

Limbaje formale și tehnici de compilare

Curs 10: 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
```

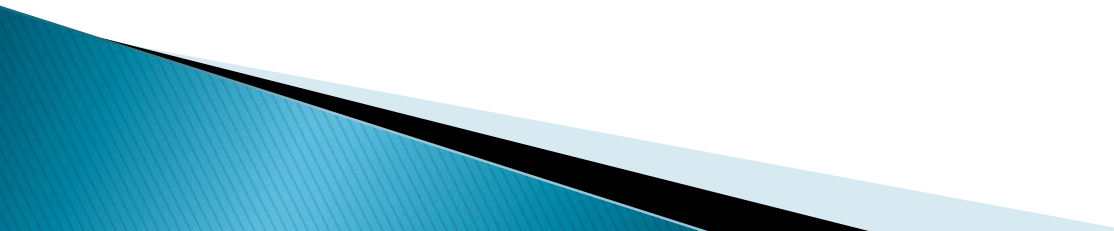
LP cu tipuri statice (statically typed)

- ▶ 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
 - ▶ Necesită 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 inteligibilitatea 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

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

var x=5;                      // x contine o valoare de tip numeric
afis(x);

x="salut";                    // x contine 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 inteligibilitatea 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 tabelii 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 identificador, 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

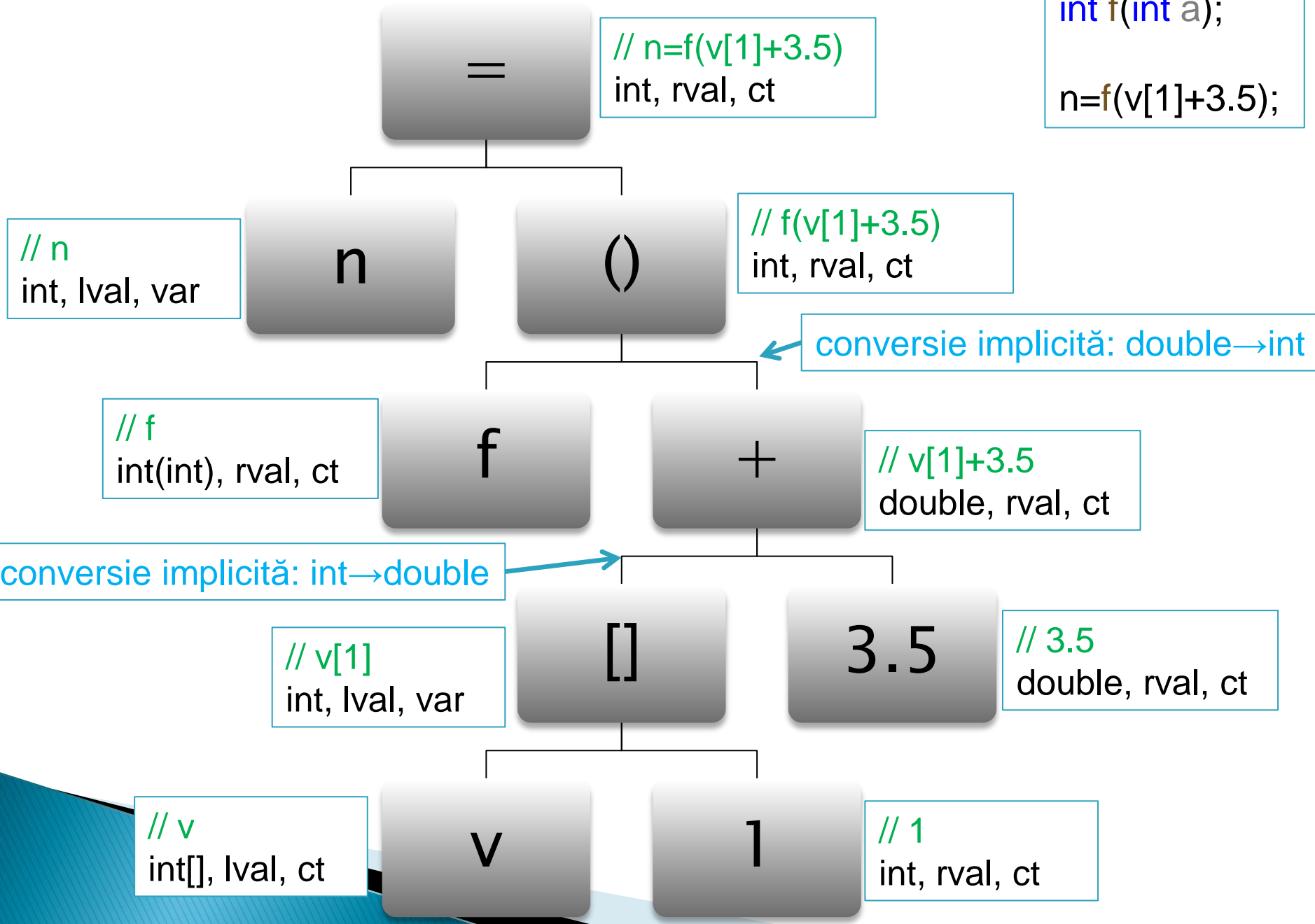
- ▶ Doar funcțiile pot fi apelate, posibil prin intermediul unor pointeri la funcții
- ▶ Numărul de argumente la apelul unei funcții trebuie să coincidă cu cel din definiția funcției
- ▶ Unele tipuri se pot converti implicit, dacă e nevoie (ex: `int` → `double` în operația `1 + 3.5`)
- ▶ Folosirea constantelor 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 ști 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 in 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 operanzii 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*