

# Limbaje formale și tehnici de compilare

Curs 9: 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
<code>int a;</code>	<code>extern int a;</code>
<code>char v[100];</code>	<code>char v[];</code>
<code>double f(int x){...}</code>	<code>double f(int x);</code>

# 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 marea majoritate a limbajelor cu structură de bloc, un bloc definește un nou domeniu care este imbricat în domeniul părinte

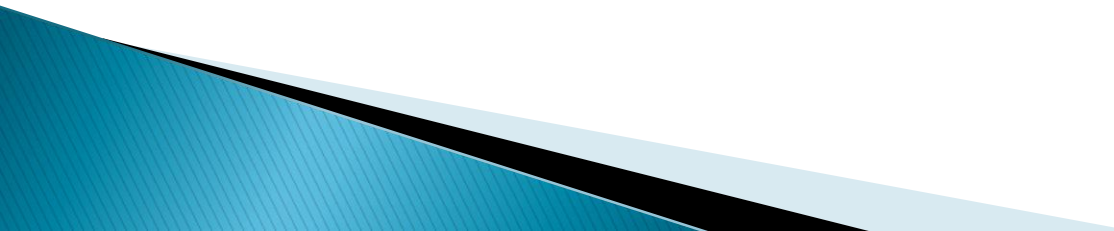
```
int a;                // domeniul lui a: tot programul (global)
int sum(int *v,int n){
    int i,s=0;        // domeniile lui i și s: funcția sum
    for(i=0;i<n;i++){
        int e=v[i];   // domeniul lui e: blocul { ... }
        s+=e;
    }
    // contextul în această locație: a, sum, v, n, i, 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 funcții: parametrii; pentru structuri: câmpurile
    Simboluri simboluri;
    Instrucțiuni instrucțiuni; // pentru funcții: 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 tabelii 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

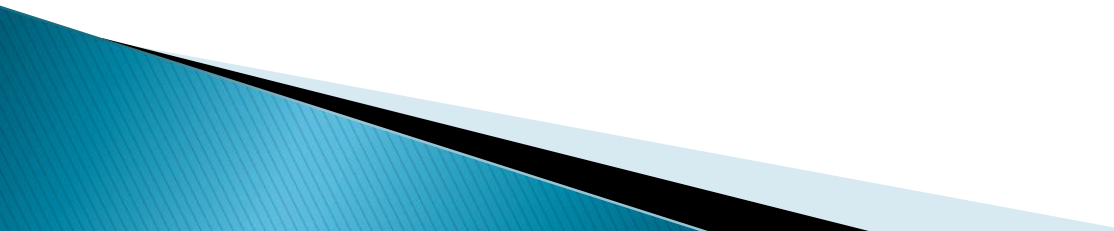
```
bool less(int a,int b){return a<b;}  
bool less(double a,double b){return a<b;}  
bool less(const char *s1,const char *s2){return strcmp(s1,s2)<0;}
```

```
// 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 părinți
  - ▶ La căutarea unui simbol, se începe cu domeniul curent, iar apoi se trece la domeniile părinți, în ordinea apropierii 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**

```
        }
```

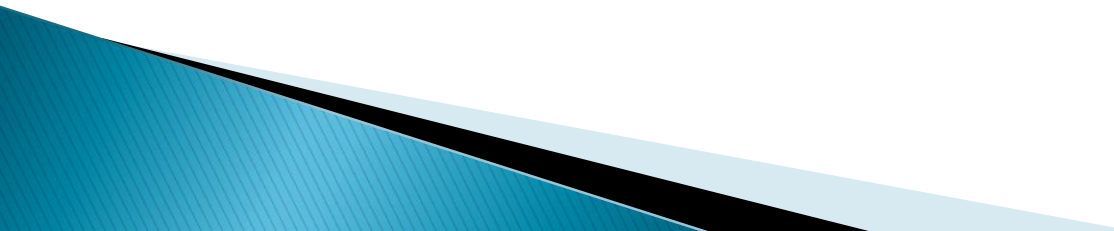
```
    }
```

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

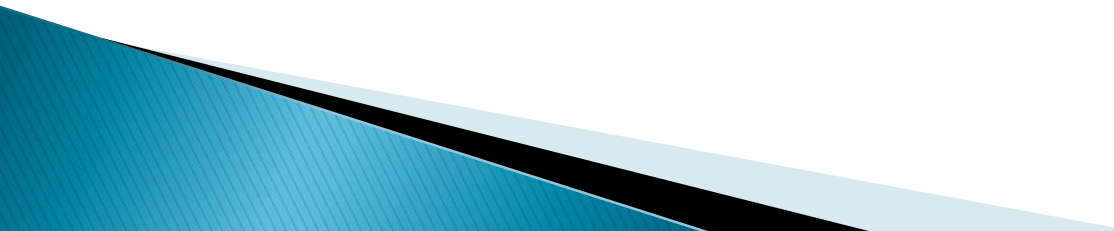
# Integrarea TS în compilator (1)

- ▶ 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 predeclaraire** (Pascal, C)  
– un simbol trebuie prima oară declarat și apoi folosit  
*(în C se permit și apeluri de funcții nedecarate încă, presupunându-se că ulterior funcțiile vor fi definite conform unor reguli predefinite)*
    - ▶ **Limbaje cu declarații statice fără predeclaraire** (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 predeclarăre

```
// 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]}
```

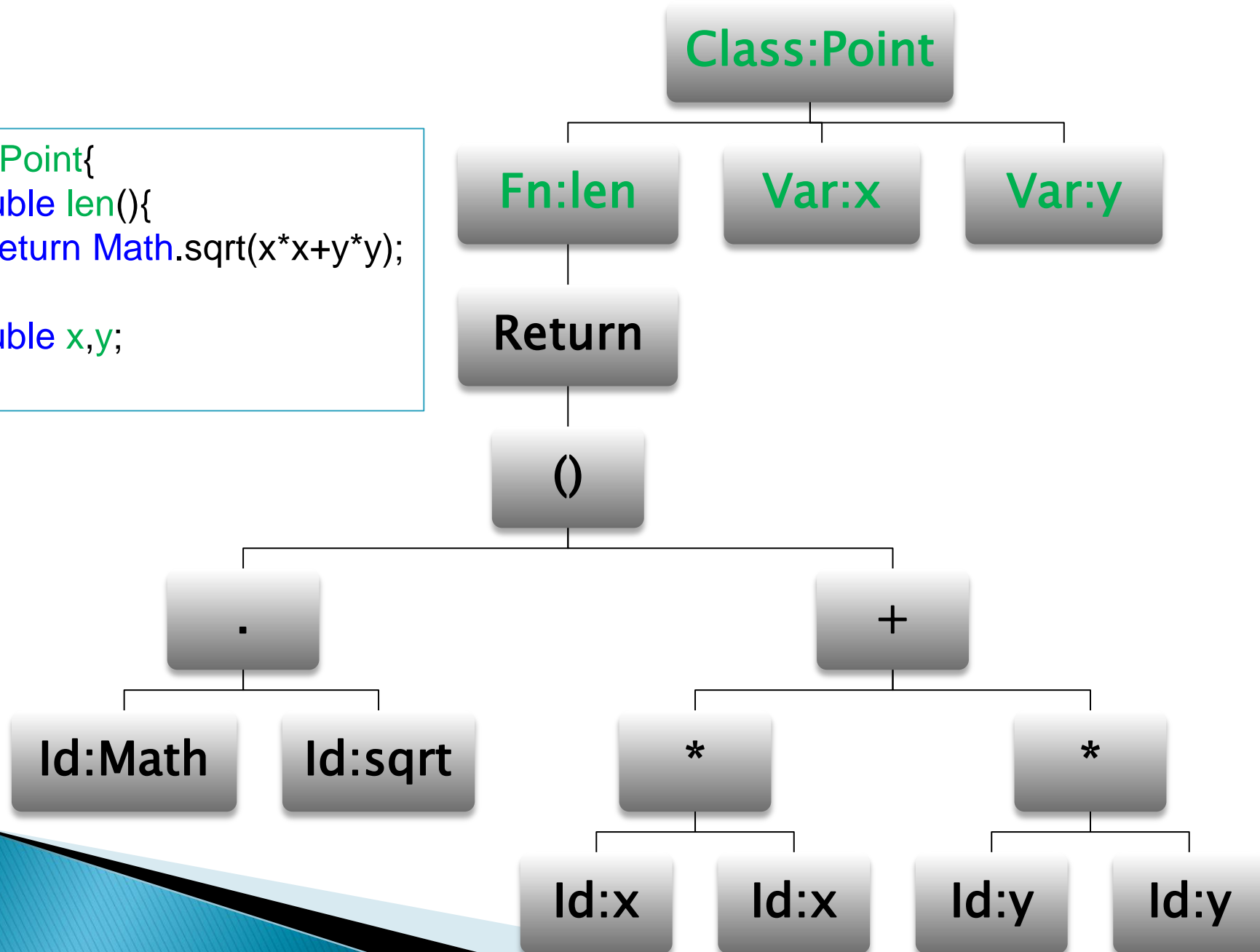
# Limbaje cu declarații statice fără predeclarare

- ▶ 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 attribute 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{ // Java  
    double len(){  
        return Math.sqrt(x*x+y*y); // x,y folosite înainte de declarare  
    }  
    double x,y;  
}
```

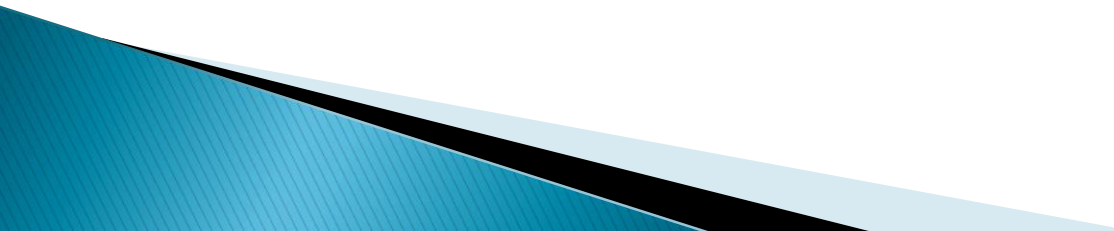
# Exemplu limbaje cu declarații statice fără predeclaraire

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





# Limbaje cu declarații 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
```