

Limbaje formale și tehnici de compilare

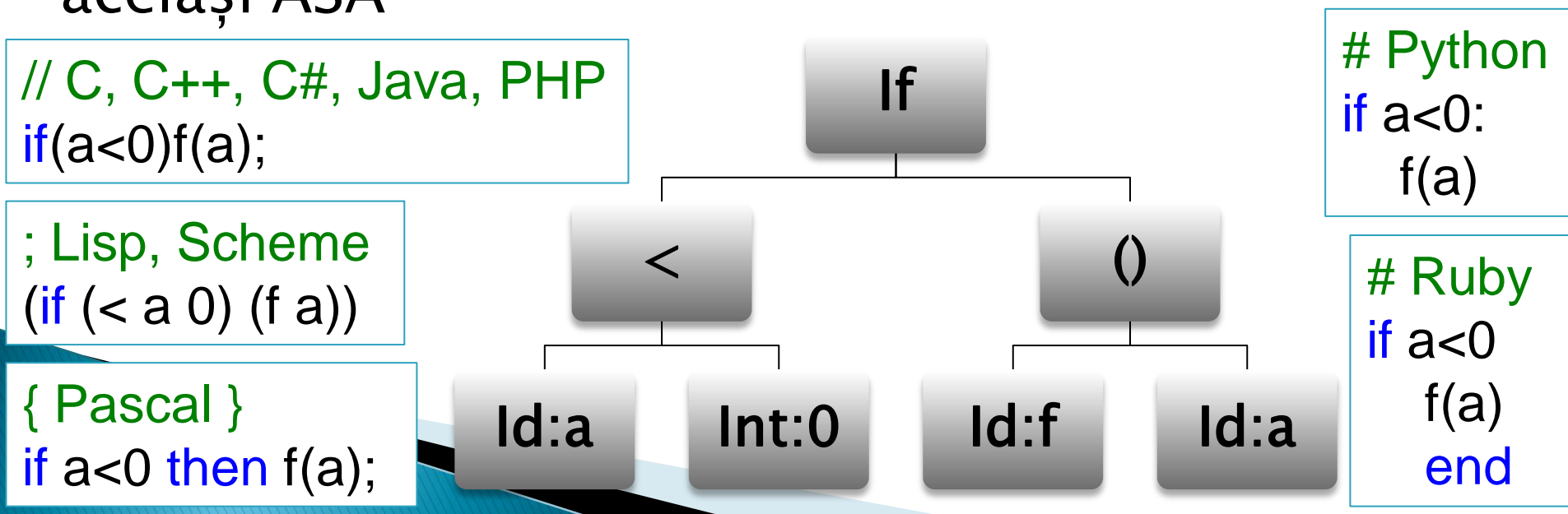
Curs 8: analiza semantică; arborele sintactic abstract; atribute moștenite și sintetizate; reguli semantice

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



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, așa că se poate renunța la aceștia
- ▶ 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, ...)

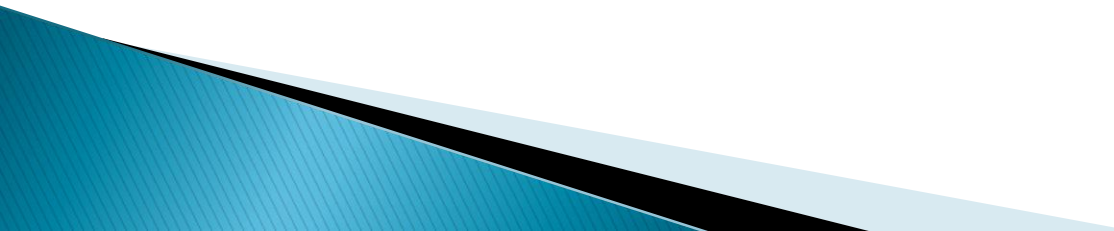
Implementarea ASA

- ▶ 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:
 - ▶ **atribute sintetizate**
 - ▶ **atribute 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 părinți sau frați. Atributele moștenite sunt echivalente parametrilor unei funcții.
- ▶ Unei reguli sintactice i se pot asocia unul sau mai multe atribute
- ▶ 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

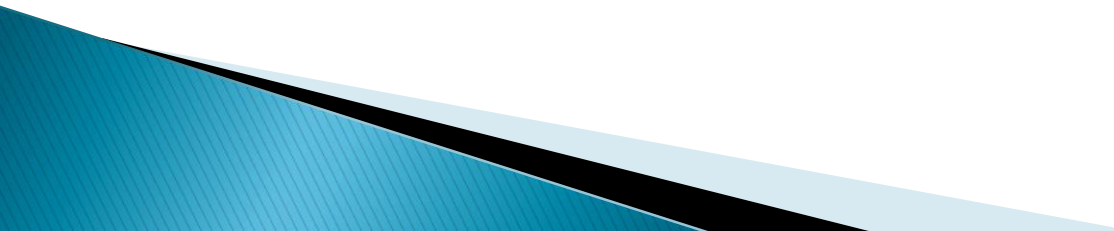
```
// regula angajat primește atributul moștenit functie  
// și returnează atributul sintetizat a  
angajat[in functie:string, out a:Angajat] = ...
```

```
#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 $e_1 \&\&e_2 = 0$)

Exemple 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 părinți
 - ▶ 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 attribute, 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#, ...), attributele (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, ...), attributele 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 atribute, 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;
}
```