

Limbaje formale și tehnici de compilare

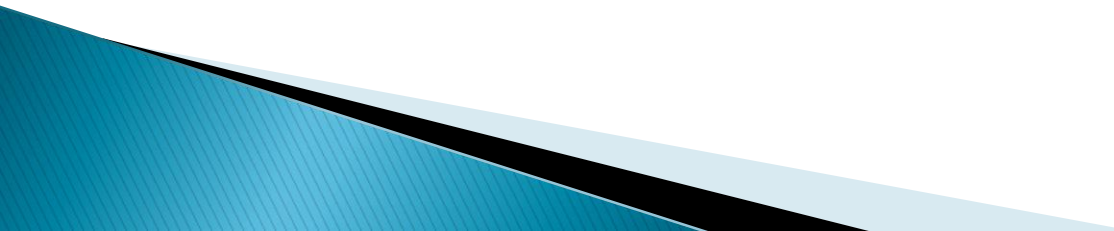
Curs 6: analizorul sintactic; conceperea
gramaticilor

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
```

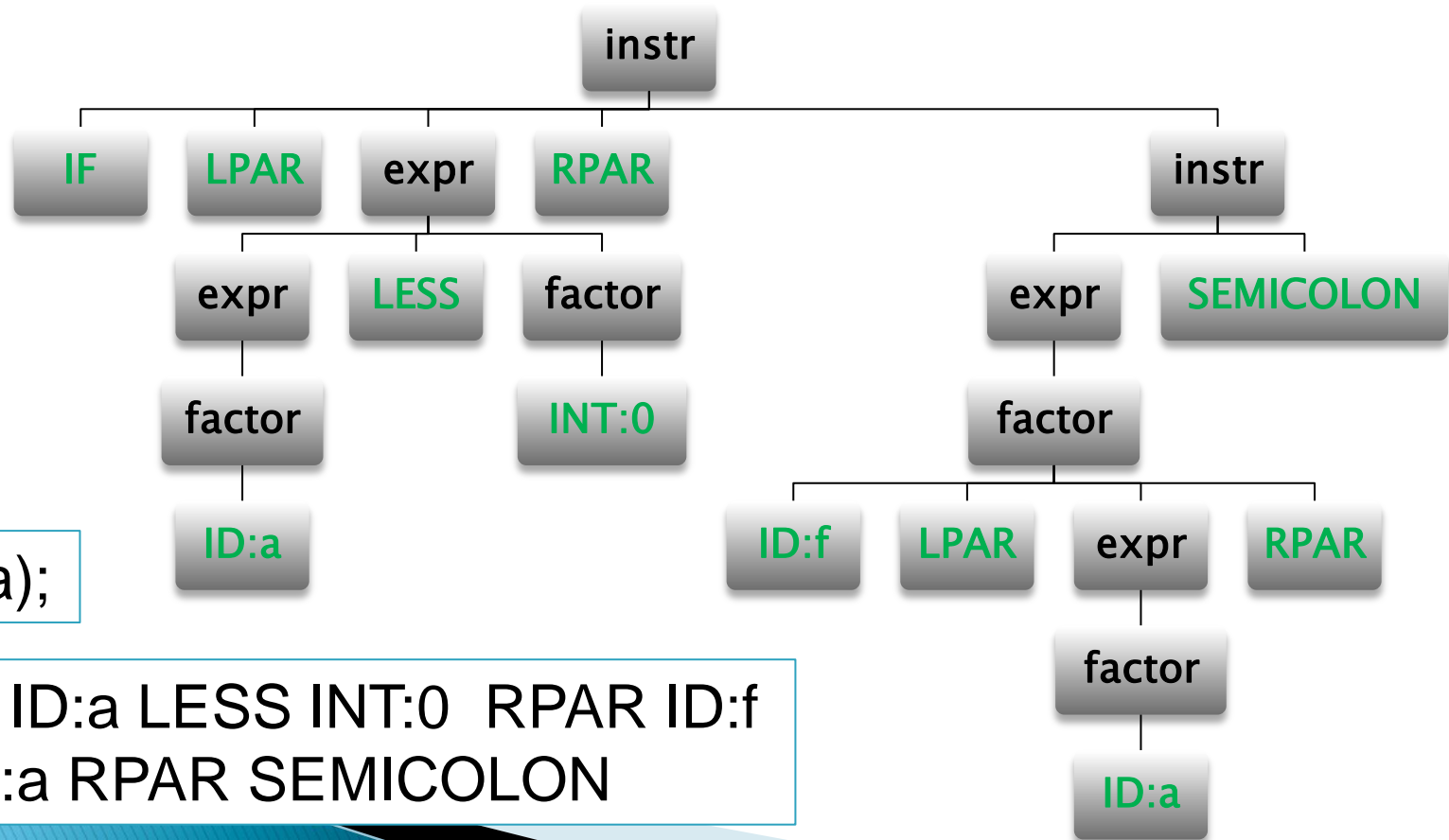
Gramaticile folosite pentru ANSIN

- ▶ 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)

$\text{instr} ::= \text{IF LPAR expr RPAR instr (ELSE instr)?} \mid \text{expr SEMICOLON}$
 $\text{expr} ::= \text{expr LESS factor} \mid \text{factor}$
 $\text{factor} ::= \text{ID (LPAR (expr (COMMA expr)?)? RPAR)?} \mid \text{INT}$



if(a<0)f(a);

IF LPAR ID:a LESS INT:0 RPAR ID:f
LPAR ID:a RPAR SEMICOLON

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 componenteii 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

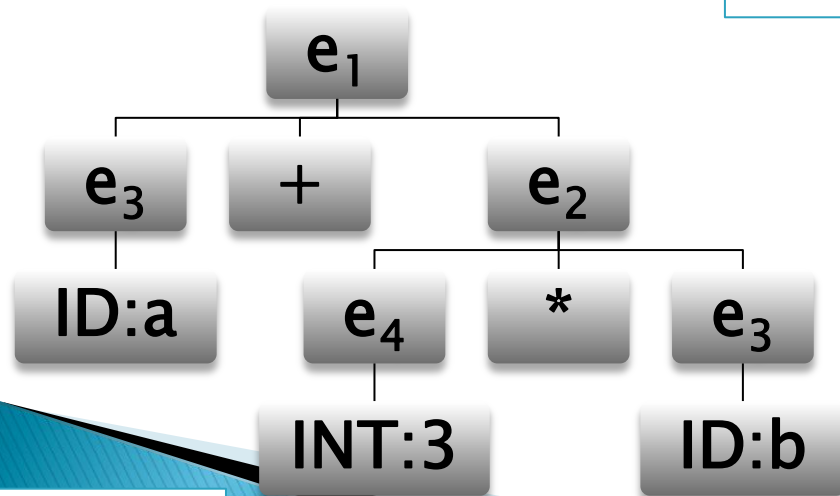
- ▶ **Precedența** – ordinea de evaluare a operatorilor. Operatorii cu precedența mai mare se vor evalua primii.
- ▶ Exemplu: în expresia $a+b/2$, împărțirea se va efectua prima (are precedența mai mare decât adunarea), deși în șirul de atomi lexicali adunarea este prima operație
- ▶ **Asociativitatea** – 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 evaluează de la stânga la dreapta
 - ▶ **dreaptă** – operatorii se evaluează 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

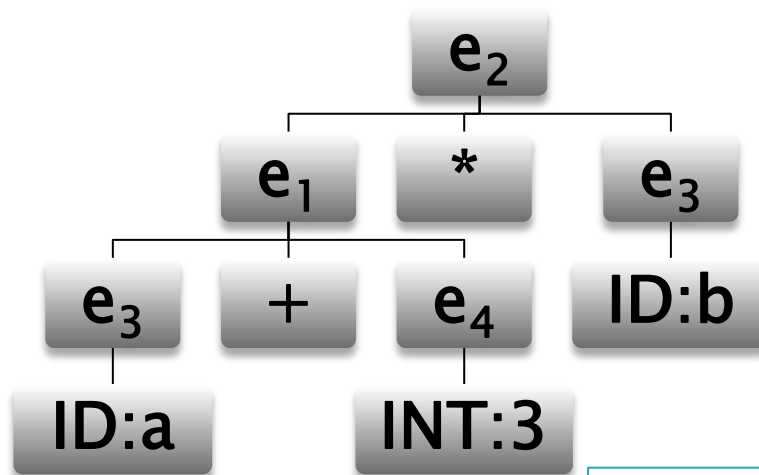
- ▶ O gramatică este ambiguă dacă permite ca pentru aceeași propoziție să existe mai multe derivări
- ▶ O gramatică ambiguă arată faptul că un limbaj nu a fost definit complet, deoarece o propoziție poate fi înțeleasă în mai multe feluri
- ▶ De obicei ambiguitatea poate fi rezolvată prin transformări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

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

$a+3*b$



$a+(3*b)$



$(a+3)*b$

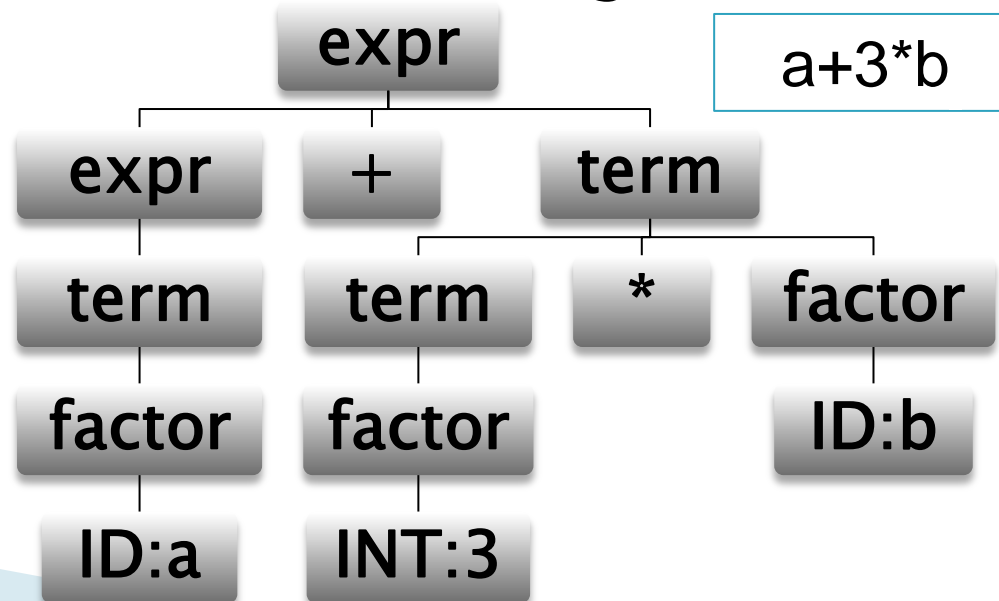
Implementarea precedenței

- ▶ Precedența se implementează folosind reguli separate pentru fiecare nivel de precedență
- ▶ Regulile cu precedența mai mică vor apela regulile cu precedența 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

$e = e \text{ '+' } e \mid e \text{ '*' } e \mid \text{ID} \mid \text{INT}$



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



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: $exprAssign = ID \text{ '=' } exprSrc$

unde $exprSrc$ nu poate conține '='

- ▶ considerând operația respectivă că nu este o expresie ci o instrucțiune (nu returnează o valoare)

Exemplu: $expr = exprAdd / exprMul / postfix / factor$
 $instrAssign = ID \text{ '=' } expr$

Implementarea asociativității stângi

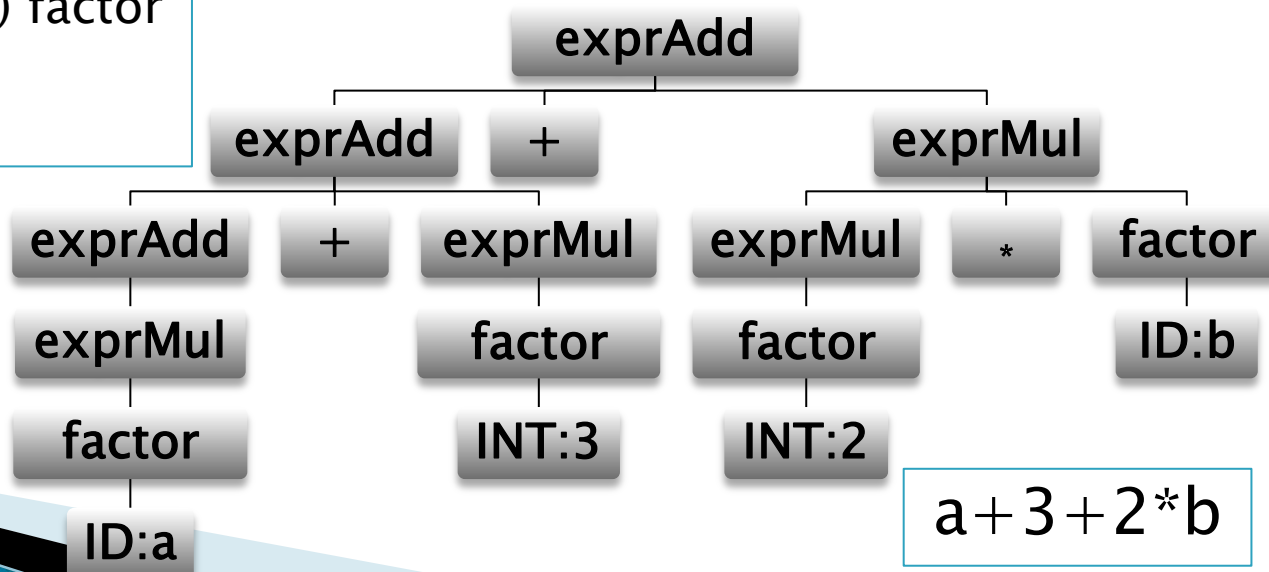
- ▶ Asociativitatea stângă se implementează folosind reguli recursive la stânga, astfel încât rădăcina arborelui va fi operatorul cel mai din dreapta și tot așa recursiv, pe subarborele stâng

- ▶ Exemplu: pentru nivelul de precedență al adunării/scăderii și ținând cont că acești operatori sunt binari (arborele trebuie să aibă 2 ramuri), putem scrie

$exprAdd = exprAdd ('+' | '-') exprMul | exprMul$

- ▶ $exprMul$ este regula pentru precedența imediat mai mare

$exprMul = exprMul ('*' | '/') factor$
| factor
 $factor = ID | INT$

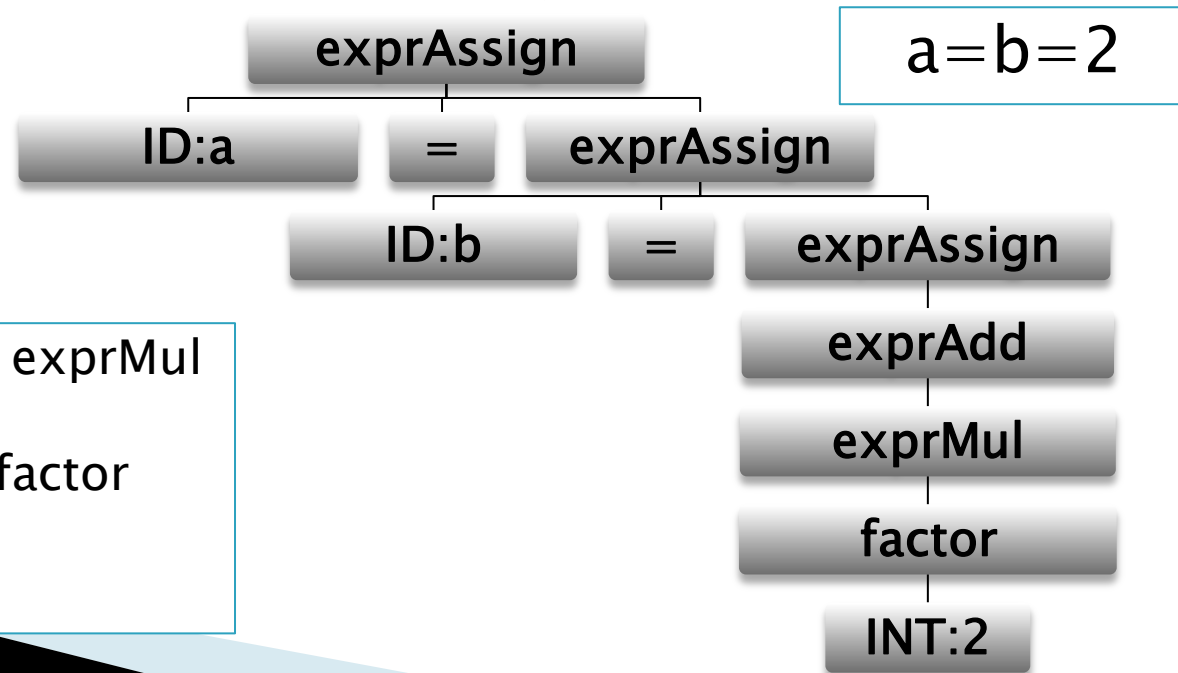


Implementarea asociativității drepte

- ▶ 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

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

- ▶ $exprAdd$ este regula pentru precedența imediat mai mare



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

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 acolade. 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 acolade
- ▶ Expresiile pot conține identificatori, numere reale/întregi, apeluri de funcții și operatori: = < + - (binar și unar) * /. Acești operatori au precedența 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 precedența 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
```