a economisi spatiu, arborii desenati nu vor fi pe deplin arbori de derivare, intrucat vor lipsi neterminalii si parantezele. Structura lor este similara cu a unui arbore de sintaxa abstracta. De exemplu, un arbore de derivare pentru expresia **(2 \* 7)** ar fi:



Ceilalti operatori (5 \* 3 + (2 \* 7) + 4 - cei subliniati) duc la constructii diferite.

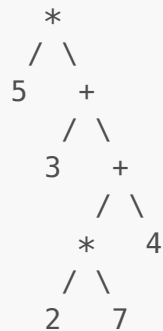
1. Punem primul operator (\* din **5 \* 3 ...**) in radacina. Asta inseamna ca grupam "punand paranteze" in jurul expresiei **3 + (2 \* 7) + 4**.



La randul ei, expresia  **$3 + (2 * 7) + 4$**  poate fi reprezentata prin mai multi arbori (subarbori ai arborelui intregii expresii).

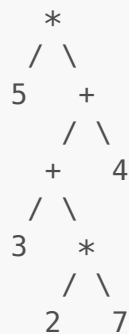
1. Punem operatorul **+** (din ...  **$3 + ( ... )$** ) in radacina. Asta inseamna ca grupam "punand paranteze" in jurul expresiei  **$(2 * 7) + 4$** .

$$5 * (3 + ((2 * 7) + 4))$$



2. Punem operatorul **+** (din ...  **$) + 4$** ) in radacina. Asta inseamna ca grupam "punand paranteze" in jurul expresiei  **$3 + (2 * 7)$** .

$$5 * ((3 + (2 * 7)) + 4)$$



Astfel, am obtinut 2 arbori pentru primul operator in radacina.

2. Punem ultimul operator (**+** din ...  **$+ 4$** ) in radacina. Rationamentul este similar cu cel explicat la punctul anterior. Se obtin tot 2 arbori.
3. Punem operatorul **+** (din ...  **$3 + ( ... )$** ) in radacina. Asta inseamna ca grupam "punand paranteze" in jurul expresiilor  **$5 * 3$**  si  **$(2 * 7) + 4$** .

$$(5 * 3) + ((2 * 7) + 4)$$





Se obtine un singur arbore.

In final, se obtin  $2 + 2 + 1 = 5$  arbori de derivare.

***Ce concluzie se obtine despre gramatica data, tinand cont de numarul de arbori de derivare obtinuti?***

Gramatica este ambigua (toti arborii sunt diferiti).

Nu era o problema daca obtineam acelasi arbore prin diferite metode de derivare.

***Ce constrangere ar trebui sa adaugam la aceasta gramatica astfel incat arborele de derivare sa fie unic?***

Sa stabilim prioritatea operatorilor (\* mai prioritar decat +) si modul in care se realizeaza asocierea (la stanga sau la dreapta).

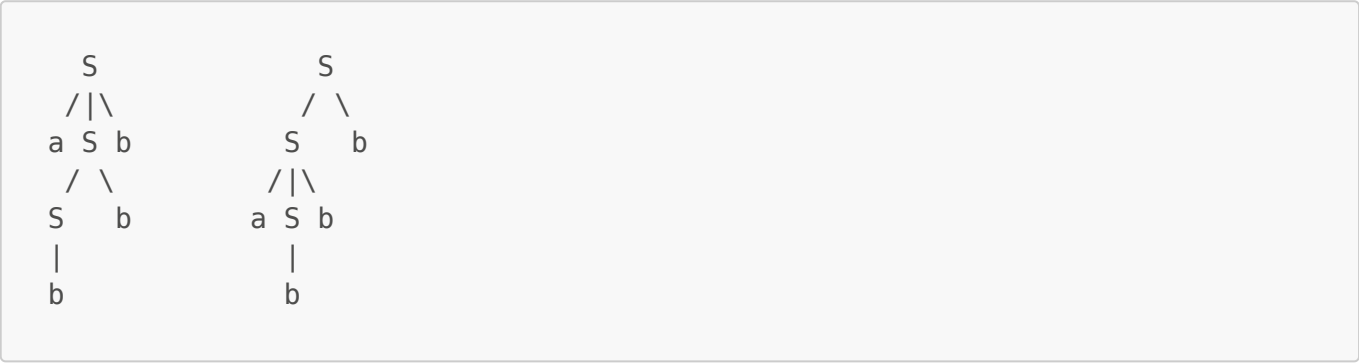
# Gramatica ambigua

Fie urmatoarea gramatica:

$$S \rightarrow aSb \mid Sb \mid b$$

***Este gramatica ambigua?***

Da, pentru ca nu putem stabili o ordine clara intre **aSb** si **Sb**. Putem demonstra folosind sirul **abbb** pentru care se pot construi 2 arbori de derivare.





# Recursive descent

---

Fie urmatoarea gramatica:

$$S \rightarrow A \mid B \mid OS$$
$$A \rightarrow OA \mid O$$
$$B \rightarrow 1$$

si urmatorul sir:

$$0^n1 \quad (0 \text{ de } n \text{ ori si apoi } 1)$$

**Folosind o strategie recursive descent, care este numarul de pasi pe care ii realizeaza parser-ul, in notatie asimptotica  $\Theta(\dots)$ , in functie de  $n$ ?**

Adaugam simbolul **\$** la sfarsitul sirului si imbogatim gramatica cu o noua productie:

$$S' \rightarrow S\$ \quad S' \text{ va fi noul neterminal cu care incepe derivarea}$$

Simulam analiza pe un exemplu. Fiecare pas prezinta sirul de la intrare, pozitia cursorului, derivarea obtinuta pana in prezent si evidentiaza terminalii si token-ii de la intrare a caror egalitate se verifica.

Sirul exemplu este **001\$**.

**|001\$**

**S'**

Folosim prima (si singura) productie a lui **S'**, adica **S' → S\$**.

**|001\$**

**S' → S\$**

Nu am intalnit terminal, continuam derivarea, folosind prima productie a lui **S**, adica **S → A**.

**|001\$**

**S' → S\$ → A\$**

Situatia anterioara se repeta, folosim prima productie a lui **A**, adica **A → OA**.

**|001\$**

**S' → S\$ → A\$ → OA\$**

Terminalul obtinut **O** se potriveste cu token-ul curent de la intrare, asa ca avansam cursorul si continuam derivarea. Folosim prima productie a lui **A**, adica **A → OA**.

**0|01\$**

**S' → S\$ → A\$ → OA\$ → OOA\$**

Terminalul obtinut **O** se potriveste cu token-ul curent de la intrare, asa ca avansam cursorul si continuam derivarea. Folosim prima productie a lui **A**, adica **A → OA**.

**00|1\$**

**S' → S\$ → A\$ → OA\$ → OOA\$ → OOOA\$**

00|1\$  
 $S' \rightarrow S\$ \rightarrow A\$ \rightarrow 0A\$ \rightarrow 00A\$ \rightarrow 000A\$$   
 $\qquad \qquad \qquad \hookrightarrow 000\$$

0|01\$  
 $S' \rightarrow S\$ \rightarrow A\$ \rightarrow 0A\$ \rightarrow 00A\$ \rightarrow 000A\$$   
                           |                   $\hookrightarrow 000\$$   
                            $\hookrightarrow 00\$$

00|1\$  
 $S' \rightarrow S\$ \rightarrow A\$ \rightarrow 0A\$ \rightarrow 00A\$ \rightarrow 000A\$$   
                   |                   $\hookrightarrow 000\$$   
                    $\hookrightarrow 00\$$

**|001\$**

**S' → S\$ → A\$ → 0A\$ → 00A\$ → 000A\$**

**|            |            ↳ 000\$**

**|            ↳ 00\$**

**↳ 0\$**

0|01\$

$S' \rightarrow S\$ \rightarrow A\$ \rightarrow 0A\$ \rightarrow 00A\$ \rightarrow 000A\$$

                  |                  |                   $\hookrightarrow 000\$$

                  |                   $\hookrightarrow 00\$$

$\hookrightarrow 0\$$

**|001\$**

**$S' \rightarrow S\$ \rightarrow A\$ \rightarrow 0A\$ \rightarrow 00A\$ \rightarrow 000A\$$**

			↪ 000\$
			↪ 00\$
			↪ 0\$
			↪ B\$

Nu am intalnit terminal, continuam derivarea. Folosim prima (si singura) productie a lui B, adica **B → 1**.

0001\$

**S' → S\$ → A\$ → 0A\$ → 00A\$ → 000A\$**

			↪ 000\$
			↪ 00\$
			↪ 0\$
			↪ B\$ → <u>1</u> \$

Terminalul obtinut **1** *nu* se potriveste cu token-ul curent de la intrare (**0**), asa ca mergem o derivare inapoi pentru B si, cum B nu are alta productie, mergem inca o derivare inapoi si incercam urmatoarea productie a lui S. Folosim ultima productie a lui S, adica **S → 0S**.

0001\$

**S' → S\$ → A\$ → 0A\$ → 00A\$ → 000A\$**

			↪ 000\$
			↪ 00\$
			↪ 0\$
			↪ B\$ → 1\$
			↪ <u>0</u> S\$

Terminalul obtinut **0** se potriveste cu token-ul curent de la intrare, asa ca avansam cursorul si continuam derivarea. Folosim prima productie a lui S, adica **S → A**.

0001\$

**S' → S\$ → A\$ → 0A\$ → 00A\$ → 000A\$**

			↪ 000\$
			↪ 00\$
			↪ 0\$
			↪ B\$ → 1\$
			↪ 0S\$ → 0A\$

Procesul continua in mod similar, cu mentiunea ca acum exista un **0** "mai putin".

Ideile pe care ne vom baza pentru a calcula numarul de pasi sunt urmatoarele:

- orice productie **S → A** consuma **0**-urile si da eroare la final, neputand consuma si ultimul token, **1**;
- trebuie sa ajungem la productia **S → 0S** care consuma **0**-urile de la intrare, pentru a putea avansa in sir si a-l consuma cu adevarat;
- un pas al parser-ului este o operatie de verificare a unui terminal cu un token de la intrare.

Consumarea "corecta" a primului **0** se face intr-un numar de pasi proportional cu **n+1** (productia **S → A** executa  $3n+2$  pasi, **S → B** executa un pas, iar **S → 0S** executa tot un pas, deci  $3(n+1)+1$ ).

Formula verifica pentru pasii prezentati mai sus: n este 2, deci ar trebui sa fie  $3*2+4 = 10$  verificari token-terminal, iar mai sus sunt fix 10 diagrame in care exista tokeni si terminali subliniati (adica verificari facute de

parser).

Urmeaza apoi consumarea urmatorului **0** care se face in maniera similara, proportional cu **n**, apoi proportional cu **n-1** pentru urmatorul **0**, continuand pana se consuma toate **0**-urile. Consumarea lui **1** se face mereu in 3 pasi (2 de la **S → A** si 1 de la **S → B**). Caracterul **\$** se valideaza intr-un singur pas, imediat dupa **1**.

Astfel, considerand valorile proportionale ale numerelor de pasi in raport cu **n**, se obtine o suma **n+1 + n + (n-1) + ...**, aproximativ egala cu **(n+1)(n+2)/2**.

Deci, numarul de pasi pe care ii realizeaza parser-ul cu o strategiei recursive descent pentru gramatica si sirurile date este, in notatie asimptotica,  **$\Theta(n^2)$** .

# Multimile First si Follow

---

Fie urmatoarea gramatica:

$$S \rightarrow A(S)B \mid \epsilon$$

$$A \rightarrow S \mid SB \mid x \mid \epsilon$$

$$B \rightarrow SB \mid y$$

**Cand se afla un terminal  $t$  in multimea  $First$ ?**

Un terminal  $t$  se afla in multimea **First( $X$ )** daca o derivare care incepe cu  $X$  rezulta intr-o secventa care incepe cu terminalul  $t$ . Formal, acest lucru s-ar scrie:  $X \rightarrow^* t... \Rightarrow t \in First(X)$ .

Similar se poate spune si despre apartenenta lui  $\epsilon$  la multimea **First( $X$ )**, diferenta fiind ca se obtine doar  $\epsilon$  in urma derivarii. Intr-o maniera formala:  $X \rightarrow^* \epsilon \Rightarrow \epsilon \in First(X)$ .

**Care sunt regulile multimilor  $First$  pentru o secventa de neterminali?**

$$\epsilon \notin First(X_1) \Rightarrow First(X_1X_2) = First(X_1)$$

$$\epsilon \in First(X_1) \Rightarrow First(X_1X_2) = First(X_1) \setminus \{\epsilon\} \cup First(X_2)$$

**Care sunt multimile  $First$  pentru neterminalii  $S$ ,  $A$  si  $B$ ?**

$$First(S) \supseteq First(A(S)B) \cup \{\epsilon\} \quad (1)$$

$$First(A) \supseteq First(S) \cup First(SB) \cup \{x, \epsilon\} \quad (2)$$

$$First(B) \supseteq First(SB) \cup \{y\} \quad (3)$$

$$First(A(S)B) = First(A) \setminus \{\epsilon\} \cup \{()\} \quad (4)$$

$$1 \ \& \ 4 \Rightarrow First(S) \supseteq First(A) \setminus \{\epsilon\} \cup \{(), \epsilon\}$$

$$\underline{First(S)} \supseteq First(A) \cup \{(), \epsilon\} \quad (5)$$

$$First(SB) = First(S) \setminus \{\epsilon\} \cup First(B) \quad (6)$$

$$2 \ \& \ 6 \Rightarrow First(A) \supseteq First(S) \cup First(S) \setminus \{\epsilon\} \cup First(B) \cup \{x, \epsilon\}$$

$$\underline{First(A)} \supseteq First(S) \cup First(B) \cup \{x, \epsilon\} \quad (7)$$

$$3 \ \& \ 6 \Rightarrow First(B) \supseteq First(S) \setminus \{\epsilon\} \cup First(B) \cup \{y\}$$

$$\underline{First(B)} \supseteq First(S) \setminus \{\epsilon\} \cup \{y\}$$

$$5 \ \& \ 7 \Rightarrow First(A) = First(S)$$

Multimile  $First$  pentru neterminalii  $S$ ,  $A$  si  $B$  sunt:

$$First(S) = First(A) = \{(), \epsilon, y, x\}$$

$$First(B) = \{(), y, x\}$$

**Cand se afla un terminal  $t$  in multimea  $Follow$ ?**

Un terminal  $t$  se afla in multimea **Follow( $X$ )** daca exista o derivare care rezulta intr-o secventa care contine neterminalul  $X$  urmat imediat de terminalul  $t$ . Formal, acest lucru s-ar scrie:  $Y \rightarrow^* ...Xt... \Rightarrow t \in Follow(X)$ .

**Care sunt multimile  $Follow$  pentru neterminalii  $S$ ,  $A$  si  $B$ ?**

$$\text{First}(A) = \text{First}(S) = \{ (, \varepsilon, y, x \} \quad (01)$$

$$\text{First}(B) = \{ (, y, x \} \quad (02)$$

$$\text{Follow}(S) \supseteq \{ \$ \} \quad (1)$$

$$S \rightarrow A(S)B \Rightarrow \text{Follow}(S) \supseteq \{ \} \quad (2)$$

$$A \rightarrow S \Rightarrow \text{Follow}(S) \supseteq \text{Follow}(A) \quad (3)$$

$$A \rightarrow SB \ \& \ 02 \Rightarrow \text{Follow}(S) \supseteq \text{First}(B) \setminus \{ \varepsilon \} \quad (4)$$

$$B \rightarrow SB \ \& \ 02 \Rightarrow \text{Follow}(S) \supseteq \text{First}(B) \setminus \{ \varepsilon \} \quad (5)$$

$$1-5 \ \& \ 02 \Rightarrow \text{Follow}(S) \supseteq \{ \$ \} \cup \{ \} \cup \text{Follow}(A) \cup \{ (, y, x \}$$

$$\underline{\text{Follow}(S)} \supseteq \text{Follow}(A) \cup \{ \$, ), (, y, x \} \quad (6)$$

$$S \rightarrow A(S)B \Rightarrow \text{Follow}(A) \supseteq \{ \}$$

$$\underline{\text{Follow}(A)} \supseteq \{ \}$$

$$S \rightarrow A(S)B \Rightarrow \text{Follow}(B) \supseteq \text{Follow}(S) \quad (7)$$

$$A \rightarrow SB \Rightarrow \text{Follow}(B) \supseteq \text{Follow}(A) \quad (8)$$

$$B \rightarrow SB \Rightarrow \text{Follow}(B) \supseteq \text{Follow}(B) \quad (9)$$

$$7-9 \Rightarrow \text{Follow}(B) \supseteq \text{Follow}(S) \cup \text{Follow}(A) \cup \text{Follow}(B)$$

$$\text{Follow}(B) \supseteq \text{Follow}(S) \cup \text{Follow}(A) \quad (10)$$

$$6 \ \& \ 10 \Rightarrow \underline{\text{Follow}(B)} \supseteq \text{Follow}(S)$$

Multimile Follow pentru neterminalii **S**, **A** si **B** sunt:

$$\mathbf{\text{Follow}(A) = \{ \}}$$

$$\mathbf{\text{Follow}(S) = \text{Follow}(B) = \{ \$, ), (, y, x \}}$$

# Prefixe viabile si conflicte

---

Fie urmatoarea gramatica:

$$S \rightarrow A(S)B \mid \varepsilon$$

$$A \rightarrow S \mid SB \mid x \mid \varepsilon$$

$$B \rightarrow SB \mid y$$

cu urmatoarele multimi First si Follow

$$\text{First}(S) = \{ (, \varepsilon, y, x \}$$

$$\text{First}(A) = \{ (, \varepsilon, y, x \}$$

$$\text{First}(B) = \{ (, y, x \}$$

$$\text{Follow}(S) = \{ \$, ), (, y, x \}$$

$$\text{Follow}(A) = \{ \}$$

$$\text{Follow}(B) = \{ \$, ), (, y, x \}$$

**Cum se construiește automatul care recunoaște prefixele viabile?**

- Se creează un nou simbol de start **S'**.
- Se introduce o nouă producție **S' → S**.

Se construiește un AFN:

- stările sunt **itemi**, adică producții în care se evidențiază cât a fost observat din partile lor drepte folosindu-se simbolul **•**;
- starea inițială este itemul **S' → •S**, care sugerează că încă nu am parcurs nimic, urmând să parcurgem S;
- se adaugă tranziții pe primul simbol din dreapta punctului (terminal sau neterminal), ajungând într-o stare (item) ce conține punctul după simbolul respectiv (de exemplu, o tranziție pe **S** duce automatul în itemul **S' → S•**);
- se adaugă  $\varepsilon$ -tranziții pentru fiecare prim neterminal din dreapta punctului, ajungând în itemi (stări) corespunzatori producțiilor acelui neterminal, păstrând punctul în fața producției;
- ultimii 2 pași se repetă până când automatul este complet.

Se transformă AFN-ul într-un AFD:

- stările sunt **multimi de itemi**;
- o stare și toate stările în care se ajunge prin  $\varepsilon$ -tranziții sunt grupate într-o nouă stare a AFD-ului.

Se poate construi direct AFD-ul, fără a construi și AFN-ul:

- în starea inițială se adaugă itemul **S' → •S**;
- pentru orice item care are punctul în fața unui neterminal, se adaugă în aceeași stare itemi pentru toate producțiile acelui neterminal, păstrând punctul în fața producției;
- se adaugă tranziții pe primele simboluri din dreapta punctelor (terminal sau neterminal);
- ultimii 2 pași se repetă până când automatul este complet.

**Care este starea inițială a automatului care recunoaște prefixele viabile ale acestei gramatici?**

Starea initiala cuprinde, pentru inceput, noul item creat:

$$S' \rightarrow \cdot S$$

Avand punctul in fata neterminalului S, se adauga in stare toti itemii productiilor lui S:

$$S \rightarrow \cdot A(S)B$$
$$S \rightarrow \cdot$$

Intrucat punctul este in fata lui A in primul item, procesul se repeta pentru A, adaugandu-se urmatorii itemi in stare:

$$A \rightarrow \cdot S$$
$$A \rightarrow \cdot SB$$
$$A \rightarrow \cdot x$$
$$A \rightarrow \cdot$$

Ar trebui repetat procesul pentru S, insa itemii rezultati sunt deja adaugati in stare.

Starea initiala este:

$$S' \rightarrow \cdot S$$
$$S \rightarrow \cdot A(S)B$$
$$S \rightarrow \cdot$$
$$A \rightarrow \cdot S$$
$$A \rightarrow \cdot SB$$
$$A \rightarrow \cdot x$$
$$A \rightarrow \cdot$$

### ***Ce este un item shift si un item reduce?***

Un item **shift** este un item care are un terminal imediat in dreapta punctului (de exemplu,  $X \rightarrow \alpha \cdot t \beta$ ). Cu alte cuvinte, el sugereaza ca putem avansa (*shift*) in sirul de la intrare ca sa consumam acel token de langa punct.

Un item **reduce** este un item in care punctul nu are nimic la dreapta lui (de exemplu,  $X \rightarrow \alpha \cdot$ ). Altfel spus, itemul sugereaza ca se poate *reduce* secventa  $\alpha$  la neterminalul X.

### ***Care sunt conflictele posibile intr-o stare a acestui tip de automat?***

Un conflict este **Shift-Reduce**, adica existenta a unui item shift si a unui item reduce in aceeasi stare, ceea ce sugereaza ca nu se stie ce operatie se poate face (avansare in sir sau reductie).

Celalalt conflict este **Reduce-Reduce**, adica existenta a doi itemi reduce in aceeasi stare, ceea ce duce la necunoasterea a ce reductie trebuie aplicata.

### ***Exista conflicte in starea initiala a automatului pentru aceasta gramatica cu un analizor LR(0)?***

Pentru un analizor LR(0), existenta conflictelor intr-o stare este suficienta pentru a intampina probleme la parsare.

Starea initiala cuprinde atat itemi shift, cat si itemi reduce:

$$S \rightarrow \cdot \quad \text{reduce} - 1$$



$A \rightarrow \cdot x$           shift    - 2

$A \rightarrow \cdot$                 reduce - 3

Conflictele starii initiale sunt:

**Shift-Reduce - 2 & 1**

**Shift-Reduce - 2 & 3**

**Reduce-Reduce - 1 & 3**

***Exista conflicte in starea initiala a automatului pentru aceasta gramatica cu un analizor SLR(1)?***

Fiind SLR(1), putem folosi un token de lookahead pentru a vedea ce urmeaza in sirul de la intrare.

Anumiti itemi reduce pot fi considerati "neutilizabili" pentru un anumit token din sirul de la intrare, daca acel token nu se afla in multimea Follow a partii stangi a itemului. Cu alte cuvinte, nu are rost aplicarea unei reductii, atat timp cat se ajunge intr-o stare in care dupa neterminatul obtinut (dupa reductie) se afla un token ce nu ii poate urma vreodata. Formal, acest lucru s-ar scrie:

$X \rightarrow \alpha \cdot$

sirul de la intrare: ... | t ... (cursorul inaintea lui t)

$t \notin \text{Follow}(X)$

$\Rightarrow X \rightarrow \alpha \cdot$  - item neutilizabil (pentru stadiul sirului din ipoteza)

Pentru un conflict Shift-Reduce, token-ul poate fi considerat terminalul din dreapta punctului itemului shift.

Considerand conflictele identificate anterior, exista doua posibile conflicte Shift-Reduce:

$S \rightarrow \cdot$

$A \rightarrow \cdot x$

$A \rightarrow \cdot x$

$A \rightarrow \cdot$

Terminalul x apartine multimii Follow(S), ceea ce inseamna ca primul conflict este valabil si in SLR(1) pentru token-ul "x". Al doilea conflict nu mai este valabil deoarece "x" nu apartine multimii Follow(A).

Expandand rationamentul initial despre un item reduce "neutilizabil", pentru 2 itemi reduce, conflictul dintre ei se rezolva daca token-ul t nu apartine simultan celor doua multimii Follow (dar poate sa apartina doar uneia).

Expandand mai departe ideea, un conflict Reduce-Reduce este pe deplin rezolvat atunci cand multimile Follow ale partilor stangi nu au elemente comune (adica nu exista niciun terminal care sa fie simultan in ambele multimii; vom avea o singura posibilitate de reduce de fiecare data).

Considerand conflictele identificate anterior, exista un posibil conflict Reduce-Reduce:

$S \rightarrow \cdot$

$A \rightarrow \cdot$

$\text{Follow}(S) \cap \text{Follow}(A) = \{\$, \), (, y, x\} \cap \{(\} = \{(\}$

Cum intersectia celor doua multimii Follow nu este vida, inseamna ca acest conflict Reduce-Reduce este valabil si pentru SLR(1) pentru token-ul "(".

In concluzie, exista 2 conflicte ale starii initiale:

$S \rightarrow \cdot$  reduce - 1

$A \rightarrow \cdot x$  shift - 2

$A \rightarrow \cdot$  reduce - 3

**Shift-Reduce** pentru  $x$  - 2 & 1

**Reduce-Reduce** pentru  $($  - 1 & 3

***Este gramatica SLR(1)?***

Nu, deoarece exista cel putin o stare (starea initiala) in care exista cel putin un conflict (Shift-Reduce si Reduce-Reduce).

# Ambiguitate si conflicte

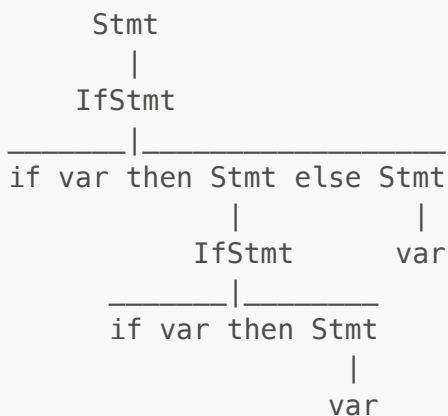
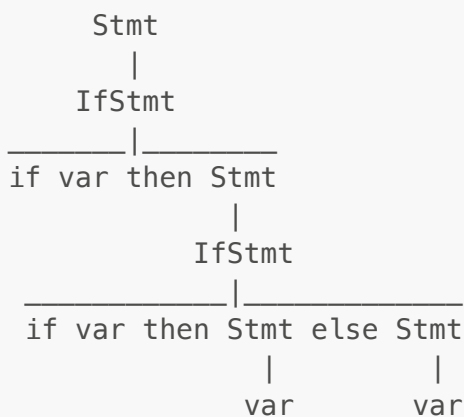
Fie urmatoarea gramatica:

$\text{Stmt} \rightarrow \text{var} \mid \text{IfStmt}$

$\text{IfStmt} \rightarrow \text{if var then Stmt} \mid \text{if var then Stmt else Stmt}$

***Este gramatica ambigua?***

Da, pentru ca nu putem stabili o ordine clara intre cele doua productii ale neterminalului **IfStmt**. Putem demonstra folosind sirul **if var then if var then var else var** pentru care se pot construi 2 arbori de derivare.



Problema mai poarta numele de *dangling else*, deoarece nu se stie carei instructiuni *if* ii apartine *else*-ul.

***Cum se reflecta problema dangling else pentru un parser bottom-up?***

Simulam operatiile pe care le face parser-ul:

if var then if var then var else var	⇒ shift
if   var then if var then var else var	⇒ shift
if var   then if var then var else var	⇒ shift
if var then   if var then var else var	⇒ shift
if var then if   var then var else var	⇒ shift
if var then if var   then var else var	⇒ shift

if var then if var then   var else var	⇒ shift
if var then if var then var   else var	⇒ reduce

Am aplicat reduce pentru ca altfel nu vom putea forma niciodata simbolul de start, intrucat dupa *then* este intotdeauna *Stmt*. Daca nu am face acum reduce, am ramane cu *then var* in partea stanga a cursorului si nu exista nicio reductie care sa cuprinda aceasta structura.

Ajungem in urmatoarea situatie:

if var then if var then Stmt | else var

Parser-ul poate sa faca atat shift, cat si reduce. In cazul operatiei shift, va grupa *else*-ul cu *if*-ul "interior", iar in cazul operatiei reduce, *else*-ul va fi grupat cu *if*-ul "exterior".

Asadar, daca am construi automatul care recunoaste prefixele variabile ale acestei gramatici, am obtine un conflict Shift-Reduce intr-o stare, conflict cauzat de ambiguitatea gramaticii.

# LL(1) si SLR(1)

---

Fie urmatoarea gramatica:

$$S \rightarrow Sb \mid a$$

## ***Este gramatica LL(1)?***

Doar analizand gramatica, reiese ca este recursiva la stanga (din cauza productiei  **$S \rightarrow Sb$** ), iar cum acest tip de gramatici nu pot fi LL(1), rezulta ca **gramatica nu este LL(1)**.

## ***Cum putem face o astfel de gramatica sa fie analizabila de un parser LL(1)?***

In primul rand, trebuie eliminata recursivitatea la stanga, indiferent daca este directa sau indirecta.

Apoi trebuie facuta factorizare la stanga:

$$X \rightarrow tY \mid tZ \quad \Rightarrow \quad X \rightarrow tW \quad W \rightarrow Y \mid Z$$

Astfel, la intalnirea token-ului  $t$ , avem o singura productie posibila, nu doua sau mai multe.

## ***Care este tabelul LL(1) pentru aceasta gramatica?***

Pentru inceput, calculam multimile First si Follow:

$$\text{First}(S) = \{a\}$$

$$\text{Follow}(S) = \{b, \$\}$$

Completam tabelul LL(1):

- pe linii se pun neterminalii si pe coloane terminalii si simbolul \$ (sfarsitul sirului);
- pentru fiecare productie  **$X \rightarrow \alpha$** , se obtine  $\text{First}(\alpha)$  si pentru fiecare terminal  **$t$**  din multime se pune in tabel la intersectia  **$[X, t]$**  partea dreapta a productiei, adica  **$\alpha$** ;
- daca  **$\epsilon \in \text{First}(X)$** , se obtine  $\text{Follow}(X)$  si pentru fiecare terminal  **$t$**  din multime se pune  **$\epsilon$**  in tabel la intersectia  **$[X, t]$** .

Tabelul este initial gol:

	<b>a</b>	<b>b</b>	<b>\$</b>
<b>S</b>			

Pentru productia  **$S \rightarrow Sb$** :

$$\text{First}(Sb) = \text{First}(S) = \{a\}$$

$$[S, a] = Sb$$

	<b>a</b>	<b>b</b>	<b>\$</b>
<b>S</b>	Sb		

Pentru productia  $S \rightarrow a$ :

$$\text{First}(\{a\}) = \{a\}$$

$$[S, a] = a$$

	a	b	\$
S	Sb a		

Acesta este tabelul LL(1) al gramaticii date. Se observa ca in celula  $[S, a]$  avem doua intrari, inca o dovada ca **gramatica nu este LL(1)**.

**Cum ar arata tabelul LL(1) daca ar mai exista o productie  $S \rightarrow \epsilon$ ?**

Gramatica arata acum astfel:

$$S \rightarrow Sb \mid a \mid \epsilon$$

Recalculam multimile First si Follow:

$$\text{First}(S) = \{a, b, \epsilon\}$$

$$\text{Follow}(S) = \{b, \$\}$$

Tabelul este initial gol:

	a	b	\$
S			

Pentru productia  $S \rightarrow Sb$ :

$$\text{First}(Sb) = \text{First}(S) \setminus \epsilon \cup \{b\} = \{a, b\}$$

$$[S, a] = Sb$$

$$[S, b] = Sb$$

	a	b	\$
S	Sb	Sb	

Pentru productia  $S \rightarrow a$ :

$$\text{First}(\{a\}) = \{a\}$$

$$[S, a] = a$$

	a	b	\$
S	Sb a	Sb	

Pentru productia  $S \rightarrow \epsilon$ :

$$\text{Follow}(S) = \{b, \$\}$$

$$[S, b] = \varepsilon$$

$$[S, \$] = \varepsilon$$

	a	b	\$
S	Sb a	Sb $\varepsilon$	$\varepsilon$

Acesta ar fi tabelul LL(1) al gramaticii dupa ce a fost imbogatita cu productia  $S \rightarrow \varepsilon$ .

### ***Este gramatica initiala SLR(1)?***

Se construiesc automatul complet pentru gramatica data.

Starea initiala (0) este:

$$S' \rightarrow \bullet S$$

$$S \rightarrow \bullet Sb$$

$$S \rightarrow \bullet a$$

Tranzitia din starea 0 pe **S** duce in urmatoarea stare (1):

$$S' \rightarrow S \bullet$$

$$S \rightarrow S \bullet b$$

Tranzitia din starea 0 pe **a** duce in urmatoarea stare (2):

$$S \rightarrow a \bullet$$

Tranzitia din starea 1 pe **b** duce in urmatoarea stare (3):

$$S \rightarrow Sb \bullet$$

Verificam daca exista conflicte in starile automatului. Se incepe cu identificarea itemilor reduce, deoarece starile fara itemi reduce nu pot avea conflicte.

**Starea 0** - un item shift  $\Rightarrow$  fara conflicte

**Starea 1** - un item shift si unul reduce  $\Rightarrow$  posibil conflict Shift-Reduce

**Starea 2** - un singur item  $\Rightarrow$  fara conflicte

**Starea 3** - un singur item  $\Rightarrow$  fara conflicte

Pentru starea **1** verificam daca **b** se afla in Follow(S').

$$\text{Follow}(S') = \{\$, \varepsilon\}$$

$$b \notin \text{Follow}(S')$$

Cum **b** nu apartine multimii Follow(S'), nu este conflict Shift-Reduce in starea **1**.

Cum nu exista stari ale automatului care sa aiba cel putin un conflict, inseamna ca **gramatica este SLR(1)**.

### ***Daca modificam productia $S \rightarrow Sb$ in $S \rightarrow SbS$ , este gramatica LL(1) si/sau SLR(1)?***

Gramatica arata acum astfel:

$$S \rightarrow SbS \mid a$$

Gramatica este in continuare recursiva la stanga, deci **nu este LL(1)**. De asemenea,  $\text{First}(SbS) = \text{First}(Sb)$ , deci crearea tabelului LL(1) ar avea iar doua intrari la intersectia  $[S, a]$ , anume **SbS** si **a**.

	<b>a</b>	<b>b</b>	<b>\$</b>
<b>S</b>	SbS a		

Structura tabelului dovedeste iarasi ca **gramatica nu este LL(1)**.

Despre SLR(1), incepem cu a analiza gramatica. Ea pare a fi ambigua, pentru ca in fiecare pas putem expanda **S**-ul din stanga sau din dreapta, pentru productia **S → SbS**.

O gramatica ambigua nu se va incadra in nicio clasa, nici LL, LR, LALR, SLR, etc. Aceste clase contin exclusiv gramatici neambigue.

Pentru siguranta, construim automatul complet pentru gramatica data.

Starea initiala (0) este:

**S' → •S**  
**S → •SbS**  
**S → •a**

Tranzitia din starea 0 pe **S** duce in urmatoarea stare (1):

**S' → S•**  
**S → S•bS**

Tranzitia din starea 0 pe **a** duce in urmatoarea stare (2):

**S → a•**

Tranzitia din starea 1 pe **b** duce in urmatoarea stare (3):

**S → Sb•S**  
**S → •SbS**  
**S → •a**

Tranzitia din starea 3 pe **S** duce in urmatoarea stare (4):

**S → SbS•**  
**S → S•bS**

Tranzitia din starea 3 pe **a** duce in starea 2.

Tranzitia din starea 4 pe **b** duce in starea 3.

Verificam daca exista conflicte in starile automatului. Se incepe cu identificarea itemilor reduce, deoarece starile fara itemi reduce nu pot avea conflicte.

**Starea 0** - un item shift ⇒ fara conflicte  
**Starea 1** - un item shift si unul reduce ⇒ posibil conflict Shift-Reduce  
**Starea 2** - un singur item ⇒ fara conflicte  
**Starea 3** - un item shift ⇒ fara conflicte  
**Starea 4** - un item shift si unul reduce ⇒ posibil conflict Shift-Reduce

Pentru starea **1** verificam daca **b** se afla in Follow(S').

Follow(S') = {\$}  
**b** ∉ Follow(S')



Cum **b** nu apartine multimii  $\text{Follow}(S')$ , nu este conflict Shift-Reduce in starea **1**.

Analizam similar si pentru starea **4**.

$\text{Follow}(S) = \{b\}$

$b \in \text{Follow}(S)$

Observam ca **b** apartine multimii  $\text{Follow}(S)$ , ceea ce inseamna ca potentialul conflict chiar este un conflict Shift-Reduce in starea **4**.

Cum am gasit o stare a automatului care are un conflict, inseamna ca **gramatica nu este SLR(1)**.

# Stiva analizei bottom-up

---

Fie urmatoarele doua gramatici:

$S \rightarrow (T)$

$T \rightarrow T + \text{int} \mid \text{int}$

$S \rightarrow (T)$

$T \rightarrow \text{int} + T \mid \text{int}$

si urmatorul sir:

( int + int + int + int )

**Caror tipuri de asociativitate le corespund cele doua gramatici?**

Prima gramatica asociaza la stanga, iar a doua asociaza la dreapta.

**Pentru cele doua gramatici s-ar desfasura acelasi numar de actiuni shift?**

Intotdeauna. Actiunea shift se executa mereu o singura data per token si sirul trebuie parcurs pana la sfarsit. Numarul de actiuni shift este egal cu numarul de tokeni ai sirului.

**In ce ordine se fac actiunile shift si reduce pentru cele doua gramatici?**

Prima gramatica:

( int + int + int + int )	⇒ shift
(   int + int + int + int )	⇒ shift
( int   + int + int + int )	⇒ reduce
( T   + int + int + int )	⇒ shift
( T +   int + int + int )	⇒ shift
( T + int   + int + int )	⇒ reduce
( T   + int + int )	⇒ shift
( T +   int + int )	⇒ shift
( T + int   + int )	⇒ reduce
( T   + int )	⇒ shift
( T +   int )	⇒ shift
( T + int   )	⇒ reduce
( T   )	⇒ shift
( T )	⇒ reduce
S	

A doua gramatica:

( int + int + int + int )	⇒ shift
(   int + int + int + int )	⇒ shift
( int   + int + int + int )	⇒ shift
( int +   int + int + int )	⇒ shift
( int + int   + int + int )	⇒ shift
( int + int +   int + int )	⇒ shift

( int + int + int   + int )	⇒ shift
( int + int + int +   int )	⇒ shift
( int + int + int + int   )	⇒ reduce
( int + int + int + T   )	⇒ reduce
( int + int + T   )	⇒ reduce
( int + T   )	⇒ reduce
( T   )	⇒ shift
( T )	⇒ reduce
S	

Operatiile shift adauga elemente pe stiva, iar operatiile reduce consuma portiuni din stiva si adauga un neterminat pe stiva.

Trebuie remarcat faptul ca prima gramatica utilizeaza un spatiu constant pe stiva. Spatiul ocupat este mai mic fata de a doua gramatica, unde spatiul este liniar in numarul de tokeni, deoarece se incarca aproape tot sirul pe stiva inainte sa inceapa reducerea sa.

Analiza bottom-up nu are probleme cu recursivitatea la stanga, spre deosebire de analiza top-down. Ea chiar duce la un proces mai eficient, care utilizeaza mai putina memorie.

# Ordinea productiilor

---

Fie urmatoarea gramatica:

$S \rightarrow baSab \mid baS \mid b$

si urmatorul sir:

babab

***Pentru o strategie recursive descent, care este cea mai buna ordine a productiilor pentru a minimiza numarul de pasi?***

Gramatica ar trebui rescrisa astfel:

**$S \rightarrow b \mid baSab \mid baS$**

Simulam analiza pe sir. Fiecare pas prezinta sirul de la intrare si pozitia cursorului, derivarea obtinuta pana in prezent si evidentiaza terminalii si token-ii de la intrare a caror egalitate se verifica.

**|babab**

**S**

Folosim prima productie a lui **S**, adica  **$S \rightarrow b$** .

**|babab**

**S  $\rightarrow$  b**

Terminalul obtinut **b** se potriveste cu token-ul curent de la intrare, insa derivarea s-a epuizat (nu mai sunt neterminali de expandat) inaintea sirul de la intrare. Astfel mergem un pas inapoi la **S**. Folosim a doua productie a lui S, adica  **$S \rightarrow baSab$** .

**|babab**

**S  $\rightarrow$  b**

**$\hookrightarrow$  baSab**

Terminalii obtinuti **b** si **a** se potrivesc cu primii token-i de dupa cursor, asa ca avansam cursorul si continuam derivarea. Folosim prima productie a lui S, adica  **$S \rightarrow b$** .

**ba|bab**

**S  $\rightarrow$  b**

**$\hookrightarrow$  baSab  $\rightarrow$  babab**

Terminalii obtinuti **b**, **a** si **b** se potrivesc cu primii token-i de dupa cursor, asa ca avansam cursorul. Atat sirul, cat si derivarea s-au terminat, asa ca sirul a fost recunoscut cu succes.

A fost nevoie de un singur pas de backtracking si de doua derivari corecte. S-au verificat terminalii cu token-ii de 6 ori ( $1_b$  din  $S \rightarrow b$  +  $2_{ba}$  din  $S \rightarrow baSab$  +  $3_{bab}$  din  $baSab \rightarrow babab$ ).

# Tipare - aspecte generale

---

## Relatia de mostenire

Considerand urmatoarele doua clase:

```
class Y {};  
  
class X inherits Y {};
```

O relatie de mostenire intre clasele X si Y, sugerand faptul ca X o *mosteneste* pe Y, se noteaza astfel:  $X \leq Y$ .

Pentru orice clasa X, este valabila relatia  $X \leq X$ .

Intr-un program COOL, orice clasa fara clasa parinte extinde implicit clasa **Object**.

Principiul se aplica si claselor predefinite din COOL. Astfel, si ele extind implicit clasa **Object**:

```
Int ≤ Object  
Bool ≤ Object  
String ≤ Object  
IO ≤ Object
```

## SELF\_TYPE

Tipul **SELF\_TYPE** este disponibil doar la compilare pentru a relaxa restrictiile sistemului de tipuri. El are sens numai in raport cu o clasa, de aceea in cadrul evaluarii tipurilor se vor obtine tipuri de forma **SELF\_TYPE<sub>T</sub>**, sugerand posibilele tipuri dinamice ale obiectului, adica T sau chiar subtipuri ale lui T, chiar daca **SELF\_TYPE** este folosit in clasa T.

De exemplu, avand clasele urmatoare:

```
class A {  
    a : SELF_TYPE;  
  
    f() : SELF_TYPE {  
        self  
    };  
};  
  
class B inherits A {  
    b : SELF_TYPE;  
};  
  
class C inherits B {  
    c : SELF_TYPE;  
};  
  
class X {
```

```

xa : A <- new A;
xb : B <- new B;
xc : C <- new C;

xab : A <- new B;
xac : A <- new C;
xbc : B <- new C;
}

```

În cazul apelurilor de metode care întorc **SELF\_TYPE**, tipul apelului trebuie adaptat la o clasă, iar acea clasă este reprezentată de tipul **static** al obiectului pe care se apelează metoda.

Astfel, pentru apelurile **xa.f()**, **xb.f()** și **xc.f()**, vom obține tipurile **A**, **B**, respectiv **C**. Ce a contat în determinarea tipurilor a fost tipul **static** al obiectelor **xa**, **xb** și **xc**, nu tipul dinamic, chiar dacă ele sunt aceleași pentru fiecare obiect în parte.

Acest aspect este mai clar evidențiat în cazul apelurilor **xab.f()**, **xac.f()** și **xbc.f()**, care au tipurile **A**, **A**, respectiv **B**, ele fiind în concordanță cu tipurile **static** ale obiectelor: **A** (**xab**), **A** (**xac**) și **B** (**xbc**). Observăm că **tipurile dinamice** (determinate de construcțiile **new** - anume **B** (**xab**), **C** (**xac**) și **C** (**xbc**)), **nu intervin în determinarea tipurilor apelurilor**.

Analiza **statică** de tip consideră doar tipurile **static** ale obiectelor.

În cazul apelurilor **a.f()**, **b.f()** și **c.f()** (fiecare în interiorul clasei unde este definit obiectul - adică **A**, **B**, respectiv **C**), tipurile acestor apeluri vor fi **SELF\_TYPE<sub>A</sub>**, **SELF\_TYPE<sub>B</sub>**, respectiv **SELF\_TYPE<sub>C</sub>**.

## Relația de mostenire

Relațiile următoare sunt mereu adevărate:

**SELF\_TYPE<sub>C</sub> ≤ C**, pentru orice clasă **C**

**SELF\_TYPE<sub>C</sub> ≤ T**, dacă **C ≤ T** (deoarece **C** sau orice subclasă a sa sunt subclase ale lui **T**)

**SELF\_TYPE<sub>C</sub> ≤ SELF\_TYPE<sub>C</sub>**

În general, o relație care îl include pe **SELF\_TYPE<sub>C</sub>** este adevărată dacă este adevărată pentru orice înlocuire a lui **SELF\_TYPE<sub>C</sub>** cu un tip concret compatibil cu el (adică **C** sau orice subclasă a sa, chiar și posibile subclase viitoare).

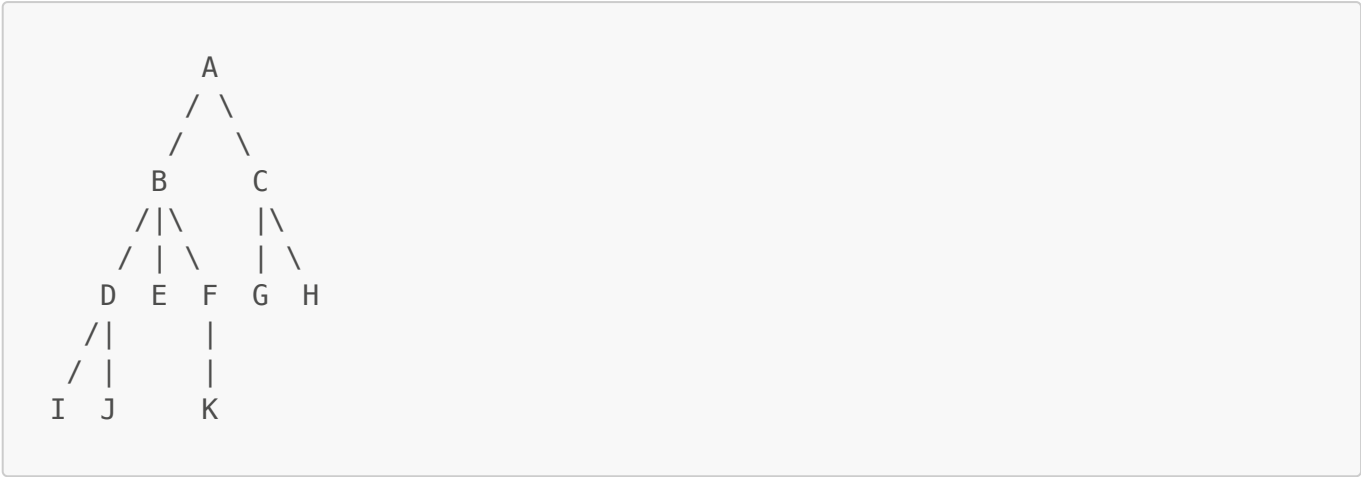
Din acest considerent, relația **T ≤ SELF\_TYPE<sub>C</sub>** este **falsă** întotdeauna, deoarece, considerând ierarhia **A ≤ T ≤ C**, **SELF\_TYPE<sub>C</sub>** ar putea fi înlocuit cu clasă **A**, iar relația obținută **T ≤ A** ar fi invalidă.

Regulile de tipare nu vor duce niciodată la comparația a două tipuri **SELF\_TYPE** raportate la clase diferite, astfel că **nu există** o relație de mostenire de forma **SELF\_TYPE<sub>C<sub>1</sub></sub> ≤ SELF\_TYPE<sub>C<sub>2</sub></sub>**, unde **C<sub>1</sub> ≠ C<sub>2</sub>**.

## Least upper bound

Există situații când, pentru determinarea tipului unei expresii cu mai multe ramuri, trebuie găsit cel mai restrictiv tip care cuprinde toate tipurile ramurilor. Acest tip poartă denumirea de **least upper bound** (*margine superioară minimă*).

De exemplu, consideram urmatoarea ierarhie de clase:

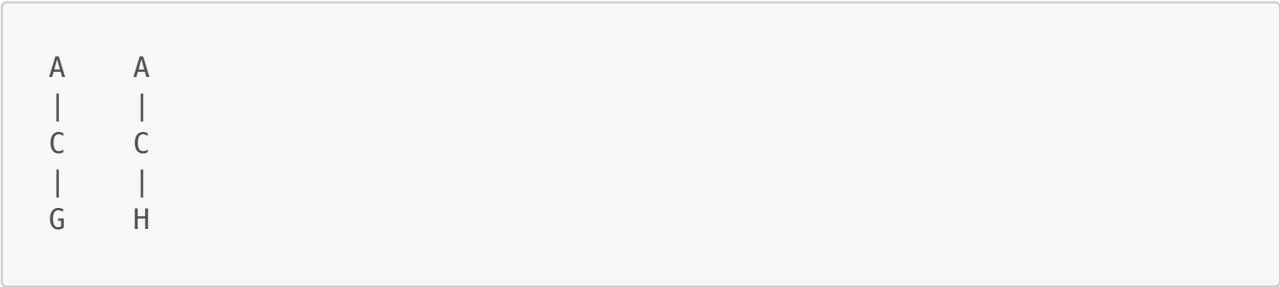


In continuare sunt reprezentate diferite grupuri de clase si *least upper bound*, precum si relatiile si lanturile de mostenire ale claselor:

- $G, H \Rightarrow C$

$$G \leq C \leq A$$

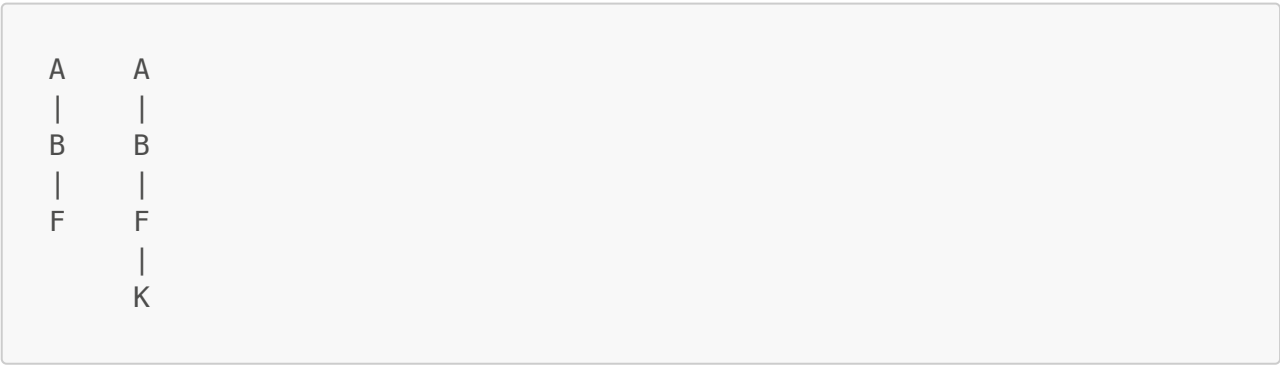
$$H \leq C \leq A$$



- $F, K \Rightarrow F$

$$F \leq B \leq A$$

$$K \leq F \leq B \leq A$$

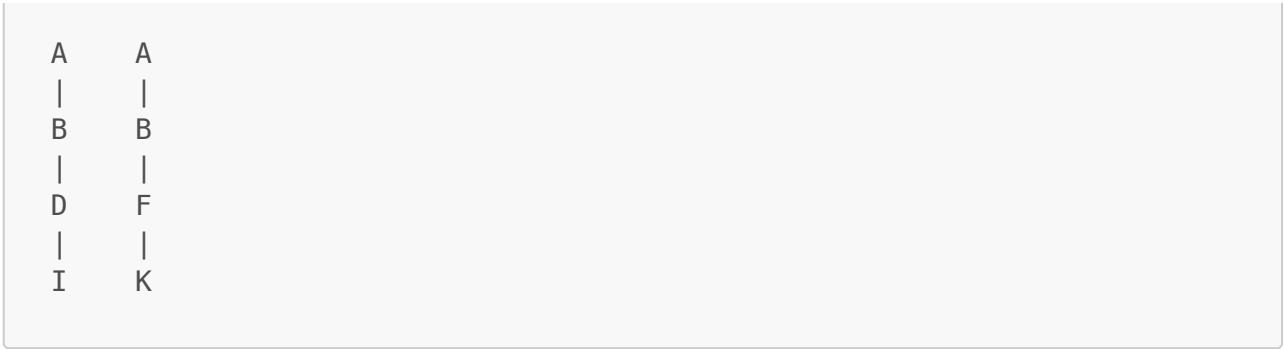


- $I, K \Rightarrow B$

$$I \leq D \leq B \leq A$$

$$K \leq F \leq B \leq A$$

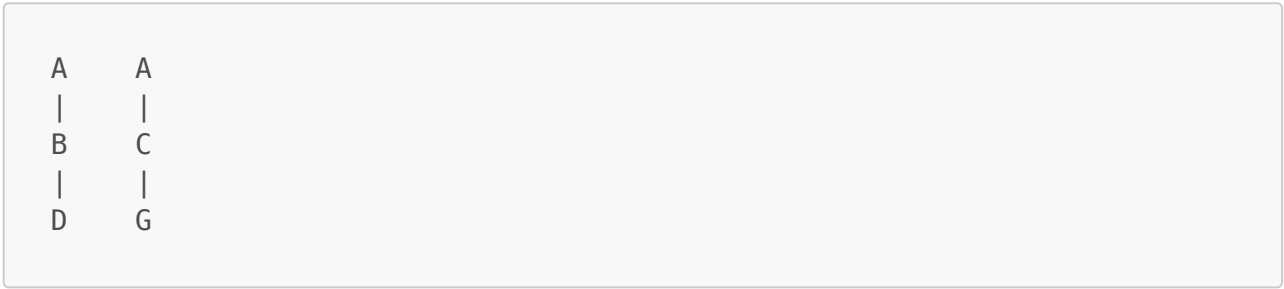




- D, G => A

$$D \leq B \leq A$$

$$G \leq C \leq A$$



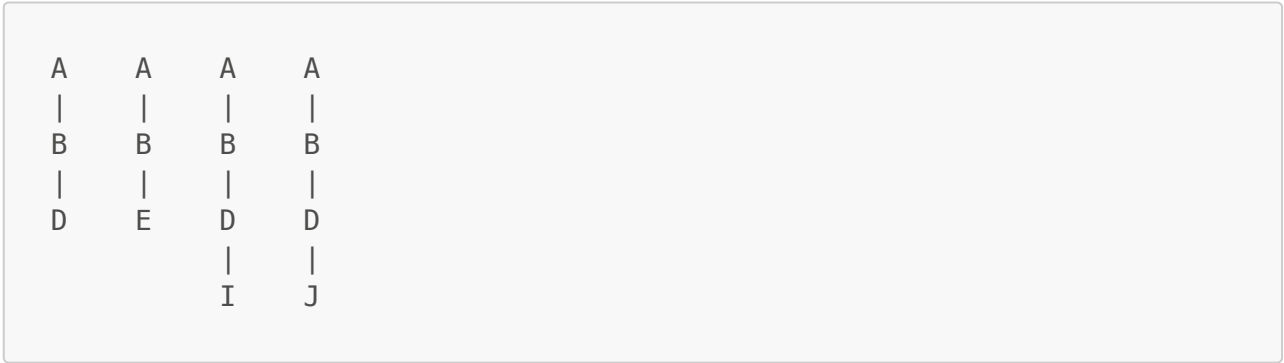
- D, E, I, J => B

$$D \leq B \leq A$$

$$E \leq B \leq A$$

$$I \leq D \leq B \leq A$$

$$J \leq D \leq B \leq A$$



- D, E, H, I, J => A

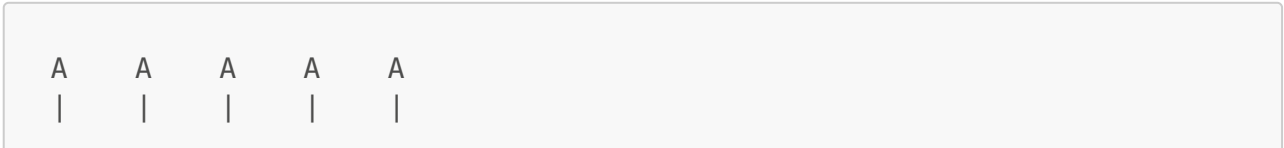
$$D \leq B \leq A$$

$$E \leq B \leq A$$

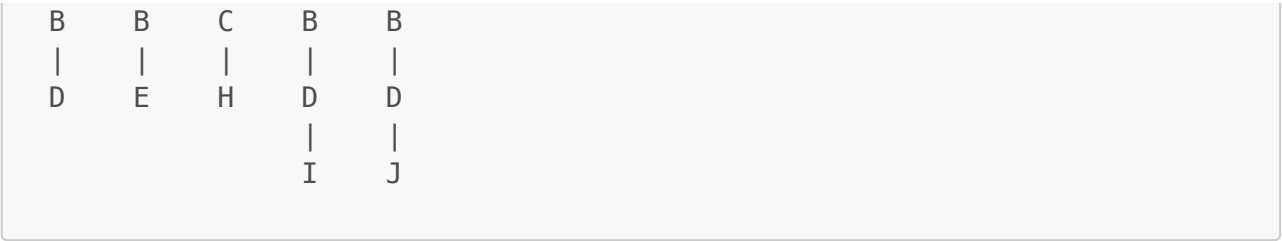
$$H \leq C \leq A$$

$$I \leq D \leq B \leq A$$

$$J \leq D \leq B \leq A$$







Se observa ca *least upper bound* pentru orice grup de clase (tipuri) este, de fapt, clasa *comuna* (care este parte din lantul de mostenire al fiecărei clase din grup) cea mai *joasa* in ierarhia de clase. In terminologia arborilor, un astfel de nod poarta denumirea de *lowest common ancestor* (*cel mai adanc stramos comun*).

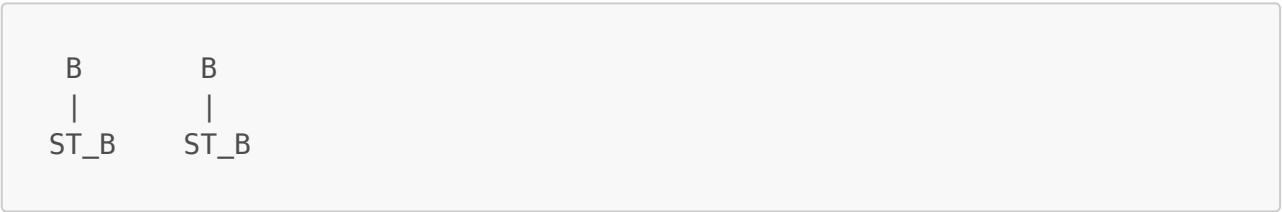
In ceea ce priveste **SELF\_TYPE**, regulile se aplica in aceeasi maniera. Putem considera ca exista o clasa **SELF\_TYPE<sub>T</sub>** care extinde clasa **T**.

Reprezentam diferite grupuri de tipuri, ce cuprind si **SELF\_TYPE**, si *least upper bound*, precum si relatiile si lanturile de mostenire ale tipurilor:

- SELF\_TYPE<sub>B</sub>, SELF\_TYPE<sub>B</sub> => SELF\_TYPE<sub>B</sub>

SELF\_TYPE<sub>B</sub> ≤ B

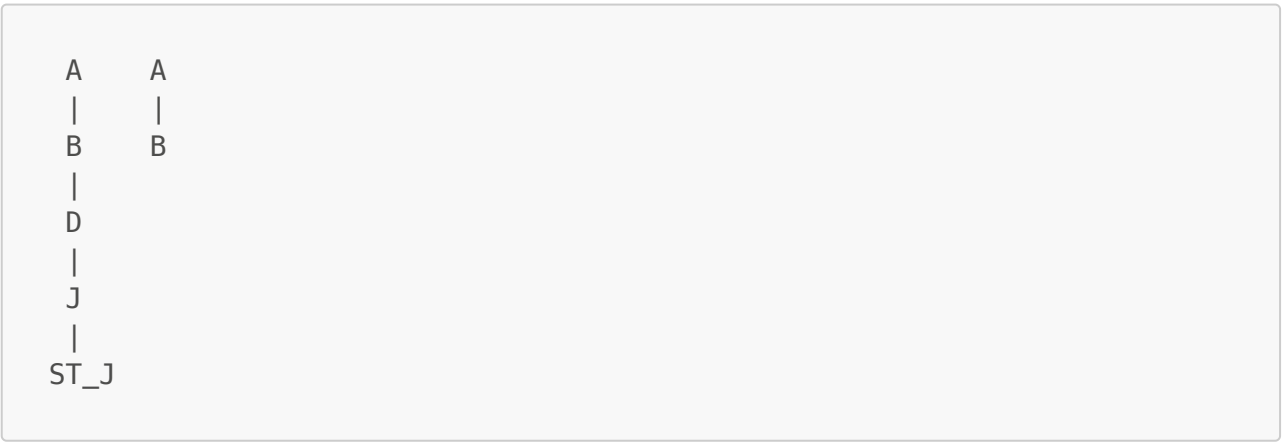
SELF\_TYPE<sub>B</sub> ≤ B



- SELF\_TYPE<sub>J</sub>, B => B

SELF\_TYPE<sub>J</sub> ≤ J ≤ D ≤ B ≤ A

SELF\_TYPE<sub>B</sub> ≤ B ≤ A



- SELF\_TYPE<sub>K</sub>, SELF\_TYPE<sub>G</sub> => A

SELF\_TYPE<sub>K</sub> ≤ K ≤ F ≤ B ≤ A

SELF\_TYPE<sub>G</sub> ≤ G ≤ C ≤ A



A  
|  
B  
|  
F  
|  
K  
|  
ST\_K

A  
|  
C  
|  
G  
|  
ST\_G

# Tipare

---

Considerand un program COOL cu urmatoarele clase:

```
class Bazz {};  
  
class Bar inherits Bazz {};  
  
class Foo inherits Bar {};
```

si urmatoarele expresii:

```
case 3 of y : Int => y; z: Bool => z; esac;           -- 1  
  
if 0 = 1 then new Bar else new Foo fi;               -- 2  
  
let x : Bazz <- new Bar in x;                         -- 3
```

**Care sunt tipurile celor trei expresii, in conformitate cu ierarhia de clase prezentata?**

Intre clasele de mai sus se poate stabili urmatoarea relatie de mostenire care cuprinde toate clasele:

**$\text{Foo} \leq \text{Bar} \leq \text{Bazz} \leq \text{Object}$**

## Expresia 1

Prima ramura a expresiei case,  **$y : \text{Int} \Rightarrow y$** ; are tipul **Int**, deoarece corpul este doar identificatorul **y** care are tipul **Int**, conform tiparii  **$y : \text{Int}$** .

In mod similar, a doua ramura a expresiei case,  **$z : \text{Bool} \Rightarrow z$** ; are tipul **Bool**.

Cum tipul expresiei case este *least upper bound* al tipurilor ramurilor, si cum avem relatiile  $\text{Int} \leq \text{Object}$  si  $\text{Bool} \leq \text{Object}$ , rezulta ca **tipul expresiei 1 este Object**.

## Expresia 2

Ambele ramuri ale expresiei *if* reprezinta expresii *new*. Obiectele create au tipul dinamic determinat de numele clasei instantiate, insa in acest caz, in lipsa unui tip static atribuit, tipul static este acelasi ca tipul dinamic. Astfel ramurile au tipurile **Bar**, respectiv **Foo**.

Conform *least upper bound* necesar pentru evaluare tipului expresiei *if* si conform relatiilor de mostenire  $\text{Bar} \leq \text{Bazz} \leq \text{Object}$  si  $\text{Foo} \leq \text{Bar} \leq \text{Bazz} \leq \text{Object}$ , **expresia 2 are tipul Bar**.

## Expresia 3

Tipul expresiei *let* este tipul corpului acesteia, introdus de *in*. In expresia 3, corpul este doar identificatorului **x**, cu tipul **Bazz** din punct de vedere static. Chiar daca in timpul executiei tipul lui **x** va fi, dinamic, **Bar** (datorita expresiei *new Bar*), la compilare ne intereseaza tipul static. Asadar, **expresia 3 are tipul Bazz**.

# Scoping (Domenii de vizibilitate)

Fie urmatorul program COOL:

```
class Main inherits IO {
  x : Int <- 5;

  foo(z : Int) : Int {
    x + z
  };

  bar(y : Int) : Int {{
    let x : Int <- 1 in
    let z : Int <- 2 in
    foo(y);
  }};

  main() : Object {{
    let x : Int <- 7 in
    out_int(foo(bar(3)));
  }};
};
```

**Ce output va avea programul daca este "statically scoped" (cu domenii de vizibilitate statice)?**

Un program *statically scoped* "leaga" identificatorii de simbolurile lor la compilare, astfel ca la rulare evaluarea unei referinte la o variabila duce la aceeaasi adresa de memorie, indiferent de cum s-a ajuns la acea evaluare. Aproape toate limbajele de programare sunt *statically scoped*: C/C++, Java, Python, JavaScript, COOL, etc.

De exemplu, in clasa de mai sus, identificatorul **x** din metoda **foo** va fi intotdeauna atributul **x** al clasei **Main**, iar identificatorul **z** din aceeasi metoda va fi intotdeauna parametrul acesteia.

Codul urmator este programul COOL de mai sus, avand identificatorii (doar variabile sau parametri ai metodelor) adnotati cu scopul in care acestia sunt definiti sau din care vor fi folositi:

```
class Main inherits IO {
  xMain : Int <- 5;

  foo(zfoo : Int) : Int {
    xMain + zfoo
  };

  bar(ybar : Int) : Int {{
    let xlet1_bar : Int <- 1 in
    let zlet2_bar : Int <- 2 in
```

```

        foo(y_bar);
    });

    main() : Object {
        let x_let_main : Int <- 7 in
            out_int(foo(bar(3)));
    };
};

```

Rularea programului va executa automat metoda **main** care apeleaza metoda **out\_int** din clasa **IO**, afisand rezultatul evaluarii expresiei **foo(bar(3))**.

Apelul **bar(3)** din metoda **main** va "seta" parametrul **y\_bar** cu valoarea **3**. Variabilele introduse prin cele doua instructiuni *let* pot fi ignorate intrucat apelul **foo(y\_bar)** nu refera vreun identificator **x** si **z**. Cum **y\_bar** este **3**, metoda apelata este **foo(3)**. Ca mai devreme, **z\_foo** este "setat" la **3**, iar corpul metodei se evalueaza la **5 + 3**, adica **8**. Prin urmare, apelul **bar(3)** se evalueaza la **8**.

Urmatorul apel efectuat din metoda **main** este **foo(8)**. Parametrul **z\_foo** este acum "setat" la valoarea **8** si corpul metodei se evalueaza la **5 + 8**, adica **13**.

Asadar, output-ul programului, intr-o rulare cu domenii de vizibilitate statice (statically scoped), este **13**.

### ***Ce output ar avea programul daca ar fi "dynamically scoped" (cu domenii de vizibilitate dinamice)?***

Intr-o rulare cu domenii de vizibilitate dinamice, referirea unei variabile poate duce la zone diferite de memorie, in functie de cum s-a ajuns la acea evaluare. Pe fluxul de rulare a programului, fiecare redefinire a unei variabile ascunde valoarea anterioara incepand din acel punct. Exista limbaje de programare care sunt *dynamically scoped* (sau care permit selectarea acestui tip de scoping): bash, LaTeX, Perl, etc.

Trebuie mentionat ca la "intoarcerea" din functii, redefinirile se pierd si definirile vechi de variabile pot fi folosite din nou. Simularea unei evaluari poate fi facuta cu ajutorul unei stive, ca in rezolvarea urmatoare.

Ca si in exercitiul anterior, rezultatul evaluarii expresiei **foo(bar(3))** este si output-ul programului. In continuare se va evalua expresia pas cu pas, reprezentand stiva cu valorile variabilelor si evidentiind noile intrari pe stiva (font cursiv/italic) si intrarile folosite in evaluari (font subliniat).

Incepem cu punctul initial, in metoda **main**, in care exista doar variabila **x**, cea definita ca atribut al clasei **Main**.

```

main
  x = 5

```

Se evalueaza instructiunea *let*, adaugand o noua valoare pentru **x** pe stiva.

```

main
  x = 7
  x = 5

```

Apoi se incepe evaluarea expresiei **bar(3)**. Se ajunge in metoda **bar**, punand pe stiva valoarea parametrului **y**.

```
main → bar
x = 7    y = 3
x = 5    x = 7
         x = 5
```

Se evalueaza cele doua instructiuni *let*, punand pe stiva noi intrari, pentru **x**, apoi pentru **z**.

```
main → bar
x = 7    z = 2
x = 5    x = 1
         y = 3
         x = 7
         x = 5
```

Se apeleaza apoi metoda **foo** cu parametrul **3**, intrucat aceasta este prima valoare intalnita in stiva pentru identificatorul **y**. Pe stiva se va scrie o noua valoare pentru parametrul **z** al metodei.

```
main → bar → foo
x = 7    z = 2    z = 3
x = 5    x = 1    z = 2
         y = 3    x = 1
         x = 7    y = 3
         x = 5    x = 7
                 x = 5
```

Se evalueaza expresia **x + z** citind de pe stiva cele mai recente valori pentru acesti identificatori. Rezulta valoarea **4**, care este si rezultatul apelului metodei **bar**. Se revine in metoda **main** si se evalueaza **foo(4)**, care pune pe stiva o noua valoare pentru parametrul **z**.

```
main → foo
x = 7    z = 4
x = 5    x = 7
         x = 5
```

Se evalueaza iar expresia **x + z**, insa de aceasta data sunt alte valori recente pe stiva. Valoarea rezultata este **11**, valoare ce ajunge sa fie afisata de metoda **out\_int**.

Remarcam faptul ca, desi s-a apelat de doua ori metoda **foo**, modul cum s-a ajuns la apelarea acesteia a contat, intrucat variabila **x** din corpul acesteia a avut valori diferite (1, respectiv 7).

Prin urmare, daca programul ar rula cu domenii de vizibilitate dinamice (dynamically scoped), output-ul ar fi **11**.

# Dispatch

---

Fie urmatorul program COOL:

```
class Main {
  main() : Object {
    (new Bar).bar()
  };
};

class Foo inherits IO {
  foo() : SELF_TYPE {{
    out_string("Foo.foo()\n");
    foo();
    self;
  }};
};

class Bar inherits Foo {
  foo() : SELF_TYPE {{
    out_string("Bar.foo()\n");
    new SELF_TYPE;
  }};

  bar() : Object {
    case foo() of
      f : Foo => f@Foo.foo();
      b : Bar => (new Bazz).foo();
      o : Object => foo();
    esac
  };
};

class Bazz inherits Bar {
  foo() : SELF_TYPE {{
    out_string("Bazz.foo()\n");
    (new Bar)@Foo.foo();
    self;
  }};
};
```

**Care este output-ul obtinut in urma rularii programului?**

Executia programului incepe din metoda **main**, in care se creeaza un obiect de tipul **Bar** atat dinamic, dat de instructiunea **new**, cat si static, intrucat nu exista un tip static declarat (lucru posibil doar la definirea atributelor unei clase sau a variabilelor unei constructii *let*). Pe acest obiect de tipul **Bar** se apeleaza metoda **bar**.

### Main.main → Bar.bar

În această metodă, construcția `case` evaluează tipul întors de apelul **foo()**. Apelul este un *implicit dispatch*, adică obiectul pe care se apelează metoda este **self**, deci apelul este echivalent cu **self.foo()**. Cum **self** este obiectul **Bar** definit mai sus, tipul dinamic este **Bar**, deci se apelează metoda **foo** din clasa **Bar**.

### Main.main → Bar.bar → Bar.foo

În metoda **foo** se afișează un prim mesaj: **"Bar.foo()\n"**. Obiectul pe care s-a apelat metoda are tipul dinamic **Bar**, așadar tipul de retur al funcției este **SELF\_TYPE<sub>Bar</sub>**.

### Main.main → Bar.bar

Înapoi în metoda **bar**, construcția `case` evaluează tipul apelului **foo** la **Bar** (deoarece **SELF\_TYPE<sub>Bar</sub> ≤ Bar**), folosindu-se, astfel, a doua ramură. Se instantiază prin instrucțiunea `new` un obiect cu tipul (static și dinamic) **Bazz**, asupra căruia se apelează metoda **foo**, cea din clasa **Bazz**.

### Main.main → Bar.bar → Bazz.foo

Se afișează al doilea mesaj: **"Bazz.foo()\n"**. Se creează apoi un nou obiect cu tipul dinamic (și static) **Bar**, și, prin *static dispatch*, se apelează metoda **foo** din clasa **Foo**.

### Main.main → Bar.bar → Bazz.foo → Foo.foo

Metoda **foo** a clasei **Foo** afișează un nou mesaj: **"Foo.foo()\n"**. Se apelează apoi **foo()**, care este echivalent cu **self.foo()**, și cum tipul dinamic al lui **self** este **Bar**, se apelează metoda **foo** din clasa **Bar**.

### Main.main → Bar.bar → Bazz.foo → Foo.foo → Bar.foo

Metoda afișează al patrulea mesaj: **Bar.foo()\n**. Execuția se întoarce în metoda **foo** a clasei **Foo**, apoi în metoda **foo** a clasei **Bazz**, apoi în metoda **bar** din clasa **Bar**.

### Main.main → Bar.bar

Ramura a doua a expresiei `case` s-a executat în întregime, deci întreaga expresie s-a încheiat, execuția revenind în metoda **main**.

### Main.main

Cum metoda **main** nu mai conține alte expresii, execuția programului se încheie. În concluzie, output-ul obținut la rularea programului este:

```
Bar.foo()
Bazz.foo()
Foo.foo()
Bar.foo()
```



# Tipare avansata

Fie urmatorul program COOL (o simplificare a programului de la exercitiul anterior):

```
class Main {
  main() : Object {
    (new Bar).bar()
  };
};

class Foo inherits IO {
  foo() : SELF_TYPE { self };
};

class Bar inherits Foo {
  foo() : SELF_TYPE { new SELF_TYPE };

  bar() : (*MISSING*) {
    case foo() of
      f : Foo => f@Foo.foo();
      b : Bar => (new Bazz).foo();
      o : Object => foo();
    esac
  };
};

class Bazz inherits Bar {
  foo() : SELF_TYPE { self };
};
```

**Ce tipuri sunt valide ca tip de retur al metodei `bar` din clasa `Bar`, conform regulilor de verificare statica a tipurilor din COOL?**

Tipul de retur al metodei **bar** trebuie sa fie un tip care sa cuprinda tipul corpului functiei, adica tipul expresiei case:

$$T_{\text{case}} \leq T_{\text{bar}}$$

Pentru a afla tipul expresiei case, trebuie determinate mai intai tipurile celor trei ramuri ale sale.

## Constructia case - Ramura 1

Prima ramura are tipul dat de *static dispatch*-ul **f@Foo.foo()**. Cum obiectul pe care se apeleaza metoda, **f**, are tipul static **Foo** si metoda apelata este cea din clasa **Foo**, tipul de retur **SELF\_TYPE** al metodei **foo** face ca tipul apelurilor catre metoda sa fie **Foo**. Astfel, tipul primei ramuri este **Foo**.

Intr-un mod formal, conform manualului COOL si informatiilor din curs, regulile de tipare folosite sunt (cu tipurile raportate la cele din regulile generale - indicii din fata tipurilor):

$$O(f) = \text{Foo}_{\tau}$$


---

[Var]

$$O, M, \text{Bar} \vdash f : \text{Foo}_{\tau}$$

$$O, M, \text{Bar} \vdash f : \text{Foo}_{\tau_0}$$

$$\text{Foo}_{\tau_0} \leq \text{Foo}_{\tau}$$

$$M(\text{Foo}, \text{foo}) = (\text{SELF\_TYPE}_{\tau_1})$$

$$\text{Foo}_{\tau_1} = \text{Foo}_{\tau_0} \quad (T'_1 = \text{SELF\_TYPE})$$


---

[StaticDispatch]

$$O, M, \text{Bar} \vdash f@ \text{Foo.foo}() : \text{Foo}_{\tau_1}$$

### Constructia case - Ramura 2

In a doua ramura, constructia *new* creaza un obiect cu tipul (static si dinamic) **Bazz**, ceea ce duce la evaluarea apelului metodei **foo** din clasa **Bazz** pe un obiect de acelasi tip. Aplicand rationamentul folosit pentru ramura anterioara, rezulta ca tipul ramurii 2 este **Bazz**.

Intr-un mod formal, conform manualului COOL si informatiilor din curs, regulile de tipare folosite sunt (cu tipurile raportate la cele din regulile generale - indicii din fata tipurilor):

$$\text{Bazz}_{\tau'} = \text{Bazz}_{\tau} \quad (T \neq \text{SELF\_TYPE})$$


---

[New]

$$O, M, \text{Bar} \vdash \text{new } \text{Bazz} : \text{Bazz}_{\tau'}$$

$$O, M, \text{Bar} \vdash \text{new } \text{Bazz} : \text{Bazz}_{\tau_0}$$

$$\text{Bazz}_{\tau'_0} = \text{Bazz}_{\tau_0} \quad (T_0 \neq \text{SELF\_TYPE}_{\text{Bar}})$$

$$M(\text{Bazz}_{\tau'_0}, \text{foo}) = (\text{SELF\_TYPE}_{\tau'_1})$$

$$\text{Bazz}_{\tau_1} = \text{Bazz}_{\tau_0} \quad (T'_1 = \text{SELF\_TYPE})$$


---

[Dispatch]

$$O, M, \text{Bar} \vdash (\text{new } \text{Bazz}).\text{foo}() : \text{Bazz}_{\tau_1}$$

### Constructia case - Ramura 3

In cazul ramurii 3, apelul *implicit dispatch* este, de fapt, **self.foo()**. Cum **self** are "tipul" static **SELF\_TYPE<sub>Bar</sub>**, iar metoda apelata **foo**, din clasa **Bar**, are tipul de retur **SELF\_TYPE**, atunci apelul de metoda are tipul **SELF\_TYPE<sub>Bar</sub>**. Tipul celei de-a treia ramura este, asadar, **SELF\_TYPE<sub>Bar</sub>**.

Intr-un mod formal, conform manualului COOL si informatiilor din curs, regulile de tipare folosite sunt (cu tipurile raportate la cele din regulile generale - indicii din fata tipurilor):

---

**[Self]**

$O, M, Bar \vdash self : \text{SELF\_TYPE}_{Bar}$   
 $\tau$

$O, M, Bar \vdash self : \text{SELF\_TYPE}_{Bar}$   
 $\tau_0$

$\tau'_0 \quad Bar = \text{SELF\_TYPE}_{Bar} \quad (T_0 = \text{SELF\_TYPE}_{Bar})$

$M(\tau'_0 \quad Bar, foo) = (\tau'_1 \quad \text{SELF\_TYPE})$

$\tau_1 \quad \text{SELF\_TYPE}_{Bar} = \text{SELF\_TYPE}_{Bar} \quad (T'_1 = \text{SELF\_TYPE})$

---

**[Dispatch]**

$O, M, Bar \vdash foo() : \text{SELF\_TYPE}_{Bar}$   
 $\tau_1$

### Constructia case

Cele trei tipuri ale ramurilor constructiei case sunt:

**Foo**

**Bazz**

**SELF\\_TYPE<sub>Bar</sub>**

Tipul intregii constructii case este *least upper bound* al tipurilor ramurilor, asa cum reiese si din regula de tipare, conform manualului COOL si informatiilor din curs (cu tipurile raportate la cele din regula generala - indicii din fata tipurilor):

$O, M, Bar \vdash foo() : \text{SELF\_TYPE}_{Bar}$   
 $\tau_0$

$O[\tau_1 \quad Foo / f], M, Bar \vdash f@Foo.foo() : Foo$   
 $\tau'_1$

$O[\tau_2 \quad Bar / b], M, Bar \vdash (new Bazz).foo() : Bazz$   
 $\tau'_2$

$O[\tau_3 \quad Object / o], M, Bar \vdash foo() : \text{SELF\_TYPE}_{Bar}$   
 $\tau'_3$

$\tau \quad Foo = lub(\tau'_1 \quad Foo, \tau'_2 \quad Bazz, \tau'_3 \quad \text{SELF\_TYPE}_{Bar})$

---

**[Case]**

$O, M, Bar \vdash case\ foo() \text{ of } \dots \text{ esac} : Foo$   
 $\tau$

Pentru a determina *least upper bound*, au fost luate in calcul relatiile de mostenire ale celor trei tipuri:

$Foo \leq IO \leq Object$

$Bazz \leq Bar \leq Foo \leq IO \leq Object$

$\text{SELF\_TYPE}_{Bar} \leq Bar \leq Foo \leq IO \leq Object$

Se observa ca cea mai restrictiva clasa comuna este clasa **Foo**.

### Metoda bar

Asa cum am mentionat la inceput, tipul de retur al metodei **bar** poate sa fie orice tip care cuprinde tipul constructiei case:  $T_{case} \leq T_{bar}$ .

Tinand cont de tipul **Foo** al constructiei *case* si de ierarhia de clase  $Foo \leq IO \leq Object$ , rezulta ca metoda **bar** poate avea ca tip de retur 3 clase:

**Foo**

**IO**

**Object**