Analiza lexicala

Fie urmatoarea specificatie lexicala:

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

Care este secventa de reguli lexicale (1-3) utilizate in analiza urmatorului sir abbaabbababababbaaababaa?

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 lexicale 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 *$*.

$\vert^{123}$ abbaabbababababbaaababaa	-
a $\vert^{12}$ bbaabbababababbaaababaa	-
ab $\vert^{12}$ baabbababababbaaababaa	-
abb $\vert^{12}$ aabbababababbaaababaa	-
abba $\vert^{12}$ abbababababbaaababaa	2
abba $\vert^{2}$ a $\vert^{1}$ bbababababbaaababaa	2
abba $\vert^{2}$ ab $\vert^{1}$ bababababbaaababaa	2
abba $\vert^{2}$ abb $\vert^{1}$ abababbaaababaa	2
abba $\vert^{2}$ abba $\vert^{1}$ bababbaaababaa	2
abba $\vert^{2}$ abbab $\vert^{1}$ ababbaaababaa	2
abba $\vert^{2}$ abbaba $\vert^{1}$ babbaaababaa	2
abba $\vert^{2}$ abbabab $\vert^{1}$ abbaaababaa	2
abba $\vert^{2}$ abbababa $\vert^{1}$ bbaaababaa	2
abba $\vert^{2}$ abbababab $\vert^{1}$ baaababaa	2
abba $\vert^{2}$ abbabababb $\vert$ 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$_2$	$|^{123}$ abbababaabbaaababaa	-
abba$_2$	a $|^{12}$ bbabababbaaababaa	-
abba$_2$	ab $|^{12}$ bababa bbaaababaa	-
abba$_2$	abb $|^{12}$ abababbaaababaa	-
abba$_2$	abba $|^{12}$ bababbaaababaa	2
abba$_2$	abba $|^{2}$ b $|^{1}$ ababbaaababaa	2
abba$_2$	abba $|^{2}$ ba $|^{1}$ babbaaababaa	2
abba$_2$	abba $|^{2}$ bab $|^{1}$ abbaaababaa	2
abba$_2$	abba $|^{2}$ baba $|^{1}$ bbaaababaa	2
abba$_2$	abba $|^{2}$ babab $|^{1}$ baaababaa	2
abba$_2$	abba $|^{2}$ bababb | aaababaa	**2**

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$_2$ abba$_2$	$|^{123}$ bababbaaababaa	-
abba$_2$ abba$_2$	b $|^{13}$ ababbaaababaa	-
abba$_2$ abba$_2$	ba $|^{13}$ babbaaababaa	-
abba$_2$ abba$_2$	bab $|^{13}$ abbaaababaa	3
abba$_2$ abba$_2$	bab $|^{3}$ a $|^{1}$ bbaaababaa	3
abba$_2$ abba$_2$	bab $|^{3}$ ab $|^{1}$ baaababaa	3
abba$_2$ abba$_2$	bab $|^{3}$ abb | aaababaa	**3**

Singura regula care a recunoscut o portiune din sir este regula **3**. Regula **1** a intampinat aceeasi problema ca si in parcurgerile anterioare.
Se utilizeaza regula **3** pentru subsirul **bab** si se continua analiza.

abba$_2$ abba$_2$ bab$_3$	$|^{123}$ abbaaababaa	-
abba$_2$ abba$_2$ bab$_3$	a $|^{12}$ bbaaababaa	-
abba$_2$ abba$_2$ bab$_3$	ab $|^{12}$ baaababaa	-
abba$_2$ abba$_2$ bab$_3$	abb $|^{12}$ aaababaa	-
abba$_2$ abba$_2$ bab$_3$	abba $|^{12}$ aababaa	2
abba$_2$ abba$_2$ bab$_3$	abba $|^{2}$ a $|^{1}$ ababaa	2
abba$_2$ abba$_2$ bab$_3$	abba $|^{2}$ aa $|^{1}$ babaa	2 & **1**

De data aceasta, atat regula **1**, cat si regula **2** au recunoscut cu succes subsiruri, insa conform regulii de baza *maximal munch*, se va utiliza regula care a consumat cel mai mult din sir.
Se utilizeaza, deci, regula **1** care recunoaste **abbaaa** din sirul de la intrare si analiza continua.

$abba_2$ $abba_2$ bab_3 $abbaaa_1$	$\|^{123}$ babaa	-
$abba_2$ $abba_2$ bab_3 $abbaaa_1$	b $\|^{13}$ abaa	-
$abba_2$ $abba_2$ bab_3 $abbaaa_1$	ba $\|^{13}$ baa	-
$abba_2$ $abba_2$ bab_3 $abbaaa_1$	bab $\|^{13}$ aa	3
$abba_2$ $abba_2$ bab_3 $abbaaa_1$	bab $\|^{3}$ a $\|^{1}$ a	3
$abba_2$ $abba_2$ bab_3 $abbaaa_1$	bab $\|^{3}$ aa $\|^{1}$	3
$abba_2$ $abba_2$ bab_3 $abbaaa_1$	bab $\|^{3}$ aa | $	**3**

Situatia actuala este una noua. Regula **1** esueaza intrucat sirul de la intrare s-a consumat inainte ca regula sa recunoasca un subsir. Regula astepta sa regaseasca *ab* sau *aa* in sir, dar a gasit doar *a*.
Intrucat este singura regula care a recunoscut un subsir, se utilizeaza regula **3** pentru subsirul **bab** si se continua analiza.

$abba_2$ $abba_2$ bab_3 $abbaaa_1$ bab_3	$\|^{123}$ aa	-
$abba_2$ $abba_2$ bab_3 $abbaaa_1$ bab_3	a $\|^{12}$ a	-
$abba_2$ $abba_2$ bab_3 $abbaaa_1$ bab_3	aa $\|^{1}$	1
$abba_2$ $abba_2$ bab_3 $abbaaa_1$ bab_3	aa $\|^{1}$ $	**1**

Regula **1** este singura care recunoaste o portiune din sirul ramas, mai exact tot ce a ramas din sir. Se utilizeaza regula **1** care recunoaste **aa** din sirul de la intrare si analiza se opreste intrucat s-a consumat tot sirul.

Asadar, secventa de reguli lexicale utilizate pentru analiza sirului este

2 2 3 1 3 1

$abba_2$ $abba_2$ bab_3 $abbaaa_1$ bab_3 aa_1

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:

$S \to A\alpha \mid \delta$

$A \to S\beta$

* α si β 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: $\mathbf{S \to^* S\ldots}$ (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: $\mathbf{S \to S\beta\alpha \mid \delta}$.
Construim cativa pasi de derivare: $\mathbf{S \to S\beta\alpha \to S\beta\alpha\beta\alpha \to S\beta\alpha\beta\alpha\beta\alpha \to \delta\beta\alpha\beta\alpha\beta\alpha}$.
Observam ca sirurile care se pot genera incep cu δ si apoi au perechi $\beta\alpha$ (sau niciuna, cand $\mathbf{S \to \delta}$ inca din primul pas).

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

$\mathbf{S \to \delta A}$

$\mathbf{A \to \beta\alpha A \mid \varepsilon}$

Arbori de derivare

Fie urmatoarea gramatica:

E → E ∗ E | E + E | (E) | 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:

```
  *
 / \
2   7
```

> 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:

```
      E
     /|\
    / | \
   (  E  )
     /|\
    / | \
   E  *  E
   |     |
int:2   int:7
```

Ceilalti operatori (**5 <u>∗</u> 3 <u>+</u> (2 ∗ 7) <u>+</u> 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**.

   ```
   5 * (3 + (2 * 7) + 4)
       ^               ^


     *
    / \
   5   ...
   ```

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))
         ^            ^

  *
 / \
5   +
   / \
  3   +
     / \
    *   4
   / \
  2   7
```

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

```
5 * ((3 + (2 * 7)) + 4)
     ^            ^

  *
 / \
5   +
   / \
  +   4
 / \
3   *
   / \
  2   7
```

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))
^      ^   ^            ^

     +
    / \
```

```
     /    \
    *      +
   / \    | \
  5   3  *   4
        / \
       2   7
```

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 → aSb | Sb | 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.

```
   S              S
  /|\            / \
 a S b          S   b
  / \          /|\
 S   b        a S b
 |              |
 b              b
```

Recursive descent

Fie urmatoarea gramatica:

S → A | B | 0S
A → 0A | 0
B → 1

si urmatorul sir:

0^n1 (0 de n ori si apoi 1)

Folosind o strategie recursive descent, care este numarul de pasi pe care ii realizeaza parser-ul, in notatie asimptotica Θ(...), in functie de n?

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

S' → S$ S' 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 → 0A**.

|001$
S' → S$ → A$ → 0A$

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

0|01$
S' → S$ → A$ → 0A$ → 00A$

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

00|1$
S' → S$ → A$ → 0A$ → 00A$ → 000A$

Terminalul obtinut **0** *nu* se potriveste cu token-ul curent de la intrare (**1**), asa ca mergem o derivare inapoi si incercam urmatoarea productie. Folosim a doua productie a lui A, adica **A → 0**.

00|1$
S' → S$ → A$ → 0A$ → 00A$ → 000A$
↳ 000$

Terminalul obtinut **0** *nu* se potriveste cu token-ul curent de la intrare (**1**), asa ca mergem o derivare inapoi si incercam urmatoarea productie. Cum nu mai sunt productii posibile, derivarea se intoarce la **0A**, se intoarce si cursorul si se incearca urmatoare productie. Folosim a doua productie a lui A, adica **A → 0**.

0|01$
S' → S$ → A$ → 0A$ → 00A$ → 000A$
| ↳ 000$
↳ 00$

Terminalul obtinut **0** se potriveste cu token-ul curent de la intrare, iar cum dupa acesta este tot un terminal, avansam cursorul continuam verificarea token-terminal.

00|1$
S' → S$ → A$ → 0A$ → 00A$ → 000A$
| ↳ 000$
↳ 00$

Terminalul obtinut **$** *nu* se potriveste cu token-ul curent de la intrare (**1**). In situatia actuala, derivarea s-a epuizat (a esuat potrivirea lui **$** si nu mai sunt neterminali de expandat) inaintea sirului de la intrare. Astfel mergem iar un pas inapoi la **A**, insa cursorul se muta 2 pozitii, deoarece s-au facut 2 verificari de tokeni cu terminali. Folosim a doua productie a lui A, adica **A → 0**.

|001$
S' → S$ → A$ → 0A$ → 00A$ → 000A$
| | ↳ 000$
| ↳ 00$
↳ 0$

Terminalul obtinut **0** se potriveste cu token-ul curent de la intrare, iar cum dupa acesta este tot un terminal, avansam cursorul si continuam verificarea token-terminal.

0|01$
S' → S$ → A$ → 0A$ → 00A$ → 000A$
| | ↳ 000$
| ↳ 00$
↳ 0$

Terminalul obtinut **$** *nu* se potriveste cu token-ul curent de la intrare (**0**). Ca in situatia precedenta, acest lucru sugereaza ca derivarea s-a epuizat inaintea sirului de la intrare. Mergem un pas inapoi la **S**, cursorul se intoarce 2 pozitii, si incercam urmatoarea productie. Folosim a doua productie a lui S, adica **S → B**.

|001$
S' → S$ → A$ → 0A$ → 00A$ → 000A$

| | | ↳ 000$
| | ↳ 00$
| ↳ 0$
↳ B$

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

|001$
S' → S$ → A$ → 0A$ → 00A$ → 000A$
| | | ↳ 000$
| | ↳ 00$
| ↳ 0$
↳ B$ → 1$

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**.

|001$
S' → S$ → A$ → 0A$ → 00A$ → 000A$
| | | ↳ 000$
| | ↳ 00$
| ↳ 0$
↳ B$ → 1$
↳ 0S$

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**.

0|01$
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:

$S \rightarrow A(S)B \mid \varepsilon$
$A \rightarrow S \mid SB \mid x \mid \varepsilon$
$B \rightarrow SB \mid y$

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

Care sunt regulile multimilor First pentru o secventa de neterminali?

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

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

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

$First(S) \supseteq First(A(S)B) \cup \{\varepsilon\}$ *(1)*
$First(A) \supseteq First(S) \cup First(SB) \cup \{x, \varepsilon\}$ *(2)*
$First(B) \supseteq First(SB) \cup \{y\}$ *(3)*
$First(A(S)B) = First(A) \setminus \{\varepsilon\} \cup \{(\}$ *(4)*
1 & 4 $\Rightarrow First(S) \supseteq First(A) \setminus \{\varepsilon\} \cup \{(, \varepsilon\}$
$\underline{First(S)} \supseteq First(A) \cup \{(, \varepsilon\}$ *(5)*
$First(SB) = First(S) \setminus \{\varepsilon\} \cup First(B)$ *(6)*
2 & 6 $\Rightarrow First(A) \supseteq First(S) \cup First(S) \setminus \{\varepsilon\} \cup First(B) \cup \{x, \varepsilon\}$
$\underline{First(A)} \supseteq First(S) \cup First(B) \cup \{x, \varepsilon\}$ *(7)*
3 & 6 $\Rightarrow First(B) \supseteq First(S) \setminus \{\varepsilon\} \cup First(B) \cup \{y\}$
$\underline{First(B)} \supseteq First(S) \setminus \{\varepsilon\} \cup \{y\}$
5 & 7 $\Rightarrow First(A) = First(S)$

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

$\mathbf{First(S) = First(A) = \{(, \varepsilon, y, x\}}$
$\mathbf{First(B) = \{(, y, x\}}$

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: $\mathbf{Y \rightarrow^* \ldots Xt\ldots \Rightarrow t \in Follow(X)}$.

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

First(A) = First(S) = {(, ε, y, x} *(01)*

First(B) = {(, y, x} *(02)*

Follow(S) ⊇ {$} *(1)*

S → A(S)B ⇒ Follow(S) ⊇ {)} *(2)*

A → S ⇒ Follow(S) ⊇ Follow(A) *(3)*

A → SB & 02 ⇒ Follow(S) ⊇ First(B) \ {ε} *(4)*

B → SB & 02 ⇒ Follow(S) ⊇ First(B) \ {ε} *(5)*

1-5 & 02 ⇒ Follow(S) ⊇ {$} ∪ {)} ∪ Follow(A) ∪ {(, y, x}

<u>Follow(S)</u> ⊇ Follow(A) ∪ {$,), (, y, x} *(6)*

S → A(S)B ⇒ Follow(A) ⊇ {(}

<u>Follow(A)</u> ⊇ {(}

S → A(S)B ⇒ Follow(B) ⊇ Follow(S) *(7)*

A → SB ⇒ Follow(B) ⊇ Follow(A) *(8)*

B → SB ⇒ Follow(B) ⊇ Follow(B) *(9)*

7-9 ⇒ Follow(B) ⊇ Follow(S) ∪ Follow(A) ∪ Follow(B)

Follow(B) ⊇ Follow(S) ∪ Follow(A) *(10)*

6 & 10 ⇒ <u>Follow(B)</u> ⊇ Follow(S)

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

Follow(A) = {(}

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

Prefixe viabile si conflicte

Fie urmatoarea gramatica:

S → A(S)B | ε
A → S | SB | x | ε
B → SB | y

cu urmatoarele multimi First si Follow

First(S) = {(, ε, y, x}
First(A) = {(, ε, y, x}
First(B) = {(, y, x}
Follow(S) = {$,), (, y, x}
Follow(A) = {(}
Follow(B) = {$,), (, y, x}

Cum se construieste automatul care recunoaste prefixele viabile?

- Se creaza un nou simbol de start **S'**.
- Se introduce o noua productie **S' → S**.

Se contruieste un AFN:

- starile sunt **itemi**, adica productii in care se evidentiaza cat a fost observat din partile lor drepte folosindu-se simbolul •;
- starea initiala este itemul **S' → •S**, care sugereaza ca inca nu am parcurs nimic, urmand sa parcurgem S;
- se adauga tranzitii pe primul simbol din dreapta punctului (terminal sau neterminal), ajungand intr-o stare (item) ce contine punctul dupa simbolul respectiv (de exemplu, o tranzitie pe **S** duce automatul in itemul **S' → S•**);
- se adauga ε-tranzitii pentru fiecare prim neterminal din dreapta punctului, ajungand in itemi (stari) corespunzatori productiilor acelui neterminal, pastrand punctul in fata productiei;
- ultimii 2 pasi se repeta pana cand automatul este complet.

Se transforma AFN-ul intr-un AFD:

- starile sunt **multimi de itemi**;
- o stare si toate starile in care se ajunge prin ε-tranzitii sunt grupare intr-o noua stare a AFD-ului.

Se poate construi direct AFD-ul, fara a construi si AFN-ul:

- in starea initiala se adauga itemul **S' → •S**;
- pentru orice item care are punctul in fata unui neterminal, se adauga in aceeasi stare itemi pentru toate productiile acelui neterminal, pastrand punctul in fata productiei;
- se adauga tranzitii pe primele simboluri din dreapta punctelor (terminal sau neterminal);
- ultimii 2 pasi se repeta pana cand automatul este complet.

Care este starea initiala a automatului care recunoaste prefixele viabile ale acestei gramatici?

Starea initiala cuprinde, pentru inceput, noul item creat:

S' → •S

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

S → •A(S)B

S → •

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

A → •S

A → •SB

A → •x

A → •

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

Starea initiala este:

S' → •S

S → •A(S)B

S → •

A → •S

A → •SB

A → •x

A → •

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 → α•tβ**). 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 → α•**). Altfel spus, itemul sugereaza ca se poate *reduce* secventa α 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 → •   reduce - 1

A → •x shift - 2
A → • 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 neterminalul obtinut (dupa reductie) se afla un token ce nu ii poate urma vreodata. Formal, acest lucru s-ar scrie:

X → α•
sirul de la intrare: ... | t ... (cursorul inaintea lui *t*)
t ∉ Follow(X)

⇒ X → α• - 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 → •
A → •x

A → •x
A → •

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 multimi 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 multimi; vom avea o singura posibilitate de reduce de fiecare data).

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

S → •
A → •

Follow(S) ∩ Follow(A) = {$,), (, y, x} ∩ {(} = {(}

Cum intersectia celor doua multimi 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 → •	reduce	- 1
A → •x	shift	- 2
A → •	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:

Stmt → var | IfStmt

IfStmt → if var then Stmt | 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.

```
          Stmt
            |
         IfStmt
 _______|________
 if var then Stmt
             |
          IfStmt
  ____________|_____________
  if var then Stmt else Stmt
               |         |
              var       var

          Stmt
            |
         IfStmt
 _______|__________________
 if var then Stmt else Stmt
             |          |
          IfStmt       var
       _______|________
       if var then Stmt
                    |
                   var
```

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:

$S \to Sb \mid a$

Este gramatica LL(1)?

Doar analizand gramatica, reiese ca este recursiva la stanga (din cauza productiei **S → 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:

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

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:

First(S) = {a}
Follow(S) = {b, $}

Completam tabelul LL(1):

- pe linii se pun neterminalii si pe coloane terminalii si simbolul $ (sfarsitul sirului);
- pentru fiecare productie **X → α**, se obtine First(α) si pentru fiecare terminal **t** din multime se pune in tabel la insersectia **[X, t]** partea dreapta a productiei, adica **α**;
- daca **ε** ∊ **First(X)**, se obtine Follow(X) si pentru fiecare terminal **t** din multime se pune **ε** in tabel la intersectia **[X, t]**.

Tabelul este initial gol:

	a	**b**	**$**
S			

Pentru productia **S → Sb**:

First(Sb) = First (S) = {a}
[S, a] = Sb

	a	**b**	**$**
S	Sb		

Pentru productia **S → a**:

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 → ε?

Gramatica arata acum astfel:

S → Sb | a | ε

Recalculam multimile First si Follow:

First(S) = {a, b, ε}

Follow(S) = {b, $}

Tabelul este initial gol:

	a	b	$
S			

Pentru productia **S → Sb**:

First(Sb) = First (S) \ ε ∪ {b} = {a, b}

[S, a] = Sb

[S, b] = Sb

	a	b	$
S	Sb	Sb	

Pentru productia **S → a**:

First({a}) = {a}

[S, a] = a

	a	b	$
S	Sb a	Sb	

Pentru productia **S → ε**:

Follow(S) = {b, $}

[S, b] = ε
[S, $] = ε

	a	b	$
S	Sb a	Sb ε	ε

Acesta ar fi tabelul LL(1) al gramaticii dupa ce a fost imbogatita cu productia S → ε.

Este gramatica initiala SLR(1)?

Se construieste automatul complet pentru gramatica data.

Starea initiala (0) este:

S' → •S
S → •Sb
S → •a

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

S' → S•
S → S•b

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•

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 singur item ⇒ fara conflicte

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

Follow(S') = {$}
b ∉ 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 → Sb in S → SbS, este gramatica LL(1) si/sau SLR(1)?

Gramatica arata acum astfel:

S → SbS | a

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

	a	b	$
S	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 Follow(S'), nu este conflict Shift-Reduce in starea **1**.

Analizam similar si pentru starea **4**.

Follow(S) = {b}

b $\in$ Follow(S)

Observam ca **b** apartine multimii 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 → (T)
T → T + int | int

S → (T)
T → int + T | 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 neterminal 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 → baSab | baS | 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 → b | baSab | 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 → b**.

|babab
S → 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 → baSab**.

|babab
S → b
↳ 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 → b**.

ba|bab
S → b
↳ baSab → 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 ($1_{\text{b din S → b}}$ + $2_{\text{ba din S → baSab}}$ + $3_{\text{bab din baSab → babab}}$).

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 ≤ Y**.

Pentru orice clasa **X**, este valabila relatia **X ≤ 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 **$\textbf{SELF\_TYPE}_T$**, 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;
}
```

In cazul apelurilor de metode care intorc **SELF_TYPE**, tipul apelului trebuie adaptat la o clasa, iar acea clasa este reprezentata de tipul **static** al obiectului pe care se apeleaza metoda.

Astfel, pentru apelurile **xa.f()**, **xb.f()** si **xc.f()**, vom obtine tipurile **A**, **B**, respectiv **C**. Ce a contat in determinarea tipurilor a fost tipul **static** al obiectelor **xa**, **xb** si **xc**, nu tipul dinamic, chiar daca ele sunt aceleasi pentru fiecare obiect in parte.

Acest aspect este mai clar evidentiat in cazul apelurilor **xab.f()**, **xac.f()** si **xbc.f()**, care au tipurile **A**, **A**, respectiv **B**, ele fiind in concordanta cu tipurile **statice** ale obiectelor: **A** (**xab**), **A** (**xac**) si **B** (**xbc**). Observam ca **tipurile dinamice** (determinate de constructiile **new** - anume **B** (**xab**), **C** (**xac**) si **C** (**xbc**)), **nu intervin in determinarea tipurilor apelurilor**.

> Analiza **statica** de tip considera doar tipurile **statice** ale obiectelor.

In cazul apelurilor **a.f()**, **b.f()** si **c.f()** (fiecare in interiorul clasei unde este definit obiectul - adica **A**, **B**, respectiv **C**), tipurile acestor apeluri vor fi $\mathbf{SELF\_TYPE_A}$, $\mathbf{SELF\_TYPE_B}$, respectiv $\mathbf{SELF\_TYPE_C}$.

Relatia de mostenire

Relatiile urmatoare sunt mereu adevarate:

$\mathbf{SELF\_TYPE_C \leq C}$, pentru orice clasa **C**

$\mathbf{SELF\_TYPE_C \leq T}$, daca $\mathbf{C \leq T}$ (deoarece **C** sau orice subclasa a sa sunt subclase ale lui **T**)

$\mathbf{SELF\_TYPE_C \leq SELF\_TYPE_C}$

In general, o relatie care il include pe $\mathbf{SELF\_TYPE_C}$ este adevarata daca este adevarata pentru orice inlocuire a lui $\mathbf{SELF\_TYPE_C}$ cu un tip concret compatibil cu el (adica **C** sau orice subclasa a sa, chiar si posibile subclase viitoare).
Din acest considerent, relatia $T \leq SELF\_TYPE_C$ este **falsa** intotdeauna, deoarece, considerand ierarhia $A \leq T \leq C$, $SELF\_TYPE_C$ ar putea fi inlocuit cu clasa A, iar relatia obtinuta $T \leq A$ ar fi invalida.

Regulile de tipare nu vor duce niciodata la comparatia a doua tipuri SELF_TYPE raportate la clase diferite, astfel ca **nu exista** o relatie de mostenire de forma $SELF\_TYPE_{C_1} \leq SELF\_TYPE_{C_2}$, unde $C_1 \neq C_2$.

Least upper bound

Exista situatii cand, pentru determinarea tipului unei expresii cu mai multe ramuri, trebuie gasit cel mai restrictiv tip care cuprinde toate tipurile ramurilor. Acest tip poarta denumirea de **least upper bound** (*margine superioara minimala*).

De exemplu, consideram urmatoarea ierarhie de clase:

```
        A
       / \
      /   \
     B     C
    /|\    |\
   / | \   | \
  D  E  F  G  H
 /|     |
/ |     |
I  J     K
```

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

- G, H => C

  G ≤ C ≤ A
  H ≤ C ≤ A

  ```
  A    A
  |    |
  C    C
  |    |
  G    H
  ```

- F, K => F

  F ≤ B ≤ A
  K ≤ F ≤ B ≤ A

  ```
  A    A
  |    |
  B    B
  |    |
  F    F
       |
       K
  ```

- I, K => B

  I ≤ D ≤ B ≤ A
  K ≤ F ≤ B ≤ A

```
A     A
|     |
B     B
|     |
D     F
|     |
I     K
```

- D, G => A

  D ≤ B ≤ A
  G ≤ C ≤ A

```
A     A
|     |
B     C
|     |
D     G
```

- D, E, I, J => B

  D ≤ B ≤ A
  E ≤ B ≤ A
  I ≤ D ≤ B ≤ A
  J ≤ D ≤ B ≤ A

```
A     A     A     A
|     |     |     |
B     B     B     B
|     |     |     |
D     E     D     D
            |     |
            I     J
```

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

  D ≤ B ≤ A
  E ≤ B ≤ A
  H ≤ C ≤ A
  I ≤ D ≤ B ≤ A
  J ≤ D ≤ B ≤ A

```
A     A     A     A     A
|     |     |     |     |
```

```
  B    B    C    B    B
  |    |    |    |    |
  D    E    H    D    D
                 |    |
                 I    J
```

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 fiecarei 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_T** 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_B, SELF_TYPE_B => SELF_TYPE_B

  $\text{SELF\_TYPE}_B \leq B$

  $\text{SELF\_TYPE}_B \leq B$

  ```
    B          B
    |          |
   ST_B       ST_B
  ```

- SELF_TYPE_J, B => B

  $\text{SELF\_TYPE}_J \leq J \leq D \leq B \leq A$

  $\text{SELF\_TYPE}_B \leq B \leq A$

  ```
    A      A
    |      |
    B      B
    |
    D
    |
    J
    |
   ST_J
  ```

- SELF_TYPE_K, SELF_TYPE_G => A

  $\text{SELF\_TYPE}_K \leq K \leq F \leq B \leq A$

  $\text{SELF\_TYPE}_G \leq G \leq C \leq A$

```
   A       A
   |       |
   B       C
   |       |
   F       G
   |       |
   K     ST_G
   |
 ST_K
```

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:

Foo ≤ Bar ≤ Bazz ≤ Object

Expresia 1

Prima ramura a expresiei *case*, **y : Int => y;** are tipul **Int**, deoarece corpul este doar identificatorul **y** care are tipul **Int**, conform tiparii **y : Int**.
In mod similar, a doua ramura a expresiei *case*, **z : Bool => z;** are tipul **Bool**.

Cum tipul expresiei *case* este *least upper bound* al tipurilor ramurilor, si cum avem relatiile *Int ≤ Object* si *Bool ≤ 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 *Bar ≤ Bazz ≤ Object* si *Foo ≤ Bar ≤ Bazz ≤ 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 aceeasi 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 {
    x_Main : Int <- 5;

    foo(z_foo : Int) : Int {
        x_Main + z_foo
    };

    bar(y_bar : Int) : Int {{
        let x_let1_bar : Int <- 1 in
            let z_let2_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 $\mathbf{y_{bar}}$ cu valoarea **3**. Variabilele introduse prin cele doua instructiuni *let* pot fi ignorate intrucat apelul **foo(**$\mathbf{y_{bar}}$**)** nu refera vreun identificator **x** si **z**. Cum $\mathbf{y_{bar}}$ este **3**, metoda apelata este **foo(3)**. Ca mai devreme, $\mathbf{z_{foo}}$ 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 $\mathbf{z_{foo}}$ 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 vizibiliate 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

In aceasta metoda, constructia *case* evalueaza tipul intors de apelul **foo()**. Apelul este un *implicit dispatch*, adica obiectul pe care se apeleaza metoda este **self**, deci apelul este echivalent cu **self.foo()**. Cum **self** este obiectul **Bar** definit mai sus, tipul dinamic este **Bar**, deci se apeleaza metoda **foo** din clasa **Bar**.

Main.main → Bar.bar → Bar.foo

In metoda **foo** se afiseaza un prim mesaj: **"Bar.foo()\n"**. Obiectul pe care s-a apelat metoda are tipul dinamic **Bar**, asadar tipul de retur al functiei este $\mathbf{SELF\_TYPE_{Bar}}$.

Main.main → Bar.bar

Inapoi in metoda **bar**, constructia *case* evalueaza tipul apelului **foo** la **Bar** (deoarece $\mathbf{SELF\_TYPE_{Bar} \leq Bar}$), folosindu-se, astfel, a doua ramura. Se instantiaza prin instructiunea *new* un obiect cu tipul (static si dinamic) **Bazz**, asupra caruia se apeleaza metoda **foo**, cea din clasa **Bazz**.

Main.main → Bar.bar → Bazz.foo

Se afiseaza al doilea mesaj: **"Bazz.foo()\n"**. Se creaza apoi un nou obiect cu tipul dinamic (si static) **Bar**, si, prin *static dispatch*, se apeleaza metoda **foo** din clasa **Foo**.

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

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

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

Metoda afiseaza al patrulea mesaj: **Bar.foo()\n**. Executia se intoarce in metoda **foo** a clasei **Foo**, apoi in metoda **foo** a clasei **Bazz**, apoi in metoda **bar** din clasa **Bar**.

Main.main → Bar.bar

Ramura a doua a expresiei *case* s-a executat in intregime, deci intreaga expresie s-a incheiat, executia revenind in metoda **main**.

Main.main

Cum metoda **main** nu mai contine alte expresii, executia programului se incheie. In concluzie, output-ul obtinut 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*:

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

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

<u>Constructia *case* - Ramura 1</u>

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):

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

$$\frac{O(f) = {}_{T}\text{Foo}}{O, M, \text{Bar} \vdash f : {}_{T}\text{Foo}} \; [\text{Var}]$$

$$\frac{\begin{array}{l} O, M, \text{Bar} \vdash f : {}_{T_0}\text{Foo} \\ {}_{T_0}\text{Foo} \leq {}_{T}\text{Foo} \\ M({}_{T}\text{Foo}, \text{foo}) = ({}_{T'_1}\text{SELF\_TYPE}) \\ {}_{T_1}\text{Foo} = {}_{T_0}\text{Foo} \qquad \textit{(T'}_1 \textit{ = SELF\_TYPE)} \end{array}}{O, M, \text{Bar} \vdash f@{}_{T}\text{Foo.foo()} : {}_{T_1}\text{Foo}} \; [\text{StaticDispatch}]$$

<u>Constructia *case* - Ramura 2</u>

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):

$$\frac{{}_{T'}\text{Bazz} = {}_{T}\text{Bazz} \qquad \textit{(T} \neq \textit{SELF\_TYPE)}}{O, M, \text{Bar} \vdash \text{new } {}_{T}\text{Bazz} : {}_{T'}\text{Bazz}} \; [\text{New}]$$

$$\frac{\begin{array}{l} O, M, \text{Bar} \vdash \text{new Bazz} : {}_{T_0}\text{Bazz} \\ {}_{T'_0}\text{Bazz} = {}_{T_0}\text{Bazz} \qquad \textit{(T}_0 \neq \textit{SELF\_TYPE}_{Bar}\textit{)} \\ M({}_{T'_0}\text{Bazz}, \text{foo}) = ({}_{T'_1}\text{SELF\_TYPE}) \\ {}_{T_1}\text{Bazz} = {}_{T_0}\text{Bazz} \qquad \textit{(T'}_1 \textit{ = SELF\_TYPE)} \end{array}}{O, M, \text{Bar} \vdash \text{(new Bazz).foo()} : {}_{T_1}\text{Bazz}} \; [\text{Dispatch}]$$

<u>Constructia *case* - Ramura 3</u>

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

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):

$$\frac{}{\textbf{O, M, Bar} \vdash \textbf{self} : {}_{T}\textbf{SELF\_TYPE}_{\textbf{Bar}}}\ \textbf{[Self]}$$

$$\frac{\begin{array}{l} \textbf{O, M, Bar} \vdash \textbf{self} : {}_{T_0}\textbf{SELF\_TYPE}_{\textbf{Bar}} \\ {}_{T'_0}\textbf{Bar} = {}_{T_0}\textbf{SELF\_TYPE}_{\textbf{Bar}} \qquad (T_0 = SELF\_TYPE_{Bar}) \\ \textbf{M(}{}_{T'_0}\textbf{Bar, foo) = (}{}_{T'_1}\textbf{SELF\_TYPE)} \\ {}_{T_1}\textbf{SELF\_TYPE}_{\textbf{Bar}} = {}_{T_0}\textbf{SELF\_TYPE}_{\textbf{Bar}} \qquad (T'_1 = SELF\_TYPE) \end{array}}{\textbf{O, M, Bar} \vdash \textbf{foo()} : {}_{T_1}\textbf{SELF\_TYPE}_{\textbf{Bar}}}\ \textbf{[Dispatch]}$$

Constructia *case*

Cele trei tipuri ale ramurilor constructiei *case* sunt:

Foo
Bazz
$\textbf{SELF\_TYPE}_{\textbf{Bar}}$

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):

$$\frac{\begin{array}{l} \textbf{O, M, Bar} \vdash \textbf{foo()} : {}_{T_0}\textbf{SELF\_TYPE}_{\textbf{Bar}} \\ \textbf{O[}\ {}_{T_1}\textbf{Foo / f], M, Bar} \vdash \textbf{f@Foo.foo()} : {}_{T'_1}\textbf{Foo} \\ \textbf{O[}\ {}_{T_2}\textbf{Bar / b], M, Bar} \vdash \textbf{(new Bazz).foo()} : {}_{T'_2}\textbf{Bazz} \\ \textbf{O[}\ {}_{T_3}\textbf{Object / o], M, Bar} \vdash \textbf{foo()} : {}_{T'_3}\textbf{SELF\_TYPE}_{\textbf{Bar}} \\ {}_{T}\textbf{Foo = lub(}{}_{T'_1}\textbf{Foo, }{}_{T'_2}\textbf{Bazz, }{}_{T'_3}\textbf{SELF\_TYPE}_{\textbf{Bar}}\textbf{)} \end{array}}{\textbf{O, M, Bar} \vdash \textbf{case foo() of ... esac} : {}_{T}\textbf{Foo}}\ \textbf{[Case]}$$

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

Foo ≤ IO ≤ Object
Bazz ≤ Bar ≤ Foo ≤ IO ≤ Object
$SELF\_TYPE_{Bar}$ ≤ Bar ≤ Foo ≤ IO ≤ Object

Se observa ca cea mai retrictiva 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*: $\mathbf{T_{case} \leq T_{bar}}$.

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

- **Foo**
- **IO**
- **Object**