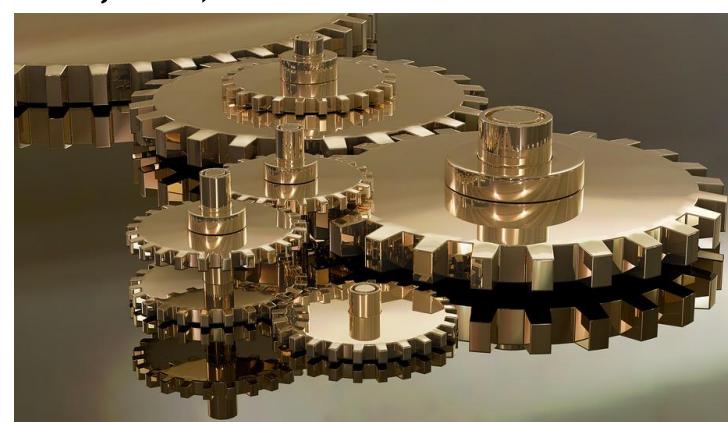


Limbaje formale și tehnici de compilare

Curs 1: introducere; compilatoare și
translatoare; fazele unui compilator

Domeniul LFTC

- ▶ Limbajul de programare (LP) este una dintre principalele unele a unui programator. La materiile de programare și algoritmi se învață folosirea LP, iar la LFTC se învață alcătuirea și funcționarea sa internă.
- ▶ LFTC se ocupă cu algoritmii și teoria pentru definirea și implementarea LP și, la modul general, pentru procesarea de text structurat: compilatoare, translatoare, interpretoare, verificare formală, validare de date, data mining, mașini virtuale
- ▶ LFTC face posibilă învățarea mult mai rapidă și mai profundă a limbajelor de programare
- ▶ Cunoașterea modului de lucru a unui compilator permite scrierea de programe care să maximizeze folosirea resurselor disponibile (CPU, memorie, ...)



Algoritmii de la LFTC fac posibile:

- ▶ Citirea fișierelor sau a altor date de intrare scrise în formate mult mai complexe decât cele accesibile prin folosirea expresiilor regulate și pentru care nu există biblioteci potrivite cu aplicația cerută
- ▶ Validarea datelor de intrare
- ▶ Extragerea informațiilor din datele de intrare
- ▶ Scrierea unor LP pentru un domeniu specific (DSL – Domain Specific Language) sau a unor LP generale
- ▶ Implementare suport de scripting în aplicații
- ▶ Verificarea formală a programelor
- ▶ Dezvoltarea unor aplicații de procesare a limbajului natural, de exemplu pentru traducere automată sau pentru agenți artificiali

Limbaje de programare (1)

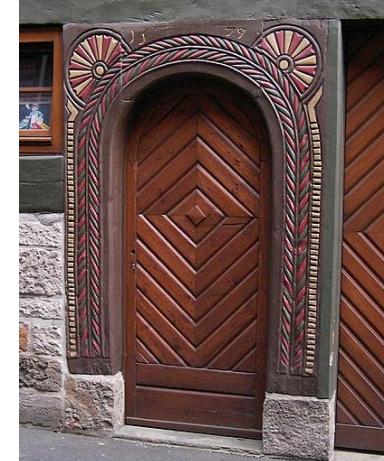
- ▶ Orice act de comunicare se bazează pe existența unui **limbaj** înțeles de ambele părți implicate în comunicare
- ▶ Limbajele pot fi de multe feluri: bazate pe cuvinte vorbite sau scrise, nonverbale, simbolice, ...
- ▶ Comunicarea în limbajele uzuale în anumite situații este ambiguă și atunci este necesar să se facă apel la context pentru a se stabili sensul ei
- ▶ Și pentru comunicarea cu calculatorul este necesar să existe un limbaj comun: **limbajul de programare**

*Ana vorbea cu Maria.
Ea era binedispusă.*

Ești un erou!

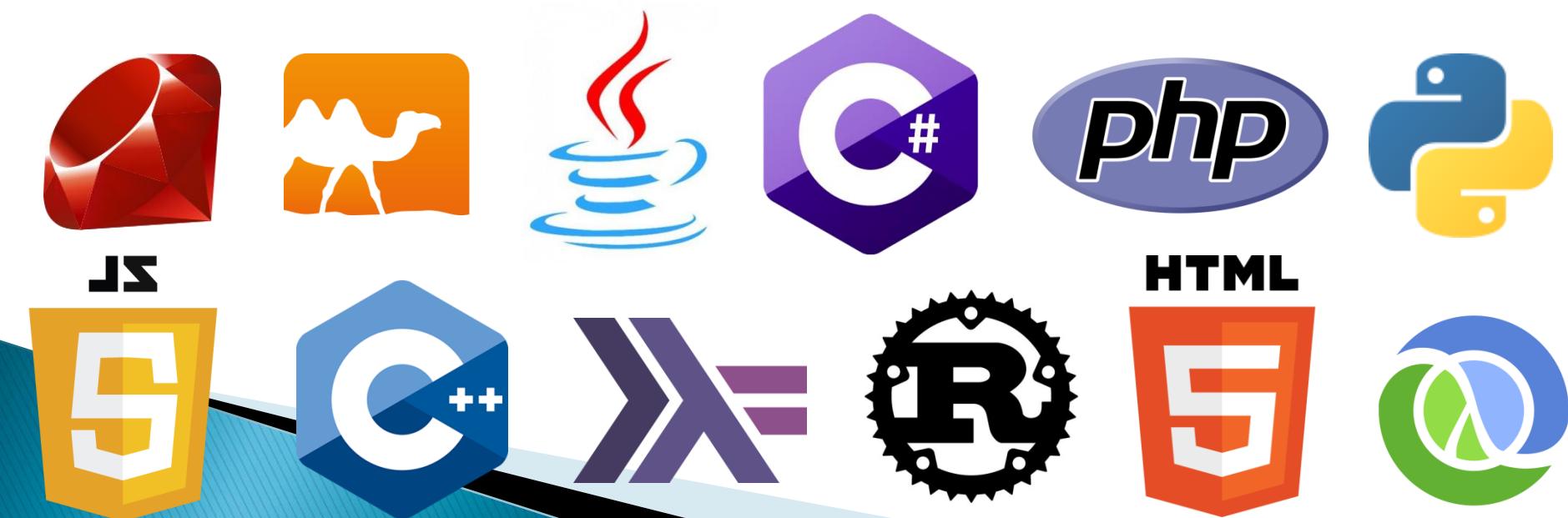


Tocul este solid.



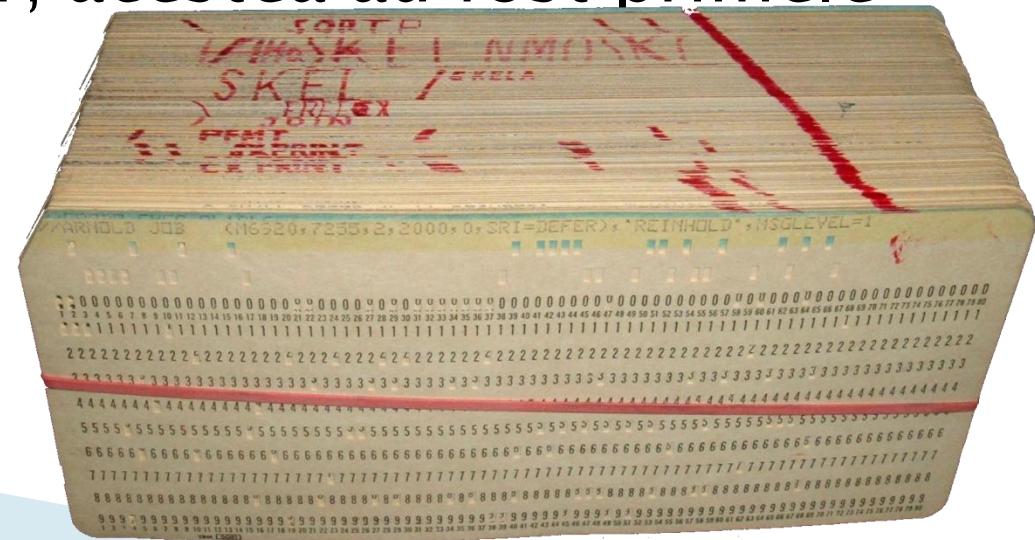
Limbaje de programare (2)

- ▶ Spre deosebire de limbajele obișnuite, un limbaj de programare (LP) trebuie să îndeplinească unele condiții:
 - **Să aibă o sintaxă complet definită** – alcătuirea și ordinea cuvintelor, folosirea semnelor de punctuație sau a operatorilor sunt specificate strict
 - **Să nu fie ambiguu** – orice comunicare prin LP trebuie să aibă strict o singură semnificație (semantică)



Compilatorul (1)

- ▶ Un CPU știe să interpreteze instrucțiuni în cod binar
- ▶ La început, deoarece nu existau limbaje de programare, primele calculatoare trebuiau programate folosind doar succesiuni de 0 și 1, ceea ce era foarte complicat
- ▶ Ulterior au început să apară programe al căror rol era să convertească un program dintr-o reprezentare mai accesibilă omului în codul binar necesar calculatoarelor; acestea au fost primele compilatoare



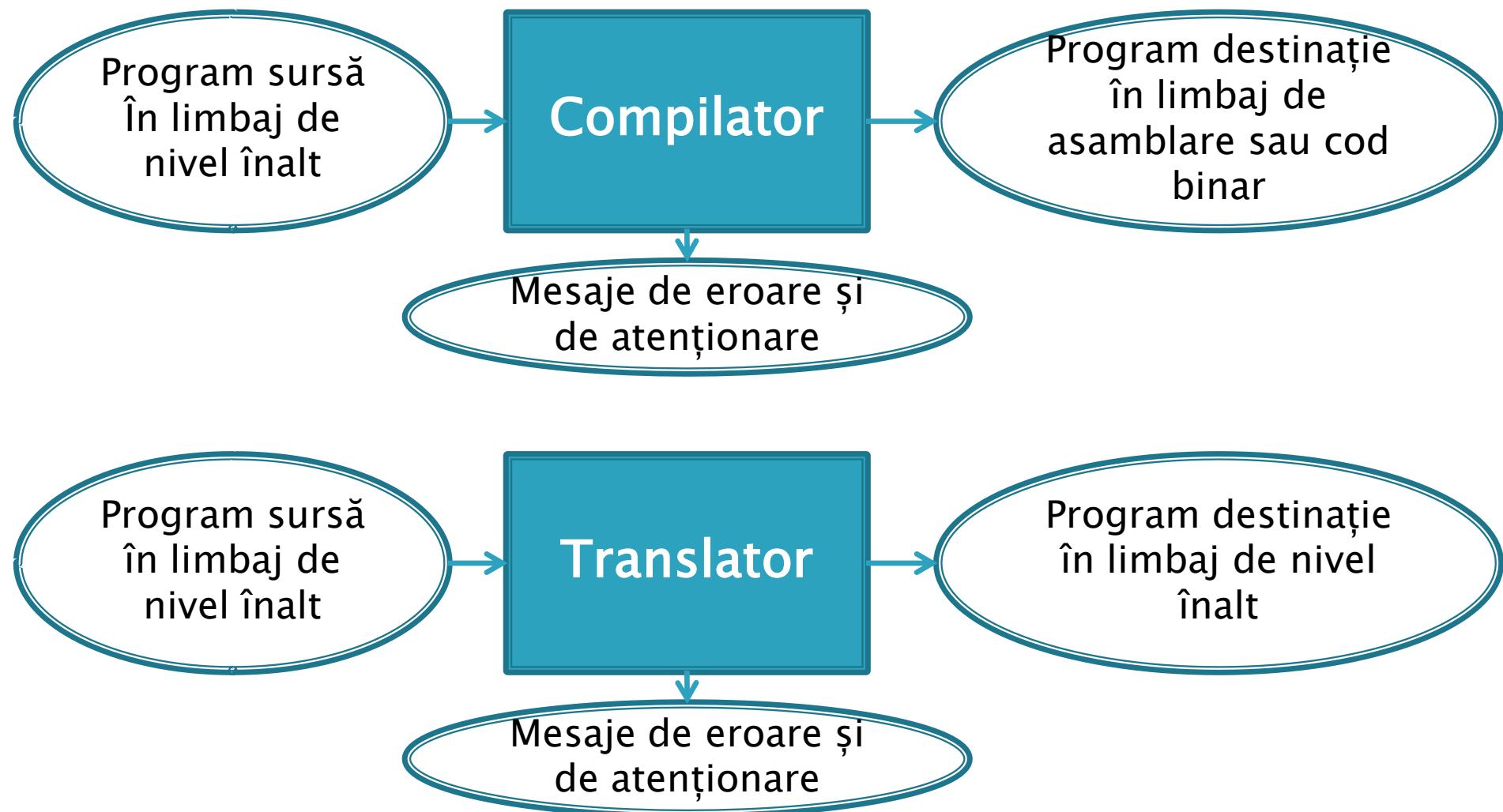
Compilatorul (2)

- ▶ Primul compilator a fost realizat în anul 1957, pentru limbajul de programare Fortran; dezvoltarea sa a durat 18 luni, deoarece nu exista o teorie a limbajelor formale și totodată a trebuit scris în limbaj de asamblare
- ▶ Dezvoltarea ulterioară a compilatoarelor a fost un proces iterativ, pe mai multe planuri:
 - Prin dezvoltarea părții teoretice s-a putut automatiza și optimiza o bună parte din procesul de dezvoltare
 - Având la dispoziție un limbaj de programare mai puternic, a fost mai ușor să se scrie programe mai complexe
- ▶ Au apărut compilatoare pentru limbaje de programare din ce în ce mai complexe: Pascal, C, Ada, C++, Java, C#

Compilatorul (3)

- ▶ Funcția de bază a unui compilator este să translateze un **program sursă**, scris într-un limbaj de nivel înalt (mai apropiat de reprezentarea umană), într-o reprezentare echivalentă (**program destinație**), accesibil unui CPU
- ▶ Ieșirea compilatorului poate fi un program în limbaj de asamblare (care ulterior va fi transformat în cod binar) sau se va genera direct cod binar
- ▶ Pe lângă această funcție de bază, un compilator mai trebuie să fie capabil să verifice dacă programul de compilat este corect (și dacă nu este, să emită mesaje de eroare sau atenționări)
- ▶ Unele compilatoare mai pot avea funcții de optimizare, pentru a genera variante mai eficiente ca timp de execuție sau ca necesar de memorie

Compilatorul (4)

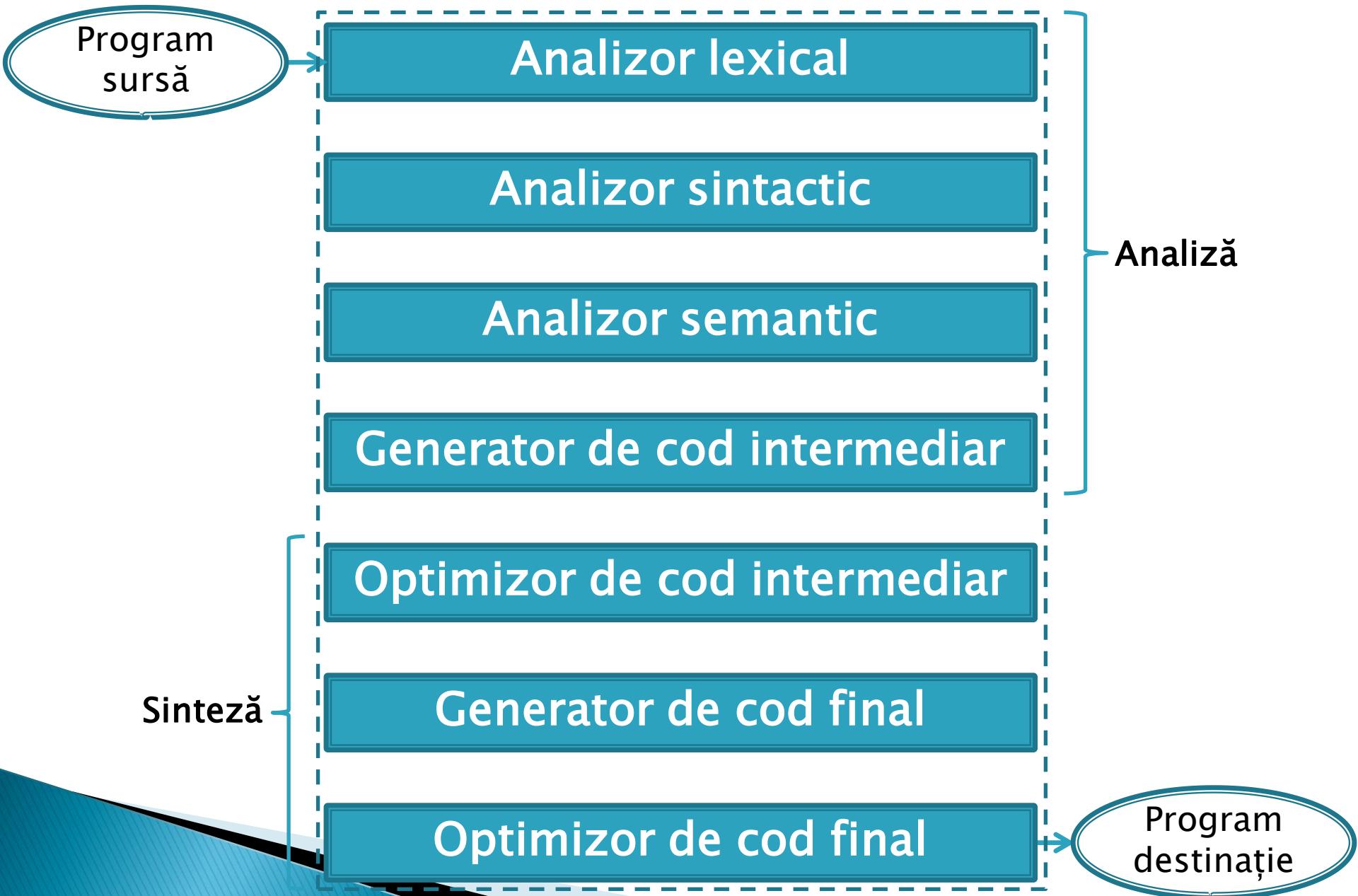


- ▶ **Translatorul** transformă programul sursă într-un program destinație, care este tot într-un limbaj de nivel înalt
- ▶ Primele compilatoare de C++ au fost de fapt translatoare, care translateau din C++ în C, iar apoi se foloseau compilatoare obișnuite de C

Modelul analiză–sinteză al compilării

- ▶ În principiu, un compilator are două mari funcții:
 - Analiza programului sursă (front-end)
 - Sinteză programului destinație (back-end)
- ▶ **Analiza programului sursă** – împarte programul sursă în componentele sale de bază și, pentru compilatoarele mai avansate, creează o reprezentare intermediară a programului sursă. Uneori prelucrarea codului intermediar se consideră ca fiind o funcție separată: **middle-end**
- ▶ **Sinteză programului destinație** – construiește programul destinație direct din informațiile de la faza de analiză sau pornind de la reprezentarea intermediară

Fazele unui compilator



Analizorul lexical

- ▶ Grupează caracterele de intrare în atomi lexicali
- ▶ Reține doar atomii lexicali utili
- ▶ Asociază atomilor lexicali informații conexe: linia din fișierul de intrare, caracterele constituente, valoarea numerică

```
// valoarea absolută
int abs(int v){
    if(a<0)return -a;
    return a;
}
```

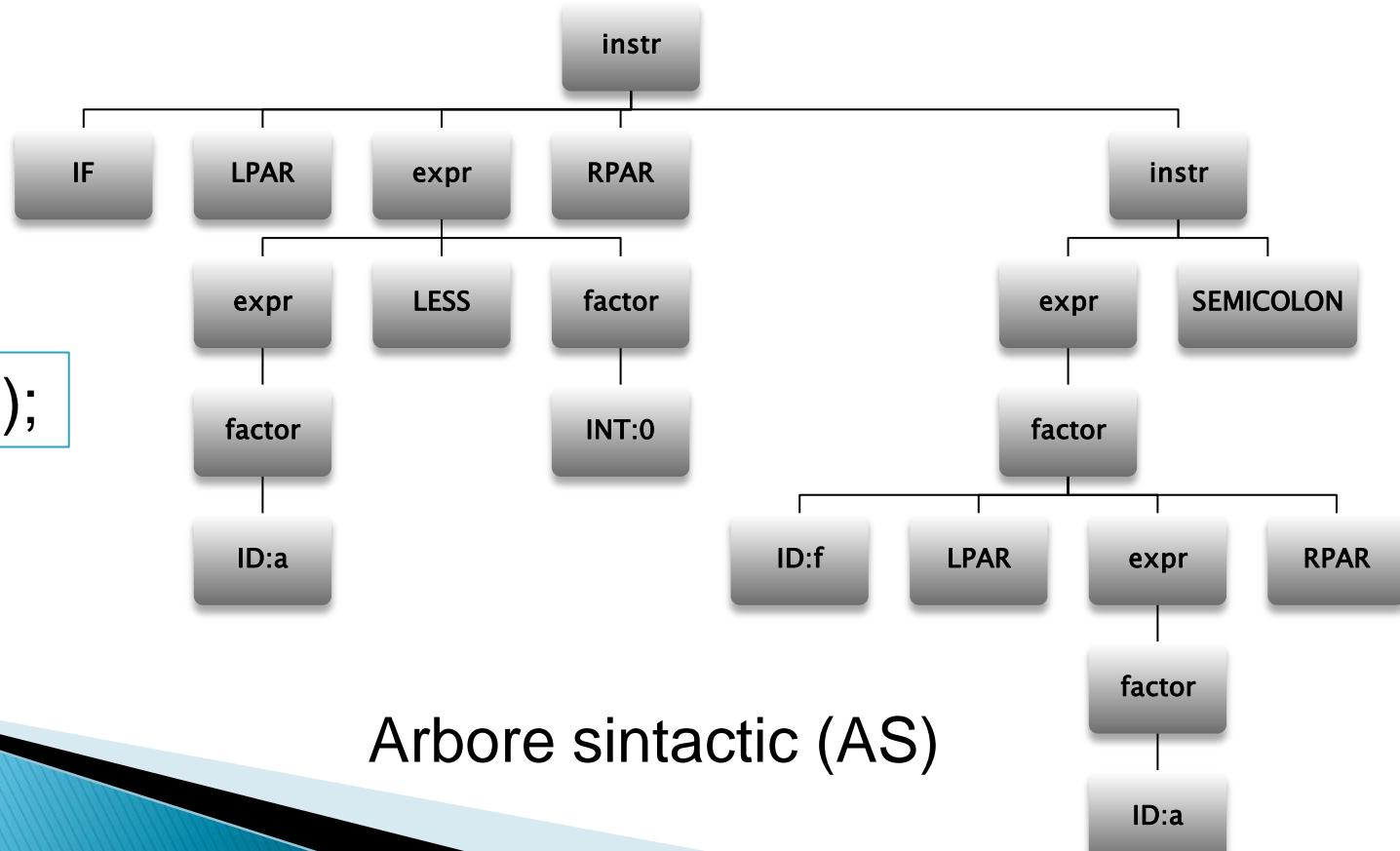
```
ID ::= [a-zA-Z_] [a-zA-Z0-9_]*  
LACC ::= {  
    ...
```

```
TYPE_INT ID:abs LPAR TYPE_INT ID:v RPAR LACC IF LPAR
ID:a LESS INT:0 RPAR RETURN SUB ID:a SEMICOLON
RETURN ID:A SEMICOLON RACC
```

Analizorul sintactic

- ▶ Grupează atomii lexicali în secvențe specifice limbajului: expresii, declarații, instrucțiuni, ...

```
instr ::= IF LPAR expr RPAR instr ( ELSE instr )? | expr SEMICOLON  
expr ::= expr LESS factor | factor  
factor ::= ID ( LPAR ( expr ( COMMA expr )* )? RPAR )? | INT
```



Analiza semantică

- ▶ Testează dacă programul respectă **regulile semantice**
- ▶ În general se compune din:
 - **Analiză de domeniu** – memorează simbolurile din declarații și verifică să nu existe redefiniri de simboluri
 - **Analiză de tip** – testează dacă într-o expresie componentele sunt folosite conform tipurilor lor și deduce tipul expresiei
- ▶ Se bazează pe **tabela de simboluri**

```
// analiză de domeniu
int a;
int a; // eroare: a este redefinit
```

```
// analiză de tip
a(5); // eroare: un întreg nu poate fi apelat ca o funcție
printf("%g",7.8+a); // double + int => double
```

Generatorul de cod intermediar

Codul intermediar permite unui compilator să utilizeze același procesor de cod și back-end-uri pentru oricâte front-end-uri



```
for(i=0;i<n;i++)  
    printf("%d",i*2);
```

```
i=0  
L1: T1=i<n  
    if not T1 goto L2  
    T2=i*2  
    printf("%d",T2)  
    i=i+1  
    goto L1  
L2:
```

Optimizorul de cod intermediu

Optimizarea codului implică aplicarea unor transformări care măresc eficiența codului din diverse puncte de vedere (timp de execuție, necesar de memorie, ...), dar îi păstrează neschimbată semantica

```
i=0  
L1: T1=s[i]  
     T2=T1!=0  
     if not T2 goto L3  
     T3=s[i]  
     T4=islower(T3)  
     if not T4 goto L2  
     T5= 'A'-'a'  
     T6=s[i]  
     T7=T6+T5  
     s[i]=T7  
  
L2: i=i+1  
     goto L1  
  
L3:
```

```
for(i=0;s[i]!=0;i++)  
    if(islower(s[i]))s[i]+='A'-'a';
```

```
i=0  
L1: T1=s[i]  
     if not T1 goto L3  
     T4=islower(T1)  
     if not T4 goto L2  
     T7=T1-32  
     s[i]=T7  
  
L2: i=i+1  
     goto L1  
  
L3:
```

Generatorul de cod final

Este generat codul programului destinație. Generarea se poate face în limbaj de asamblare sau direct în cod binar.

```
i=0  
L1: T1=s[i]  
    if not T1 goto L3  
    T4=islower(T1)  
    if not T4 goto L2  
    T7=T1-32  
    s[i]=T7  
L2: i=i+1  
    goto L1  
L3:
```

```
L1: mov edi,0  
    mov bl,[s+edi]  
    cmp bl,0  
    jz L3  
    movsx ecx,bl  
    push ecx  
    call islower  
    add esp,4  
    cmp eax,0  
    jz L2  
    sub bl,32  
    mov [s+edi],bl  
    inc edi  
    jmp L1  
L3:
```

Optimizorul de cod final

Aplică pe codul final optimizări care sunt specifice unei anumite platforme și, din acest motiv, nu se pot include în optimizările generale de cod intermedian

```
do{...}while(--i>0);
```

```
L1: ...
dec edi
cmp edi,0
jnz L1
```

```
L1: ...
dec edi
jnz L1
```

cmp edi,0 nu este necesară, deoarece dec edi setează deja EFLAGS cu valorile corecte

Unelte pentru dezvoltarea compilatoarelor

- ▶ **ANTLR** – generator de analizoare lexicale și sintactice pentru mai multe limbaje de programare
- ▶ **Bison** – generator de analizoare sintactice pentru C/C++/Java. Este considerat ca un succesor a lui Yacc.
- ▶ **Flex** – generator de analizoare lexicale pentru C/C++
- ▶ **LLVM** – o reprezentare intermediară de cod, cu unele dintre cele mai bune optimizări și multe back-end-uri. Se poate folosi din mai multe limbaje de programare.
- ▶ **re2c** – generator de analizoare lexicale pentru C/C++
- ▶ **Yacc** – generator de analizoare sintactice pentru C/C++
- ▶ Există asemenea unelte pentru multe limbaje de programare: Python, PHP, Ruby, OCaml, Haskell, ...
- ▶ Mai pot fi utile programe/biblioteci pentru: expresii regulate, generare cod mașină din asembler, ...

Bibliografie

- ▶ "Compilers – Principles, Techniques, & Tools", Aho A.V., Lam M.S., Sethi R., Ullman J.D., 2nd edition, 2007
- ▶ "Limbaje formale și translatoare", Ciocârlie H., 2017
- ▶ "Limbaje formale și tehnici de compilare – laboratoare pentru programele de studii Calculatoare și Informatică", Aciu R.
- ▶ "Engineering a Compiler", Cooper K.D., Linda T., 2nd edition, 2012

Limbaje formale și tehnici de compilare

Curs 2: definirea formală a unui limbaj; sintaxa;
gramatici; reprezentare; arborele sintactic

Formalizarea limbajelor de programare

- ▶ Formalizarea unui limbaj de programare (LP) se bazează pe definirea strictă atât a sintaxei cât și a semanticii sale
- ▶ Un limbaj de programare este definit prin tripletul
 $\langle \text{St}, \text{Sm}, F: \text{St} \rightarrow \text{Sm} \rangle$
 - ▶ **St – sintaxa** – forma în care este reprezentat conținutul comunicării
 - ▶ **Sm – semantica** – înțelesul pe care îl are comunicarea
 - ▶ **F** – o funcție care asociază o semantică unei sintactici

```
// C
if(a<0)r=-a;
else r=a;
```

```
# Python
if a<0:
    r=-a
else:
    r=a
```

Alfabetul unui limbaj

- ▶ **Alfabet (A)** – mulțimea finită și nevidă a simbolurilor utilizate într-un limbaj
- ▶ **Propoziții** – sirurile de simboluri care se pot forma cu elementele din A
- ▶ **A*** – mulțimea tuturor propozițiilor care se pot forma cu simbolurile din alfabetul A, incluzând propoziția vidă (ϵ)
- ▶ **A⁺** – A* din care s-a exclus propoziția vidă

$$A = \{0, 1\}$$

$$A^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\} \text{ (fără } \epsilon \text{ devine } A^+\text{)}$$

Limbaje formale

- ▶ Un **limbaj formal** peste alfabetul A este o submulțime a lui A^*
- ▶ Orice submulțime a lui A^* reprezintă un **limbaj formal**, iar A^* în ansamblu se numește **limbaj universal**
- ▶ **Gramatica unui limbaj** – totalitatea regulilor care definesc propozițiile valide din acel limbaj
- ▶ Sarcina esențială a verificării sintactice pentru un LP este de a decide dacă o anumită propoziție din A^* este conformă gramaticii aceluia LP

Limbă	Scriere	Transliterare
arabă	سلام	salām
ebraică	שלום	shalom
greacă	Ειρήνη	iríni
hindu	शांति	sānti
japoneză	平和	heiwa
rusă	Мир	mir

Gramatici formale

- ▶ O gramatică $G = \langle T, N, P, s \rangle$ are următoarele componente:
 - ▶ T – mulțimea tuturor terminalelor (simbolurile din alfabetul gramaticii). Vom nota terminalele cu litere mari sau între apostroafe/ghilimele.
 - ▶ N – mulțimea tuturor neterminalelor (simboluri care pot fi înlocuite prin alte siruri, folosind regulile gramaticii). Vom nota neterminalele cu litere mici.
 - ▶ P – mulțimea tuturor regulilor (producțiilor) gramaticii
 - ▶ s – neterminalul de start. De obicei regula de start este prima regulă din gramatică și se notează cu s , *start*, *program*, ...
- ▶ Fie V_G mulțimea tuturor terminalelor și a neterminalelor: $V_G = T \cup N$
- ▶ V_G^* este mulțimea sirurilor formate cu elementele din V
- ▶ Notăm cu litere grecești elemente din V_G^* : $\alpha, \beta, \gamma, \delta \in V_G^*$

// α

'while' '(' expr ')' instr

Reguli (producții)

- ▶ O regulă este de forma $\alpha \rightarrow \beta$ și are semnificația că prin intermediul ei sirul α va fi transformat în sirul β
- ▶ Partea stângă (α) a unei reguli se numește *cap*, iar partea dreaptă (β) se numește *coadă*
- ▶ De multe ori regulile se numesc **producții**, fiindcă prin intermediul lor se produc noi propoziții

```
'while' '(' expr ')' instr → 'while' '(' 'a' '>' expr ')' instr
```

// terminale: while, (,), a, >, 0

// neterminale: expr, instr

Transformări bazate pe gramatici (1)

- ▶ Fie $\alpha, \beta, \gamma, \delta \in V_G^*$
- ▶ Dacă $(\beta \rightarrow \delta) \in P$, atunci $\alpha\beta\gamma$ se poate transforma în $\alpha\delta\gamma$ ($\alpha\beta\gamma \rightarrow \alpha\delta\gamma$).
- ▶ Această transformare, prin care într-un sir se identifică partea stângă a unei reguli (capul ei) și ea se înlocuiește cu partea dreaptă (coada) a regulii respective, se numește **derivare**
- ▶ Operația opusă derivării, de identificare într-un sir a părții drepte a unei reguli și de înlocuire a ei cu partea stângă corespunzătoare, se numește **reducere** ($\alpha\delta\gamma \rightarrow \alpha\beta\gamma$)

Transformări bazate pe gramatici (2)

- ▶ Sirurile care se obțin prin derivări pornind de la simbolul de start al gramaticii, se numesc **forme propoziționale**
- ▶ O formă propozițională formată doar din terminale, se numește propoziție (ex: '**a' '+' '1'**)
- ▶ Totalitatea propozițiilor generate de o gramatică formează **limbajul** generat de ea, $L(G)$
- ▶ Dacă două gramatici, G_1 și G_2 generează același limbaj, $L(G_1)=L(G_2)$, gramaticile se numesc **echivalente**: $G_1 \sim G_2$

Exemplu de gramatică

- ▶ Fie gramatica $G = \langle T, N, P, S \rangle$, unde:
 - ▶ $T = \{0, 1\}$
 - ▶ $N = \{S\}$
 - ▶ $P = \{S \rightarrow 0S1, S \rightarrow 01\}$
- ▶ Aplicând prima regulă de $n-1$ ori și a doua regulă o dată, avem următorul sir de transformări:
 $S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 000S111 \rightarrow \dots \rightarrow 0^{n-1}S1^{n-1} \rightarrow 0^n1^n$
- ▶ Această propoziție se numește derivata de ordinul n a lui s conform gramaticii G . Pentru exemplul de mai sus, limbajul $L(G)$ poate fi notat $L(G) = \{0^n1^n \mid n \geq 1\}$

Tipuri de gramatici (1)

- ▶ După forma regulilor, Chomsky a împărțit gramaticile în 4 categorii
- ▶ **Gramatici de tipul 0** – sunt gramatici fără restricții, având forma generală $\alpha \rightarrow \beta$. Există totuși 3 cerințe pentru aceste gramatici:
 - a) Să se poată defini o procedură recursivă care să permită să se asocieze fiecărei propoziții câte un număr. Din acest motiv, limbajele generate de aceste gramatici se numesc **limbaje enumerabile recursiv**.
 - b) Să nu existe ambiguități
 - c) Fiecare parte stângă să conțină cel puțin un neterminal
- ▶ **Gramatici de tipul 1** – gramatici dependente de context (sensibile la context). Regulile sunt de forma $\alpha a \beta \rightarrow \alpha y \beta$, adică $a \rightarrow y$ doar dacă se găsește în contextul $\alpha \dots \beta$

Tipuri de gramatici (2)

- ▶ Gramatici de tipul 2 – gramatici independente de context (GIC). Producțiile sunt de forma $a \rightarrow \alpha$, adică a se înlocuiește cu α independent de contextul în care apare
- ▶ Gramatici de tipul 3 – gramatici regulate (GR). Producțiile sunt de forma $a \rightarrow Ab|A$ sau $a \rightarrow bA|A$

▶ Dacă notăm cu G_i multimea gramaticilor de tipul i și cu L_i clasa limbajelor generate de G_i , există următoarele incluziuni:

$$G_3 \subseteq G_2 \subseteq G_1 \subseteq G_0$$

$$L_3 \subset L_2 \subset L_1 \subset L_0$$

- ▶ În teoria limbajelor formale se poate demonstra că pentru orice limbaj dependent de context (implicit și pentru cele independente de context) se poate decide cu certitudine dacă o propoziție aparține sau nu limbajului respectiv
- ▶ La ora actuală, limbajele de programare sunt limbaje independente de context.

Notății pentru specificarea gramaticilor

- ▶ O notație foarte răspândită pentru specificarea unei gramatici este BNF (Backus Naur Form)
- ▶ Există multe variante ale acestei notății, de exemplu EBNF (Extended BNF), ABNF (Augmented BNF), ...
- ▶ Vom folosi o formă de EBNF care utilizează unii dintre operatorii specifici expresiilor regulate

Construcție	Semnificație
nume_regulă	un cuvânt cu litere mici denotă numele unei reguli (neterminal)
NUME_ATOM	un cuvânt cu litere mari denotă un atom lexical (terminal)
‘a’ sau “A”	caractere propriu-zise sau siruri, fără semnificație ca operatori
$\alpha \mid \beta$	alternativă – oricare variantă este validă
$\alpha \beta$	secvență – trebuie să fie îndeplinite ambele siruri, în ordine
α^*	repetiție optională – α poate să apară ori de câte ori, sau poate lipsi
α^+	repetiție obligatorie – α trebuie să apară cel puțin o dată
$\alpha?$	optional – α poate sau nu să apară
(α)	parantezele sunt folosite pentru a modifica ordinea operațiilor

Ordinea operatorilor

- ▶ La fel ca la operatorii din matematică, există o ordine de execuție și pentru operatorii din gramatică
- ▶ Prima oară se execută operatorii postfixați: * + ? (au precendența cea mai mare). Acești operatori acționează asupra elementului din fața lor
- ▶ Ulterior se execută secvența
- ▶ La final se execută alternativa
- ▶ Parantezele se folosesc pentru a schimba această ordine

r1 = 'a' 'b' 'c'? // ab, abc
r2 = 'a' 'b' 'c'* // ab, abc, abcc, abccc, ...
r3 = 'a' ('b' 'c')+ // abc, abcabc, ...
r4 = 'a' 'b' 'c' // ab, c
r5 = 'a' ('b' 'c') // ab, ac
r6 = 'a' 'b' 'c'* // ab, cccc
r7 = 'a' ('b' 'c')* // abbbccb

Folosirea unei gramatici

- ▶ Pentru a se determina dacă o propoziție este validă, trebuie să se găsească o secvență de operații prin care de la regula de start să se ajungă la acea propoziție (**derivare**).
- ▶ Determinarea validității unei propoziții se poate face și în sens invers, pornind de la propoziție și substituind pe rând în ea corpuri de reguli cu numele lor, până când rămâne doar numele regulii de start (**reducere**).

$s = 'a' \ s \ 'b'$ // s_1

$s = 'ba'$ // s_2

// echivalent: $s = 'a' \ s \ 'b' \mid 'ba'$

// notăm cu **s** pe oricare dintre cele două alternative

Propoziția '**aababb**' este validă?

Șir curent	Substituție
ϵ	$\epsilon \rightarrow s_1$
$'a' \ s \ 'b'$	$s \rightarrow s_1$
$'aa' \ s \ 'bb'$	$s \rightarrow s_2$
$'aababb'$	

Analiză lexicală și sintactică

- ▶ De multe ori este convenabil să se împartă verificarea sintaxei unui program în mai multe părți. De obicei se folosesc două etape: **analiza lexicală și analiza sintactică**
- ▶ De exemplu, în C, comentariile, caracterele TAB și NL (new line) pot apărea oriunde este valid un spațiu; ar fi complicat să se reprezinte toate posibilității în toate locurile posibile și atunci se preferă folosirea unei prime faze (analiza lexicală) care, printre altele, elimină aceste secvențe

// gramatică folosind o singură fază

comentariu = ...

spatiu = ' ' | '\t' | '\n'

skip = (comentariu | spatiu)*

instr = 'while' skip '(' skip expr skip ')' skip instr skip

// analiza lexicală: elimină skip

comentariu = ...

spatiu = ' ' | '\t' | '\n'

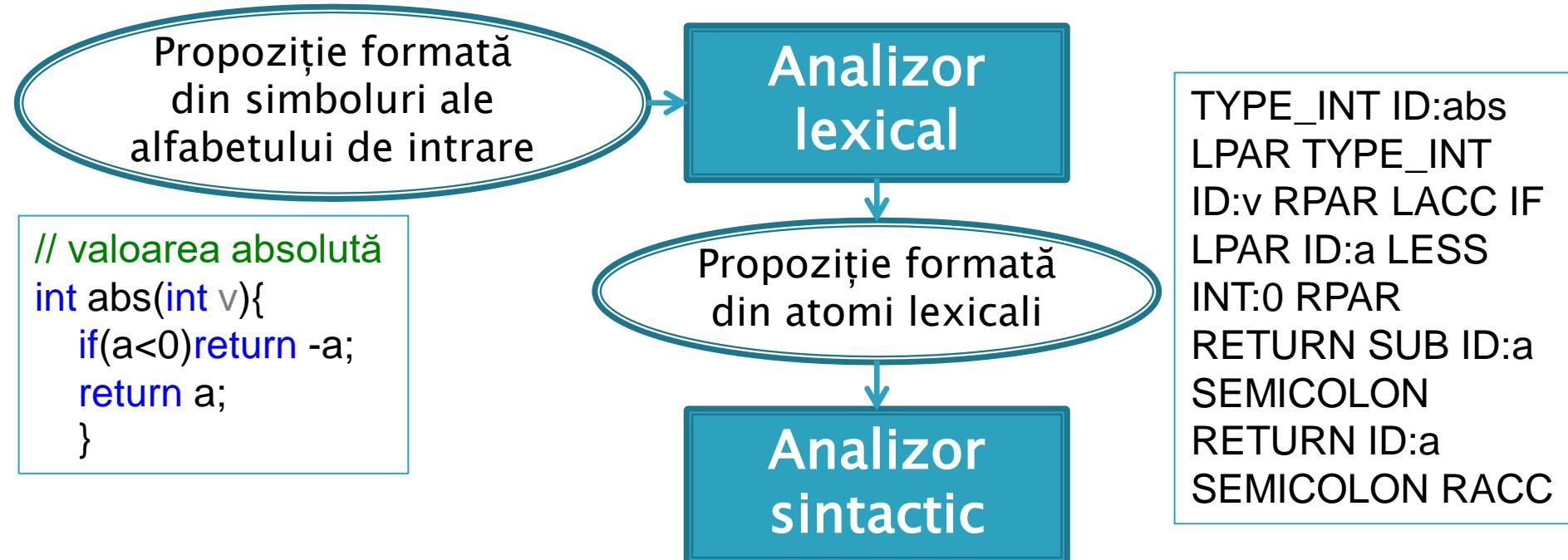
skip = (comentariu | spatiu)*

// analiza sintactică

instr = 'while' '(' expr ')' instr

Terminale și neterminale (1)

- Dacă verificarea sintaxei se face în două faze (analiza lexicală și cea sintactică), fluxul de date are loc conform diagramei:



- Se poate constata că de fapt analizorul lexical generează o propoziție compusă din simbolurile unui nou alfabet: alfabetul care cuprinde toți atomii lexicali (ex: ID, TYPE_INT, INT, ...). În acest alfabet nu se regăsesc simbolurile de la intrarea analizorului lexical, decât cel mult sub formă de atrbute ale atomilor.
- Deoarece atomii lexicali reprezintă simboluri de intrare în analizorul sintactic, ei sunt **terminale** pentru analiza sintactică

Terminale și neterminale (2)

- ▶ Pentru ușurință reprezentării, în faza de analiză sintactică se pot totuși folosi caractere în regulile sintactice, subînțelengând că de fapt acele caractere sunt atomi lexicali
- ▶ Astfel, se respectă faptul că analiza sintactică se face pe baza alfabetului atomilor lexicali, nu pe cel al caracterelor de la intrarea analizorului lexical

```
// reprezentarea implicită a atomilor lexicali  
instr = 'while' '(' expr ')' instr
```

```
// reprezentarea explicită a atomilor lexicali  
instr = WHILE LPAR expr RPAR instr
```

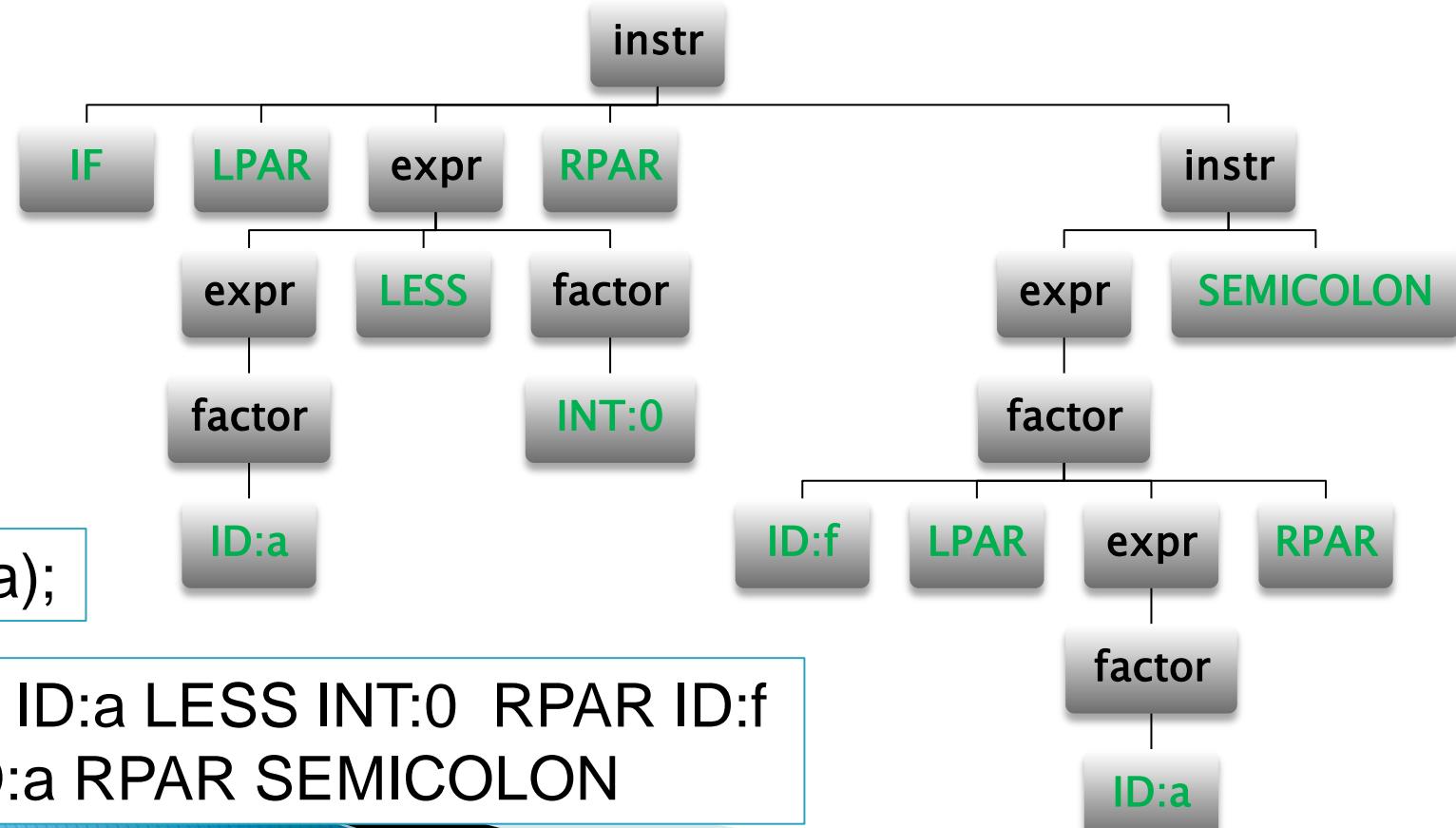
Arborele sintactic

- Este o reprezentare grafică a procesului de derivare
- Nodurile sunt regulile sintactice (neterminalele)
- Frunzele sunt simbolurile din limbajul sursă (terminalele)

instr ::= IF LPAR expr RPAR instr (ELSE instr)? | expr SEMICOLON

expr ::= expr LESS factor | factor

factor ::= ID (LPAR (expr (COMMA expr)*)? RPAR)? | INT

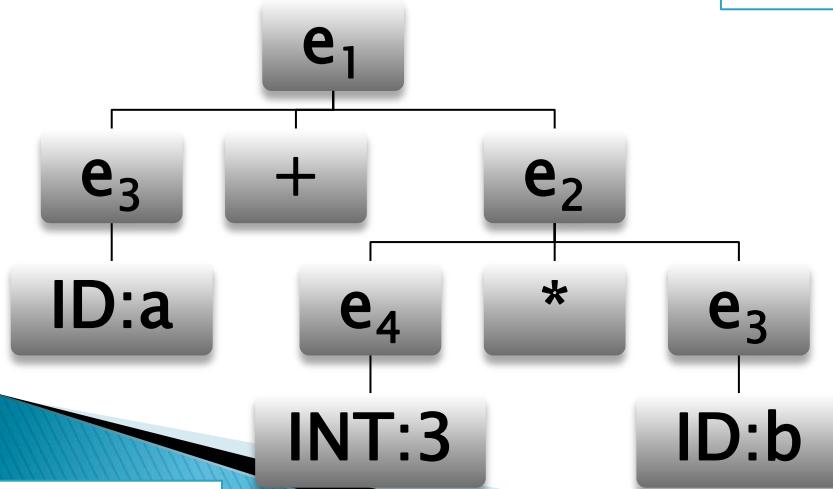


Gramatică ambiguă

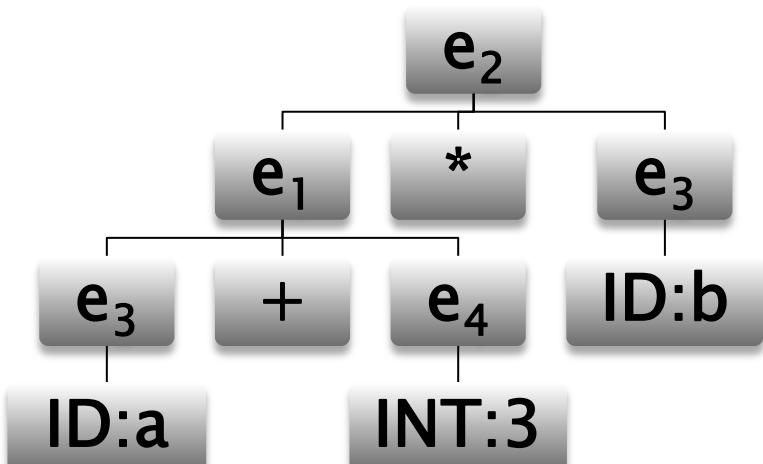
- ▶ O gramatică este ambiguă dacă permite ca pentru aceeași propoziție să existe mai multe derivări
- ▶ O gramatică ambiguă arată faptul că un limbaj nu a fost definit complet, deoarece o propoziție poate fi înțeleasă în mai multe feluri
- ▶ De obicei ambiguitatea poate fi rezolvată prin transformări în gramatică

$e = e' + e \quad // e_1$
 $e = e' * e \quad // e_2$
 $e = ID \quad // e_3$
 $e = INT \quad // e_4$

a+3*b



$a+(3*b)$



$(a+3)^*b$

Gramatică recursivă

- ▶ Fie o gramatică G și un neterminal oarecare, a . Se consideră următorul sir de transformări:
 $a \rightarrow \dots \rightarrow \alpha a \beta$, cu $\alpha, \beta \in V_G^*$, $V_G = N \cup T$
- ▶ Dacă $\alpha = \epsilon$ ($a \rightarrow \dots \rightarrow a \beta$), atunci a se numește **stâng recursiv** (corpul regulii începe cu ea însăși).
- ▶ Dacă $\beta = \epsilon$ ($a \rightarrow \dots \rightarrow \alpha a$), atunci a se numește **drept recursiv** (corpul regulii se termină cu ea însăși).
- ▶ Dacă într-o gramatică există cel puțin un neterminal stâng recursiv, atunci acea gramatică este **recursivă la stânga**. Similar, dacă există cel puțin un neterminal recursiv la dreapta, gramatica este **recursivă la dreapta**.

Limbaje formale și tehnici de compilare

**Curs 3: limbaje regulate; expresii regulate:
sintaxă, semantică, exemple**

Limbaje regulate

- ▶ Sunt limbaje ale căror reguli se pot implementa folosind un **automat cu stări finite**
- ▶ Corespund gramaticilor de tip 3 din clasificarea lui Chomsky (producțiile sunt de forma $a \rightarrow Ab|A$ sau $a \rightarrow bA|A$)
- ▶ Implementarea lor este simplă și viteza analizei sintactice este cea mai mare dintre toate clasele de limbaje
- ▶ Deficiența lor majoră este faptul că nu admit construcții recursive ("nu știu să numere") și atunci nu pot fi folosite pentru a testa de exemplu împerecheri de paranteze imbricate: $((a+3)^*2-1)/5$
- ▶ Sunt folosite pentru definirea expresiilor regulate și în analiza lexicală, deoarece regulile lexicale în general nu sunt recursive (un exemplu de recursivitate ar fi dacă limbajul C ar admite comentarii imbricate: /*.../* */..*/)

Definirea limbajelor regulate

- ▶ Multimea limbajelor regulate peste un alfabet A, se definește astfel:
- ▶ Limbajul vid {} (\emptyset) și cel care conține doar sirul vid { ϵ } sunt limbaje regulate
- ▶ Pentru $\forall a \in A$, {a} este un limbaj regulat
- ▶ Dacă A și B sunt limbaje regulate, atunci $A \cup B$ (reuniune), $A \bullet B$ (concatenare, scris simplificat AB) și A^* (repetiție optională sau *operatorul/închiderea/stea Kleene*) sunt limbaje regulate
- ▶ Nici un alt limbaj peste A nu mai este regulat

Expresii regulate (ER)

- ▶ Sunt o notație (un formalism) care implementează limbaje regulate. O ER este o regulă care **recunoaște** (match) o secvență de text
- ▶ Există mai multe dialecte de ER (ex: POSIX, Perl/PCRE, ...), fiecare cu diversi operatori și reguli
- ▶ Unele dialecte au operatori care permit să se scrie ER care depășesc clasa limbajelor regulate (ex: *referințe anterioare*: (a*)\1 → {ε, aa, aaaa, aaaaaa} = {aⁿaⁿ|n≥0})
- ▶ Sunt foarte răspândite în multe de domenii: validarea intrărilor, extragere de date, conversii de formate, căutare/înlocuire, analiza lexicală, etc. Practic aproape toate aplicațiile de prelucrare de text beneficiază de pe urma ER.
- ▶ Caracterele care au o semnificație specială (ex: . () * + ?) se numesc **metacaractere**

Caracterele din ER

- ▶ Literele, cifrele și toate caracterele simple (care nu sunt metacaractere) se recunosc pe ele însese (ex: ion va recunoaște toate secvențele de 3 litere *ion* din text). Dacă nu se specifică altfel, se face diferența între literele mari și mici (ex: Mare este diferit de mare).
- ▶ Spațiile sunt semnificative și sunt considerate caractere simple (ex: ion ana recunoaște sirul de 7 caractere dat și este diferit de ionana)
- ▶ . (punct) recunoaște orice caracter (ex: ra. → *ram*, *ras*, *rar*, *ra#*). În funcție de dialectul de ER și de unii modificatori, punctul poate sau nu să recunoască \n (new line).
- ▶ \ (backslash) pus în fața unui metacaracter îi anulează semnificația specială și acel caracter va fi recunoscut ca el însuși (ex: \. → \.)

Clase de caractere

- ▶ [...] – **clasă de caractere** – recunoaște orice caracter (unul singur) care face parte dintre caracterele specificate (ex: **[aeiou]** recunoaște o vocală).
- ▶ În interiorul [] se pot defini intervale de caractere, punând – (cratimă) între capetele intervalului (ex: **[a-zA-Z]** recunoaște orice literă). Dacă – apare la început sau la sfârșit, atunci se consideră caracter obișnuit (ex: **[+-]**)
- ▶ [^...] – **clasă de caractere negată** – Caracterul ^ (căciulă / hat) pe prima poziție într-o clasă de caractere are semnificația de “orice caracter cu **excepția celor din clasă**” (ex: **[^0-9]** va recunoaște un caracter care nu este cifră)
- ▶ În interiorul claselor de caractere aproape toate metacaracterele își pierd semnificația specială, cu excepția – \] (ex: **[0.*]** va recunoaște paranteze, punct sau steluță). Metacaracterele rămase se pot transforma în caractere simple folosind \ (ex: **\\\\"** recunoaște \ sau]). Unele dialecte de ER nu acceptă nici \ în interiorul []).

Operatori (1)

- ▶ **$e_1 e_2$ – secvență** – ambele ER, e_1 și e_2 , trebuie să fie îndeplinite în ordine (ex: = recunoaște o succesiune de două semne =)
- ▶ **e^* – repetiție optională** – e se poate repeta de oricâte ori, sau poate lipsi (ex: [a-z][0-9]* – o literă obligatorie urmată optional de oricâte cifre)
- ▶ **e^+ – repetiție obligatorie** – e trebuie să apară cel puțin o dată (ex: [0-9]+ – cel puțin o cifră)
- ▶ **$e^?$ – optional** – e poate sau nu să apară (ex: -?[0-9]+ – cratimă optională, urmată de cel puțin o cifră)
- ▶ **$e_1 | e_2$ – alternativă** – una dintre cele două ER trebuie să fie adevărată. Ele se testează în ordine și dacă o ER este adevărată iar ER se termină, următoarele ER nu mai sunt testate (ex: \?|! – testează dacă apare semnul întrebării sau al exclamării)

Operatori (2)

- ▶ **e{n}** – repetă **e** de **n** ori (ex: **[0-9]{4}** o cifră repetată exact de 4 ori)
- ▶ **e{n,}** – repetă **e** de minim **n** ori (ex: **[a-z]{2,}** cuvinte de minim 2 litere)
- ▶ **e{m,n}** – repetă **e** între **m** și **n** ori (ex: **[0-9]{1,4}** numere între 1 și 4 cifre)
- ▶ **^e** – recunoaște **e** doar dacă ea este la început de linie sau sir. **^** în sine nu consumă niciun caracter, ci doar ancorează ER în acea poziție.
Atenție: dacă **^** este în interiorul **[]**, el are semnificația prezentată anterior la clase de caractere. (ex: **^#[^\\n]*** recunoaște toate liniile care încep cu #, urmat de orice caracter până la \n)
- ▶ **e\$** – recunoaște **e** doar dacă ea este la sfârșit de linie sau sir. **\$** în sine nu consumă niciun caracter, ci doar ancorează ER în acea poziție (ex: **;\$** recunoaște toate ; care apar la sfârșit de linie)

Modificatori pentru ER

- ▶ Dacă o ER este pusă între /.../, la sfârșit pot fi specificate unele litere cu rol de modificatori. Aceștia vor influența aplicarea ER. Modificatori pot fi:
 - ▶ i (**insensitive**) – nu se face diferență între litere mari și mici
 - ▶ g (**global**) – caută toate aparițiile ER în text, nu doar prima apariție
 - ▶ s (**single line**) – punctul recunoaște și \n
 - ▶ m (**multi line**) – textul este considerat ca fiind format din mai multe linii și se pot aplica ^...\$ pentru fiecare linie. Altfel, ^...\$ ancorează la începutul și sfârșitul întregului text.
 - ▶ x (**extended**) – nu se ține cont de spații în ER. Astfel ER se poate scrie mai lizibil, iar dacă este nevoie de spații, ele se vor pune în []
- ▶ ex: /ana/ig – recunoaște toate aparițiile cuvântului *ana* în text, indiferent de literele cu care este scris

Capturi și referințe la ele

- ▶ În ER parantezele, pe lângă rolul de a schimba ordinea operațiilor, au și rolul de captura ER din interiorul lor
- ▶ Această captură se poate folosi ulterior cu \0..9 (*backslash urmat de o cifră*). Cifra indică la a câta paranteză deschisă se face referire, începând cu 1. \0 înseamnă toată ER.
- ▶ Captura este foarte utilă pentru extragerea informațiilor utile din datele de intrare sau pentru operații de genul search/replace.
- ▶ **Exemplu:** dacă avem un fișier CSV cu linii cu formatul *nume,email,data_nastere* putem folosi o ER de forma $\wedge([\wedge,]+),([\wedge,]+),([\wedge,]+)\$$ pentru a extrage numele în \1, emailul în \2 și data nașterii în \3.

Ordinea operațiilor

- ▶ Prima oară se execută operatorii postfixați: * + ? {}
- ▶ Ulterior se execută secvențele: e_1e_2
- ▶ În final se execută alternativele: $e_1|e_2$
- ▶ Pentru a se modifica ordinea operațiilor, se folosesc parantezele
- ▶ Se va ține cont de faptul că dacă o ER a fost îndeplinită complet pe o anumită ramură, nu se vor mai testa și celelalte ramuri, deși s-ar putea ca și ele să fie adevărate. Din acest motiv, alternativele cele mai lungi se vor pune primele. În caz contrar, ER se va îndeplini prima oară pe o ramură mai scurtă și nu va consuma toate caracterele pe care ar trebui să le consume.
Ex: pentru sirul de intrare "*ionescu citește*", ER *ion|ionescu* va recunoaște doar subșirul "*ion*", pe când ER *ionescu|ion* va recunoaște subșirul "*ionescu*"

Comportamentul *greedy*

- ▶ Implicit operatorii din ER încearcă să consume cât mai multe caractere (au comportament *greedy=iacom*)
- ▶ Dacă sunt mai multe caractere la fel și vrem să ne oprim la primul dintre ele, atunci va trebui să scriem ER în aşa fel încât caracterul de final să nu facă parte din repetiție
- ▶ **Exemplu:** se dă propoziția
"and", "or" și "not" sunt cuvinte în engleză și dorim să scriem o ER care să recunoască fiecare cuvânt dintre ghilimele
- ▶ **Greșit:** ".*" – va recunoaște tot subșirul de la primele până la ultimele ghilimele ("and", "or" și "not"). Deoarece se încearcă să se consume cât de mult posibil, iar punctul se poate substitui inclusiv cu ghilimele, se vor consuma și ghilimele intermediare.
- ▶ **Corect:** "[^"]*"" – după ghilimele inițiale se vor consuma doar caractere care nu sunt ghilimele, până când se vor întâlni următoarele ghilimele

Exemple

- ▶ sun|mon|tue|wed|thu|fri|sat
- ▶ 0?[1-9]|[12][0-9]|3[01]
- ▶ [+-]?[0-9]+(\.[0-9]+)?
- ▶ \/\/[^\n\r\0]*
- ▶ ^[\t]*([\^, \t]+)[\t]*,[\t]*([0-9]+)[\t]*\$

Exemplu în C++ (1)

- ▶ Considerăm că avem un fișier CSV, numit *persons.csv*, care conține linii de forma: *nume_persoana,zi/luna/an*
- ▶ Fișierul mai poate conține și linii care sunt goale sau au conținut în alt format – aceste linii trebuie ignore
- ▶ Numele poate conține litere mari, mici și diverse alte caractere gen cratimă sau apostrof
- ▶ La început și sfârșit de linie, precum și în jurul virgulei pot să fie spații sau taburi care trebuie ignore
- ▶ Programul va trebui să afișeze toate numele persoanelor care au ziua de naștere în luna curentă, precum și vîrstă pe care o împlinesc

Ion , 23/10/1990

; Ana s-a nascut la Brasov
Ana,4/3/1992

Maria ,9/10/2001

10/2019
Ion implinește 29 ani
Maria implinește 18 ani

Exemplu în C++ (2)

```
int main(){
    time_t tt=time(nullptr);
    struct tm *t=localtime(&tt);
    int crtMonth=t->tm_mon+1,crtYear=t->tm_year+1900;
    cout<<crtMonth<<'/'<<crtYear<<'\n';
    ifstream fis("persons.csv");
    string line;
    regex e("[ \t]*([^\t,\n]+)[ \t]*,[ \t]*[0-9]{1,2}\W([0-9]{1,2})\W([0-9]{1,4})[ \t]*$");
    while(getline(fis,line)){
        smatch captures;
        if(regex_match(line,captures,e)){
            int month=atoi(captures[2].str().c_str());
            int year=atoi(captures[3].str().c_str());
            if(month==crtMonth){
                cout<<captures[1]<<" implinește "<<(crtYear-year)<<" ani\n";
            }
        }
    }
    return 0;
}
```

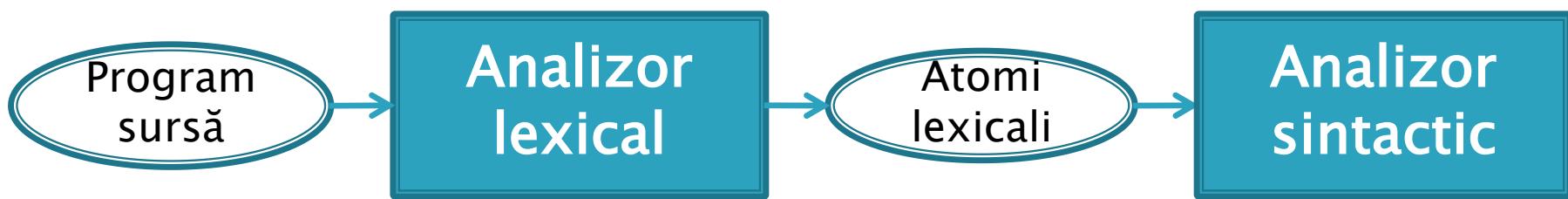
```
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iostream>
#include <regex>
using namespace std;
```

Limbaje formale și tehnici de compilare

Curs 4: analiza lexicală

Analiza lexicală

- ▶ Analiza lexicală este prima fază a translației (compilării)
- ▶ Ea se realizează de către **analizorul lexical** (ANLEX)
- ▶ ANLEX primește la intrare textul sursă la programului sub forma unui sir de caractere și va genera un sir de **atomii lexicali (tokens)** corespondent. Atomii lexicali produși vor fi folosiți în faza următoare, analiza sintactică



```
// valoarea absolută
int abs(int v){
    if(a<0) return -a;
    return a;
}
```

```
TYPE_INT ID:abs LPAR TYPE_INT
ID:v RPAR LACC IF LPAR ID:a LESS
INT:0 RPAR RETURN SUB ID:a
SEMICOLON RETURN ID:A
SEMICOLON RACC
```

Sarcinile analizei lexicale

- ▶ Obține sirul de atomi lexicali corespondent codului sursă
- ▶ Elimină construcțiile care nu mai sunt necesare în fazele ulterioare: comentarii, spații, linii noi, ...
- ▶ Reține locațiile (linie, coloană) din codul sursă corespunzătoare atomilor lexicali pentru a se genera ulterior, dacă este nevoie, mesaje de eroare localizate
- ▶ Realizează unele transformări din caractere în alte formate, cum ar fi: constantele numerice → tipuri numerice, secvențe escape → coduri corespunzătoare, ...
- ▶ Separă cuvintele cheie de identificatori

3.14159265358979323846 → **double**

"Ion\nAna" → **char []={**'I','o','n','\r','\n','A','n','a','\0'**}** // Windows

Avantajele analizei lexicale

- ▶ Permite simplificarea gramaticii sintactice și implicit a fazei de analiză sintactică
- ▶ De obicei ANLEX poate fi implementat cu limbaje simple (limbaje regulate), ceea ce mărește viteza de procesare
- ▶ Distribuie complexitatea verificării sintactice în mai multe module, care pot fi implementate independent

```
// gramatică folosind o singură fază
```

```
comentariu = ...
```

```
spatiu = ' ' | '\t' | '\n'
```

```
skip = ( comentariu | spatiu )*
```

```
instr = 'while' skip '(' skip expr skip ')' skip instr skip
```

```
// analiza lexicală: elimină skip
```

```
comentariu = ...
```

```
spatiu = ' ' | '\t' | '\n'
```

```
skip = ( comentariu | spatiu )*
```

```
// analiza sintactică
```

```
instr = 'while' '(' expr ')' instr
```

Atomi lexicali (tokens)

- ▶ Unități primare de informație, care sunt formate pe baza caracterelor din programul sursă
- ▶ Constituie alfabetul de intrare al analizorului sintactic
- ▶ Atomii reprezintă diverse constituente ale unui LP:
 - ▶ identificatori: tmp, i, x0, A, main, printf
 - ▶ cuvinte cheie (keywords): int, void, for, return
 - ▶ constante numerice (întregi și reale): 0, 108II, 0.5, 93f
 - ▶ constante caracter: '#', '\\', '\x5C'
 - ▶ constante sir de caractere: "salut\n", "#", ""
 - ▶ operatori: + -- && ^=
 - ▶ separatori/delimitatori: ; { } ,
- ▶ În funcție de locul lor, unii atomi pot reprezenta diverse constituente. De exemplu, parantezele pot fi atât separatori cât și operatori: int f(int x){...}, k*f(x)

Componentele atomilor lexicali

- ▶ **cod/tip** – de obicei constante numerice, care arată despre ce atom este vorba: ID, TYPE_INT, INT, LPAR, RPAR, ...
- ▶ **poziție în codul sursă** – toate informațiile care localizează originea atomului în sursă: linie, coloană, nume fișier
- ▶ **informații lexicale asociate** – caracterele din care sunt compuși identificatorii sau constantele caracter/șir de caractere, valorile constantelor numerice

```
typedef struct{  
    int linie, coloana;  
    char *numeFis;  
}Pozitie;
```

```
enum{ID, TYPE_INT, INT, LPAR, RPAR};  
typedef struct{  
    int cod;  
    Pozitie poz;  
    union{  
        double r;      // numere reale  
        long i;       // numere intregi  
        char c;        // caractere  
        char *text;    // identificatori, siruri de caractere  
    };  
}Atom;
```

Exemplu de extragere atomi lexicali

```
// aria cercului
int main(){
    double r;          // raza
    printf("raza: ");
    scanf("%g",&r);
    printf("aria=%g\n",3.14*r*r);
    return 0;
}
```

Linie	Atomi
2	TYPE_INT, ID:main, LPAR, RPAR, LACC
3	TYPE_DOUBLE, ID:r, SEMICOLON
4	ID:printf, LPAR, STR:"raza: ", RPAR, SEMICOLON
5	ID:scanf, LPAR, STR:"%g", COMMA, AND, ID:r, RPAR, SEMICOLON
6	ID:printf, LPAR, STR:"aria=%g\n", COMMA, DOUBLE:3.14, MUL, ID:r, MUL, ID:r, RPAR, SEMICOLON
7	RETURN, INT:0, SEMICOLON
8	RACC

Definirea atomilor lexicali

- ▶ Atomii lexicali se definesc în cadrul **gramaticii lexicale**. Această gramatică cuprinde doar definițiile atomilor lexicali, specificându-se și care atomi trebuie eliberați (ex: spații, comentarii, ...)
- ▶ Pentru marea majoritatea a LP, definirea atomilor lexicali se poate realiza folosind doar limbaje regulate. În această situație se va folosi o formă ușor modificată a expresiilor regulate, numită **definiții regulate**
- ▶ În cazul unor LP care au reguli mai complexe de definire a atomilor lexicali (ex: comentarii imbricate), limbajele regulate nu mai sunt suficiente și atunci este nevoie de un formalism mai puternic (ex: gramatici independente de context – GIC)

Definiții regulate

- ▶ Definițiile regulate (DR) se obțin din expresii regulate (ER), prin următoarele modificări:
 - ▶ Fiecarei ER i se va atribui un nume. Toate aceste nume formează o mulțime $\{d_1, d_2, \dots, d_n\}$
 - ▶ Spațiile nu mai sunt semnificative
 - ▶ În interiorul DR pot să apară numele altor DR. Pentru a se evita recursivitatea, în interiorul DR_i vor putea apărea doar numele din mulțimea $\{d_1, d_2, \dots, d_{i-1}\}$
 - ▶ Pentru a se face distincția între nume de DR și caractere obișnuite, caracterele vor fi puse în clase de caractere sau între apostroafe sau ghilimele. Din acest motiv și apostroafele și ghilimele devin metacaractere.

WHILE = 'while'

ESC = [\] [abfnrtv'?"\\0]

CHAR = ['] (ESC | [^'\\]) [']

STRING = [""] (ESC | [^"\\])* [""]

Extragerea atomilor lexicali

- ▶ Se poate face:
 - ▶ **Simultan cu citirea caracterelor de intrare** (codul sursă)
– necesarul de memorie este mai mic; se pot produce rezultate înainte de terminarea sursei (ex: când se procesează date din rețea)
 - ▶ **După citirea caracterelor într-un buffer** – se poate integra mai ușor cu alte module, deoarece este independentă de proveniența codului; în general este mai rapidă
- ▶ Extragerea se poate face folosind o funcție care extrage câte un atom la fiecare apel (`getNextToken`) sau se pot extrage toți atomii cu un singur apel (`getTokens`)

Algoritm de extragere atomi

- ▶ Pentru fiecare DR se implementează componentele acesteia, folosind construcții specifice de cod
- ▶ Dacă două sau mai multe DR încep cu același prefix, atunci se parsează prefixul comun într-un singur loc în cod. Apoi se testează următoarele caractere pentru a se diferenția ce DR specifică continuă după prefixul comun.
- ▶ Optional, secvențele de if-uri care testează caractere simple se pot înlocui cu o instrucțiune **switch**. Dacă între testelete pentru caractere simple se află și teste mai complexe (ex: pentru intervale de caractere), acestea se pot muta pe ramura de **default** de la **switch**.
- ▶ Se adaugă în cod alte aspecte necesare, gen preluarea informațiilor lexicale asociate și a locației, recunoașterea cuvintelor cheie

Implementarea componentelor DR

```
// e  
if(e){  
    next  
    ...  
}
```

```
// e1 e2  
if(e1){  
    next  
    if(e2){  
        next  
        ...  
    }  
}
```

```
// e1 | e2  
if(e1){  
    next  
    ...  
}  
else if(e2){  
    next  
    ...  
}
```

```
// e*  
while(e)next  
...
```

```
// e?  
if(e)next  
...
```

- ▶ **next** – se trece la următorul caracter
- ▶ ... – continuarea codului cu următoarele componente
- ▶ **e+ → e e***

Consumarea caracterelor

```
int getNextTk(){
    char ch=*pch;      // preluare caracter curent
    for(;;){
        if(isalpha(ch)||ch=='_'){
            ch=*++pch;          // next
            while(isalnum(ch)||ch=='_')ch=*++pch;
            addTk(ID);return ID;
        }
        if(ch=='='){
            ch=*++pch;
            if(ch=='='){
                pch++;           // nu mai este nevoie de setarea lui ch
                addTk(EQUAL);return EQUAL;
            }else{               // tranziția de tip else
                addTk(ASSIGN);return ASSIGN;
            }
        }
        if(ch==' '||ch=='\r'||ch=='\n'||ch=='\t')ch=*++pch;      // rămâne în bucla for
    }
}
```

ID = [a-zA-Z_] [a-zA-Z0-9_]*
ASSIGN = [=]
EQUAL = [=][=]

// cuvinte cheie
IF = 'if'
ELSE = 'else'

SKIP = [\r\n\t] // se elimină

Folosire switch

```
int getNextTk(){
    char ch=*pch;                                // preluare caracter curent
    for(;;){
        switch(ch){
            case ' ':case '\r':case '\n':case '\t':ch=*&gt;+&gt;pch;break;
            case ';':pch++;addTk(SEMICOLON);return SEMICOLON;      // adăugat: SEMICOLON = ;
            case ':':pch++;addTk(COLON);return COLON;                // adăugat: COLON = :
            case '(':pch++;addTk(LPAR);return LPAR;                  // adăugat: LPAR = \(
            case ')':pch++;addTk(RPAR);return RPAR;                  // adăugat: RPAR = \)
            case '=':
                ch=*&gt;+&gt;pch;
                if(ch=='='){
                    pch++;
                    addTk(EQUAL);return EQUAL;
                }else{
                    addTk(ASSIGN);return ASSIGN;
                }
            default:
                if(isalpha(ch)||ch=='_'){
                    for(ch=*&gt;+&gt;pch;isalnum(ch)||ch=='_';ch=*&gt;+&gt;pch){}
                    addTk(ID);return ID;
                }
        }
    }
}
```

Tratarea erorilor

- ▶ În cod se pot adăuga teste pentru detectarea situațiilor de eroare, pentru ca acestea să poată fi afișate
- ▶ Se vor raporta erori în toate cazurile în care caracterul curent nu permite avansarea către sfârșitul unei DR
- ▶ Funcția `lexErr` (lexical error) folosită în cod, realizează următoarele acțiuni:
 - ▶ folosește informația de localizare curentă pentru a afișa poziția actuală a erorii
 - ▶ afișează mesajul de eroare, inclusiv cu valoarea unor posibile variabile, aşa cum face `printf`
 - ▶ ieșe din program (nu se face revenire din eroare)

Cod care tratează erorile

```
switch(ch){  
    case '\\"':  
        ch=*&+&pch;  
        if(ch!="\\"){  
            ch=*&+&pch;  
            if(ch=='\\'){  
                pch++;  
                addTk(CHAR);return CHAR;  
            }else lexErr("lipseste ' la sfarsitul constantei caracter");  
        }else lexErr("constanta caracter vida");  
    default:  
        if(isdigit(ch)){  
            for(ch=*&+&pch;isdigit(ch);ch=*&+&pch){}
            if(ch=='.'){  
                ch=*&+&pch;  
                if(isdigit(ch)){  
                    for(ch=*&+&pch;isdigit(ch);ch=*&+&pch){}
                    addTk REAL;return REAL;  
                }else lexErr("dupa punctul zecimal trebuie sa fie minim un digit")
            }else{
                addTk INT;return INT;
            }
        }
        else lexErr("caracter invalid: %c (ASCII %d)",ch,ch);
}
```

```
CHAR = '\[^' \'  
INT = [0-9]+  
REAL = [0-9]+ \. [0-9]+
```

Preluarea informațiilor lexicale

- ▶ În cazul identificatorilor și a constantelor (numere, caractere, siruri de caractere) este necesar să se preia literele/valorile lor din sirul de intrare
- ▶ Se pot aplica două metode:
 1. Pe măsură ce se consumă caractere, acestea vor fi acumulate într-un buffer, care va fi prelucrat ulterior. Dacă bufferul este de dimensiune fixă, la adăugare de caractere trebuie efectuate verificări de depășire.
 2. Se setează un pointer/iterator/index la începutul atomului. La sfârșitul atomului se preiau toate caracterele dintre începutul atomului (setat anterior) și poziția curentă.

Identificarea cuvintelor cheie

- ▶ Deoarece cuvintele cheie (keywords) trebuie tratate diferit de identificatorii obișnuiți, ele vor fi extrase ca atomi cu coduri specifice
- ▶ Se pot folosi două metode:
 1. După ce s-a extras un identificator, se va face verificarea dacă acesta este un cuvânt cheie. Dacă este, se va adăuga cuvântul cheie respectiv, altfel se va adăuga ca identificator. Există metode de a se optimiza găsirea cuvintelor cheie, de exemplu știindu-se lungimea identificatorului găsit, prima literă din el, etc, în aşa fel încât să se reducă numărul de comparații necesare
 2. Fiecare cuvânt cheie se extrage direct prin parcurgerea DR care îl determină, la fel ca alți atomi. Este o modalitate rapidă, dar complică codul, datorită testelor de eliminare ambiguități (acest proces se poate automatiza cu unele specifice)

Setarea informațiilor de localizare (IL)

- ▶ Informațiile de localizare (nume fișier, linie, coloană) se pot prelua folosind variabile globale inițializate corespunzător
- ▶ Când se consumă caractere, se vor actualiza și IL (ex: când se întâlnește \n, se incrementează linia)
- ▶ Dacă LP permite ca un fișier să includă alt fișier (C/C++), se poate folosi o stivă de structuri IL. Când se include un fișier nou, IL curente se depun în stivă și se inițializează un nou set de IL pentru fișierul nou inclus. Când se revine în fișierul original, se scot de pe stivă IL salvate, astfel încât se refac poziția de la includere.
- ▶ Când se adaugă un nou atom, subrutina de adăgare a atomului (addTk) va prelua IL și le va seta în atomul nou adăugat

Cod care implementează și alte sarcini

```
int getNextTk(){
    char buf[ID_MAX], ch=*pch;
    int nBuf=0;                                // numărul de caractere din buf
    for(;;){
        switch(ch){
            case ' ':case '\r':case '\t':ch=*&gt;+&gt;pch;break;
            case '\n':                         // \n se tratează separat, pentru a se actualiza linia curentă
                ch=*&gt;+&gt;pch;
                linie++;
                break;
            default:
                if(isalpha(ch)||ch=='_'){
                    buf[nBuf++]=ch;
                    for(ch=*&gt;+&gt;pch;isalnum(ch)||ch=='_';ch=*&gt;+&gt;pch)CHECKADD
                    buf[nBuf]='\0';
                    if(!strcmp(buf, "if")){addTk(IF);return IF;}
                    if...                      // teste pentru toate cuvintele cheie
                    addTk(ID);
                    setTkText(buf);
                    return ID;
                }
        }
    }
}
```

Limbaje formale și tehnici de compilare

Curs 5: analizorul sintactic; conceperea
gramaticilor

Rolul analizorului sintactic (ANSIN)

- ▶ Pe baza gramaticii sintactice, grupează atomii lexicali în secvențe specifice limbajului: expresii, declarații, instrucțiuni, ...
- ▶ Alfabetul de intrare este constituit de atomii lexicali produși de ANLEX
- ▶ Secvențele recunoscute pot fi imbricate: o instrucțiune poate conține expresii sau instrucțiuni, care la rândul lor pot conține alte expresii, ...
- ▶ În caz de eroare, produce mesaje de eroare localizate
- ▶ Pentru a se mări productivitatea programatorului, la apariția unei erori se poate continua compilarea, astfel încât să se depisteze cât mai multe erori la o singură compilare

```
instr ::= IF LPAR expr RPAR instr ( ELSE instr )? | expr SEMICOLON  
expr ::= expr LESS factor | factor  
factor ::= ID ( LPAR ( expr ( COMMA expr )* )? RPAR )? | INT
```

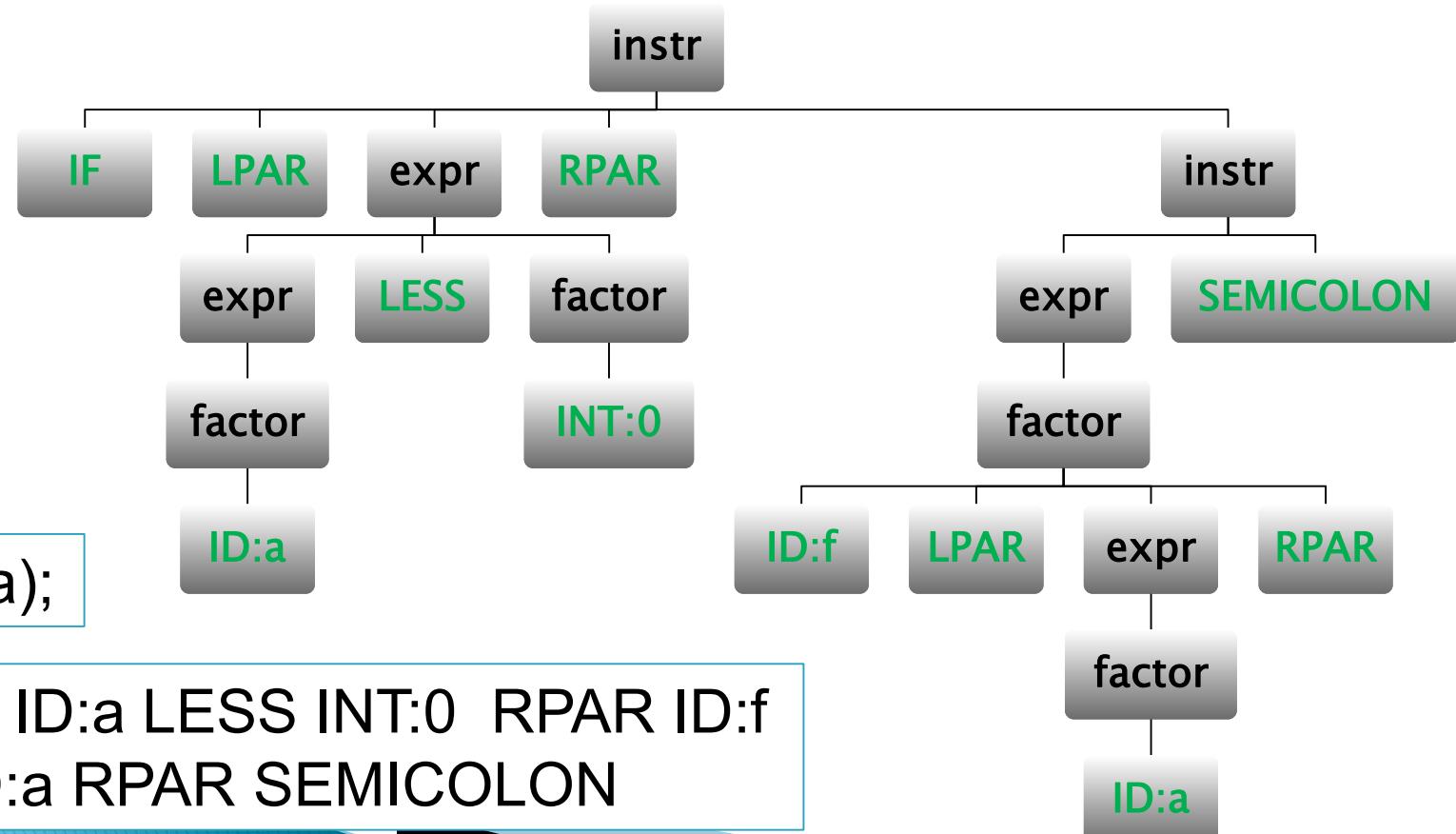
Gramaticile folosite pentru ANSIN

- ▶ În general construcțiile sintactice sunt recursive și atunci nu se mai pot folosi definiții regulate
- ▶ Marea majoritate a LP folosesc **gramatici independente de context (GIC)** – gramatici de tipul 2 după Chomsky
- ▶ GIC au producții de forma $a \rightarrow \alpha$, adică a se înlocuiește cu α independent de contextul în care apare
- ▶ Pentru implementarea GIC nu se mai pot folosi automate cu stări finite/diagrame de tranziții, ci sunt necesari alți algoritmi

Arborele sintactic

- Este o reprezentare grafică a procesului de derivare
- Nodurile sunt regulile sintactice (neterminalele)
- Frunzele sunt atomii lexicali (terminalele)

```
instr ::= IF LPAR expr RPAR instr ( ELSE instr )? | expr SEMICOLON  
expr ::= expr LESS factor | factor  
factor ::= ID ( LPAR ( expr ( COMMA expr )* )? RPAR )? | INT
```



Conceperea unei gramatici

- ▶ Gramatica trebuie să cuprindă reguli (neterminale) pentru toate construcțiile LP. Dacă o construcție este prea complexă, se poate împărți în componente și defini câte o regulă pentru fiecare componentă.
- ▶ Se poate folosi o abordare *top-down*, în care se pornește de la regula de start și ulterior detaliază componentele ei
- ▶ Regula de start definește un întreg program
- ▶ Se explicitează fiecare componentă a regulii de start și, în mod recursiv, fiecare dintre componentele componentei explicitate
- ▶ La implementarea unor construcții de genul expresiilor matematice, trebuie ținut cont de precedența și asociativitatea operatorilor, pentru a nu rezulta gramatici ambigue

Precedență și asociativitate

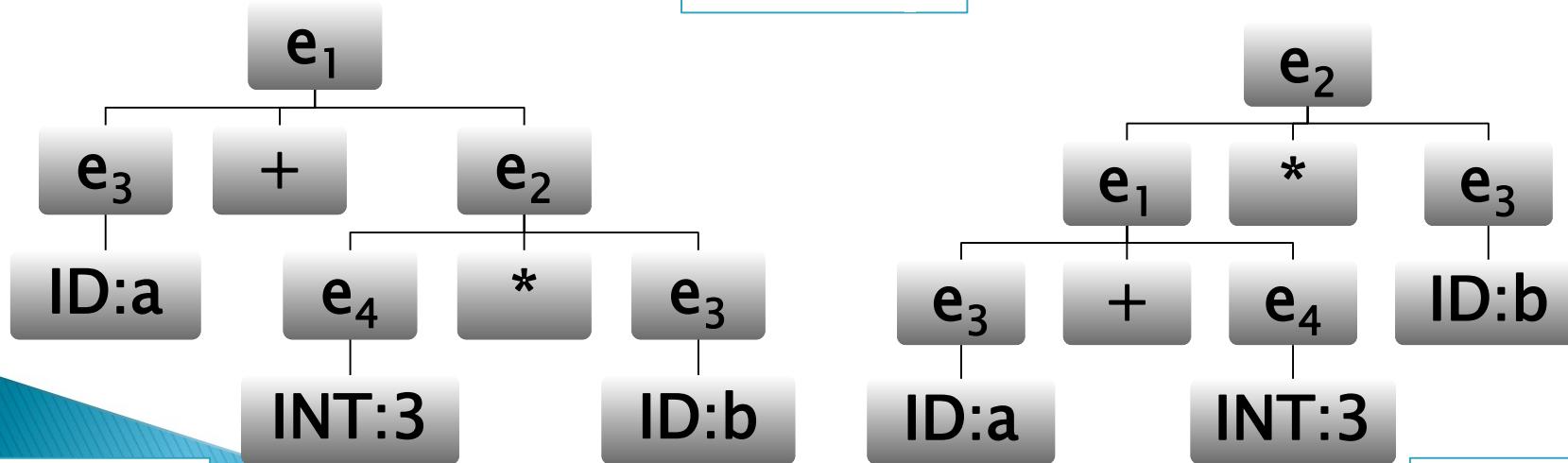
- ▶ **Precedență** – ordinea de evaluare a operatorilor. Operatorii cu precedență mai mare se vor evalua primii.
- ▶ Exemplu: în expresia $a+b/2$, împărțirea se va efectua prima (are precedență mai mare decât adunarea), deși în sirul de atomi lexicali adunarea este prima operație
- ▶ **Asociativitatea** – ordinea de evaluare a operatorilor cu aceeași precedență. Asociativitatea poate fi:
 - ▶ **non-asociativă** – nu se permite înlănțuirea operatorilor (exemplu: $a+=b+=1$ este invalid în Python)
 - ▶ **stângă** – operatorii se evaluatează de la stânga la dreapta
 - ▶ **dreaptă** – operatorii se evaluatează de la dreapta la stânga
- ▶ Exemplu: expresia $18/6/3$, în funcție de asociativitate, se va evalua astfel:
 - ▶ stângă: $(18/6)/3 \rightarrow 1$
 - ▶ dreaptă: $18/(6/3) \rightarrow 9$

Gramatică ambiguă

- ▶ O gramatică este ambiguă dacă permite ca pentru aceeași propoziție să existe mai multe derivări
- ▶ O gramatică ambiguă arată faptul că un limbaj nu a fost definit complet, deoarece o propoziție poate fi înțeleasă în mai multe feluri
- ▶ De obicei ambiguitatea poate fi rezolvată prin transformări în gramatică, ținând cont de precedența și asociativitatea operatorilor

$e = e' + e \quad // e_1$
 $e = e' * e \quad // e_2$
 $e = ID \quad // e_3$
 $e = INT \quad // e_4$

a+3*b



$a+(3*b)$

$(a+3)*b$

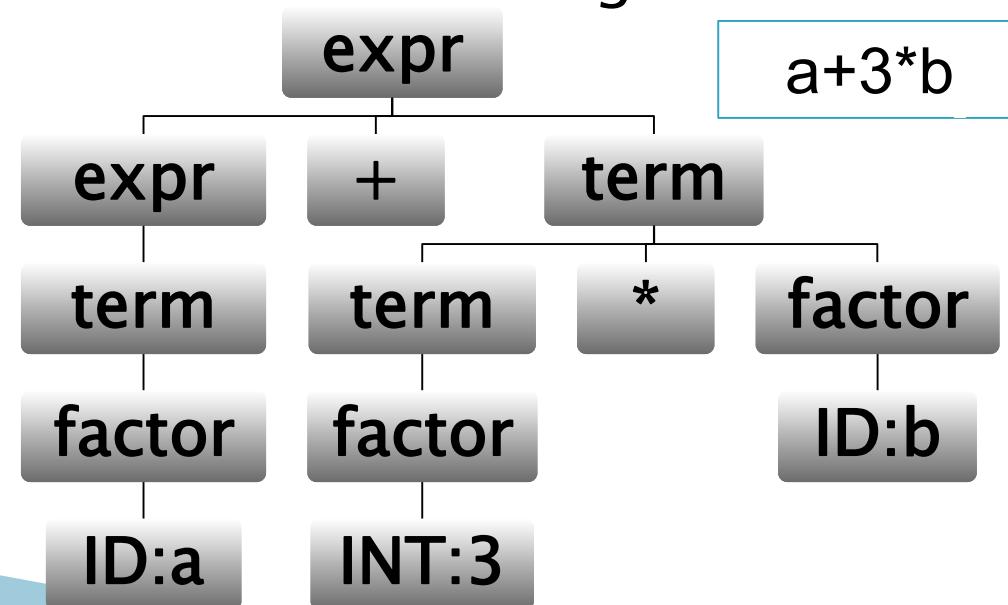
Implementarea precedenței

- ▶ Precedența se implementează folosind reguli separate pentru fiecare nivel de precedență
- ▶ Regulile cu precedență mai mică vor apela regulile cu precedență mai mare, astfel încât regulile cu precedență mai mare să se execute primele, și abia apoi să se revină în regulile cu precedență mai mică
- ▶ Regulile cu precedență mai mică vor trebui să aibă posibilitatea să se reducă la regulile cu precedență mai mare, astfel încât operatorii lor să nu fie obligatorii în expresie

$e = e '+' e \mid e '*' e \mid ID \mid INT$

\downarrow

$expr = expr '+' term \mid term$
 $term = term '*' factor \mid factor$
 $factor = ID \mid INT$



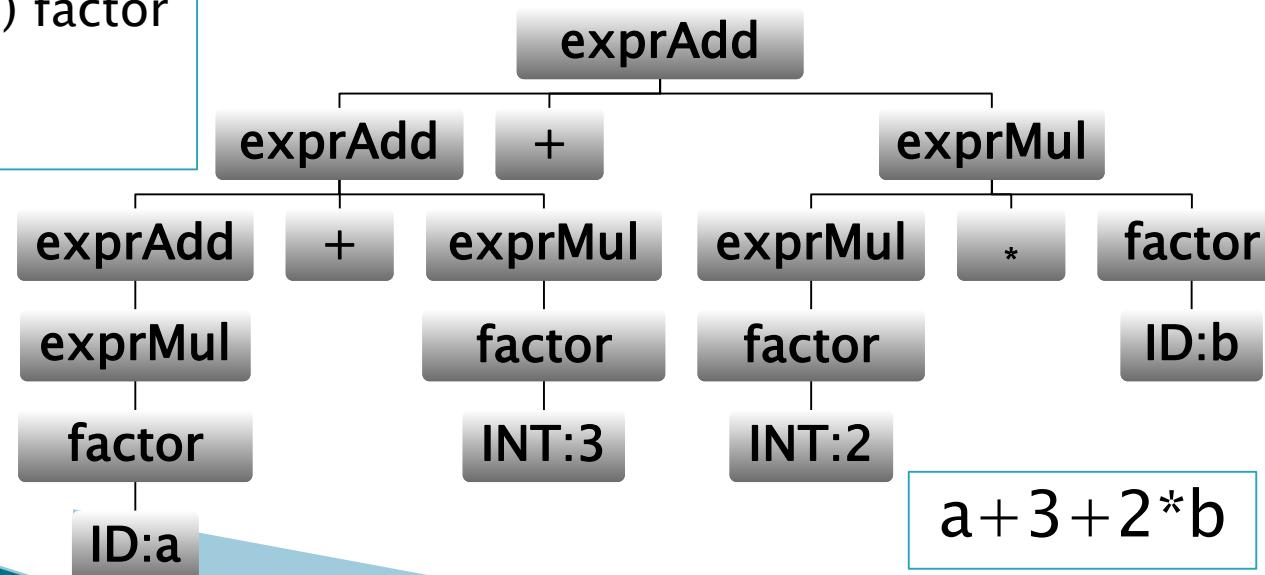
Implementarea non-asociativității

- ▶ Non-asociativitatea se poate implementa:
 - ▶ limitând recursivitatea în interiorul regulii, astfel încât operatorul respectiv să nu mai poată să apară în mod recursiv
Exemplu: $\text{exprAssign} = \text{ID} \stackrel{!}{=} \text{exprSrc}$
unde exprSrc nu poate conține $\stackrel{!}{=}$
 - ▶ considerând operația respectivă că nu este o expresie ci o instrucțiune (nu returnează o valoare)
Exemplu: $\text{expr} = \text{exprAdd} / \text{exprMul} / \text{postfix} / \text{factor}$
 $\text{instrAssign} = \text{ID} \stackrel{!}{=} \text{expr}$

Implementarea asociativității stângi

- ▶ Asociativitatea stângă se implementează folosind reguli recursive la stânga, astfel încât rădăcina arborelui va fi operatorul cel mai din dreapta și tot aşa recursiv, pe subarborele stâng
- ▶ Exemplu: pentru nivelul de precedență al adunării/scăderii și ținând cont că acești operatori sunt binari (arborele trebuie să aibă 2 ramuri), putem scrie
$$\text{exprAdd} = \text{exprAdd} (\text{+'} | \text{-'}) \text{exprMul} | \text{exprMul}$$
- ▶ exprMul este regula pentru precedența imediat mai mare

$\text{exprMul} = \text{exprMul} (\text{*'} | \text{/'}) \text{factor}$
| factor
factor = ID | INT



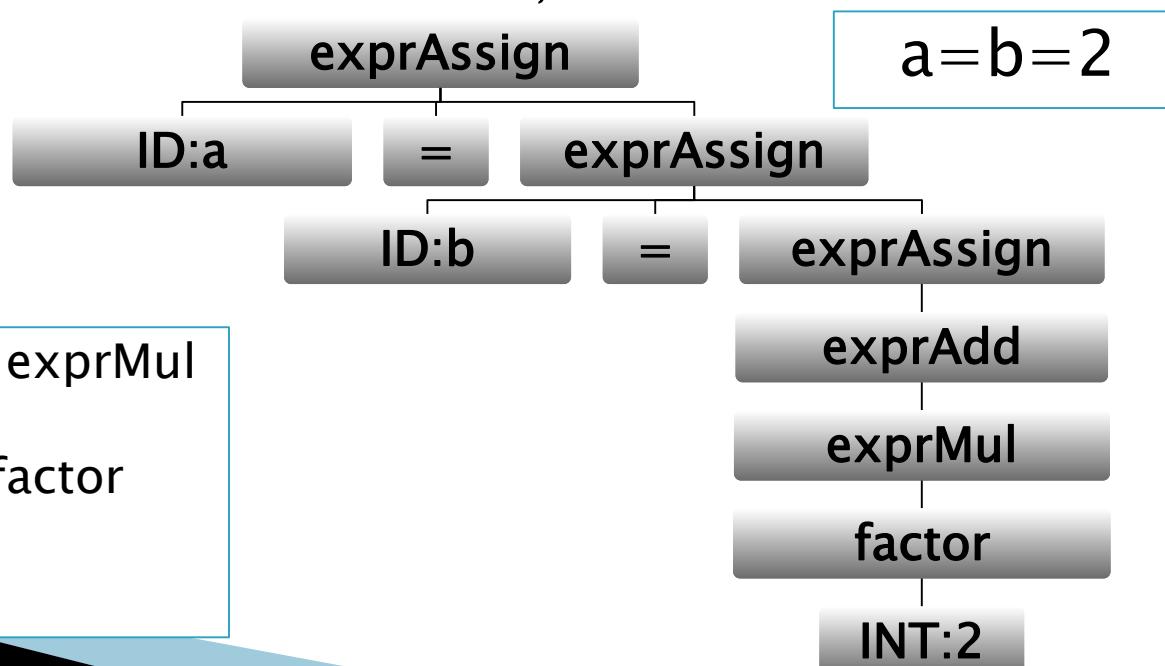
Implementarea asociativității drepte

- Asociativitatea dreaptă se implementează folosind reguli recursive la dreapta, astfel încât rădăcina arborelui va fi operatorul cel mai din stânga și tot așa recursiv, pe subarborele drept
- Exemplu: pentru nivelul de precedență al atribuirii în C, putem scrie

$\text{exprAssign} = \text{ID} \text{ '}' \text{exprAssign} \mid \text{exprAdd}$

- exprAdd este regula pentru precedența imediat mai mare

```
exprAdd=exprAdd ( '*' | '/' ) exprMul  
        | exprMul  
exprMul=exprMul ( '*' | '/' ) factor  
        | factor  
factor=ID | INT
```



Exemplu: conceperea unei gramatici

- ▶ Fie un subset al limbajului C, numit C1, cu următoarele cerințe, pentru care trebuie concepută o gramatică (pentru posibile aspecte nespecificate, se vor folosi regulile din C):
- ▶ Un program este alcătuit dintr-o secvență, posibil vidă, de definiții de funcții și de variabile
- ▶ Corpurile funcțiilor se definesc între accolade. În funcții se pot defini variabile locale, dar nu și alte funcții.
- ▶ Funcțiile nu pot fi *void*, ci returnează *int/float*
- ▶ O variabilă sau un argument de funcție poate fi de tipul *int* sau *float*. Mai multe variabile se pot separa cu virgulă.
- ▶ Există instrucțiunile *if/else*, *return* și expresii
- ▶ *if/else* poate conține instrucțiuni simple sau blocuri de instrucțiuni, date între accolade
- ▶ Expresiile pot conține identificatori, numere reale/întregi, apeluri de funcții și operatorii: = < + - (binar și unar) * /. Acești operatori au precedență și asociativitatea din C.
- ▶ În expresii, parantezele modifică ordinea operațiilor
- ▶ Pot exista doar comentarii linie (//...)

Definirea atomilor lexicali pentru C1

// operatori și separatori

LPAR = '('

RPAR = ')'

LACC = '{'

RACC = '}'

ASSIGN = '='

LESS = '<'

ADD = '+'

SUB = '-'

MUL = '*'

DIV = '/'

SEMICOLON = ';

COMMA = ','

FINISH = '\0'

SKIP = [\r\n\t] | '//' [^\r\n\0]*

ID = [a-zA-Z_] [a-zA-Z0-9_]*

INT = [0-9]+

FLOAT = [0-9]+ '.' [0-9]+

// cuvinte cheie

TYPE_INT = 'int'

TYPE_FLOAT = 'float'

IF = 'if'

ELSE = 'else'

RETURN = 'return'

// funcția factorial

int fact(int n){

if(n<2)

 return n;

 else

 return n*f(n-1);

}

Componentele gramaticii C1 (1)

- ▶ *Un program este alcătuit dintr-o secvență, posibil vidă, de definiții de funcții și de variabile*

program = (defFn | defVar)* FINISH

- ▶ *Corpurile funcțiilor se definesc între acolade. În funcții se pot defini variabile locale, dar nu și alte funcții.*
- ▶ *Funcțiile nu pot fi void, ci returnează int/float*

defFn = type ID LPAR args RPAR LACC body RACC

args = (arg (COMMA arg)*)?

arg = type ID

body = (defVar | instr)*

- ▶ *O variabilă sau un argument de funcție poate fi de tipul int sau float. Mai multe variabile se pot separa cu virgulă.*

type = TYPE_INT | TYPE_FLOAT

defVar = type ID (COMMA ID)* SEMICOLON

Componentele gramaticii C1 (2)

- ▶ Există instrucțiunile *if/else*, *return* și expresii
instr = if | return | expr SEMICOLON
return = RETURN expr SEMICOLON
- ▶ *if/else* poate conține instrucțiuni simple sau blocuri de instrucțiuni, date între acolade
if = IF LPAR expr RPAR instrCompusa (ELSE
 instrCompusa)?
instrCompusa = instr | LACC instr* RACC

Componentele gramaticii C1 (3)

- ▶ *Expresiile pot conține identificatori, numere reale/întregi, apeluri de funcții și operatorii: = < + - (binar și unar) * /. Acești operatori au precedență și asociativitatea din C.*
- ▶ *În expresii, parantezele modifică ordinea operațiilor*

expr = exprAssign

exprAssign = ID ASSIGN exprAssign | exprCmp

exprCmp = exprCmp LESS exprAdd | exprAdd

exprAdd = exprAdd (ADD | SUB) exprMul

| exprMul

exprMul = exprMul (MUL | DIV) exprUnary

| exprUnary

exprUnary = SUB exprUnary | exprFactor

exprFactor = ID (LPAR (expr (COMMA expr)*)? RPAR)?

| INT | REAL | LPAR expr RPAR

Gramatica C1

program = (defFn | defVar)* FINISH
defFn = type ID LPAR args RPAR LACC body RACC
args = (arg (COMMA arg)*)?
arg = type ID
body = (defVar | instr)*
type = TYPE_INT | TYPE_FLOAT
defVar = type ID (COMMA ID)* SEMICOLON
instr = if | return | expr SEMICOLON
return = RETURN expr SEMICOLON
if = IF LPAR expr RPAR instrCompusa (ELSE instrCompusa)?
instrCompusa = instr | LACC instr* RACC
expr = exprAssign
exprAssign = ID ASSIGN exprAssign | exprCmp
exprCmp = exprCmp LESS exprAdd | exprAdd
exprAdd = exprAdd (ADD | SUB) exprMul | exprMul
exprMul = exprMul (MUL | DIV) exprUnary | exprUnary
exprUnary = SUB exprUnary | exprFactor
exprFactor = ID (LPAR (expr (COMMA expr)*)? RPAR)?
| INT | REAL | LPAR expr RPAR

Limbaje formale și tehnici de compilare

Curs 6: analizorul sintactic descendent recursiv;
eliminarea recursivității stângi; tratarea erorilor

Implementarea analizorului sintactic

- ▶ Există două metode generale de implementare a unui analizor sintactic (parser):
- ▶ **Analiza sintactică descendentă (top-down parsing)** – se pornește de la regula de start a gramaticii și se derivă regulile până când se ajunge la terminale. Arborele sintactic este construit de la rădăcină către frunze.
- ▶ **Analiza sintactică ascendentă (bottom-up parsing)** – se pornește de la terminale și acestea se reduc la reguli sintactice, până când se ajunge la regula de start. Arborele sintactic este construit de la frunze către rădăcină.

Integrarea ANLEX în ANSIN

- ▶ Analiza sintactică se poate desfășura după ce a avut loc analiza lexicală și s-a produs lista de atomi lexicali. Această metodă permite decuplarea ANLEX de ANSIN, este mai rapidă, deoarece atomii sunt parsați o singură dată, dar necesită memorie pentru păstrarea tuturor atomilor.
- ▶ Analiza sintactică se poate desfășura concomitent cu cea lexicală: de fiecare dată când ANSIN are nevoie de un atom, va apela ANLEX pentru a i-l furniza. Această metodă nu necesită memorie pentru păstrarea listei de atomi, dar necesită o metodă de revenire în sirul de intrare, ceea ce uneori (ex: date citite din rețea) rezultă în necesitatea implementării unui buffer. Dacă ANSIN revine dintr-o alternativă greșită, un atom lexical se va parsa de mai multe ori.

Analizorul sintactic descendental recursiv

- ▶ ASDR este o metodă de implementare de tip descendantă (top-down) a ANSIN
- ▶ ASDR se pretează foarte de bine la implementarea manuală, de către programator, a gramaticilor
- ▶ Codul rezultat are o structură foarte asemănătoare cu cea a gramaticii
- ▶ Dacă este nevoie de o viteză mai mare de analiză, în special în cazul gramaticilor cu multe alternative, există metode de optimizare a ASDR
- ▶ Modalitatea de implementare a ASDR prezentată în curs este adaptată gramaticilor **PEG** (Parsing Expression Grammar). Diferența între acestea și formalismul GIC (Gramatică Independentă de Context), este faptul că în PEG alternativa ($e_1 \mid e_2$) este ordonată, la fel ca în expreziile regulate: dacă e_1 este recunoscută (match), iar regula se îndeplinește, atunci nu se mai analizează și e_2 .

Funcția *consume*

- ▶ Funcția **consume** furnizează ANSIN următorul atom lexical de prelucrat
- ▶ **consume** primește ca parametru un cod de atom
- ▶ Dacă atomul curent are codul cerut, el se consumă (se trece la următorul atom), iar **consume** va returna *true*
- ▶ Dacă atomul curent are alt cod decât cel cerut, **consume** va returna *false*, rămânând la poziția curentă

```
#include <stdbool.h> // În C, pentru bool, true, false
```

```
Atom *pCrtAtom;      // iterator atom curent
bool consume(int cod){
    if(pCrtAtom->cod==cod){
        ++pCrtAtom;    // considerăm că atomii sunt păstrați într-un vector
        return true;
    }
    return false;
}
```

Algoritm implementare ASDR

- ▶ Se elimină recursivitatea stângă din gramatică
- ▶ Fiecare regulă se va implementa printr-o funcție separată
- ▶ O funcție nu are niciun parametru și returnează *true/false*, dacă ea a putut fi aplicată (match) sau nu cu succes pe atomii începând cu poziția curentă
- ▶ Dacă o funcție s-a aplicat cu succes, ea consumă toți atomii corespunzători corpului ei; dacă funcția nu s-a putut aplica, ea nu trebuie să consume niciun atom (*totul sau nimic*). Acest mecanism se poate implementa salvând la intrarea în funcție poziția curentă din sirul de atomi. De fiecare dată când funcția trebuie să returneze *false*, poziția salvată inițial va fi refăcută.

```
bool regula()
{
    Atom *pStartAtom=pCrtAtom;      // salvare poziție inițială
    ... return true; ...
    pCrtAtom=pStartAtom;           // refacere poziție inițială
    return false;
}
```

Implementarea corpului unei reguli

- ▶ Atomii lexicali (**terminalele**) se consumă folosind funcția `consume`
- ▶ Neterminalele se consumă apelând funcțiile care le implementează
- ▶ $e_1 e_2$ – se consumă cu *if*-uri imbricate, deci se ajunge la următoarea componentă doar dacă prima a fost consumată
- ▶ $e_1 | e_2$ – se consumă cu succesiuni de *if*-uri, fiecare *if* testând o variantă din alternativă. Dacă o variantă a fost îndeplinită, nu se mai trece la următoarea variantă. Pentru aceasta se poate folosi un *return* în fiecare *if* sau succesiuni de *if/else if/.../else*. Dacă este posibil ca într-o alternativă să se fi consumat atomi și alternativa nu se aplică, trebuie refăcută poziția inițială.
- ▶ e^* – se implementează printr-o buclă infinită, din care seiese atunci când nu se mai poate consuma e
- ▶ e^+ – se aplică identitatea $e^+ = e \ e^*$
- ▶ $e^?$ – se consumă fără a se ține cont de rezultatul consumării, astfel încât e poate să lipsească
- ▶ $e | \epsilon$ – este echivalentă cu $e^?$

Implementare e₁ e₂

- ▶ *if*-uri imbricate, astfel încât se ajunge la următoarea componentă doar dacă prima a fost consumată

```
instrWhile = WHILE LPAR expr RPAR instr
```

```
bool instrWhile(){  
    Atom *pStartAtom=pCrtAtom;  
    if(consume(WHILE)){  
        if(consume(LPAR)){  
            if(expr()){  
                if(consume(RPAR)){  
                    if(instr()){  
                        return true;  
                    }  
                }  
            }  
        }  
    }  
    pCrtAtom=pStartAtom;  
    return false;  
}
```

Implementare $e_1 \perp e_2$

- succesiuni de *if*-uri, fiecare *if* testând o variantă din alternativă. Fiecare alternativă trebuie să refacă poziția inițială, dacă e posibil să se fi consumat în ea atomi.

```
factor = ID | LPAR expr RPAR | NR
```

```
bool factor(){
    Atom *pStartAtom=pCrtAtom;
    if(consume(ID)){
        return true;
    }
    if(consume(LPAR)){
        if(expr()){
            if(consume(RPAR)){
                return true;
            }
        }
        pCrtAtom=pStartAtom;
    }
    if(consume(NR)){
        return true;
    }
    return false;
}
```

Implementare e*

- se implementează printr-o buclă infinită, din care se ieșe atunci când nu se mai poate consuma e

```
varDef = type ID ( COMMA ID )* SEMICOLON
```

```
bool varDef(){  
    Atom *pStartAtom=pCrtAtom;  
    if(type()){  
        if(consume(ID)){  
            for(;;){  
                if(consume(COMMA)){  
                    if(consume(ID)){}  
                    else{ /*eroare*/ }  
                }  
                else break;  
            }  
            if(consume(SEMICOLON)){  
                return true;  
            }  
        }  
    }  
    pCrtAtom=pStartAtom;  
    return false;  
}
```

```
// variantă pentru ( COMMA ID )*  
while(consume(COMMA)){  
    if(consume(ID)){}  
    else{ /*eroare*/ }  
}
```

Implementare e?

- ▶ se consumă fără a se ține cont de rezultatul consumării, astfel încât *e* poate să lipsească
- ▶ În fazele ulterioare ale compilatorului va fi necesar să se știe dacă s-a consumat *e*. De aceea, la consumarea lui *e* se folosește un *if*, care deocamdată nu are instrucțiuni atașate.

exprUnary = SUB? factor

```
bool exprUnary(){  
    Atom *pStartAtom=pCrtAtom;  
    if(consume(SUB)){  
        }  
    if(factor()){  
        return true;  
        }  
    pCrtAtom=pStartAtom;  
    return false;  
}
```

Eliminarea recursivității stângi (1)

- ▶ În implementarea analizorului sintactic folosind ASDR, recursivitatea stângă duce în cod la recursivitate infinită

expr = **expr** (ADD | SUB) termen | termen

```
bool expr(){
    Atom *pStartAtom=pCrtAtom;
    if(expr()){
        if(consume(ADD)||consume(SUB)){
            if(termen()){
                return true;
            }
        }
        pCrtAtom=pStartAtom;
    }
    if(termen()){
        return true;
    }
    pCrtAtom=pStartAtom;
    return false;
}
```

Formulă eliminare recursivitate stângă

- ▶ Pentru eliminarea recursivității stângi, se poate folosi următoarea formulă de transformare a unei reguli gramaticale, în care:
 - ▶ A – numele regulii inițiale
 - ▶ A' – numele unei reguli nou introdusă
 - ▶ $\alpha_1, \alpha_2, \dots, \alpha_m$ – fragmente finale de expresii pe ramurile cu recursivitate stângă
 - ▶ $\beta_1, \beta_2, \dots, \beta_n$ – expresiile de pe ramurile fără recursivitate stângă

$$A = A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$



$$A = \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$
$$A' = \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Exemplu eliminare recursivitate stângă

```
postfix = postfix LBRACKET expr RBRACKET  
        | postfix INC  
        | postfix DEC  
        | postfix DOT ID  
        | factor
```



```
postfix = factor postfixPrim  
postfixPrim = LBRACKET expr RBRACKET postfixPrim  
            | INC postfixPrim  
            | DEC postfixPrim  
            | DOT ID postfixPrim  
            | ε
```

Eliminarea recursivității stângi (2)

expr = **expr** (ADD | SUB) termen | termen



expr = termen exprPrim

exprPrim = (ADD | SUB) termen exprPrim | ϵ

```
bool exprPrim(){  
    Atom *pStartAtom=pCrtAtom;  
    if(consume(ADD)||consume(SUB)){  
        if(termen()){  
            if(exprPrim()){  
                return true;  
            }  
        }  
    }  
    pCrtAtom=pStartAtom;  
    return true; //  $\epsilon$   
}
```

```
bool expr(){  
    Atom *pStartAtom=pCrtAtom;  
    if(termen()){  
        if(exprPrim()){  
            return true;  
        }  
    }  
    pCrtAtom=pStartAtom;  
    return false;  
}
```

Tratarea erorilor sintactice

- ▶ Una dintre funcțiile ANSIN este generarea de mesaje de eroare relevante, dacă a avut loc o eroare la parsarea sirului de atomi
- ▶ Mesajele de eroare trebuie să fie:
 - ▶ **localizate** – trebuie să conțină suficiente informații pentru a localiza de unde anume s-a generat eroarea
 - ▶ **destinate celor care folosesc compilatorul** – textul mesajului de eroare trebuie conceput ca să aibă înțeles din perspectiva celor care folosesc compilatorul, nu din perspectiva celor care îl implementează
 - ▶ **specifice** – secvențe diferite din gramatică trebuie să genereze erori diferite, care să permită înțelegerea regulilor care nu au fost respectate

Localizarea erorilor

- ▶ Pentru a localiza o eroare, mesajul generat poate conține:
 - ▶ numele fișierului de intrare
 - ▶ linia și coloana din fișier
 - ▶ marginile din fișierul de intrare ale regulii în interiorul căreia s-a generat eroarea
- ▶ Pentru generarea erorilor se poate folosi o funcție (ex: `err(mesaj_de_eroare)`), care să adauge automat la mesajul generat informațiile de localizare, pe baza atomului curent)

```
exprAdd = exprAdd ( ADD | SUB ) exprMul | exprMul
```

```
// test1.c
...
for(i=1;i<n;i++)
    if(v[i-]<v[i]){
        ...
    }
```



test1.c [249,13]: expresie invalidă după -

Mesaje pentru cei care folosesc compilatorul

- ▶ Perspectiva celor care implementează un compilator este diferită de perspectiva celor care îl vor folosi
- ▶ Mesajele de eroare întotdeauna vor trebui să implementeze perspectiva celor care vor folosi compilatorul

```
declVar = type ID SEMICOLON
```

```
exprAdd = exprAdd ( ADD | SUB ) exprMul | exprMul
```

// perspectiva celor care implementează compilatorul

- lipsește ID după tipul variabilei
- în exprAdd lipsește exprMul după + sau -

// perspectiva celor care folosesc compilatorul

- lipsește numele variabilei
- lipsește expresia de după + sau -

Mesaje specifice

- ▶ Erorile care apar în locuri diferite trebuie să aibă texte diferite
- ▶ Textele trebuie să reflecte cât mai bine semnificația construcțiilor gramaticale care le-au generat

declVar = type ID SEMICOLON

declFn = type ID LPAR arg (COMMA arg)* RPAR LACC instr* RACC

arg = type ID

instr = exprWhile | LACC instr* RACC

// mesaje vagi sau prea generale

- lipsește numele // numele cui? variabilei, funcției, argumentului?
- lipsește { // care accoladă? de la funcție sau de la instrucție?

// mesaje specifice

- lipsește numele variabilei
- lipsește { după antetul funcției

Determinarea erorilor posibile

- ▶ Prin analiza gramaticii se determină ce erori pot fi generate pe baza ei
- ▶ Regulă generală: **dacă într-un anumit punct din gramatică există posibilitatea ca analiza să se blocheze, în acel punct este posibilă o eroare.**
- ▶ Algoritm pentru tratarea erorilor posibile care pot apărea într-o gramatică:
 - ▶ regulile în sine nu vor genera erori, ci doar vor returna *false* dacă nu sunt îndeplinite.
 - ▶ în cadrul fiecărei reguli se determină care sunt punctele în care analiza se poate bloca
 - ▶ pentru fiecare dintre aceste puncte se va prevedea un mesaj de eroare specific

Analiza e₁_e₂

- ▶ Dacă un element din secvență nu se poate consuma, rezultă eroare
- ▶ În general erorile vor fi de forma "*lipsește ...*" sau "*invalid*", deoarece acea construcție nu a fost găsită la poziția curentă fie din cauză că nu există, fie din cauză că este greșită și deci nu poate fi consumată
- ▶ Mesajul trebuie să reflecte situația cea mai probabilă: *lipsește/invalid*

```
bool instrWhile(){  
    Atom *pStartAtom=pCrtAtom;  
    if(consume(WHILE)){  
        if(consume(LPAR)){  
            if(expr()){  
                if(consume(RPAR)){  
                    if(instr()){  
                        return true;  
                    } else err("instructiune invalida pentru corpul while");  
                } else err("conditie while invalida sau lipseste ");  
            } else err("conditie invalida pentru while");  
        } else err("lipseste ( dupa while");  
    }  
    pCrtAtom=pStartAtom;  
    return false;  
}
```

instrWhile = WHILE LPAR expr RPAR instr

Analiza primului element din $e_1 \cdot e_2$

- ▶ Primul element dintr-o secvență va putea genera eroare doar dacă secvența este obligatorie

```
program = linie+ FINISH  
linie = ID COMMA NR
```

```
bool program(){  
    if(linie()){  
        // linie+ -> linie linie*  
        while(linie()){}  
        if(consume(FINISH)){  
            return true;  
        }else err("eroare de sintaxă");  
        }else err("fișier vid");  
    return false;  
}
```

Analiza e₁ ⊥ e₂

- ▶ În general mesajele nu vor putea fi foarte specifice, deoarece nu se știe care alternativă s-a apropiat cel mai mult de forma corectă
- ▶ Se poate mări specificitatea mesajelor dacă în cursul analizei se salvează informații cu privire la calea care a fost urmată până în punctul curent

// mesaje posibile dacă nu se consumă instr: "*instructiune invalidă*" sau "*eroare de sintaxă*"

program = instr+ FINISH

instr = instrWhile | instrIf | expr SEMICOLON

// mesaj posibil: "*expresie invalidă după +/-*"

exprAdd = exprAdd (ADD | SUB) exprMul | exprMul

// se poate crește specificitatea mesajului dacă se ține minte operatorul consumat și în mesaj se va afișa acel operator: "*expresie invalidă după -*"

Analiza e? și e*

- ▶ nu se vor genera mesaje de eroare, deoarece se poate considera că atunci când nu poate fi consumată nu este vorba de o eroare ci de lipsa elementului
- ▶ **Atenție:** chiar dacă "e?" sau "e*" nu generează mesaje de eroare, este posibil ca "e" să genereze mesaje, deci va trebui analizată separat

// *MUL?* nu va genera niciodată mesaje de eroare

declVar = type *MUL?* ID SEMICOLON

// (*declVar* | *declFunc*)* nu va genera niciodată mesaje de eroare

program = (*declVar* | *declFunc*)* FINISH

// "*ELSE instr*" este o secvență optională, dar ea trebuie analizată separat: dacă s-a consumat *ELSE* și nu urmează *instr*, se generează mesaj de eroare

instrIf = IF LPAR expr RPAR instr (ELSE instr)?

// '*COMMA ID*' este o secvență cu repetiție optională, dar se analizează separat: dacă s-a consumat *COMMA* și nu urmează *ID*, se generează mesaj de eroare

declVar = type ID (COMMA ID)* SEMICOLON

Limbaje formale și tehnici de compilare

Curs 7: analiza semantică; arborele sintactic abstract; atribută moștenite și sintetizate; reguli semantice

Analiza semantică (ANSEM)

- ▶ Se ocupă cu semnificația programului
- ▶ Sintaxa stabilește forma programului, iar semantica determină conținutul său
- ▶ Pot exista mai multe forme care să aibă același conținut (ex: aproape fiecare limbaj de programare are o instrucțiune *while*, dar cu o sintaxă diferită)
- ▶ Analiza semantică se compune din:
 - ▶ **Analiza de domeniu** – analizează declarațiile din program (variabile, funcții, ...)
 - ▶ **Analiza de tipuri** – analizează folosirea constantelor și a simbolurilor

Arborele sintactic abstract (ASA)

- ▶ ASA reprezintă conținutul unui program (en: AST - Abstract Syntax Tree)
- ▶ ASA este o structură arborescentă de obiecte, fiecare obiect fiind o construcție de limbaj, iar copiii unui nod fiind componentele acelei construcții
- ▶ Din ASA este eliminată sintaxa și astfel construcții identice reprezentate în limbaje diferite conduc la același ASA

// C, C++, C#, Java, PHP
if(a<0)f(a);

; Lisp, Scheme
(if (< a 0) (f a))

{ Pascal }
if a<0 then f(a);

If

<

Id:a

Int:0

()

Id:f

Id:a

Python
if a<0:
 f(a)

Ruby
if a<0
 f(a)
end

Folosirea ASA (1)

- ▶ ASA se folosește ca o reprezentare intermediară în compilatoare, în special pentru LP mai complexe sau la care este nevoie de mai multe procesări
- ▶ ASA este util la LP care acceptă folosirea simbolurilor înainte de definirea lor (ex: Java). La aceste LP sunt necesare mai multe parcurgeri ale codului sursă, prima oară fiind pentru colectarea simbolurilor.
- ▶ Dacă se dorește generarea ASA, regulile sintactice vor cuprinde codul pentru preluarea informațiilor utile din atomii lexicali și formarea ASA cu acestea
- ▶ După ce s-a construit ASA, nu mai este nevoie de atomii lexicali, aşa că se poate renunța la aceştia
- ▶ Alternativ, dacă nu se dorește folosirea ASA, în faza de analiză sintactică se pot implementa direct următoarele etape (analiză de domeniu, de tipuri, generare de cod, ...)

Implementarea ASA

- ▶ Pentru construcția ASA, se definesc structurile de date care vor conține informațiile utile din program, iar apoi se populează aceste structuri cu datele extrase din codul sursă
- ▶ Structura de date pentru ASA se poate implementa ca o ierarhie de clase, având ca bază o clasă abstractă
- ▶ Fiecare construcție de limbaj (declarații, instrucțiuni, expresii) va fi implementată prin subclase specifice

```
// clasa de bază pentru toate nodurile ASA           C++
struct ASA{                                         // ASA
    InputPos pos;                                // poziția în sursă
};

enum class Type {INT, REAL};
struct ASAVar:ASA{                               // o definiție de variabilă
    Type type;
    string name;
};
```

Atribute sintetizate și moștenite (1)

- ▶ Pentru ca regulile sintactice să poată să returneze componentele utile care au fost recunoscute în cadrul analizei sintactice, este necesar să existe un mecanism prin care aceste reguli să poată returna componentele recunoscute
- ▶ Uneori este nevoie ca o regulă să primească informații care deja au fost extrase de alte reguli
- ▶ Pentru toate aceste situații, se folosesc **atribute**, care sunt echivalentul parametrilor unei funcții și valorilor returnate:
 - ▶ **attribute sintetizate**
 - ▶ **attribute moștenite**

Atribute sintetizate și moștenite (2)

- ▶ **Atribute sintetizate** – valori care sunt calculate în interiorul regulii sintactice și apoi returnate de ea. Atributele sintetizate sunt echivalente valorilor returnate de o funcție.
- ▶ **Atribute moștenite** – valori pe care regula sintactică le primește din exterior, în general de la regulile sintactice părinti sau frați. Atributele moștenite sunt echivalente parametrilor unei funcții.
- ▶ Unei reguli sintactice i se pot asocia unul sau mai multe atrbute
- ▶ Atributele se scriu după numele regulii sintactice, între paranteze drepte, specificându-se pentru fiecare ce fel de atribut este (**out**-sintetizat, **in**-moștenit) și tipul său

```
// regula angajat primește atributul moștenit functie
// și returnează atributul sintetizat a
angajat[in functie:string, out a:Angajat] = ...
```

```
#sofer
Ion, 2500
Marin, 2600
```

Definirea regulilor semantice (RS)

- ▶ Regulile semantice – definesc interacțiunile și transformările din cadrul limbajului
- ▶ Pentru ANSEM nu există un formalism abstract, cum sunt gramaticile pentru analiza sintactică. Din acest motiv, există mai multe metode de a defini regulile semantice:
 - ▶ **prin limbaj natural** – se explică în limbaj natural ce anume trebuie să facă fiecare RS
 - ▶ **prin descrierea într-un limbaj de programare** – se implementează regula folosind un limbaj de programare, pentru a se descrie acțiunea sa
 - ▶ **prin formalism matematic** – se descriu matematic acțiunile regulilor (ex: $e_1, e_2 \in \mathcal{R}$; dacă $e_1 \neq 0$ și $e_2 \neq 0$, atunci $e_1 \& \& e_2 \neq 0$; altfel $e_1 \& \& e_2 = 0$)

Exemplu de RS în limbaj natural

- ▶ Un bloc definește un nou domeniu
- ▶ În același domeniu nu pot exista două simboluri cu același nume
- ▶ Într-un domeniu imbricat se pot defini simboluri având același nume ca simbolurile din domeniile părinti
- ▶ Tipul rezultat din operațiile aritmetice având un operand de tip real și altul de tip întreg este tipul operandului real
- ▶ Operatorul restul împărțirii (%) acceptă doar operanzi de tip întreg

Includerea RS în gramatica sintactică

- ▶ RS se includ în analizorul sintactic, între acolade
- ▶ O RS poate fi inclusă ca text în limbaj natural, pseudocod sau ca o secvență într-un limbaj de programare
- ▶ Dacă o regulă gramaticală a fost definită ca având atribute, regula se va folosi întotdeauna cu parametri corespunzători atributelor, puși între []
- ▶ Dacă după atomii lexicali se pune [], între [] se află o variabilă care va primi valoarea atomului

```
angajati = functie* FINISH
```

```
functie = HASH ID[functie]
```

```
( angajat[functie.text,a] {bd.add(a);} )*
```

```
angajat[in functie:string, out a:Angajat] =
```

```
ID[nume] COMMA REAL[salariu] {
```

```
    a.functie=functie;
```

```
    a.nume=nume.text;
```

```
    a.salariu=salariu.r;
```

```
}
```

```
#sofer  
Ion, 2500  
Marin, 2600  
#secretara  
Ana, 2500
```

```
typedef struct{  
    ...  
    union{  
        float r; // numere reale  
        char text[MAX]; // ID  
    };  
}Atom;
```

Exemplu: calculator

```
calculator = ( expr[e] {printf("%g\n",e);} ) * FINISH
expr[out r:float] = expr[st]
  ( ADD termen[dr] {r=st+dr;}
  | SUB termen[dr] {r=st-dr;} ) | termen[r]
termen[out r:float] = termen[st]
  ( MUL factor[dr] {r=st*dr;}
  | DIV factor[dr] {r=st/dr;} ) | factor[r]
factor[out r:float] = NR[n] {r=n.r;} | LPAR expr[r] RPAR
```



Implementarea atributelor (1)

- ▶ Pentru atomii lexicali, funcția *consume* va memora atomul consumat, a.î. acesta să poată fi accesat după consumare
- ▶ Dacă LP permite (C, C++, C#, ...), atributele (sintetizate) se implementează folosind transfer prin referință, pentru a da posibilitatea regulii să modifice valoarea lor

```
// factor[out r:float] = NR[n] {r=n.r;} | LPAR expr[r] RPAR      C
int factor(float *r){
    if(consume(NR)){
        Atom *n=consumed;          // consumed – ultimul atom consumat
        *r=n->r;
        return 1;
    }
    if(consume(LPAR)){
        if(expr(r)){
            if(consume(RPAR)){
                return true;
            }else err("lipsește ) după expresie");
            }else err("expresie invalidă după (");
        }
    return 0;
}
```

Implementarea atributelor (2)

- ▶ Dacă LP nu permite transfer prin referință (Java, Python, ...), atributele se pot implementa prin comasarea lor cu semnificația valorii returnate de funcție: dacă funcția returnează *null*, acesta are semnificația de *false*, altfel se consideră că a returnat atributul
- ▶ Atributele de tip atomic în Java (ex: *float*) se vor implementa ca tip referință, pentru a fi nulabile (*float -> Float*)
- ▶ Dacă trebuie returnate mai multe atrbute, se vor comasa într-o clasă

```
// factor[out r:float] = NR[n] {r=n.r;} | LPAR expr[r] RPAR           Java
Float factor(){
    if(consume(NR)){
        Atom n=consumed;
        return n.r;          // automatic boxing (float->Float); altfel return new Float(n.r);
    }
    if(consume(LPAR)){
        Float e;
        if((e=expr())!=null){
            if(consume(RPAR)){
                return e;
            }else err("lipsește ) după expresie");
            }else err("expresie invalidă după ());
        }
    return null;
}
```

Limbaje formale și tehnici de compilare

**Curs 8: analiza de domeniu; tabela de
simboluri**

Analiza de domeniu (AD)

- ▶ Analizează structurile din program care introduc noi simboluri: definiții, declarații, ...
- ▶ **Definiție** – o structură de program care specifică în mod complet și introduce un nou simbol, posibil împreună cu conținutul său
- ▶ **Declarație** – o structură de program care specifică (posibil incomplet) un simbol definit în altă parte din program
- ▶ În AD se creează **tabela de simboluri (TS)**
- ▶ TS va fi folosită atât în AD cât și în alte etape

Definiții	Declarații
int a;	extern int a;
char v[100];	char v[];
double f(int x){...}	double f(int x);

Simboluri

- ▶ Simbolurile sunt identificatorii folosiți în cadrul unui program, împreună cu semnificația și informațiile lor asociate
- ▶ Simbolurile pot fi variabile, funcții, tipuri, ...

```
int i,v[100];          // variabile
struct Punct{           // nume de structuri
    float x,y;          // câmpuri de structuri
};
typedef double(*FnPtr)(double); // nume de tipuri
int f(int a,int b){      // funcții și parametri
    goto err;
    ...
err: // etichete pentru goto
    ...
}
```

Domeniul și contextul unui simbol

- ▶ Domeniu – toate locurile din program în care un simbol poate fi folosit
- ▶ Context – toate simbolurile care sunt disponibile într-o anumită locație a unui program
- ▶ La mareea majoritate a limbajelor cu structură de bloc, un bloc definește un nou domeniu care este imbricat în domeniul părinte

```
int a;                      // domeniul lui a: tot programul (global)
int sum(int *v,int n){
    int i,s=0;              // domeniile lui i și s: funcția sum
    for(i=0;i<n;i++){
        int e=v[i];          // domeniul lui e: blocul { ... }
        s+=e;
    }
    // contextul în această locație: a, sum, v, n, i, s
    return s;
}
// contextul în această locație: a, sum
```

Informații asociate simbolurilor

- ▶ **fel** – variabilă, funcție, parametru, structură, ...
 - ▶ **tip** – int, float[], double (*)(double)
 - ▶ **nivelul de imbricare al blocului curent** – diferențiază între variabilele globale și cele locale
 - ▶ **parametrii și codul funcțiilor, câmpurile structurilor, ...**

```
enum{FEL_VAR,FEL_FN,FEL_STRUCT,FEL_PARAM};  
typedef struct{  
    String nume;           // numele simbolului  
    int fel;               // FEL_*  
    Tip tip;  
    int imbricare;         // 0=global ...  
    // pentru functii: parametrii; pentru structuri: campurile  
    Simboluri simboluri;  
    Instructiuni instructiuni; // pentru functii: codul  
}Simbol;
```

Tabela de simboluri (TS)

- ▶ Tabela de simboluri – o structură de date care memorează toate simbolurile împreună cu informațiile asociate lor
- ▶ Operații cu TS
 - ▶ Adăugarea unui nou simbol
 - ▶ Căutarea unui simbol
 - ▶ Adăugarea unui nou domeniu
 - ▶ Ștergerea unui domeniu

Structura tableei de simboluri

- ▶ De obicei TS se implementează ca o stivă de domenii, fiecare domeniu conținând toate simbolurile definite în el
- ▶ Inițial în TS se află un singur domeniu, corespunzător simbolurilor globale
- ▶ La intrarea într-un bloc, în TS se adaugă un nou domeniu corespunzător noului bloc
- ▶ La ieșirea din blocul curent, din TS se șterge domeniul curent și toate simbolurile din acesta

```
// stiva de domenii; primul domeniu este cel global
typedef struct{
    Domeniu *domenii;           // stivă implementată folosind listă
}TS;
void domeniuNou(TS *ts){...} // creează un nou domeniu în vârful stivei
void stergeDomeniu(TS *ts){...} // șterge domeniul curent
void adaugaSimbol(TS *ts, Simbol *sim){...} // adaugă un simbol
Simbol *cautaSimbol(TS *ts, String nume){...} // caută un simbol
```

Structura unui domeniu

- ▶ În interiorul structurii de domeniu se pot folosi structuri de date optimizate pentru căutarea eficientă a unui simbol: tabele hash, arbori de căutare, sortare pentru căutare binară, ...
- ▶ În cazul limbajelor care acceptă supraîncărcarea funcțiilor (C++, Java, C#) este necesar ca un domeniu să permită stocarea mai multor simboluri cu același nume

```
bool less(int a,int b){return a<b;}  
bool less(double a,double b){return a<b;}  
bool less(const char *s1,const char *s2){return strcmp(s1,s2)<0;}
```

```
// toate simbolurile  
// care au același nume  
typedef struct{  
    String nume;  
    Simboluri simboluri;  
}AcelasiNume;
```

```
// domeniu care permite simboluri  
// cu același nume  
typedef struct{  
    AcelasiNume *numeDistincte;  
    int nNumeDistincte;  
}Domeniu;
```

Exemple reguli semantice pentru AD

- ▶ O funcție definește un nou domeniu, care cuprinde parametrii și simbolurile definite în interiorul acoladelor funcției, dar nu și în blocurile {...} interioare acestora
- ▶ Un bloc definește un nou domeniu
- ▶ În același domeniu nu pot exista două simboluri cu același nume
- ▶ Într-un domeniu imbricat se pot defini simboluri având același nume ca simbolurile din domeniile părinti
- ▶ La căutarea unui simbol, se începe cu domeniul curent, iar apoi se trece la domeniile părinti, în ordinea apropierei lor ca nivel de imbricare de cel curent (domeniu curent → părinte direct → ...)

Exemplu de analiză de domeniu

```
int a,b;
```

// parametrul **a** este OK fiindcă funcția definește un nou domeniu
void f(int x,int y,int a){

int x; // eroare: domeniul funcției este același cu domeniul
parametrilor săi => redefinire **x**

int b; // OK: **b** are voie să fie redefinit într-un nou domeniu

```
    if(a){
```

int y; // OK: blocul {...} definește un nou domeniu în care
se poate defini o nouă variabilă **y**

```
    }
```

```
}
```

int f; // eroare: în domeniul global există deja un simbol
denumit **f** => redefinire **f**

Integrarea TS în compilator (1)

- ▶ TS se creează în faza de analiză de domeniu și apoi se folosește în fazele ulterioare (analiză de tip, generare de cod, ...)
- ▶ Analiza de domeniu se poate realiza în faza de analiză sintactică. Astfel, regulile sintactice vor realiza și:
 - ▶ Verificarea TS pentru consistență (ex: noul simbol adăugat să nu fie o redefinire; funcțiile să fie declarate doar în domeniul global)
 - ▶ Adăugarea la TS a simbolurilor declarate în aceste reguli
- ▶ Pentru limbajele mai complexe, analiza sintactică poate fi folosită pentru a se genera un ASA, iar apoi fazele ulterioare să fie implementate prin parcurgeri multiple ale acestui ASA

Integrarea TS în compilator (2)

- ▶ În funcție de complexitatea RS necesare pentru analiza de domeniu, putem distinge următoarele tipuri de limbaje:
 - ▶ **Limbaje cu declarații statice și predeclarare** (Pascal, C)
– un simbol trebuie prima oară declarat și apoi folosit
(în C se permit și apeluri de funcții nedeclarate încă, presupunându-se că ulterior funcțiile vor fi definite conform unor reguli predefinite)
 - ▶ **Limbaje cu declarații statice fără predeclarare** (C++, C#, Java) – unele construcții (ex: în interiorul claselor) permit folosirea unui simbol care va fi declarat ulterior
 - ▶ **Limbaje cu declarații dinamice** (Python, JavaScript, PHP, Ruby) – simbolurile se pot declara sau șterge chiar și în faza de execuție a programului

Limbaje cu declarații statice și predeclarare

- ▶ Deoarece simbolurile trebuie prima oară declarate și apoi folosite, analiza de domeniu și popularea TS se poate face direct din faza de analiză sintactică:
 - ▶ Orice regulă sintactică ce definește un nou domeniu (ex: funcții, structuri) va adăuga domeniul respectiv în TS, care este organizată ca o stivă de domenii
 - ▶ La sfârșitul unei reguli sintactice ce definește un domeniu, din TS se șterge domeniul definit de ea (vârful stivei)
 - ▶ Când se procesează o declarație, simbolul declarat se verifică pentru consistență și se introduce în domeniul curent din TS

Exemplu limbaje cu declarații statice și predeclarare

```
// În TS este un singur domeniu, cel global: {}
int a,b;           // se introduc a și b în TS (în domeniul global): {[a,b]}

void f             // se introduce f în TS: {[a,b,f]}
    (int x,int y,int a) // se adaugă un nou domeniu și se introduc
x,y,a: {[a,b,f], [x,y,a]}}

{
    int b;          // se introduce b în TS: {[a,b,f], [x,y,a,b]}
    if(a){          // se adaugă un nou domeniu: {[a,b,f], [x,y,a,b], []}
        int y;      // se introduce y în TS: [a,b,f], [x,y,a,b], [y]}
        }
    }               // se șterge domeniul curent (al instrucțiunii if):
{[a,b,f], [x,y,a,b]}

}                   // se șterge domeniul curent (al funcției f): {[a,b,f]}
// În TS rămâne doar domeniul global: {[a,b,f]}
```

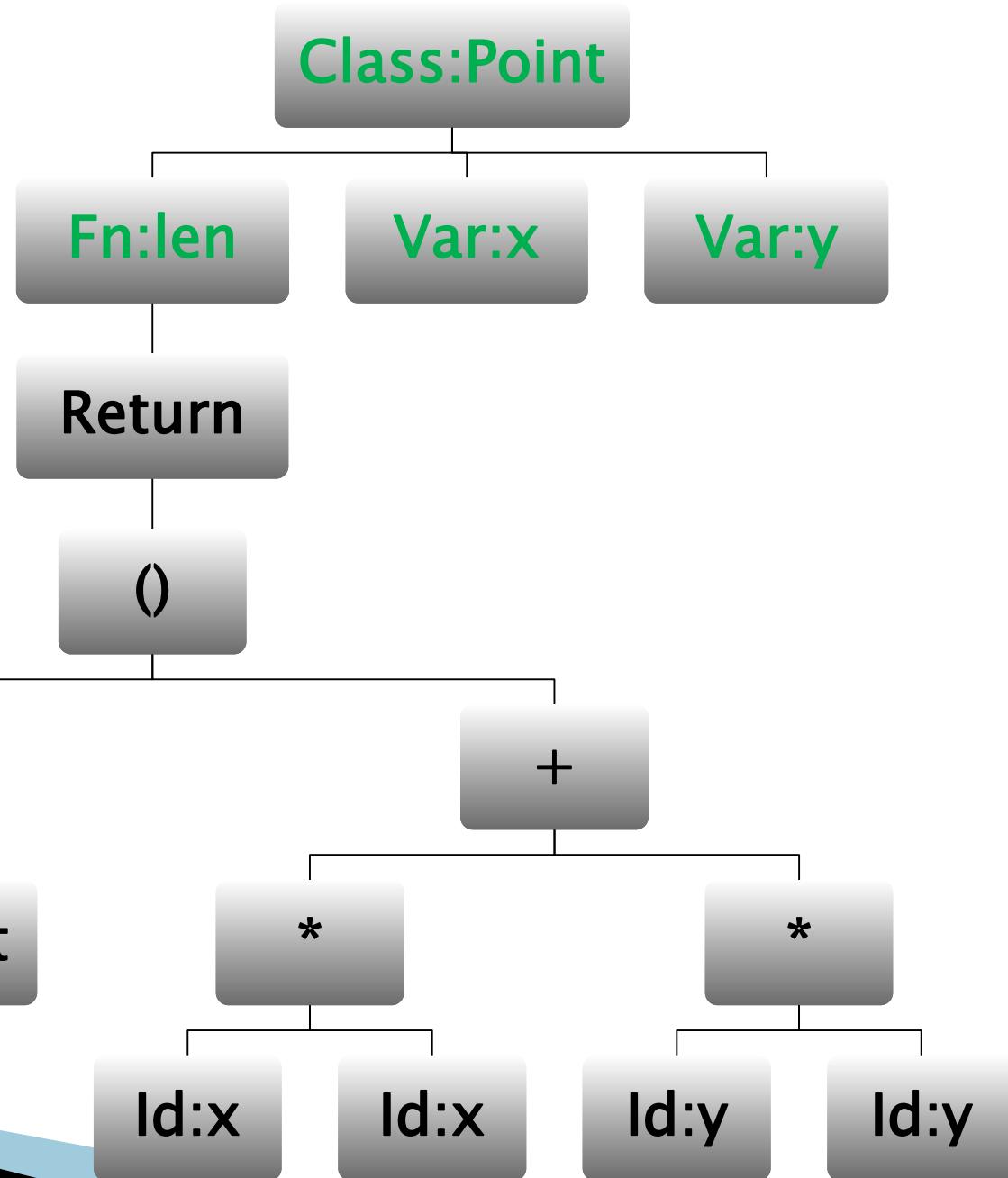
Limbaje cu declarații statice fără predeclarare

- ▶ Deoarece simbolurile pot fi folosite înainte de a fi declarate, nu este posibil ca folosind doar o singură trecere, într-un anumit punct din cod să se știe tot contextul curent
- ▶ În faza de analiză sintactică se construiește ASA și apoi acesta va fi folosit pentru etapele ulterioare
- ▶ Deoarece ASA va conține și noduri pentru declarații (ex: clase, funcții, variabile), ASA poate fi văzut ca o TS, eventual adnotându-se aceste noduri și cu alte atrbute necesare TS. La căutarea unui simbol, ASA se va parcurge din nodul curent către rădăcină, în același timp verificându-se toți copiii direcți ai nodurilor traversate ascendent
- ▶ Alternativ, se poate construi o TS separată prin parcurgerea ASA în lățime și colectarea tuturor simbolurilor de pe un anumit nivel, înainte de a se trece la următorul subnivel

```
class Point{  
    double len(){  
        return Math.sqrt(x*x+y*y);      // x,y folosite înainte de declarare  
    }  
    double x,y;  
}
```

Exemplu limbaje cu declarații statice fără predeclarare

```
class Point{  
    double len(){  
        return Math.sqrt(x*x+y*y);  
    }  
    double x,y;  
}
```



Limbaje cu declarări dinamice

- ▶ Deoarece simbolurile se pot adăuga și șterge în timpul execuției programului (la **runtime**), este necesar să existe posibilitatea de a se interoga în timpul execuției programului simbolurile disponibile într-un anumit loc
- ▶ În timpul execuției se vor păstra TS asociate domeniilor active (global, locale funcțiilor, ...)
- ▶ Deoarece fiecare instanță a unei clase poate avea simboluri specifice, care sunt diferite de cele din alte instanțe ale aceleiași clase, fiecare instanță va trebui să aibă o TS proprie
- ▶ În general, la compilare, analiza semantică pentru aceste limbaje este simplă, astfel încât se poate realiza în faza de analiză sintactică

Exemplu cu declarații dinamice

```
class Person{} // JavaScript (ECMAScript 6 pentru class)
```

```
var p1=new Person(); // p1 este o instanță de Person, fără niciun  
atribut
```

```
p1.name="Ana"; // definire dinamică de atribut doar pentru  
p1; p1 trebuie să aibă propria sa TS, pentru a memora attributele  
specifice
```

```
console.log(p1.name); // => Ana
```

```
var p2=new Person();  
console.log(p2.name); // => undefined; p2 nu are atributul name
```

Limbaje formale și tehnici de compilare

Curs 9: analiza de tipuri

Analiza de tipuri (AT)

- ▶ Analizează dacă simbolurile sunt folosite în concordanță cu tipul lor
- ▶ La limbajele de programare cu inferență de tipuri (determinarea automată a tipului unui simbol), deduce tipurile simbolurilor
- ▶ În funcție de tipul LP, AT poate fi făcută la compilare (compile time), pentru LP cu tipuri statice (statically typed) sau la execuție (runtime), pentru LP cu tipuri dinamice (dynamically typed)

```
int x,y;  
x(); // eroare: o variabila de tip int nu poate fi apelata ca functie  
y=printf+2; // eroare: operatorul + nu poate fi aplicat unei functii
```

LP cu tipuri statice (statically typed)

- ▶ Tipurile sunt asociate simbolurilor și este cunoscut încă de la compilare
- ▶ Tipul unui simbol nu mai poate fi schimbat ulterior
- ▶ Tipurile pot fi specificate explicit, de către programator, sau se poate utiliza inferența de tipuri pentru determinarea lor automată
- ▶ Exemple: C/C++, C#, Java, OCaml, Haskell, Rust

```
// C
int fact(int n){
    return n<3 ? n : n*fact(n-1);
}
```

```
(* OCaml *)
let rec fact n = if n<3 then n else n * fact(n-1)
```

Caracteristicile LP cu tipuri statice

- ▶ Sunt mai rapide decât LP cu tipuri dinamice, deoarece AT se face încă de la compilare și toate informațiile obținute despre tipuri pot fi folosite la optimizarea codului
- ▶ Necesară mai puțină memorie la execuție, deoarece informațiile despre tipuri nu mai sunt necesare la execuție
- ▶ Se pot detecta încă de la compilare erori sau atenționări (warnings) care țin de AT
- ▶ Dacă LP nu are inferență de tipuri, în general codul sursă este mai mare, deoarece trebuie specificate explicit toate tipurile
- ▶ Dacă tipurile sunt date explicit în codul sursă, se crește intelligibilitatea codului, în special în cazurile în care este vorba despre cod scris de altcineva

LP cu tipuri dinamice (dynamically typed)

- ▶ Tipurile sunt asociate valorilor simbolurilor, nu simbolurilor în sine
- ▶ În momente diferite un simbol poate conține valori de tipuri diferite
- ▶ Exemple: Javascript, Python, PHP, Ruby, Lisp, Prolog

```
// Javascript
function afis(msg){      // funcția afis primește orice tip de date
    console.log(msg);
}

var x=5;                  // x conține o valoare de tip numeric
afis(x);
x="salut";                // x conține o valoare de tip string
afis(x);
```

Caracteristicile LP cu tipuri dinamice

- ▶ Sunt mai lente decât LP cu tipuri statice, deoarece AT se face la execuție, în general înainte de fiecare operație
- ▶ Necesită mai multă memorie la execuție, deoarece fiecare valoare trebuie să aibă asociat tipul său
- ▶ Erorile specifice AT (ex: funcție apelată cu număr sau tipuri incorecte de argumente) se vor detecta doar la execuție, deci este nevoie de o testare mai complexă
- ▶ În general codul sursă este mai mic decât la LP cu tipuri statice
- ▶ Pentru a crește inteligențialitatea codului, programatorii pot specifica tipurile în comentarii sau în documentația atașată
- ▶ În general codul are tendința de a fi generic, el putând procesa valori cu tipuri diferite

Valori stânga și dreapta

- ▶ O **valoare stânga** (left-value, L-value) este o locație de memorie, care poate stoca o valoare
- ▶ Valorile stânga pot apărea în partea stângă a unei operații de atribuire, dacă nu sunt constante
- ▶ O **valoare dreapta** (right-value, R-value) nu are asociată o locație de memorie, deci nu i se pot atribui valori
- ▶ Valorile dreapta **nu** pot apărea în partea stângă a unei operații de atribuire

```
int x,v[10];
x=1;           // corect: x este lval și este variabilă
v[2]=5;         // corect: v[2] este lval
9=x;           // eroare: 9 este rval
```

Folosirea tabelei de simboluri pentru AT

- ▶ În timpul fazei de analiză de domeniu toate simbolurile din codul sursă au fost depuse în tabela de simboluri (TS)
- ▶ Căutarea în TS se face de la ultimele simboluri introduse către primele, astfel încât să se caute prima oară domeniile cele mai apropiate de locația curentă
- ▶ Dacă un identificator căutat în TS:
 - ▶ Nu este găsit, se raportează eroare: "*identificator nedefinit*"
 - ▶ Este găsit, se returnează o referință la definiția sa din TS

Folosirea TS pentru LP cu supraîncărcare

- ▶ Dacă LP permite supraîncărcarea (overloading) identificatorilor, atunci se vor returna toate definițiile posibile pentru acel identificator, urmând ca ulterior să se aleagă definiția potrivită, conform contextului din codul sursă
- ▶ Selecția definiției potrivite se face pe baza unor factori de diferențiere, cum sunt: numărul de parametri, tipurile lor, etc.

// C++

```
void afis(int nr);           // afis1
void afis(int nr,int repetare); // afis2
void afis(const char *str);   // afis3
void afis(const Punct &p);    // afis4

...
afis("salut");               // afis3
afis(9);                     // afis1
```

Exemple de reguli semantice pentru AT

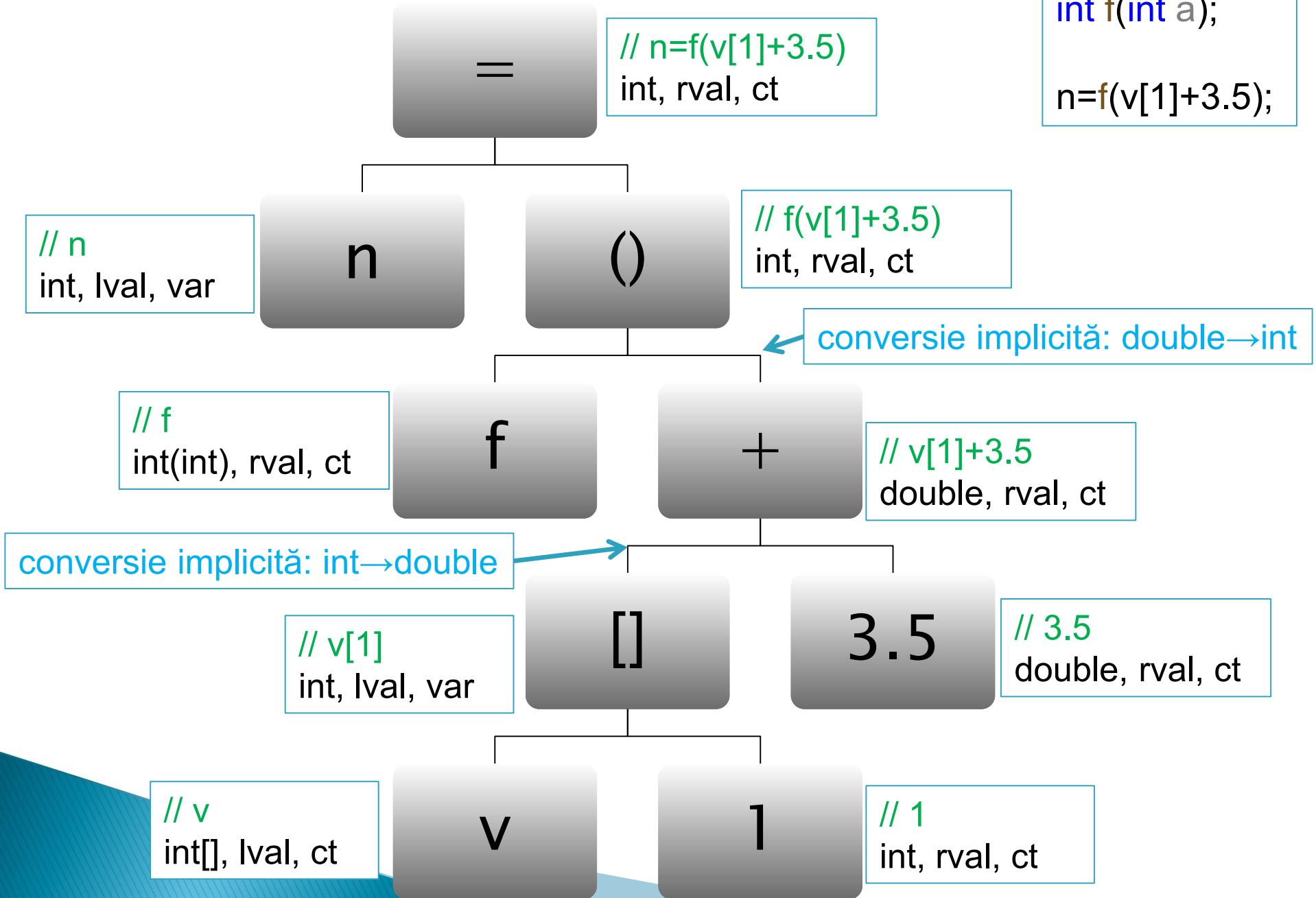
- ▶ Doar funcțiile pot fi apelate, posibil prin intermediul unor pointeri la funcții
- ▶ Numărul de argumente la apelul unei funcții trebuie să coincidă cu cel din definiția funcției
- ▶ Unele tipuri se pot converti implicit, dacă e nevoie (ex: int→double în operația $1+3.5$)
- ▶ Folosirea constantelor doar ca valori dreapta
- ▶ Destinațiile atribuirilor sau a operatorului adresa (`&x`) trebuie să fie valori stânga
- ▶ Indexarea se poate aplica doar vectorilor
- ▶ Nu se pot declara variabile de tip *void*
- ▶ La selecția unui câmp de structură, câmpul respectiv trebuie să existe
- ▶ Pointerii acceptă doar anumite operații (ex: adunarea cu un întreg, scăderea a doi pointeri)

Algoritm pentru AT la LP statice

- ▶ AT se poate face în faza de analiză sintactică sau prin parcurgerea ASA
- ▶ La analiza unei expresii, se analizează subexpresiile sale
- ▶ Pentru fiecare subexpresie, se va returna o structură de date care conține câmpurile necesare pentru RS, de exemplu:
 - ▶ Tipul
 - ▶ Dacă este valoare stânga sau dreapta
 - ▶ Dacă este constantă
- ▶ Folosind datele returnate de subexpresiile sale, se face AT pentru expresia părinte

Exemplu AT la LP statice

```
int n,v[10];  
int f(int a);  
  
n=f(v[1]+3.5);
```



AT pentru LP dinamice

- ▶ La LP dinamice este posibil ca la execuție o operație să primească la momente diferite operanzi de tipuri diferite
- ▶ Din acest motiv, în general AT se face la fiecare evaluare a unei expresii
- ▶ Valorile trebuie să aibă asociat propriul tip, astfel încât operațiile să poată să cu ce fel de date trebuie să opereze

a+b

```
enum{NB,STR};  
typedef struct{  
    int tip; // NB, STR  
    union{  
        double nb;  
        char *str;  
    };  
}Val;
```

```
Val operatorAdd(Val v1,Val v2){  
    switch(v1.tip){  
        case NB:switch(v2.tip){  
            case NB:return valNb(v1.nb+v2.nb);  
            case STR:return concat(nbToStr(v1.nb),v2.str);  
        }  
        case STR:switch(v2.tip){  
            case NB:return concat(v1.str,nbToStr(v2.nb));  
            case STR:return concat(v1.str,v2.str);  
        }  
    }  
}
```

Tipuri de date generice

- ▶ Tipurile de date generice (template, generics) permit ca un tip în sine să fie tratat ca o constantă, a cărei valoare poate fi dată de programator
- ▶ Se elimină astfel nevoia de a scrie cod foarte asemănător, atunci când codul este aproape identic la modificarea tipului de date asupra căruia acționează

```
template<class T> struct Vect{ // C++  
    T *v; // vector alocat dinamic cu n elemente  
    int n; // numărul de elemente din v  
    Vect(int n){v=new T[n];this->n=n;}  
    ~Vect(){delete []v;}
```

```
};
```

```
Vect<int> v1(10); // vector cu 10 elemente de tip int  
Vect<Pt> v1(5); // vector cu 5 elemente de tip Pt
```

AT pentru tipuri de date generice

- ▶ La instanțierea unui tip de date generic (ex: Vect<int>), se creează un nou tip de date, cu parametrii generici înlocuiți cu tipurile date la instanțiere
- ▶ Simbolurile nou create (clase, variabile, funcții, ...) se introduc în TS
- ▶ AT va trata aceste noi simboluri la fel ca pe orice alte simboluri

```
// introduce în TS tipul generic: template<class T> struct Vect
template<class T> struct Vect{...};
// introduce în TS tipul Vect<int> și variabila v1
Vect<int> v1(10);
// introduce în TS tipul Vect<Pt> și variabila v2
Vect<Pt> v2(5);
// introduce în TS variabila v3. Tipul Vect<int> este deja în TS
Vect<int> v3(10); // v1 va avea același tip ca v3
```

Inferența de tipuri (IT)

- ▶ Este deducerea automată a tipului unei construcții, fără a mai fi necesar ca programatorul să scrie explicit acel tip
- ▶ IT poate fi totală, atunci când programatorul nu mai trebuie să scrie în definiții niciun tip (ex: OCaml, Haskell) sau parțială, când LP permite inferența doar în cazul anumitor construcții (ex: C++, Rust)
- ▶ Algoritmii de inferență urmăresc să propage tipurile elementelor cunoscute la elementele ale căror tip nu se cunoaște încă
- ▶ Se ajunge astfel la un sistem de constrângeri, care specifică pentru fiecare tip ce condiții trebuie să îndeplinească
- ▶ Prin rezolvarea acestui sistem, se determină tipurile tuturor componentelor

Exemplu de inferență de tipuri

(* OCaml *)

```
let rec fact n = if n < 3 then n else n * fact(n-1)
```

- ▶ *fact* este o funcție care are un argument
- ▶ 3 și 1 sunt constante care au tipul *int*
- ▶ deoarece operatorii < (mai mic) și - (scădere) trebuie să aibă operanții de același tip, rezultă că tipul lui *n* este *int*
- ▶ deoarece la instrucțiunea *if* tipul expresiei *then* trebuie să fie identic cu tipul expresiei *else*, rezultă că operatorul * (înmulțire) trebuie să fie de tip *int*
- ▶ deoarece operatorul * este de tip *int*, ambele sale argumente trebuie să fie de tip *int*, deci funcția *fact* returnează o valoare de tip *int*
- ▶ Din condițiile de mai sus, tipul funcției *fact* este *int* → *int*

Limbaje formale și tehnici de compilare

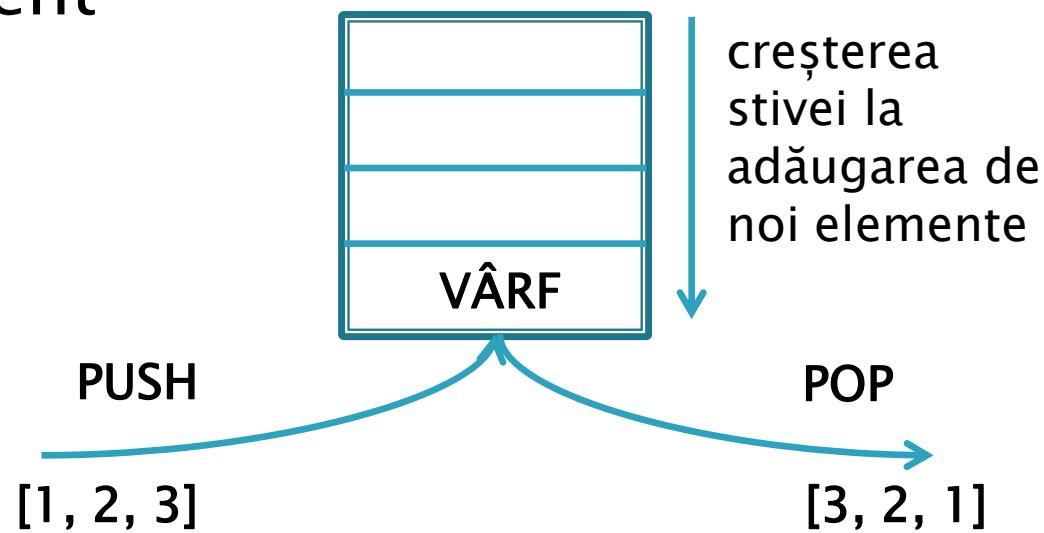
Curs 10: mașini virtuale bazate pe stivă:
structură, generare de cod, implementare

Mașini virtuale (MV)

- ▶ O MV modelează un calculator virtual (CPU, memorie...)
- ▶ Dacă un compilator emite cod pentru o MV, atunci codul emis va fi independent de sistemul gazdă, existând doar necesitatea de a exista MV instalată pe el (compile once, run everywhere)
- ▶ Deoarece trebuie emulate instrucțiunile MV folosind CPU-ul gazdă, codul MV va fi mai lent decât codul mașină echivalent pentru CPU
- ▶ Se poate evita emularea traducând instrucțiunile MV în cod mașină atunci când e nevoie să se execute (JIT – Just In Time compiling) sau la instalarea/prima rulare a programului (AOT – Ahead Of Time). Pentru AOT se pot aplica mai multe optimizări codului mașină, deoarece nu există restricții de timp limită privind compilarea, ca la JIT, unde compilarea e inclusă în timpul de execuție

Stiva

- ▶ O structură de date care implementează o colecție de elemente, cu proprietatea că la extragerea unui element, întotdeauna se va extrage ultimul element introdus (LIFO – Last In, First Out)
- ▶ **Vârful stivei** – locul prin care se adaugă și se extrag elemente din stivă
- ▶ **PUSH** – adăugare element
- ▶ **POP** – extragere element



Implementarea unei stive

- ▶ În general stiva se implementează cu ajutorul unui vector de dimensiune fixă sau variabilă. O variabilă index se folosește pentru a indica ultimul element introdus (vârful stivei).
- ▶ Se pot implementa stive și folosind liste simplu înlántuite, cu adăugare la început de listă (considerat vârful stivei)

Operație	Efect	Stivă inițială → finală (vârf stivă la dreapta)
PUSH x	adaugă valoarea x în stivă	... → ... x
POP var	extragă ultima valoare din stivă și o depune în variabila dată	... x → ... ; var=x
PEEK var	depune ultima valoare din stivă în variabila dată, fără a o elimina din stivă	... x → ...x ; var=x
DROP	șterge ultima valoare din stivă	... x → ...
DUP	duplică ultima valoare din stivă	... x → ...x x

Mașini virtuale bazate pe stivă

- ▶ Folosesc una sau mai multe stive pentru efectuarea operațiilor, păstrarea variabilelor locale și pentru apelurile de funcții
- ▶ Sunt simplu de implementat
- ▶ Generarea de cod pentru ele este simplă
- ▶ Exemple: JVM (Java Virtual Machine), CLR (Common Language Runtime – mașina virtuală pentru .NET)
- ▶ Argumentele pentru operații (aritmetice, logice, conversii, ...), apeluri de funcții și instrucțiuni se transmit prin intermediul stivei, iar rezultatul va fi returnat tot pe stivă
- ▶ Valorile din stivă pot avea tipuri diferite, dar trebuie să ocupe un număr întreg de celule de stivă

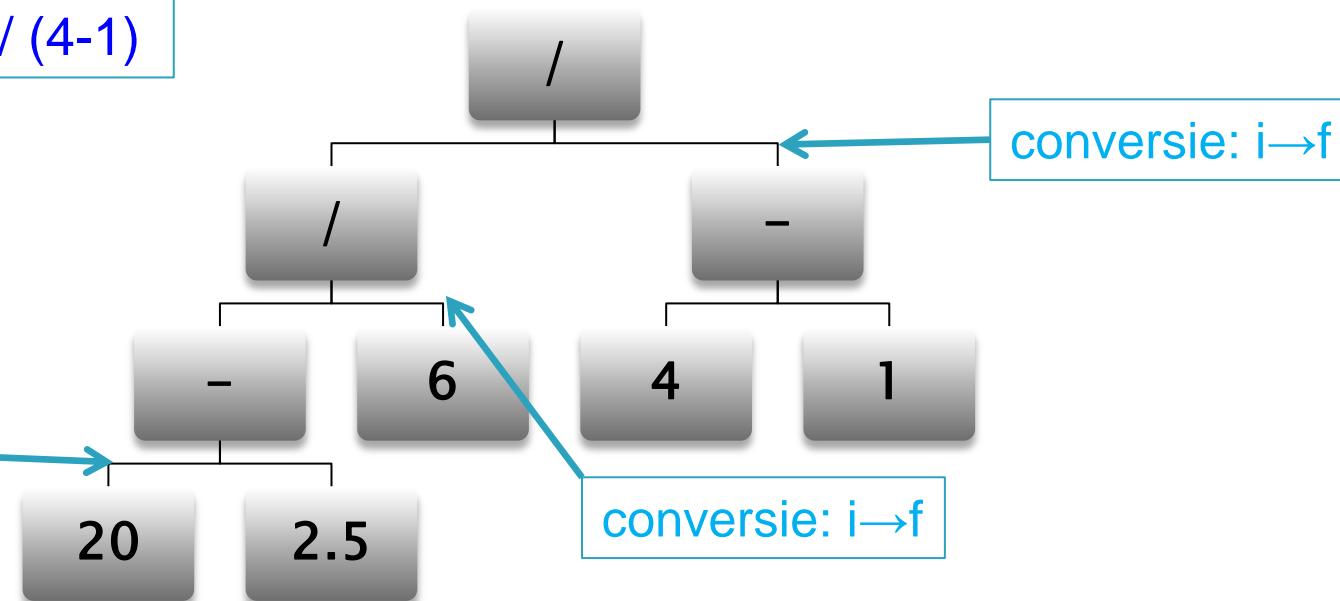
Implementarea operațiilor aritmetice

- ▶ Deoarece se poate opera cu tipuri diferite, reprezentate diferit în memorie, trebuie să existe operații distincte pentru fiecare tip de date
- ▶ Convenții:
 - ▶ o literă mică cu punct după codul operației specifică tipul operanzilor (i – întreg, f – floating point)
 - ▶ constantele de un anumit tip vor începe cu litera tipului

Operație	Stivă inițială → finală
ADD.i	... $i_1 \ i_2 \rightarrow \dots (i_1 + i_2)$
DIV.f	... $f_1 \ f_2 \rightarrow \dots (f_1 / f_2)$
CONV.i.f	... $i_1 \rightarrow \dots f_2$
AND – doar pentru întregi	... $i_1 \ i_2 \rightarrow \dots (i_1 \ \&\& \ i_2)$
BITAND – doar pentru întregi	... $i_1 \ i_2 \rightarrow \dots (i_1 \ \& \ i_2)$

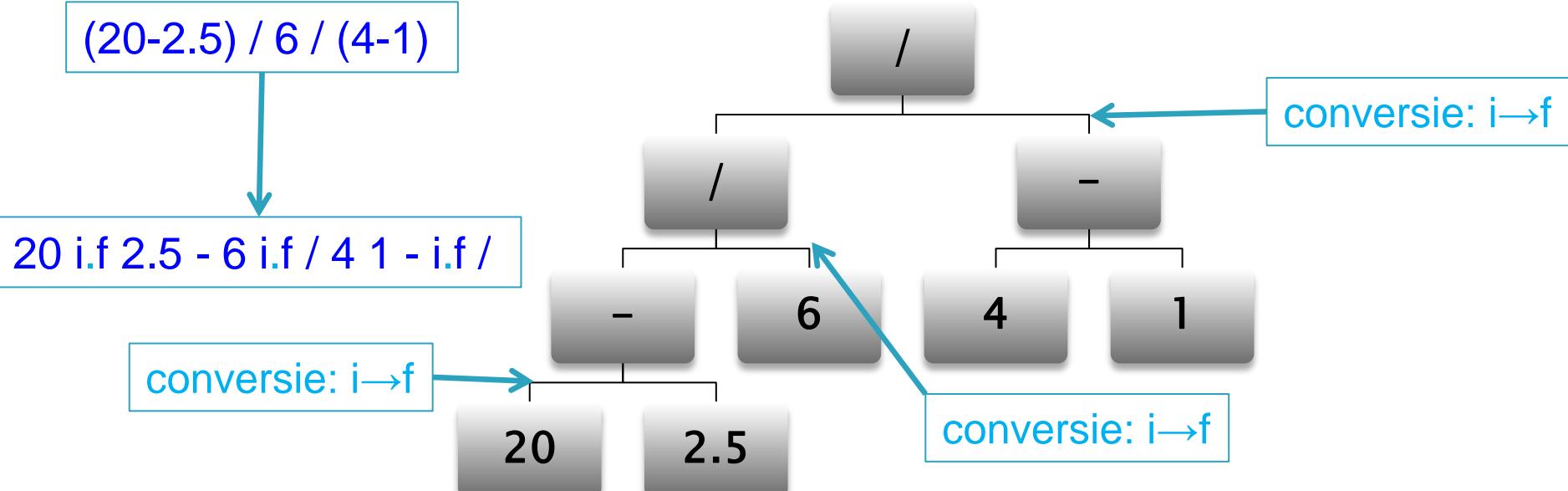
Construcția ASA pentru o expresie

(20-2.5) / 6 / (4-1)



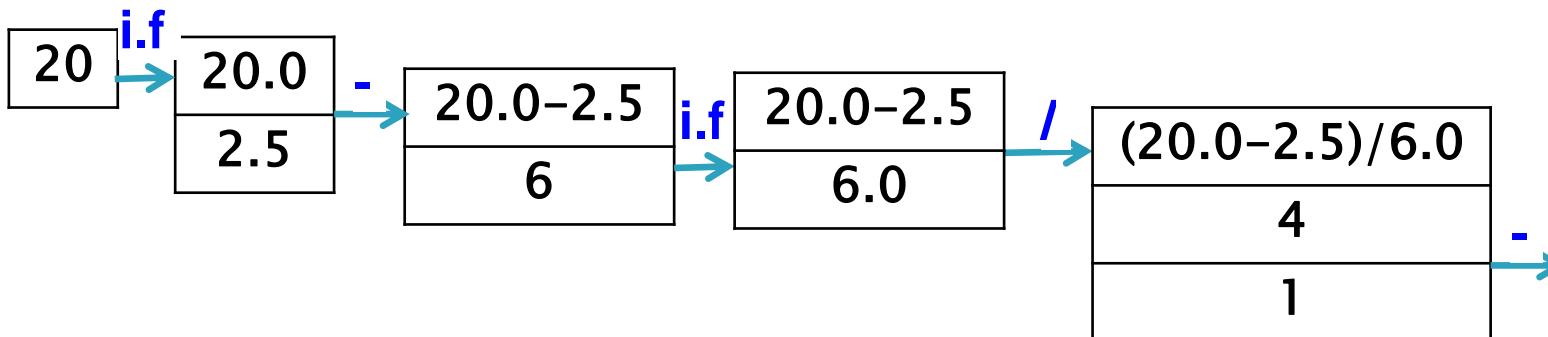
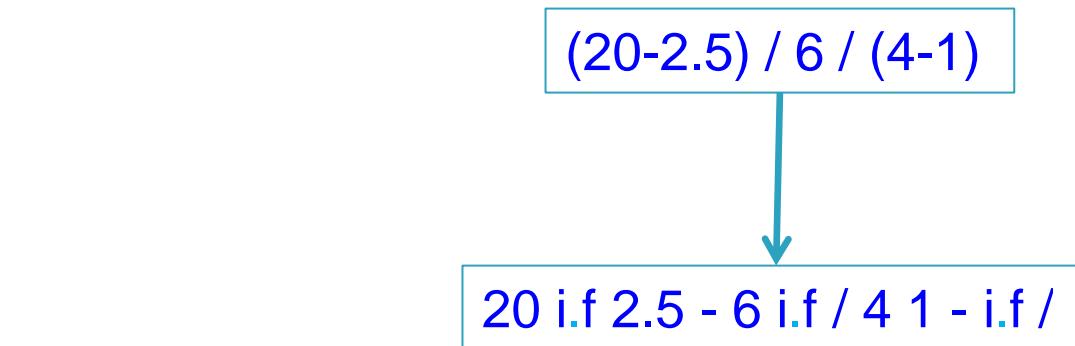
- Se consideră operația care se execută ultima și se pune ca nod curent
- Fii nodului curent vor fi subexpresiile stânga și dreapta
- Dacă subexpresiile sunt operanzi, aceștia vor deveni frunze
- Dacă subexpresiile sunt operatori, se repetă algoritmul până când se epuizează toate operațiile
- Unde este necesar, se inserează conversii de tipuri în arbore, astfel încât operațiile să aibă operanzi de același tip

Forma postfixată a unei expresii



- ▶ Forma postfixată (poloneză), se obține prin parcurgerea în postordine a ASA
- ▶ O proprietate importantă a acestei forme este faptul că folosind o stivă, se poate evalua expresia de la stânga la dreapta, după următorul algoritm:
 - ▶ dacă se întâlnește un operand, se depune în stivă
 - ▶ dacă se întâlnește un operator, se extrag din stivă operanții necesari, se face operația și rezultatul se depune înapoi în stivă

Evaluarea expresiei folosind o stivă



► vârful stivei este în partea de jos

Codul pentru expresii

(20-2.5) / 6 / (4-1)

20 i.f 2.5 - 6 i.f / 4 1 - i.f /

PUSH.i 20
CONV.i.f
PUSH.f 2.5
SUB.f
PUSH.i 6
CONV.i.f
DIV.f
PUSH.i 4
PUSH.i 1
SUB.i
CONV.i.f
DIV.f
SHOW.f

- ▶ **PUSH.*tip* ct.*tip*** – depune pe stivă constanta cu tipul dat

Algoritm generare de cod pentru expresii

- ▶ Fiecare regulă gramaticală care generează o valoare (expr, factor, ...):
 - ▶ va avea un atribut sintetizat "tip", care returnează tipul valorii generate (deoarece întotdeauna rezultatul expresiilor va fi pe stivă, nu este nevoie să se specifice și locația sa)
 - ▶ își va genera codul corespunzător atomilor consumați, astfel încât la terminarea regulii codul va fi deja generat
- ▶ **tipRezultat=combine(tip1,tip2)** – returnează tipul necesar pentru un operator care are operanze de tipurile specificate (ex: int*float → float)
- ▶ **idx=posInstr()** – returnează indexul la care va fi generată următoarea instrucțiune

Generare de cod pentru expresii

```
enum Tip = { TipInt, TipFloat }
```

```
calculator = ( expr[t] {addShow(t);} ) * FINISH
```

```
expr[out tip:Tip] = expr[t1] {idx=posInstr();}
```

```
( ADD termen[t2] /*1*/
```

```
| SUB termen[t2] /*2*/ ) | termen[tip]
```

```
termen[out tip:Tip] = termen[t1] {idx=posInstr();}
```

```
( MUL factor[t2] /*3*/
```

```
| DIV factor[t2] /*4*/ ) | factor[tip]
```

```
factor[out tip:Tip] = INT[n] {addPush(TipInt,n.i);tip=TipInt;}
```

```
| FLOAT[n] {addPush(TipFloat,n.f);tip=TipFloat;}
```

```
| LPAR expr[tip] RPAR
```

```
// 1-ADD, 2-SUB, 3-MUL, 4-DIV  
tip=combine(t1,t2);  
setConversie(idx,t1,tip);  
setConversie(posInstr(),t2,tip);  
addDiv(tip); // 1, 2, 3, 4
```

- ▶ **setConversie(idx,tipSrc,tipDst)** – dacă tipSrc!=tipDst, inserează la poziția idx conversia necesară (tipSrc→tipDst)

Codul pentru variabile și atribuiră

```
int n;  
float x;  
x=n*2.5;
```

```
LOAD.i n  
CONV.i.f  
PUSH.f 2.5  
MUL.f  
STORE.f x
```

- ▶ **LOAD.***tip addr* – ia valoarea de la adresa de memorie specificată și o depune în stivă
- ▶ **STORE.***tip addr* – ia valoarea din vârful stivei și o depune la adresa de memorie specificată

Codul pentru instrucțiunea if

```
if(expr){  
    // instructiuniTrue  
}
```

```
    // cod expresie  
    JF L1  
    // cod instructiuniTrue
```

L1:

```
if(expr){  
    // instructiuniTrue  
}else{  
    // instructiuniFalse  
}
```

```
    // cod expresie  
    JF L1  
    // cod instructiuniTrue  
    JMP L2  
L1:    // cod instructiuniFalse  
L2:
```

- ▶ JT *eticheta* – ia valoarea de pe stivă și dacă e true sare la eticheta specificată (*Jump if True*)
- ▶ JF *eticheta* – ia valoarea de pe stivă și dacă e false sare la eticheta specificată (*Jump if False*)
- ▶ JMP *eticheta* – salt necondiționat la eticheta specificată
- ▶ JT și JF folosesc doar valori de tip int, deci rezultatul *expr* trebuie să fie de tip int

Generare cod pentru instrucțiunea if

```
instrIf = IF LPAR expr[tip] RPAR {  
    addConv(tip, TipInt);  
    idxJF=posInstr();  
    addJF(0);          // se va seta cu indexul de la else sau de dupa if  
}  
  
block  
( ELSE {  
    idxJmp=posInstr();  
    addJmp(0);          // se va seta cu indexul de dupa if  
    setJIdx(idxJF,posInstr());  
}  
  
block {setJIdx(idxJmp,posInstr());}  
| ε {setJIdx(idxJF,posInstr());}  
})
```

// cod expresie
JF L1
// cod instructiuniTrue
JMP L2
L1: // cod instructiuniFalse
L2:

Codul pentru instrucțiunea while

```
while(expr){  
    // instructiuni  
}
```

```
int n;  
while(n>0){  
    n=n-1;  
}
```

```
L1:    // cod expresie  
        JF L2  
        // cod instructiuni  
        JMP L1  
L2:
```

```
L1:    LOAD.i    n  
        PUSH.i    0  
        GT.i  
        JF L2  
        LOAD.i    n  
        PUSH.i    1  
        SUB.i  
        STORE.i   n  
        JMP L1  
L2:
```

Apelurile de funcții

- ▶ O funcție poate avea simultan mai multe apelări ale ei în memorie
- ▶ Exemplu: `int fact(int n){return n<3 ? n : n*fact(n-1);}`
- ▶ Dacă apelăm *fact(4)*, în interiorul funcției trebuie să se calculeze prima oară *fact(3)*, deci se va face acest apel. La fel, pentru calculul lui *fact(3)*, trebuie calculat *fact(2)*. Rezultă că vor fi simultan în memorie apelurile pentru *fact(4)*, *fact(3)* și *fact(2)*.
- ▶ Deoarece pot exista simultan mai multe apeluri, fiecare cu valori distincte pentru argumente și variabile locale, nu este suficientă o singură locație pentru stocarea acestora, ca în cazul variabilelor globale
- ▶ Din acest motiv, valorile locale unei funcții (argumente și variabile locale) se implementează folosind stiva și un index special în stivă, numit **FP** (frame pointer)
- ▶ **Cadrul funcției** (function frame) – toate valorile locale, adresa de revenire și FP-ul funcției apelante
- ▶ Valorile din cadrul funcției se accesează folosind FP

Cadrul unei funcții

```
int min(int x, int y){  
    int r;  
    if(x<y)r=x;  
    else r=y;  
    return r;  
}
```

```
// g=min(k,108);  
    LOAD.i k  
    PUSH.i 108  
    CALL min  
ret_min: STORE.i g
```

Index față de FP	Valoare	Observații
-3	x	primul arg (k)
-2	y	al doilea arg (108)
-1	ret_addr	adresa de revenire (ret_min)
0	old_FP	vechiul FP
1	r	var locală

← FP ← SP

- ▶ **CALL *nume*** – apelează funcția cu numele dat
- ▶ La revenirea din funcție, execuția va continua cu instrucțiunea de după CALL
- ▶ Se respectă convenția de la operatori: toate argumentele trebuie puse pe stivă înainte de CALL, iar funcția le va înlocui pe stivă cu valoarea returnată:

$$\text{arg}_1 \dots \text{arg}_N \rightarrow f(\text{arg}_1 \dots \text{arg}_N)$$

Etapele unui apel de funcție

- ▶ Se depun toate argumentele pe stivă
- ▶ **Se apelează funcția folosind CALL:**
 - ▶ Depune pe stivă adresa de revenire (adresa următoarei instrucțiuni de după CALL)
 - ▶ Face JMP la codul funcției
- ▶ **La intrarea în funcție:**
 - ▶ Se salvează vechiul FP în stivă, apoi în FP se va pune indexul vârfului stivei
 - ▶ Se adună la vârful stivei numărul de variabile locale, pentru a se face loc pentru ele după vechiul FP
- ▶ **Revenirea din funcție se face cu instrucțiunea RET:**
 - ▶ Reface FP la valoarea anterioară apelului
 - ▶ Șterge din stivă tot cadrul funcției
 - ▶ În locul cadrului pune valoarea care era în vârful stivei. Aceasta va deveni valoarea returnată de funcție

Codul funcției

```
int min(int x, int y){  
    int r;  
    if(x<y)r=x;  
    else r=y;  
    return r;  
}
```

FP	Valoare
-3	x
-2	y
-1	ret_addr
0	old_FP
1	r

// funcția min

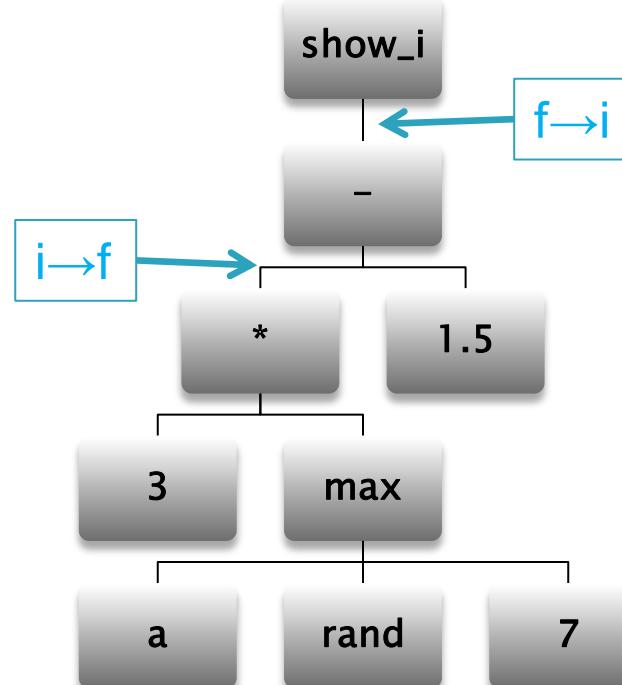
```
ENTER      1          // nr var locale  
FPOLOAD.i -3          // PUSH.i FP[-3]  
FPOLOAD.i -2  
LESS.i  
JF          L1  
FPOLOAD.i -3  
FPSTORE.i 1          // POP.i FP[-3]  
JMP L2  
L1:        FPOLOAD.i -2  
FPSTORE.i 1  
L2:        FPOLOAD.i 1  
RET         2          // nr argumente
```

- ▶ **ENTER *nr_var_locale*** – salvează vechiul FP în stivă; SP→FP; SP+=nr_var_locale
- ▶ **RET *nr_argumete*** – reface vechiul FP; șterge cadrul funcției; pune în vârful stivei de după ștergerea cadrului valoarea din vârful stivei de dinainte de ștergere; continuă execuția cu instrucțiunea de la adresa de revenire

Integrarea apelurilor de funcții în expresii

```
int a;  
int max(int x, int y , int z);  
void show_i(int x);  
int rand();  
  
show_i(3*max(a,rand(),7)-1.5);
```

```
PUSH.i 3  
LOAD.i a  
CALL rand  
PUSH.i 7  
CALL max  
MUL.i  
CONV.i.f  
PUSH.f 1.5  
SUB.f  
CONV.f.i  
CALL show_i
```



3 a rand 7 max * i.f 1.5 – f.i show_i

- ▶ Funcțiile reprezintă o extindere a noțiunii de operator, acționând asupra a n argumente (**aritate n**)
- ▶ La apelul funcției se depun pe stivă argumentele, de la stânga la dreapta, iar apoi se realizează apelul

Implementare MV stivă – structuri de date

```
typedef union _Val{  
    int i;           // nr int, index in instructiuni pentru CALL, RET  
    float f;        // nr float  
    struct _Val *addr; // pentru salvarea adreselor in stiva  
}Val;
```

```
typedef enum{SUB_F, LOAD_I, STORE_I, JF, CALL, ENTER, RET} Code;  
typedef struct{  
    Code code;  
    union{  
        Val v;  
        Val *addr; // pentru LOAD, STORE  
        int idx;   // pentru JF, JT, FPLOAD, FPSTORE, CALL, RET, ENTER  
        }arg;  
    }Instr;
```

```
Val stack[]; // stiva  
Val *SP;      // Stack Pointer - pointer la vârful stivei  
Val *FP;      // Frame Pointer - pointer la locația unde s-a salvat vechiul FP  
Instr instructions[]; // vectorul de instrucțiuni  
Instr *IP;     // Instruction Pointer - pointer la instrucțiunea curentă
```

Implementare MV stivă – cod

```
switch(IP->code){  
    case SUB_F:f2=popf();f1=popf();pushf(f1-f2);IP++;break;  
    case LOAD_I:pushi(*IP->arg.addr);IP++;break;  
    case STORE_I:*IP->arg.addr=popi();IP++;break;  
    case JF:if(popi())IP++;else IP=instructions+IP->arg.idx;break;  
    case CALL:  
        pushi(IP-instructions+1);  
        IP=instructions+IP->arg.idx;  
        break;  
    case ENTER:  
        pusha(FP);      // pusha - pune pe stiva o adresa  
        FP=SP;  
        SP+=IP->arg.idx;  
        IP++;  
        break;  
    case RET:  
        v=popv();      // popv - ia de pe stivă o celulă (Val)  
        SP=FP - IP->arg.idx - 2; // -(args, old_FP, ret_addr)  
        IP=instructions+FP[-1].i;  
        FP=FP[0].addr; pushv(v);  
        break;
```

```
typedef struct{  
    int i;  
    float f;  
    struct _Val *addr;  
}Val;  
  
typedef struct{  
    Cod cod;  
    union{  
        Val v;  
        Val *addr;  
        int idx;  
    }arg;  
}Instr;
```

Limbaje formale și tehnici de compilare

Curs 11: optimizare

Obiectivele optimizării

- ▶ Optimizarea urmărește transformarea unui program, astfel încât programul rezultat:
 - Să performeze mai bine conform anumitor criterii
 - Să fie semantic echivalent cu cel original
- ▶ Criterii de optimizare:
 - Viteză mai mare de execuție
 - Consum mai mic de memorie
 - Comportament mai bun în anumite configurații sau arhitecturi (ex: un program poate fi optimizat pentru a folosi GPU)

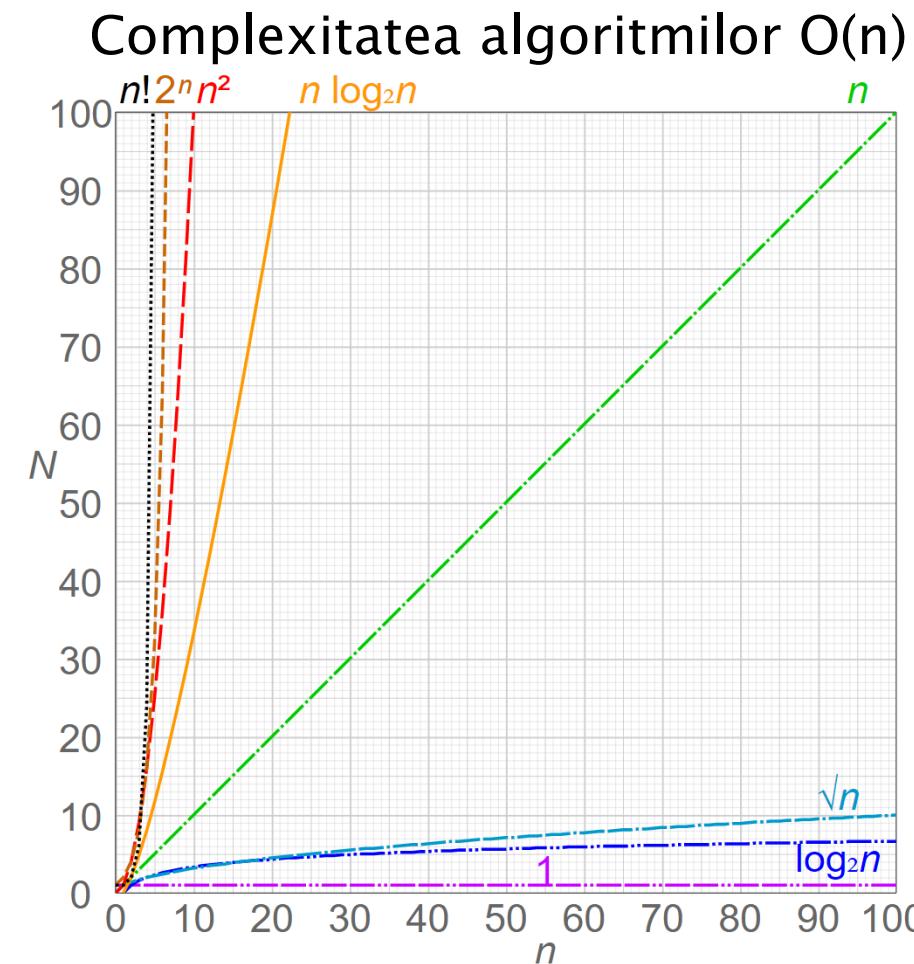
Echilibrarea diverselor optimizări

- ▶ Anumite optimizări ale unui criteriu pot duce la degradarea performanțelor conform altui criteriu
- ▶ Din acest motiv, compilatoarele au mai multe opțiuni de optimizare, astfel încât programatorul să le aleagă pe cele mai potrivite
- ▶ În general optimizările pentru spațiu se aplică în situații care memoria este limitată și codul nu ar încăpea: microcontrollere (ex: Arduino) sau programe stocate în memoria ROM (ex: BIOS-ul)
- ▶ În celealte cazuri se preferă optimizările pentru viteză
- ▶ Se poate testa un program în ambele situații, pentru a se decide cea mai potrivită

Ce nu este optimizarea?

- ▶ Optimizarea nu duce la îmbunătățirea unui algoritm, ci doar la eficientizarea execuției sale
- ▶ Exemplu: prin optimizare viteza crește de 5 ori

Tip algoritm	n=10	n=100
Constant O(1)	1	1
Logaritmic O($\log_2(n)$)	3.3	6.6
Linear O(n)	10	100
Linearitic O($n \cdot \log_2(n)$)	33.2	664
Pătratic O(n^2)	100	10000
Exponențial O(2^n)	1024	10^{30}



Includerea optimizărilor în compilator

- ▶ Optimizările pot fi relativ ușor de implementat în anumite faze ale unui compilator, dar foarte greu de implementat în altele
- ▶ Din acest motiv, fiecărei faze de compilare i se atașează optimizări specifice. Pot fi optimizări pentru:
 - Codul sursă (transformări în ASA)
 - Generarea de cod intermedian
 - Codul intermedian (CI)
 - Generarea de cod mașină
 - Codul mașină
- ▶ Unele optimizări se aplică după ce programul a fost executat de mai multe ori, pentru a se colecta date statistice despre cazurile cele mai des întâlnite

Optimizări la nivel de cod sursă

- ▶ Sunt optimizările care țin cont de concepte de nivel înalt ale LP, cunoșterea bibliotecii standard sau operează pe structuri de date mai complexe
- ▶ Din acest motiv, este greu să fie aplicate în fazele următoare, deoarece o fază succesoare renunță la informațiile nerelevante din fazele precedente
- ▶ De obicei se aplică în AST
- ▶ Exemple:
 - Transformarea unor construcții de limbaj sau expresii
 - Optimizări rezultate din inferență de tipuri
 - Concretizarea apelurilor polimorfice

Transformarea unor construcții

- ▶ Dacă se cunosc funcționalitățile pe care LP și biblioteca sa standard le oferă, devine posibil să se înlocuiască unele construcții prin altele, mai eficiente

```
Angajat a=new Angajat("Ion", 3200);           // Java
String text="nume: "+a.nume+, salariu: "+a.salariu;
// poate deveni
String s1=a.nume;
int len1=s1.length();
String s2=a.salariu.toString();
int len2=s2.length();
String text=new StringBuilder(17+len1+len2)
.append("nume: ").append(s1)
.append(", salariu: ").append(s2)
.toString();
```

Optimizări rezultate din inferență de tipuri

- ▶ În special pentru limbajele dinamice, dacă se poate infera tipul unei variabile, atunci se pot folosi în mod direct operațiile acelui tip, în loc să se testeze de fiecare dată ce tip are valoarea ei

```
require('console') //JavaScript, Node.js
var i,n=7
for(i=0;i<n;i++){
    console.log(i)
}
```

- ▶ Se inferă că **i** și **n** sunt de tip numeric și atunci se cunoaște exact cum se execută operațiile cu ele (ex: `<`, `++`)
- ▶ Altfel, ar fi trebuit înainte de fiecare operărie să se testeze tipurile operanzilor, iar apoi să se execute operația corespunzătoare tipului rezultat din testare

Concretizarea apelurilor polimorfice

- ▶ Metodele polimorfice au implementări diferite în clase diferite. Trebuie să existe un mecanism de selecție a metodei corespunzătoare obiectului curent

```
abstract class Figura{    //Cerc,Triunghi,Dreptunghi, ...
    public abstract double arie();
}

class Cerc extends Figura{    //implementare concretă
    double r;
    @Override public double arie() {
        return Math.PI*r*r;
    }
}
Figura fig=new Cerc(1);
double a=fig.arie(); //ce implementare trebuie apelată?
```

- ▶ Din ultima expresie atribuită lui **fig**, se inferă faptul că figura este de fapt un **Cerc**, astfel încât se apelează direct **Cerc.arie**, fără să mai fie necesar mecanismul de selecție

Codul intermediu (CI)

- ▶ În general, pentru optimizări avansate, se folosește o reprezentare specifică a programului, numită CI, bazată pe instrucțiuni simple, care fiecare îndeplinește o singură operație
- ▶ Astfel se elimină aspecte gen ordinea operațiilor sau analiza tuturor operațiilor care alcătuiesc o instrucțiune complexă
- ▶ Un exemplu de asemenea CI este cel folosit de LLVM, care se bazează pe o reprezentare SSA (Single–Static Assignment) a codului, în care fiecărei variabile temporare (registru) îi este atribuită o singură valoare, la inițializarea ei

Folosirea CI

CI permite unui compilator să utilizeze același optimizor de cod și back-end-uri pentru oricâte front-end-uri



```
for(int i=0;i<n;i++)  
    printf("%d",i*2);
```

```
int i=0  
L1: int T1=i<n  
     if not T1 goto L2  
     int T2=i*2  
     printf "%d",T2  
     i=i+1  
     goto L1
```

L2:

Exemplu de CI

- ▶ Toate constantele numerice fără zecimale sunt întregi, iar cele cu zecimale sunt reale
- ▶ Variabilele auxiliare se notează cu T_{index} și li se specifică tipul
- ▶ O instrucție poate eventual atribui rezultatul unei destinații

```
int min(int x, int y){  
    int r;  
    if(x<y)r=x;  
    else r=y;  
    return r;  
}
```



```
int min(int x, int y){  
    int T1=x<y  
    if not T1 goto L1  
    r=x  
    goto L2  
L1: r=y  
L2: return r  
}
```

Optimizări la generarea CI

- ▶ Aceeași instrucțiune sursă poate genera CI diferit, în funcție de felul specific în care e folosită
- ▶ De exemplu, la **switch** se poate ține cont de numărul de **case-uri**, dacă valorile sunt consecutive, etc

```
int x;
switch(x){
    case 1:...
    case 2:...
    case 3:...
    case 4:...
    case 5:...
    default:
}
```



```
labels etichete={L1, L2, L3, L4, L5}
int T1=x<1
if T1 goto L_default
T1=x>5
if T1 goto L_default
T1=x-1
goto etichete[T1]
L1: ...
L2: ...
L3: ...
L4: ...
L5: ...
L_default: ...
```

Optimizări la nivel de CI

- ▶ La acest nivel se pot aplica toate optimizările care sunt independente atât de LP cât și de arhitectura destinație:
- ▶ Eliminarea operațiilor fără efect
- ▶ Propagarea valorilor constante
- ▶ Folosirea unor operații mai simple
- ▶ Eliminarea subexpresiilor comune
- ▶ Optimizarea salturilor redundante
- ▶ Înlocuirea apelurilor de funcții cu codul funcțiilor

Eliminarea operațiilor fără efect

- ▶ Se elimină: $x+0$, $x-0$, $x*1$, $x/1$, $x=x$
- ▶ $x==x \rightarrow 1$, $x!=x \rightarrow 0$
- ▶ $\&^p$ (adresa unde se află valoarea pointată)
 $\rightarrow p$
- ▶ $*\&x$ (valoarea de la adresa variabilei) $\rightarrow x$
- ▶ Eliminarea atribuirilor fără efecte colaterale,
a căror destinație nu mai este folosită
ulterior: $x=y/2;x=1; \rightarrow x=1$

Propagarea valorilor constante

```
int DEBUG=0;  
float PI=3.14159;  
float a=2*PI*r;  
if(DEBUG){  
    printf("%g\n",a);  
}
```

```
int DEBUG=0  
float PI=3.14159  
float T1=2*PI  
float a=T1*r  
if not DEBUG goto L1  
printf "%g\n",a
```

L1:

```
a=6.28318*r  
if not 0 goto L1  
printf "%g\n",a
```

L1:

```
a=6.28318*r
```

Folosirea unor operații mai simple

- ▶ Pe arhitecturi mai simple (ex: microcontrollere), operații precum înmulțire sau împărțire pot să ia sute de tacturi pentru execuție sau pot să nu fie implementate deloc, caz în care iau chiar mai mult timp
- ▶ Din acest motiv, când se poate, se preferă implementarea lor prin operații mai simple

```
int b=a*2;  
int c=a/512; // 512=29
```



```
int b=a<<1; // sau a+a  
int c=a>>9;
```

Eliminarea subexpresiilor comune

```
float len(float x1,float y1,float x2,float y2){  
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));  
}
```

```
float T1=x2-x1  
float T2=x2-x1  
float T3=T1*T1  
float T4=y2-y1  
float T5=y2-y1  
float T6=T4*T5  
float T7=T3*T6  
float T8=sqrt T7  
return T8
```



```
float T1=x2-x1  
float T3=T1*T1  
float T4=y2-y1  
float T6=T4*T4  
float T7=T3*T6  
float T8=sqrt T7  
return T8
```

Optimizarea salturilor redundante

```
while(a>b){  
    if(a<10)a=a-1;  
    else a=a/2;  
}
```

```
L1: int T1=a>b  
    if not T1 goto L4  
    int T2=a<10  
    if not T2 goto L2  
    a=a-1  
    goto L3  
L2: a=a/2  
L3: goto L1  
L4:
```

```
L1: int T1=a>b  
    if not T1 goto L4  
    int T2=a<10  
    if not T2 goto L2  
    a=a-1  
    goto L1  
L2: a=a/2  
L3: goto L1  
L4:
```

Înlocuirea apelurilor de funcții cu codul lor

- ▶ Pentru funcțiile care constă doar în câteva instrucțiuni sau cele care se apelează dintr-un singur loc din program, se poate înlocui apelul lor cu codul funcției
- ▶ Nu mai sunt necesare crearea/ștergerea cadrului funcției și apelul, inclusiv transmiterea argumentelor
- ▶ Apar oportunități și pentru alte optimizări
- ▶ Corpul funcției se copiază la locul apelului și:
 - ▶ Parametrilor li se dau valorile de la apel
 - ▶ Pentru rezultat se introduce o variabilă temporară care va înlocui apelul funcției

```
int min(int x, int y){  
    if(x<y) return x;  
    else return y;  
}  
printf("%d\n",min(1,3));
```

```
int r;  
x=1;  
y=3;  
if(x<y)r=x;  
else r=y;  
printf("%d\n",r);
```

```
printf("%d\n",1);
```

Optimizări la generarea codului mașină

- ▶ Poate cel mai important aspect la generarea codului mașină este optimizarea folosirii regiștrilor CPU
- ▶ Folosirea memoriei poate fi de zeci de ori mai lentă decât cea a regiștrilor
- ▶ Unele instrucțiuni CPU necesită unul sau mai mulți operanzi în regiștri, deci valorile oricum trebuie aduse din memorie în regiștri

Generare de cod mașină

// fără optimizare registri

```
L1:    MOV EAX,[ESP-4] // x
        CMP EAX,[ESP-8] // y
        JE L4
        MOV EAX,[ESP-4]
        CMP EAX,[ESP-8]
        JLE L2
        MOV EAX,[ESP-4]
        SUB EAX,[ESP-8]
        MOV [ESP-4],EAX
        JMP L3
L2:    MOV EAX,[ESP-8]
        SUB EAX,[ESP-4]
        MOV [ESP-8],EAX
L3:    JMP L1
L4:    MOV EAX,[ESP-4]
        RET 8
```

```
int cmmdc(int x,int y){
    while(x!=y){
        if(x>y)x=x-y;
        else y=y-x;
    }
    return x;
}
```

// cu optimizare registri

```
L1:    MOV EAX,[ESP-4] // x
        MOV ECX,[ESP-8] // y
        CMP EAX,ECX
        JE L4
        CMP EAX,ECX
        JLE L2
        SUB EAX,ECX
        JMP L3
L2:    SUB ECX,EAX
        JMP L1
L3:    RET 8
```

Optimizarea codului mașină

- ▶ Se optimizează codul mașină generat din CI
- ▶ Exemple de optimizări:
 - Înlocuirea unei secvențe de instrucțiuni mai simple cu o instrucțiune mai complexă
 - Refolosirea valorilor care rămân constante după o operație sau care apar ca rezultate auxiliare

Înlocuirea secvențelor de instrucțiuni simple cu o instrucțiune mai complexă

```
// int i,v[100];  
// v[i]=-1;
```

MOV EAX,[i]	// EAX=i
SHL EAX,2	// EAX=i*4
ADD EAX,v	// EAX=v+i*4
MOV DWORD[EAX],-1	// *(v+i*4)=-1



MOV EAX,[i]	// EAX=i
MOV DWORD[v+EAX*4],-1	// *(v+i*4)=-1

Refolosirea valorilor constante după operații

```
MOV EAX,[ESP-4] // x  
MOV ECX,[ESP-8] // y  
L1:  CMP EAX,ECX  
      JE L4  
      CMP EAX,ECX  
      JLE L2  
      SUB EAX,ECX  
      JMP L3  
L2:  SUB ECX,EAX  
L3:  JMP L1  
L4:  RET 8
```

```
MOV EAX,[ESP-4] // x  
MOV ECX,[ESP-8] // y  
L1:  CMP EAX,ECX  
      JE L4  
      JLE L2  
      SUB EAX,ECX  
      JMP L3  
L2:  SUB ECX,EAX  
L3:  JMP L1  
L4:  RET 8
```

- ▶ Registrul EFLAGS conține indicatori pentru: egalitate, mai mic, mai mare, negativ, etc.
- ▶ Instrucțiunile aritmetice și de comparație (SUB, CMP) setează registrul EFLAGS conform rezultatului lor
- ▶ EFLAGS rămâne neschimbat până la următoarea instrucțiune care-l modifică
- ▶ Instrucțiunile de salt condiționat (JE, JLE) testează indicatorii din EFLAGS

Optimizări care folosesc date de la execuția programului

- ▶ Aceste optimizări folosesc date statistice colectate de la execuția codului, pentru a favoriza situațiile cele mai des întâlnite

```
switch(x){          // probabilități pentru x, determinate prin execuții multiple
    case 7:...break;    // 40%
    case 100:...break;   // 3%
    case 5000:...break;  // 50%
    default:...
}
```



```
switch(x){
    case 5000:...break;
    case 7:...break;
    case 100:...break;
    default:...
}
```

Limbaje formale și tehnici de compilare

Curs 12: implementarea unor facilități avansate:
metode polimorfice, managementul memoriei

Facilități avansate ale LP

- ▶ Aceste facilități automatizează crearea unui cod destul de complex, care altfel ar fi trebuit implementat manual
- ▶ (1) Facilități avansate care se implementează prin generare de cod specific:
 - Metode polimorfice
 - Funcții cu context (closures)
 - Excepții
- ▶ (2) Facilități avansate care necesită o bibliotecă mai complexă de funcții care să o implementeze:
 - Managementul automat al memoriei
 - Încărcare dinamică de module
 - Metaprogramare și accesul la funcțiile compilatorului

Metode polimorfice (MP)

- ▶ Metodele polimorfice au implementări diferite în clase diferite
 - ▶ Există un mecanism de selecție a implementării corespunzătoare obiectului pentru care se apelează metoda
 - ▶ Ele se folosesc pentru implementarea colecțiilor eterogene
 - ▶ Exemple: figuri în programe CAD, entități într-un joc, produse cu structură diferită într-un inventar, etc

```
abstract class Figura{                                // Cerc,Triunghi,Dreptunghi,...  
    public abstract double arie();  
}  
class Cerc extends Figura{                          // implementare concretă  
    double r;  
    @Override public double arie() {  
        return Math.PI*r*r;  
    }  
}  
Figura []figuri;          // figuri de tipuri diferite  
figuri[i].arie();         // ce implementare trebuie apelată?
```

Implementarea clasei abstracte

- ▶ Orice clasă care are MP va avea un membru ascuns, denumit "tabelă de selecție" (*dispatch table* sau *virtual table*) – vTable
- ▶ vTable este un pointer la o structură (VTable) ce conține pointeri la funcții
- ▶ Fiecare pointer din VTable corespunde unei MP

```
struct VTable{                                // C
    double (*arie)(Figura *fig);           // fig ⇔ this
    double (*perimetru)(Figura *fig);
}VTable;
```

```
struct Figura{
    VTable *vTable;
};
```

Implementarea claselor concrete

- ▶ O clasă concretă va avea toate câmpurile clasei de bază și propriile sale implementări ale MP

```
struct Cerc{ // C
    VTable *vTable; // moștenită din Figura
    double r;
};

double Cerc_arie(Figura *fig){ // fig ⇔ this
    double r=((Cerc*)fig)->r;
    return PI*r*r;
}

VTable Cerc_vTable={Cerc_arie, Cerc_perimetru};
```

Instanțierea claselor concrete

- ▶ La crearea unei instanțe (new), se setează vTable-ul instanței cu tabela ce cuprinde implementările MP pentru clasa respectivă

```
// Java: Cerc c=new Cerc();
```

```
Cerc *Cerc_new(){ // C
    Cerc *c=(Cerc*)malloc(sizeof(Cerc));
    c->vTable=Cerc_vTable; // vTable pentru un Cerc va
    pointa la implementarile specifice cercului
    c->r=0;
    return c;
}
```

Apelul MP

- ▶ La apelul unei MP se selectează din vTable-ul obiectului curent pointerul la funcția corespunzătoare MP
- ▶ Se apelează funcția pointată dându-i-se ca prim argument obiectul curent (this), pentru ca funcția să aibă acces la obiect

```
// Java: fig.arie();  
Figura *fig;  
double a=fig->vTable->arie(fig); // C
```

Funcții cu context (closures)

- ▶ O funcție cu context este funcție creată în altă funcție și care păstrează (captează, încide (close)) variabilele locale necesare (contextul) din funcția părinte, chiar și după ce funcția părinte s-a încheiat
- ▶ Deoarece fiecare apel al funcției părinte creează alte variabile locale, contextele vor fi diferite

```
require('console') // JavaScript + Node.js
function salutare(expresie){
    return function(nume){
        return expresie+' '+nume+'!';
    }
}
const f1=salutare("Salut");
const f2=salutare("Bine ai venit");

console.log(f1("Ion")); // Salut Ion!
console.log(f2("Ana")); // Bine ai venit Ana!
```

Clasificare management memorie (MM)

- ▶ **Manual** – programatorul eliberează explicit memoria folosind funcții gen *free(ptr)*
- ▶ **Semiautomat** – LP furnizează construcții care se apelează automat la ieșirea dintr-un bloc și eliberează resursele obiectelor care ies din domeniul de existență. Exemple: *destructori*, blocuri *using*, *pointeri inteligenți* (smart pointers – *unique_ptr*, *shared_ptr*)
- ▶ **Automat** – LP testează când o zonă de memorie nu mai este folosită de program și o eliberează automat

MM automat

- ▶ Metode de implementare:
- ▶ **Numărarea referințelor** – (reference counting) fiecare zonă alocată are un contor care memorează câți pointeri pointează la ea. Dacă acest contor devine 0, zona este eliberată.
- ▶ **Trasare** – (tracing) alocatorul urmărește (trasează) folosirea zonelor de memorie, pornind de la cele despre care știe sigur că sunt folosite. Orice zonă la care nu poate ajunge prin trasare înseamnă că este inaccesibilă și va fi eliberată.

MM prin numărarea referințelor

- ▶ La alocarea unui obiect: nRefs=0
- ▶ La atribuiră (variabile, argumente, câmpuri): nRefs++
- ▶ Când un deținător dispare: nRefs--
- ▶ La eliberarea unui obiect, se decrementează referințele câmpurilor lui

```
function f(){  
    var o1=new Obiect();  
    var o2=o1;  
}
```

```
struct Obiect{  
    unsigned nRefs; // nr referinte  
    // ... alte campuri ...  
};
```

```
tmp=malloc(sizeof(Obiect)); // new Obiect();  
tmp->nRefs=0;  
o1=tmp; // var o1=tmp;  
o1->nRefs++;  
o2=o1; // var o2=o1;  
o2->nRefs++;  
if(--o1->nRefs==0)free(o1); // ieșirea din f  
if(--o2->nRefs==0)free(o2);
```

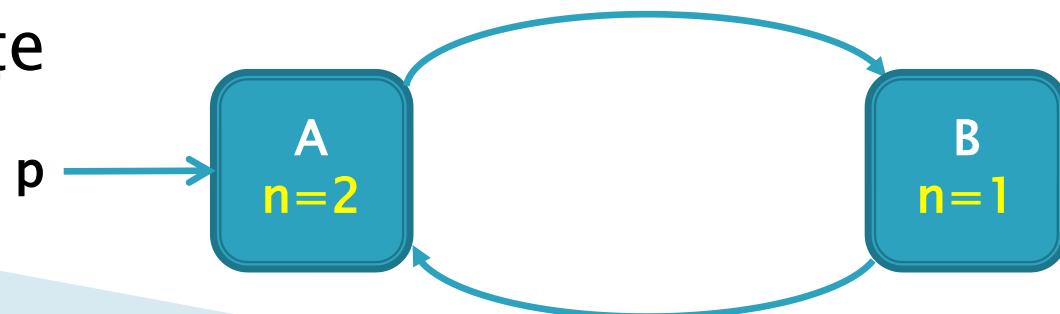
MM prin numărarea referințelor

► Avantaje:

- Simplu de implementat
- Resursele sunt eliberate în momentul în care nu mai sunt folosite

► Dezavantaje:

- Referințele ciclice (un obiect referă direct sau indirect la el însuși) nu pot fi eliberate
- Viteză mai mică a programului, din cauza operațiilor de incrementare/decrementare/testare
- Pentru multithreading, operațiile sunt mai complexe
- Necesită memorie suplimentară pentru stocarea numărului de referințe

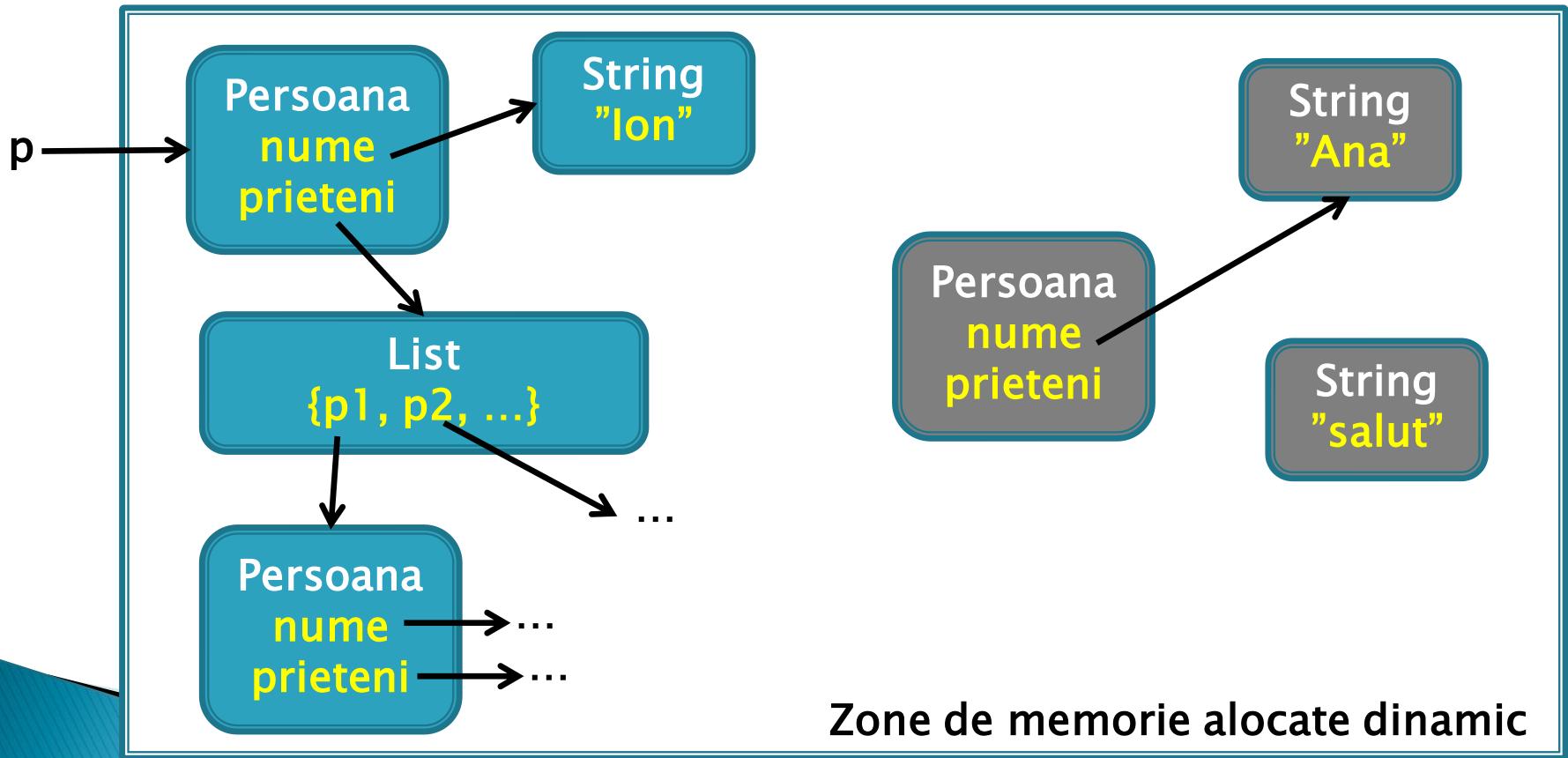


MM cu trasare (tracing) – algoritm

1. **Marcarea zonelor folosite (mark)** – se pleacă de la variabile (globale, locale) și argumentele funcțiilor în desfășurare și se marchează zonele pointate de ele ca fiind folosite.
Aceste puncte de pornire se numesc **rădăcini** (roots). Se consideră că doar acestea pointează la zone folosite în program (in use).
Se continuă recursiv cu câmpurile obiectelor marcate până când nu mai este nimic de marcat.
2. **Se parcurge toată memoria alocată și se eliberează zonele care nu au fost marcate anterior (sweep)**

MM cu trasare – exemplu

- ▶ Cu turcoaz sunt zonele care pot fi atinse (reachable) pornind de la variabile/argumente din program. Aceste zone sunt utile și vor fi menținute.
- ▶ Cu gri sunt zonele care nu pot fi atinse (unreachable). Aceste zone nu mai sunt utile și vor fi eliberate.



MM cu trasare – îmbunătățiri

- ▶ **Compactare** – (compacting) în timp, datorită alocărilor și eliberărilor de diverse dimensiuni, apar zone de memorie libere de dimensiuni din ce în ce mai mici între zone alocate. Aceste zone nu vor mai putea fi refolosite ulterior și duc la epuizarea memoriei. Prin compactare, se reorganizează memoria pentru a elimina aceste goluri.
- ▶ **Alocare pe generații** – (generational) pentru a nu se parcurge de fiecare dată toată memoria, alocările se organizează în zone distincte, după tiparul lor de folosire: alocare/eliberare **frecventă** sau **persistentă**. La reclamarea memoriei, prima oară se verifică doar zona frecventă și, doar dacă nu se recuperează suficientă memorie, se va verifica și zona persistentă.
- ▶ **Multithreading** – se folosesc mai multe fire de execuție pentru a se reclama memoria în paralel cu funcționarea programului propriu-zis.
- ▶ **Incremental** – reclamarea memoriei are loc în pași distinți (incrementi) în care se parcurg doar părți mici de memorie, pentru a nu apărea stopări prelungite ale programului

MM cu trasare

► Avantaje:

- Se eliberează și referințele ciclice
- Necessarul de memorie este mai mic decât la MM cu numărare de referințe, deoarece nu se mai stochează contorul
- Când nu are loc reclamarea memoriei, nu există operații suplimentare, ca la MM cu numărare de referințe

► Dezavantaje:

- Momentul când se face reclamarea memoriei nu este predictibil, deci nu se poate folosi pentru eliberarea resurselor critice (fișiere, porturi de rețea, ...) în momentul în care ele nu mai sunt folosite
- Reclamarea memoriei poate dura destul de mult timp (chiar pentru îmbunătățiri gen incremental + generational + multithreading), ceea ce nu este admisibil pentru aplicații cu limite stricte de timp