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

<St, Sm, F:St→Sm>

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

```python
# 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** – șirurile 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*={ϵ,0,1,00,01,10,11,000,001,…} (fără ϵ devine **A+**)

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 acelui LP

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

Gramatici formale

- O gramatică G=<T,N,P,s> 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 șiruri, 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 șirurilor 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 **α → β** și are semnificația că prin intermediul ei șirul *α* va fi transformat în șirul *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 șir 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 șir 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)

- Șirurile 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=<T,N,P,s>, unde:
  - T={0,1}
  - N={s}
  - P={s→0s1, s→01}
- Aplicând prima regulă de n-1 ori și a doua regulă o dată, avem următorul șir de transformări:
  $s \rightarrow 0s1 \rightarrow 00s11 \rightarrow 000s111 \rightarrow \ldots \rightarrow 0^{n-1}s1^{n-1} \rightarrow 0^n1^n$
- 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 $\alpha \mathbf{a} \beta \rightarrow \alpha \boldsymbol{\gamma} \beta$, adică $a \rightarrow \gamma$ doar dacă se găsește în contextul $\alpha \ldots \beta$

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 mulțimea gramaticilor de tipul *i* și cu L_i clasa limbajelor generate de 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.

Notații pentru specificarea gramaticilor

- O notație foarte răspândită pentru specificarea unei gramatici este BNF (Backus Naur Form)
- Există multe variante ale acestei notații, 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 șiruri, fără semnificație ca operatori
α | β	**alternativă** – oricare variantă este validă
α β	**secvență** – trebuie să fie îndeplinite ambele șiruri, în ordine
α*	**repetiție opțională** – α poate să apară ori de câte ori, sau poate lipsi
α+	**repetiție obligatorie** – α trebuie să apară cel puțin o dată
α?	**opțional** – 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, abcbc, ...
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'    // s1
s = 'ba'         // s2
// echivalent: s = 'a' s 'b' | 'ba'
// notăm cu s pe oricare dintre
cele două alternative
```

Propoziția '**aababb**' este validă?

Șir curent	Substituție
ϵ	ϵ → s_1
'a' s 'b'	s → s_1
'aa' s 'bb'	s → 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 aceste posibilități în toate locurile posibile și atunci se preferă folosirea unei prime faze (analiza lexicală) care, printre altele, elimină aceste secvențe

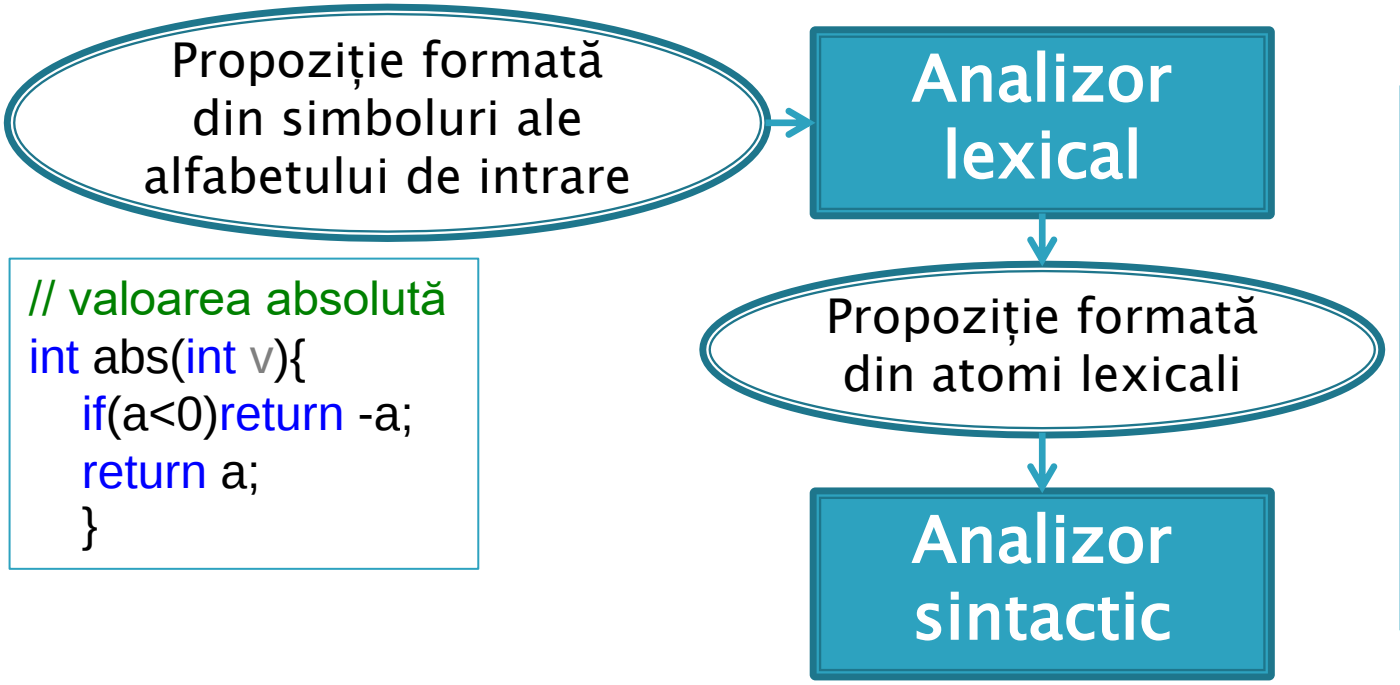
```
// gramatică folosind o singură fază
comentariu = ...
spatiu = ' ' | '\t' | '\n'
skip = ( comentariu | spatiu )*
instr = 'while' skip '(' skip expr skip ')' skip instr skip
```

```
// 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:




```
// 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
```

- 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 atribute ale atomilor.
- Deoarece atomii lexicali reprezintă simboluri de intrare în analizorul sintactic, ei sunt **terminale** pentru analiza sintactică

Terminale și neterminale (2)

- Pentru ușurința reprezentării, în faza de analiză sintactică se pot totuși folosi caractere în regulile sintactice, subînțelegând 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

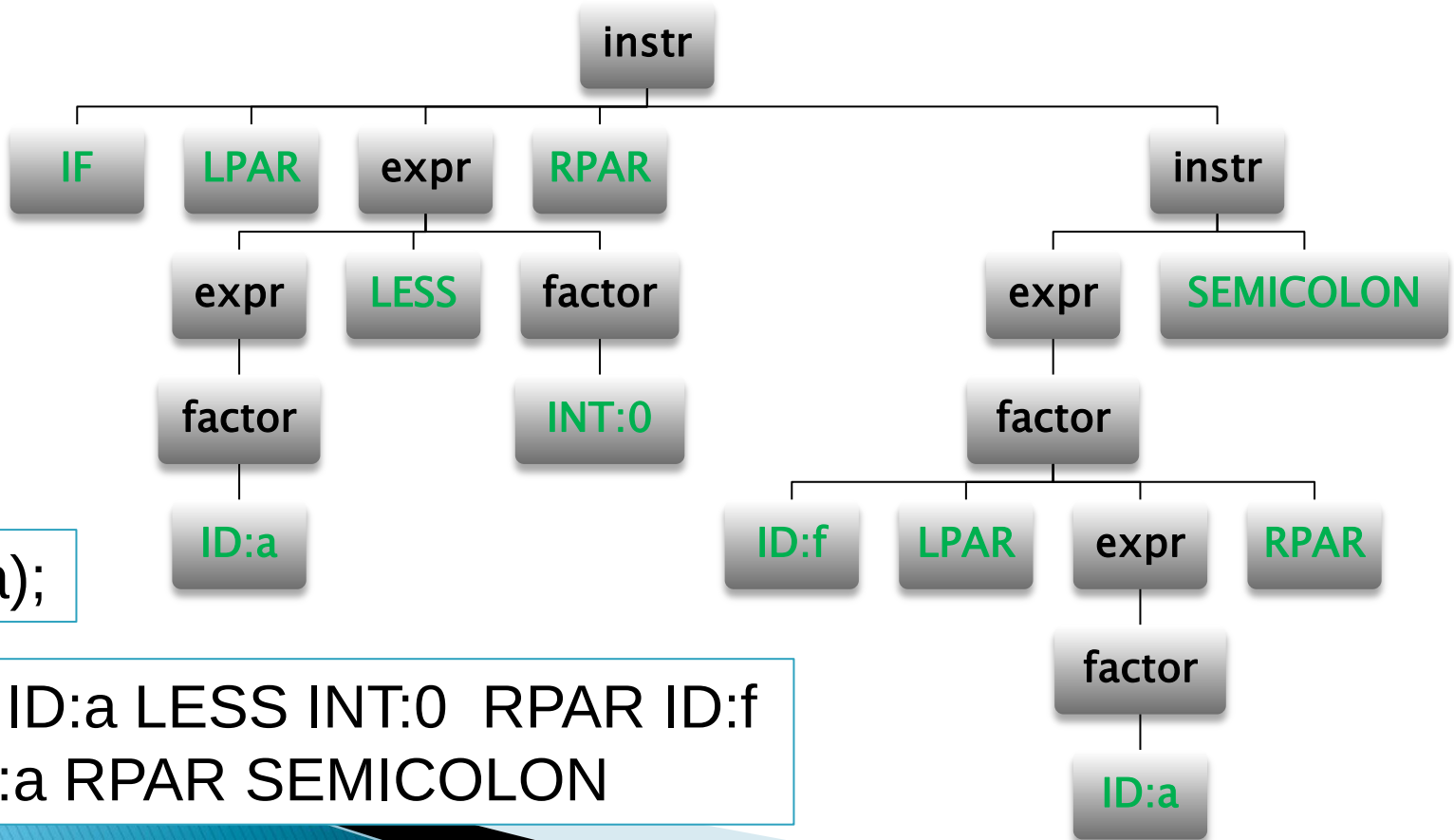
```
// reprezentarea implicită a atomilor lexicali
instr = 'while' '(' expr ')' instr
```

```
// 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
```




```
if(a<0)f(a);
```

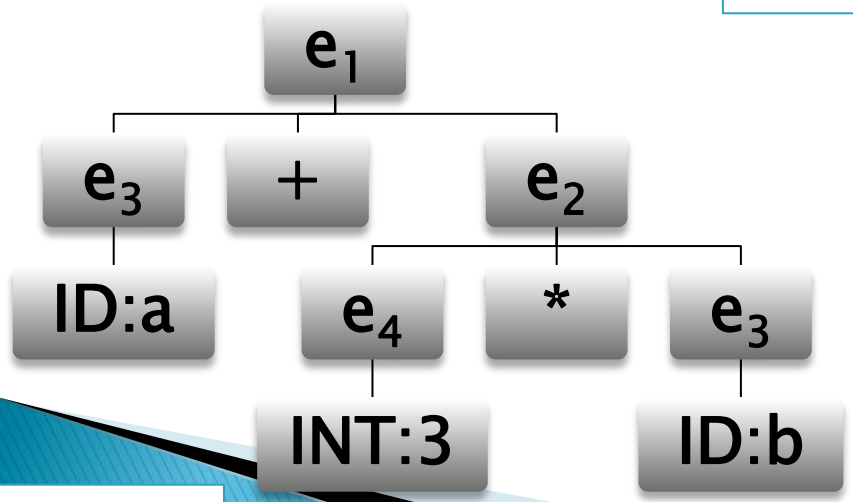
IF LPAR ID:a LESS INT:0 RPAR ID:f
LPAR ID:a RPAR SEMICOLON

Gramatică ambiguă

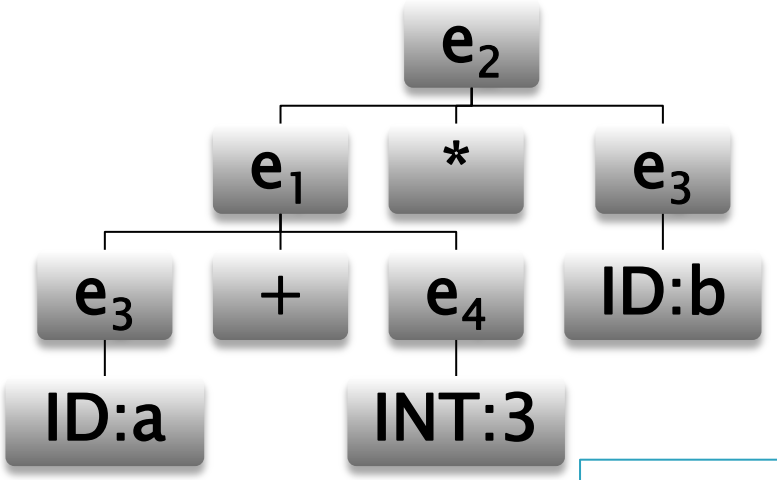
- O gramatică este ambiguă dacă permite ca pentru aceeași propoziție să existe mai multe derivări
- O gramatică ambiguă arată faptul că un limbaj nu a fost definit complet, deoarece o propoziție poate fi înțeleasă în mai multe feluri
- De obicei ambiguitatea poate fi rezolvată prin transformări în gramatică

```
e = e '+' e    // e1
e = e '*' e    // e2
e = ID         // e3
e = INT        // e4
```

a+3*b



a+(3*b)



(a+3)*b

Gramatică recursivă

- Fie o gramatică G și un neterminal oarecare, *a*. Se consideră următorul șir de transformări:
  $a \rightarrow \ldots \rightarrow \alpha a \beta$, cu $\alpha, \beta \in V_G^*$, $V_G = N \cup T$
- Dacă $\alpha = \epsilon$ ($a \rightarrow \ldots \rightarrow a\beta$), atunci *a* se numește **stâng recursiv** (corpul regulii începe cu ea însăși).
- Dacă $\beta = \epsilon$ ($a \rightarrow \ldots \rightarrow \alpha a$), 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**.