

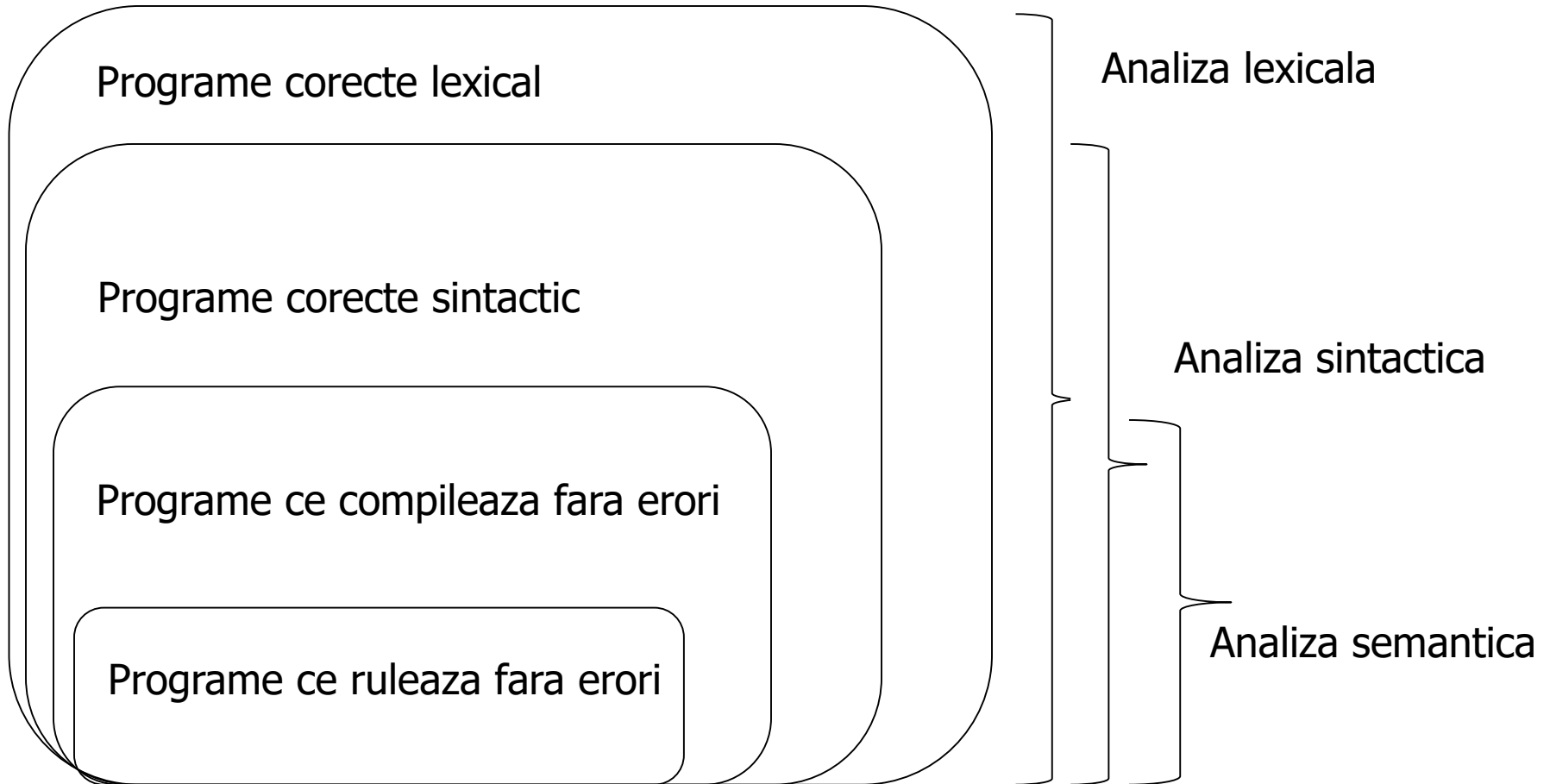
Compilatoare

Curs 4

Analiza semantica



ANALIZA SEMANTICA



ANALIZA SEMANTICA

- Calculeaza toate attributele asociate nodurilor din arborele sintactic
 - Exemple de attribute: Valoarea unei constante, numele unei variabile, tipul unei expresii
 - Attributele terminalilor se seteaza de obicei direct din analiza lexicala
 - O parte din analiza semantica se face in timpul parsarii
 - Restul - parcurgerea recursiva a arborelui sintactic. (AST = abstract syntax tree)
- Verifica daca structurile sintactic corecte au sens dpdv semantic
 - Gaseste erori semantice (toate erorile de compilare care nu sunt erori de sintaxa)

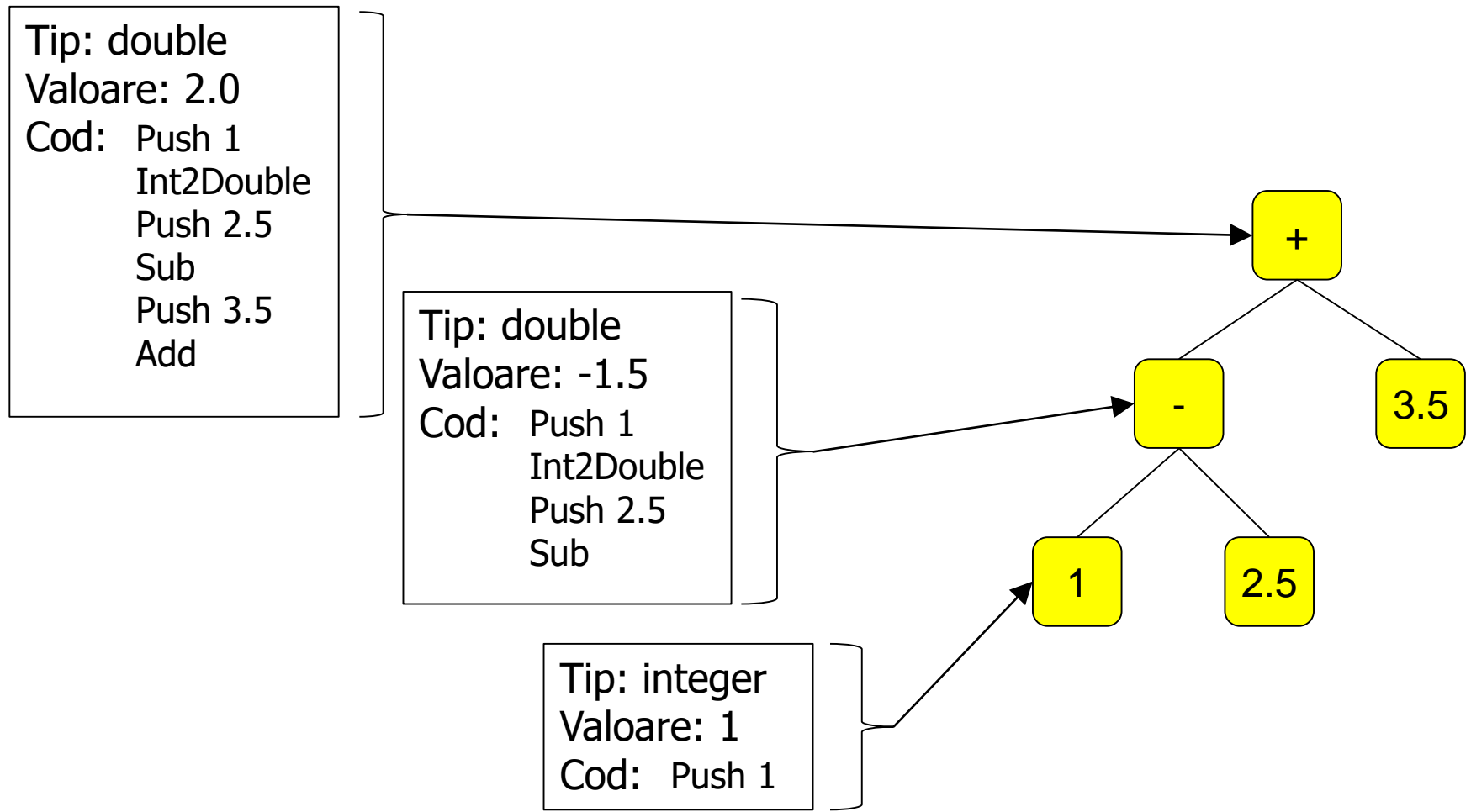
Exemple de erori semantice

- Erori de definitie – variabile, functii, tipuri folosite fara a fi definite
 - In unele limbaje avem definitii implicite.
 - "var a = 10;" se poate deduce tipul din context?
 - Rezolvare la timpul compilarii (limbaje statice), sau la rulare (limbaje dinamice).
- Erori de structura
 - $X.y=A[3]$ – X trebuie sa fie structura/clasa cu campul 'y', A trebuie sa fie array/pointer
 - $foo(3, true, 8)$ trebuie sa fie o functie ce accepta 3 parametri

Exemple de erori semantice

- Erori de tip – `'a'+5`.
 - Compatibilitatea tipurilor.
 - Ex: in Pascal, doar tipurile identice; in C, tipurile "cu aceeasi structura"; in C++/Java, subclasele sunt compatibile cu superclasele
 - Unele limbaje accepta conversii automate de tip
 - $3 + "45" = ?$ (48? "345"?)
 - Strongly / weakly typed.
- Erori de acces – `private/protected;const`

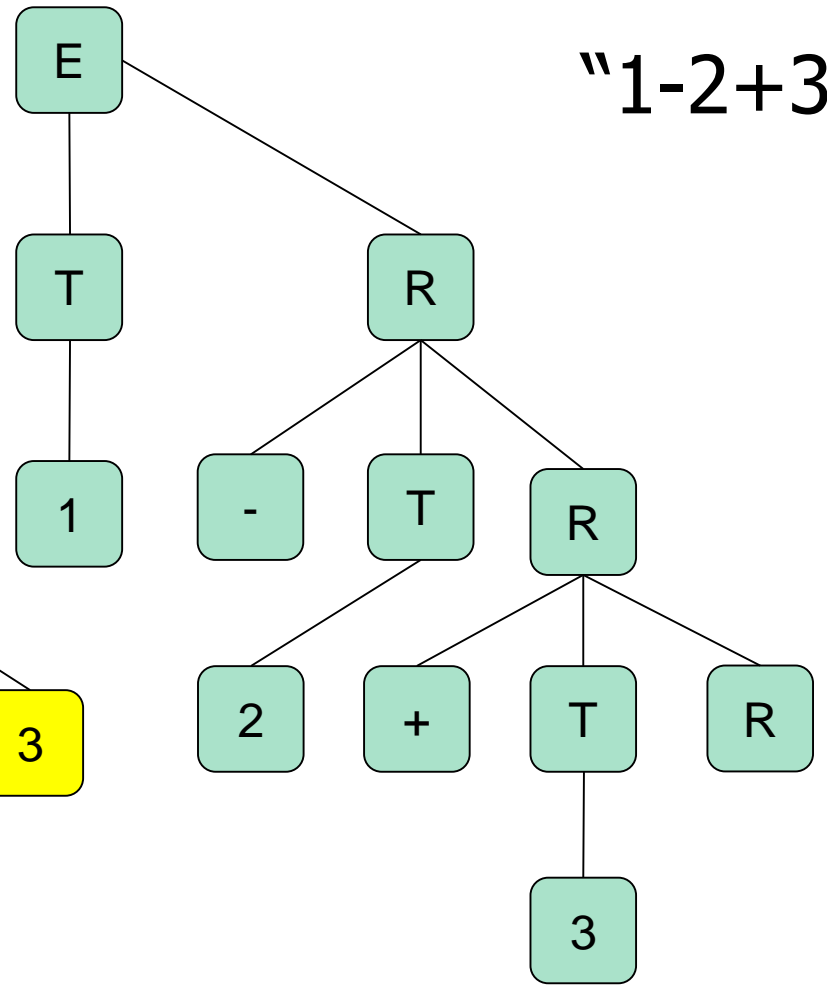
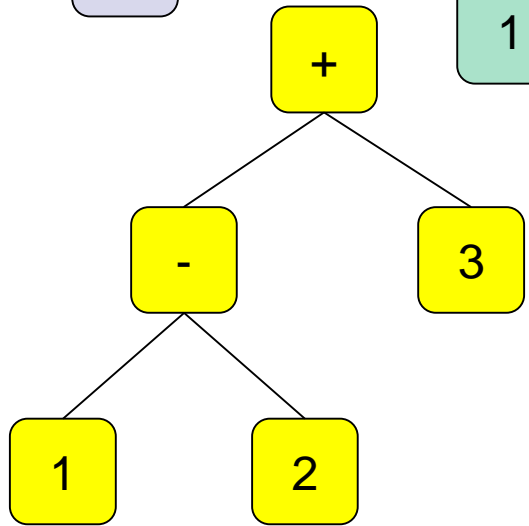
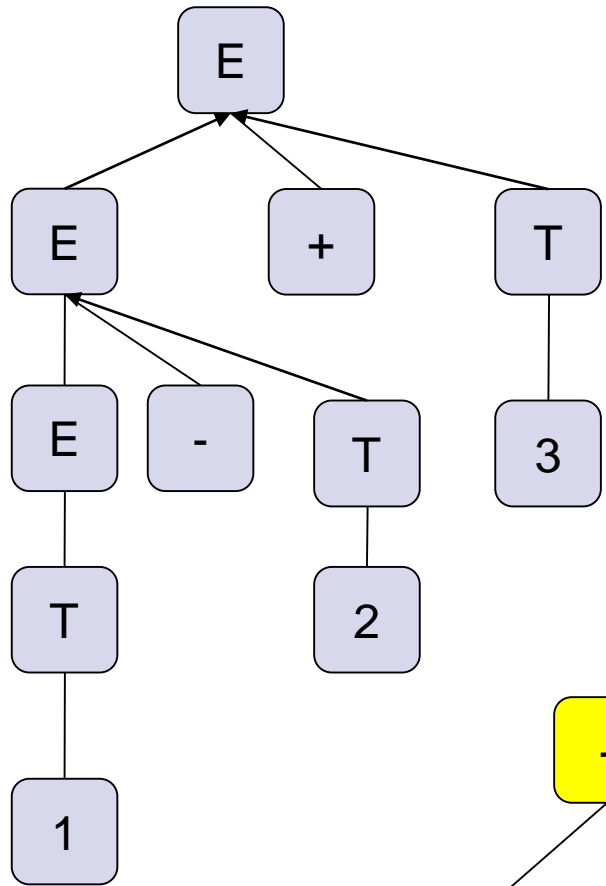
Exemple de atribute



Arbori: derivare vs. sintactic

- O gramatică "comodă" din punctul de vedere al analizei sintactice se poate dovedi "incomodă" din punctul de vedere al stabilirii regulilor semantice datorita transformarilor suferite
 - Parserul descopera un arbore de derivare.
 - Facem analiza semantica pe arborii sintactici!
- Un pas de analiza semantica – extragerea AST

Arbori: derivare vs. sintactic

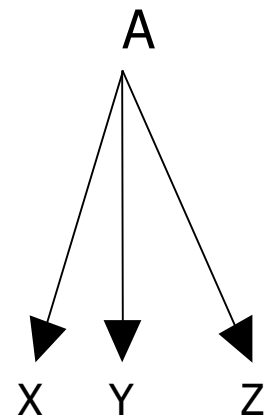


"1-2+3"

Cateva definitii

- Gramatica independenta de context + reguli de calcul ale atributelor = **definiție orientată sintaxă** (syntax directed definition).
 - Dacă funcțiile utilizate în calculul de attribute nu au efecte laterale -> **gramatică de attribute**
- Definitie orientata sintaxa + detalii de implementare = **schemă de traducere**.

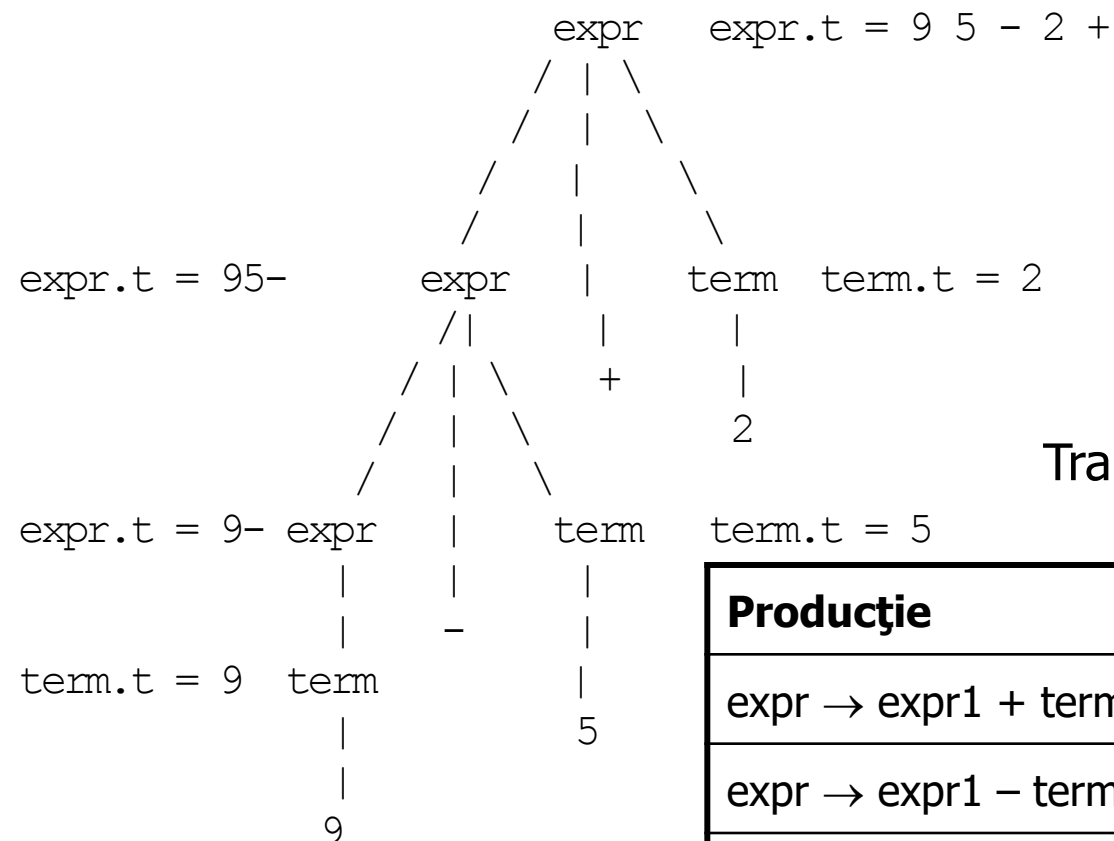
- Fie urmatorul arbore de derivare:
 - $A.a = f(X.a, Y.a, Z.a)$ – atribut sintetizat
 - $Y.a = F(A.a, X.a, Z.a)$ –atribut mostenit



Syntax directed definition

- După ce am stabilit gramatica limbajului
 - Pentru fiecare simbol din gramatică se asociază un atribut (eventual cu mai multe campuri)
 - Pentru fiecare producție se asociază o mulțime de reguli semantice (cum calculăm valoarea atributelor)
- Gramatica + reguli semantice \Rightarrow definiție orientată sintaxa
- Pt o producție $A \rightarrow X_1 \dots X_k$ regulile semantice sunt de forma:
 - $A.a := f(X_1.a, \dots, X_k.a)$
 - $X_i.a := f(A.a, X_1.a, \dots, X_k.a)$, cu X_i neterminal

Gramatica de atribute



Translatarea expresiilor in forma postfixata

9 - 5 + 2 → 9 5 - 2 +

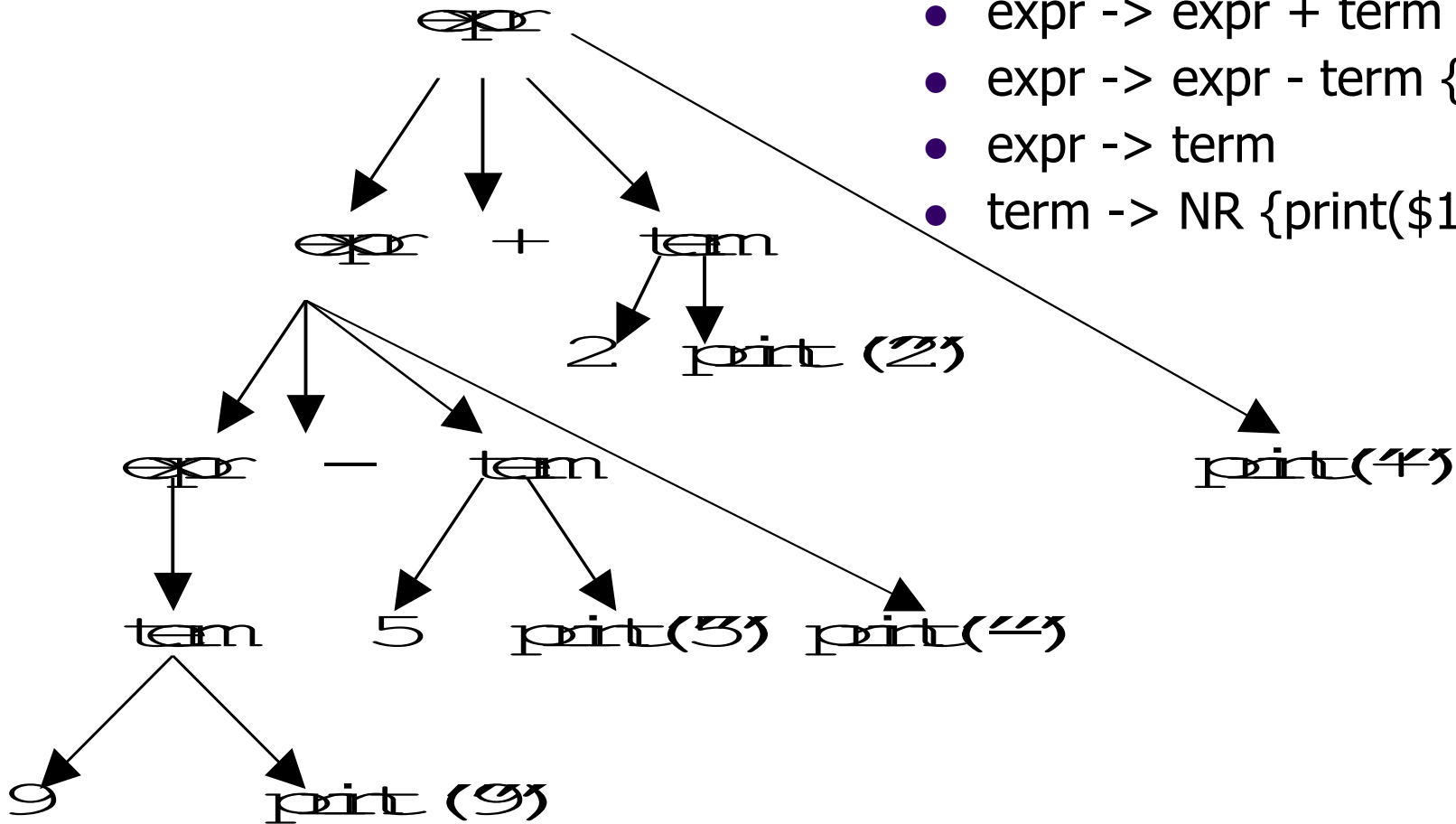
Producție	Acțiune
$\text{expr} \rightarrow \text{expr1} + \text{term}$	$\text{expr.t} := \text{expr1.t} \text{ term.t} '+'$
$\text{expr} \rightarrow \text{expr1} - \text{term}$	$\text{expr.t} := \text{expr1.t} \text{ term.t} '-'$
$\text{expr} \rightarrow \text{term}$	$\text{expr.t} := \text{term.t}$
$\text{term} \rightarrow 0$	$\text{term.t} := '0'$
...	...

Scheme de traducere

- O schema de traducere
 - GIC + acțiuni semantice (definiție orientată sintaxă)
 - Momentul în care act. semantice sunt executate în timpul parsării
- O specificare posibilă utilizează fragmente de program reprezentând acțiuni semantice intercalate între simbolii care apar în partea dreaptă a producțiilor:
 - $A \rightarrow \alpha \{ \text{print('x')} \} \beta$
 - se va afișa caracterul 'x' după ce se vizitează subarbo-rele α și înainte de traversarea subarborelui β .
 - Un nod care reprezintă o acțiune semantică nu are descendenți iar acțiunea semantică se execută atunci când este întâlnită în parcurgerea arborelui.

Exemplu: 9-5+2

- $\text{expr} \rightarrow \text{expr} + \text{term} \{\text{print}('+')\}$
- $\text{expr} \rightarrow \text{expr} - \text{term} \{\text{print}('-')\}$
- $\text{expr} \rightarrow \text{term}$
- $\text{term} \rightarrow \text{NR} \{\text{print}(\$1)\}$



Graful de dependenta

- Definițiile orientate sintaxa nu precizează când se aplică regulile semantice
 - Dar se precizează cum depind unele de altele
 - Dacă un atribut depinde de un alt atribut c , atunci regula semantică pentru calculul atributului b trebuie să fie evaluată după regula semantică care îl produce pe c
- Graful de dependenta
 - Noduri = attribute
 - Arc $n_1 \rightarrow n_2$: n_2 se calculează pe baza n_1

Calculul atributelor

- Ordinea de calcul – ordinea topologica pe graful de dependenta
- Se construiesc arborele de derivare, apoi graful de dependenta pentru toate attributele, apoi se sorteaza topologic si rezulta o ordine de calcul a atributelor
 - Calculul atributelor este posibil numai dacă graful de dependență este necircular.
- Conteaza ordinea de evaluare?
 - Nu, pentru gramaticile de attribute
 - Da, pentru schemele de traducere

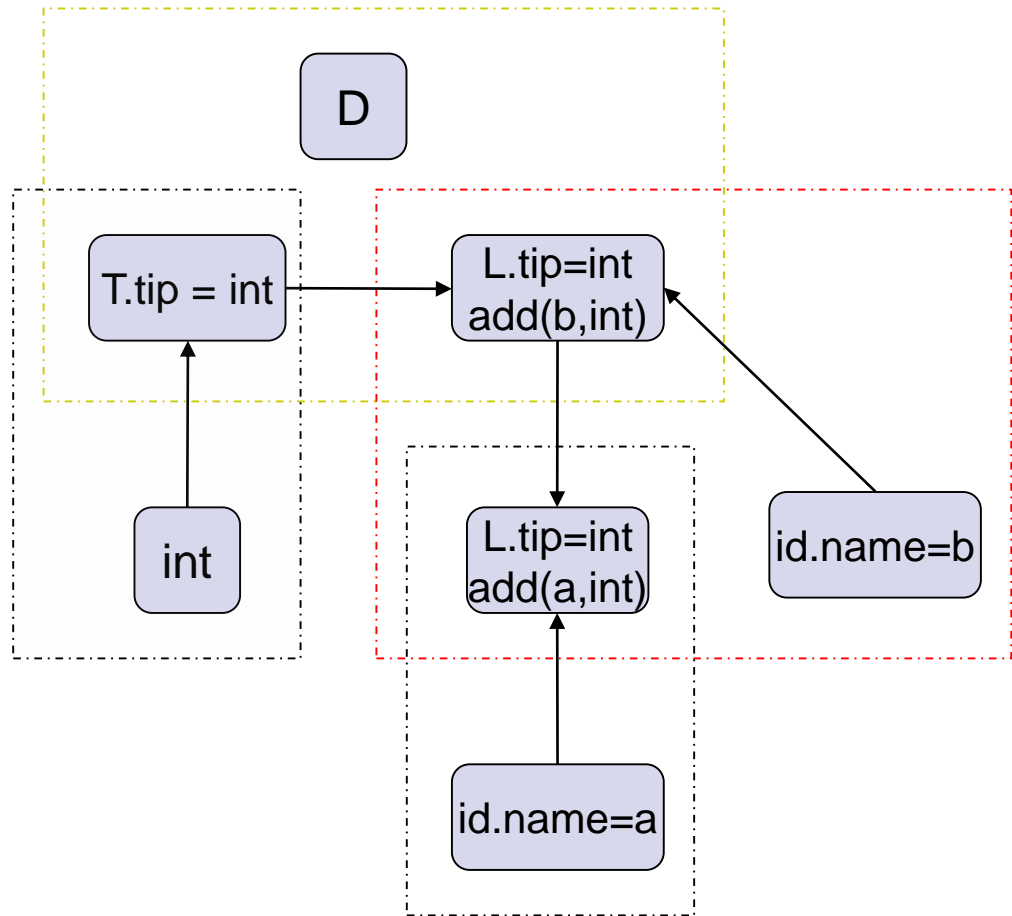
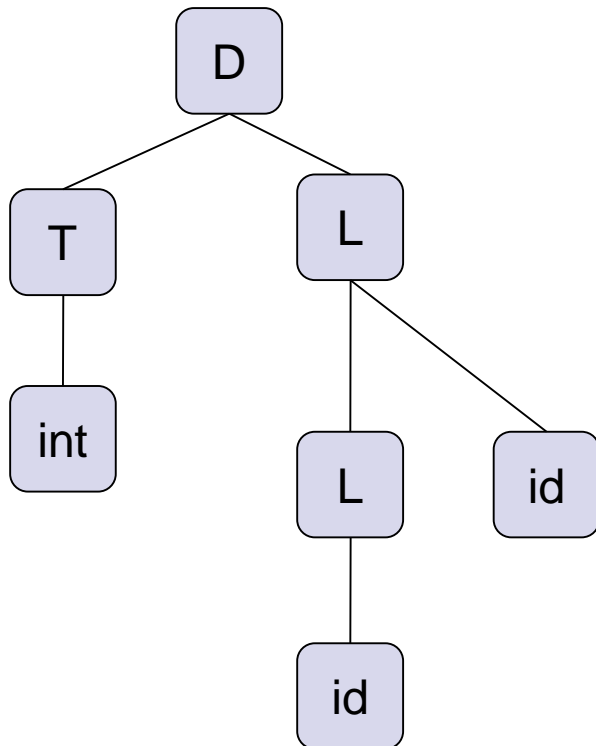
Exemplu

- float a, b, c;
- L.tip depinde de T.tip, addVar() depinde de L.tip
- L.tip, id.nume sunt mostenite
- T.tip e sintetizat
- Definitie orientata sintaxa, nu gramatica de atribute (addVar)

Producție	Regula semantică (acțiune)
D -> T L;	L.tip = T.tip
T -> int	T.tip = int
T -> float	T.tip = float
L -> L ₁ , id	L ₁ .tip = L.tip; addVar(id.nume, L.tip)
L -> id	addVar(id.nume , L.tip)

Calculul atributelor (cont.)

- "int a,b;"



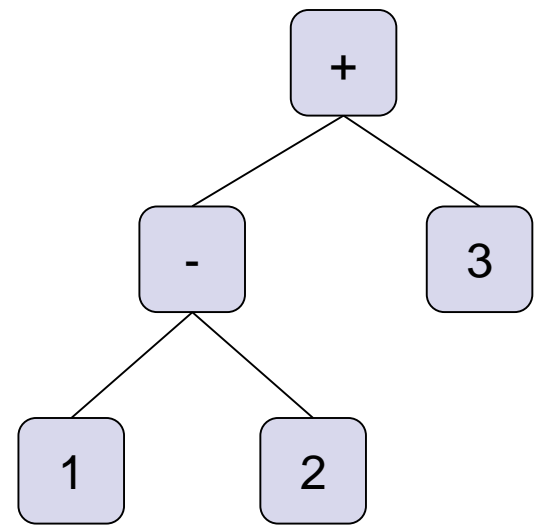
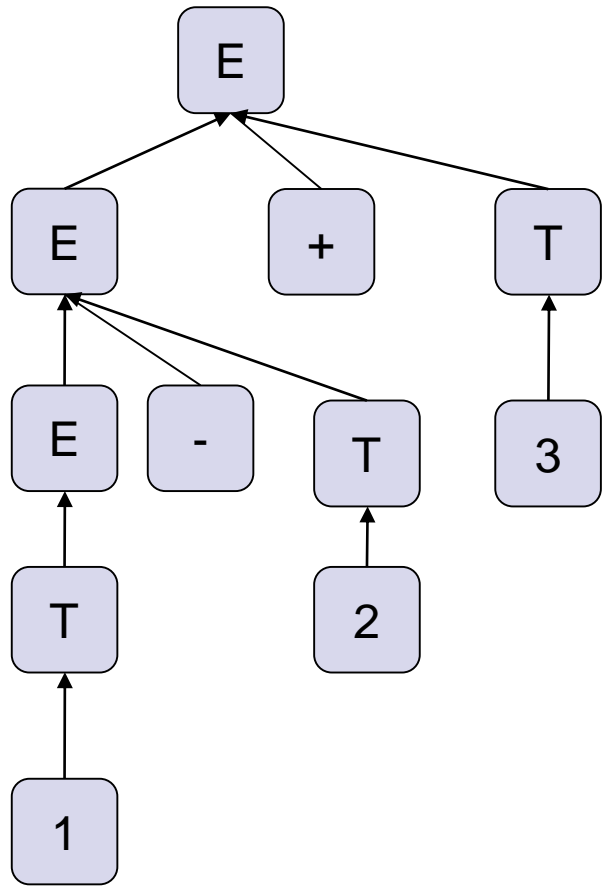
Calculul atributelor

- Dandu-se o definitie orientata sintaxa, este graful necircular pentru orice arbore de derivare?
 - Algoritm exponential in cazul general
 - Se restrictioneaza regulile de calcul ale atributelor
- Evaluare in timpul parsarii
- Algoritmi care garanteaza ordinea de evaluare

Definitii S-atributate

Producție	Regula semantică (acțiune)
$E \rightarrow E_1 + T$	$E.s := \text{Nod}('+', E_1.s, T.s);$
$E \rightarrow E_1 - T$	$E.s := \text{Nod}('-', E_1.s, T.s);$
$E \rightarrow T$	$E.s := T.s$
$T \rightarrow \text{num}$	$T.s := \text{Nod}(\text{num.val})$

Definitii S-atributate (cont.)



"1-2+3"

Definitii S-atributate (cont.)

- Doar attribute sintetizate
 - Stiva e 'imbogatita' cu informatii legate de attributele neterminalilor recunoscuti
 - De câte ori se face o reducere, valorile atributelor sintetizate sunt calculate pornind de la attributele care apar în stivă pentru simbolii din partea dreaptă a producției.
 - Naturale in analiza ascendenta, dar si in analiza descendenta

Definitii S-atributate (cont.)

Analiza descendent recursiva

```
expr returns [int value] : e=term {$value = $e.value;}  
  ( '+' e=term {$value += $e.value;} )*;
```

```
int Expr() {  
    int e = Term(), value = e;  
    while (lookahead() == PLUS) {  
        match(PLUS);  
        e = Term();  
        value += e;  
    }  
    // ... verify lookahead here ...  
    return value;  
}
```

- Dar intr-un automat cu stiva?

Definitii S-atributate

Analiza ascendenta

```
expr : expr '+' term { $$ = $1 + $3; }  
      | term { $$ = $1; } ;
```

- Cod executat la reduce:

```
switch (state) {  
  case 3: value = stack[top - 2] + stack[top]; break;  
  case 4: value = stack[top]; break;  
}
```

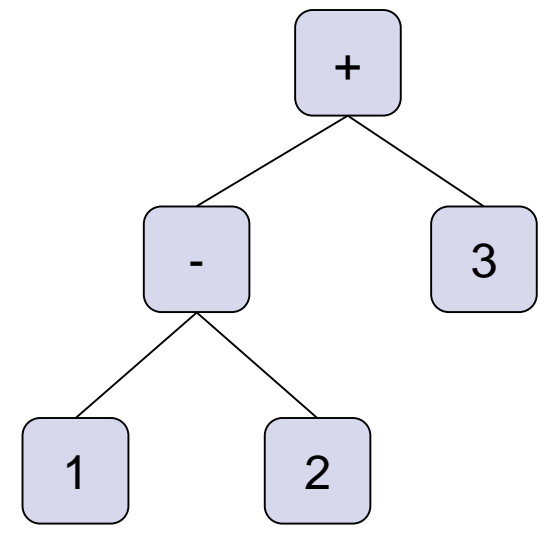
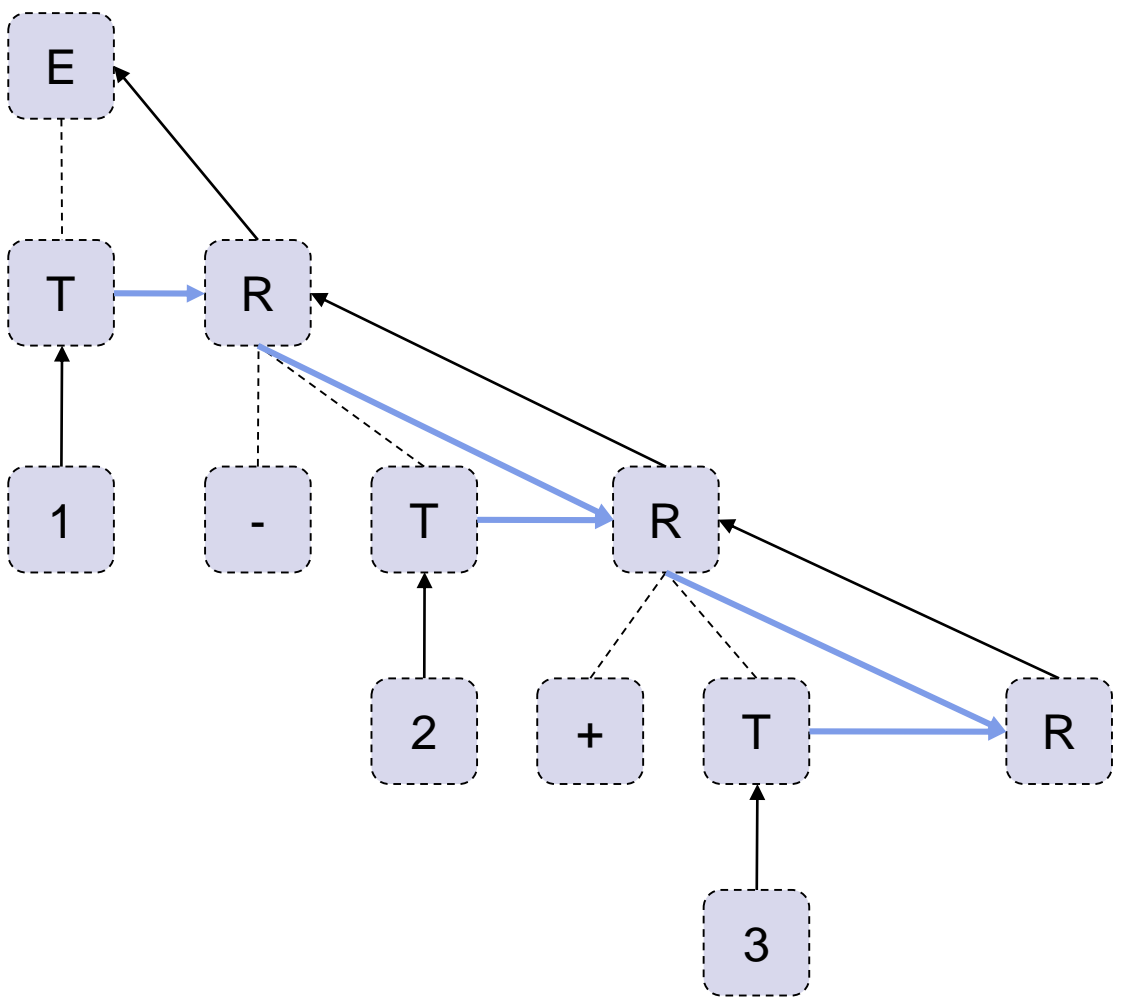
```
pop(stack, 3); push(stack, value);
```

- Care e continutul stivei?
- Ce cod se genereaza pentru shift?

Definitii L-atributate

Producție	Regula semantică (acțiune)
$E \rightarrow T R$	$R.m := T.s; E.s = R.s$
$R \rightarrow - T R_1$	$R_1.m := \text{Nod}('-', R.m, T.s); R.s = R_1.s$
$R \rightarrow + T R_1$	$R_1.m := \text{Nod}('+', R.m, T.s); R.s = R_1.s$
$R \rightarrow \lambda$	$R.s := R.m$
$T \rightarrow \text{num}$	$T.s := \text{Nod}(\text{num.val})$

Definitii L-atributate (cont.)



"1-2+3"

Definitii L-atributate (cont.)

- Orice atribut calculat printr-o regulă semantică asociată producției $A \rightarrow X_1 X_2 \dots X_n$ este
 - fie sintetizat,
 - fie este un atribut moștenit pentru neterminalul X_j care depinde numai de atributele simbolilor X_1, X_2, \dots, X_{j-1} și de atributele moștenite pentru A
- Includ definițiile S-atributate
- Naturale în analiza descendentă

Definitii L-atributate

Analiza descendent recursiva

$R \rightarrow + T R_1$	$R_1.m := \text{Nod}('+', R.m, T.s); R.s = R_1.s$
$R \rightarrow \lambda$	$R.s := R.m$

```
Nod R(Nod m) {  
    if (lookahead() == PLUS) {  
        MATCH(PLUS);  
        Nod t = T();  
        Nod r = R(new Nod(PLUS, m, t));  
    }  
    else  
        r = R(m);  
    return r;  
}
```

Implementare in analiza ascendenta

$E \rightarrow TR$

$R \rightarrow +T \{ \text{print}('+') \} R \mid -T \{ \text{print}('-') \} R \mid \lambda$

$T \rightarrow \text{numar} \{ \text{print}(\text{numar.val}) \}$

- Putem să rescriem schema de traducere sub forma:

$E \rightarrow TR$

$E \rightarrow +T M R \mid -T N R \mid \lambda$

$T \rightarrow \text{numar} \{ \text{print}(\text{numar.val}) \}$

$M \rightarrow \lambda \{ \text{print}('+') \}$

$N \rightarrow \lambda \{ \text{print}('-') \}$

- Ambele scheme de traducere reprezintă aceeași gramatică și toate acțiunile sunt executate în aceeași ordine. Prin introducerea unor simbolii neterminali suplimentari am reușit să îndeplinim condiția de a avea acțiunile semantice la sfârșitul producției

Implementare (cont.)

- Putem considera cunoscuta structura stivei
- T apare intotdeauna in stiva inaintea lui L
- Putem folosi `addVar(id.ume, Previous(stack).tip)`.
- Pt. attributele sintetizate, pozitia se stie; pt cele mostenite, e "tricky"

Producție	Regula semantică (acțiune)
$D \rightarrow T L;$	<code>L.tip = T.tip</code>
$T \rightarrow \text{int}$	<code>T.tip = int</code>
$T \rightarrow \text{float}$	<code>T.tip = float</code>
$L \rightarrow L_1, \text{id}$	<code>L₁.tip = L.tip;</code> <code>addVar(id.ume, L.tip)</code>
$L \rightarrow \text{id}$	<code>addVar(id.ume, L.tip)</code>

Implementare (cont.)

- Soluția anterioară nu e generică.
- T nu mai apare întotdeauna în stivă înaintea lui L
- Putem modifica gramatica:
 - $D \rightarrow T : X L$
 - $X \rightarrow \lambda$
 - $X.tip = T.tip$
 - $L.tip = T.tip$

Producție	Regula semantică (acțiune)
$D \rightarrow T : L;$	$L.tip = T.tip$
$D \rightarrow TL;$	$L.tip = T.tip$
$T \rightarrow int$	$T.tip = int$
$T \rightarrow float$	$T.tip = float$
$L \rightarrow L_1, id$	$L_1.tip = L.tip;$ $addVar(id.nume, L.tip)$
$L \rightarrow id$	$addVar(id.nume, L.tip)$

Implementare (cont.)

- Probleme daca gramatica nu e LL(1)

$A_1 \rightarrow A_2 x \{A_2.m = f(A_1.m); \} \mid y \{y.i = f(A_1.m); \}$

Introducem neterminali:

$A \rightarrow M1 Ax \mid M2 y$

$M1 \rightarrow \lambda$

$M2 \rightarrow \lambda$

- Apare conflict reduce-reduce M1-M2
 - Y e in FOLLOW(M1) si in FOLLOW(M2)

Ce poate contine un atribut?

- Un sub-arbore sintactic
- Valoarea unei expresii (evaluare/interpretare)
- Tipul unei expresii
- Cod intermediar / final generat
 - Syntax-directed translation

Attribute folosite in translatare

- $T \rightarrow \text{var}$

```
T.Cod =  $\epsilon$   
T.Res = var
```

- $E \rightarrow T$

```
E.Cod = T.Cod  
E.Res = T.Res
```

- $E \rightarrow E + T$

```
E.Cod =  
    E.Cod;  
    T.Cod;  
    temp = E.Res + T.Res  
E.Res = temp
```

Attribute folosite in translatare

- $I \rightarrow \text{if } E \text{ then } I_1 \text{ else } I_2$

```

I.Cod =
    E.Cod;
    if E.Res==false goto l1
    I1; goto l2
l1: I2;
l2:
    
```

- $L \rightarrow \text{var}$

```

L.Cod =  $\epsilon$ 
L.Address = addr(var)
    
```

- $I \rightarrow L = E$

```

I.Cod =
    L.Cod;
    E.Cod;
    store (L.Address, E.Res)
    
```

- $L \rightarrow \text{var } [E]$

```

L.Cod =
    E.cod;
    temp =
        addr(var) + E.Res * size
L.Address = temp
    
```

Attribute folosite in translatare

- $I \rightarrow \text{if } C \text{ then } I_1 \text{ else } I_2$

```
I.Cod =
    C(xa, xb).Cod;
xa: I1; goto xc
xb: I2
xc:
```

- $C \rightarrow E$

```
C(xtrue, xfalse).Cod =
if E==true goto xtrue
goto xfalse
```

- $C \rightarrow C_1 \text{ and } E$

```
C(xtrue, xfalse).Cod =
    C1(xn, xfalse).Cod;
xn: if E==true goto xtrue
    goto xfalse
```

- $C \rightarrow C_1 \text{ or } E$

```
C(xtrue, xfalse).Cod =
    C1(xtrue, xn).Cod;
xn: if E==true goto xtrue
    goto xfalse
```

Analiza semantica, in practica

- Practic, pe noi ne intereseaza
 - sa adnotam arborele sintactic cu informatia de tip
 - sa construim tabela (tabelele) de simbolii
 - sa modificam arborele (daca e nevoie) prin inserarea de noduri type-cast
- Mare parte din analiza semantica se refera la management-ul contextelor

Contexte (scopes)

- Contextele pastreaza definitiile/declaratiile curente
 - Numele si structura tipurilor
 - Numele si tipul variabilelor
 - Numele, tipul de 'return', numarul si tipul parametrilor pentru functii
- Pe masura ce variabilele/functiile/tipurile etc sunt declarate, sunt adaugate la contextul curent
- Cand variabilele(functii, tipuri) sunt accesate, se verifica definitia din contextul current
- Contextele sunt imbricate

Contexte - exemple

- C++
 - Contextul local (de la declaratie pana la sfarsitul blocului/fisierului)
 - Label-urile – valabile in intreaga functie.
 - Campurile/metodele – valabile in intreaga clasa.
 - Name spaces
- Java
 - Nivele: Package, Class, Inner class, Method
- Name hiding

Contextele si spatiile de nume

- Tipurile si variabilele au spatii de nume diferite in limbaje diferite:
- In C:
 - `typedef int foo; foo foo; // e legal`
 - `int int; // e ilegal – int e cuvant rezervat`
- In Java
 - `Integer Integer = new Integer(4); // e legal`
- Ilegal in C, legal in Java:
 - `int foo(x) { return x+4;}`
 - `int f(){ int foo=3; return foo(foo);}`
 - E totusi nerecomandat chiar daca e legal !!!

Implementarea contextelor

- Se face cu ajutorul tablelor de simbolii
- Actiuni pentru tabela de simbolii:
 - Deschide un context nou.
 - Adauga o pereche "cheie=valoare"
 - Cauta valoarea unei chei, daca sunt mai multe intoarce-o pe cea din contextul cel mai 'recent'
 - Inchide contextul – sterge toate perechile 'cheie=valoare' din context.
- Concret – implementare cu stiva sau hashtable

Implementarea contextelor(2)

- Varianta 1: cu stiva. In fiecare context avem cate o tabela de simbolii. Exista o stiva de contexte deschise, si cautarea unui simbol se face in din varful catre baza stivei
- Varianta2: cu hashtable. Avem o singura tabela de simbolii, in care avem `nume_identificator + nr. context`. La inchiderea unui context, se sterg toti identificatori cu numarul respectiv.

Contexte statice sau dinamice

- Contexte statice – apartenența unui simbol la un context e decisă la compilare
 - Natural in C/C++/Java
 - Pascal - o funcție imbricată în alta funcție poate accesa variabilele locale ale funcției 'mama'.
- Contexte dinamice – decizie la rulare
 - LISP : defvar - se accesează variabile din funcția apelantă

Tipuri

- Tip = setul de valori + operatiile permise pe valorile respective; 3 categorii:
 - Tipuri simple/de baza: int, float, double, char, bool – tipuri primitive, de obicei exista suport hardware direct pentru ele de ex. registri dedicati). Si 'enum' intra aici.
 - Tipuri compuse – array, pointer, struct, union, class, etc. Obtinute prin compunerea tipurilor de baza cu tipuri compuse simple (array/pointer)
 - Tipuri complexe – liste, arbori – de obicei suportate prin biblioteci, nu direct de limbaj

Informatii despre tipuri

- La tipurile de baza, nu avem nevoie de informatie suplimentara (exceptie: enum)
 - Tipurile de baza sunt create 'by default'
 - Variabilele au un pointer la tip
- Tipurile compuse
 - Au nevoie de o lista de nume de campuri, cu tipul lor
 - Poate fi tinuta ca si context!
 - Expresii de tip

Informatii despre tipuri (continuare)

- Array
 - Tipul de baza, numarul de elemente
 - Eventual range-ul indicilor, pentru array-uri declarate static
 - Pentru array-urile multidimensionale – fiecare dimensiune e un nou tip!
- Pointeri
 - Tipul de baza (poate fi tot pointer)
- Adnotari – pe toate tipurile
 - const, restricted, etc.
 - Creaza un nou tip!
 - Sunt si adnotari ce influenteaza doar variabilele (de ex. `'static'`).

Constructii care au tip asociat

- Constantele
- Variabilele
- Functiile
- Expresiile
- Instructiunile
 - De ex. 'if' asteapta o expresie de tip 'bool'
 - Tipul void
- Tipuri+constructii+reguli generale = sistem de tipuri

Limbaje si tipuri

- **Dinamice vs. Statice**
 - Unde se face verificarea de tipuri?
La rulare vs. la compilare.
- **Strongly typed vs. Weakly typed**
 - Ce se întâmplă dacă tipurile nu se potrivesc?
Se emite eroare vs. se face conversie.

Verificarea de tip

- Sinteza
 - Determinarea tipului unei constructii (e.g. expresie) pornind de la tipurile membrilor (subexpresii)
 - Daca f are tipul $S_x \times S_y \times \dots \rightarrow T$ si x are tipul S_x , y are tipul S_y atunci $f(x,y\dots)$ are tipul T
 - Overloading – pentru functii si operatori
- Inferenta
 - Determinarea tipului unei constructii din context.

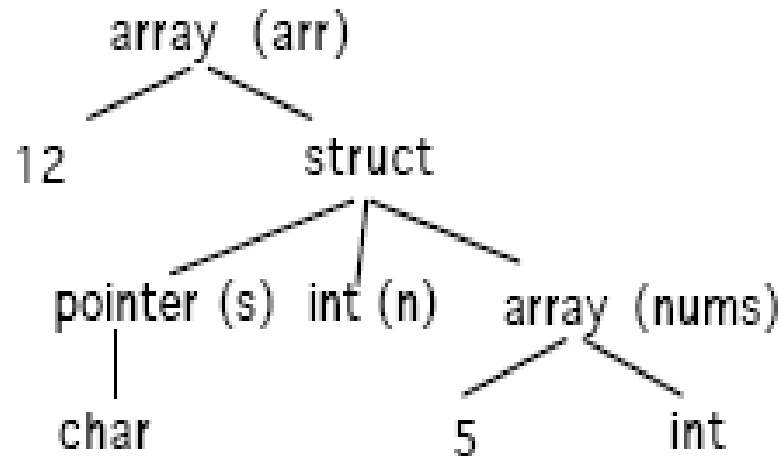
Actiuni din analiza semantica

- Declaratii -> adauga info. in tabela de simbolii; daca nu gaseste tipul, raporteaza eroare
 - Declaratii array – pot produce tipuri noi
- Instructiuni/constructii: verifica regulile specifice fiecărei instructiuni
 - $A=b;$ -> a si b exista? au tipuri compatibile?
- Prototipuri de functii -> ...

Echivalenta tipurilor compuse

- Se tine informatia de tip sub forma de arbore
- Echivalenta – de nume; structurala
- Se verifica recursiv echivalenta pe arbore
 - Atentie la tipuri recursive!

```
struct {  
    char *s;  
    int n;  
    int nums[5];  
} arr[12];
```



Tipuri compatibile, subtipuri

- int compatibil cu double
 - nu neaparat in ambele directii!!
 - Conversii implicite vs. explicite
 - Widening / Narrowing
- Subtip – poate fi folosit oricand in locul tipului 'parinte'
 - Enum in C
 - Mostenire in C++

Tipuri compatibile - variance

- Tipuri generice
- Covariant

```
PrintFullName(IEnumerable<Person> persons) {...}  
Main() { List<Employee> employees = new List<Employee>();  
        PrintFullName(employees);    }
```

- Contravariant

```
Action<Person> printFullName = (target) => { Console.WriteLine(target.Name); };  
Action<Employee> onEmployeeClick = printFullName;
```

- Invariant

```
List<Person> e;  
e = new List<Employee>() // Not allowed  
e.add(new Customer()); // This fails
```

Inferenta de tipuri

- Deducerea tipului unei expresii din context
- La compilare sau la rulare.
- De ce?
 - Verificarea tipurilor
 - Function overloading, generics/templates
 - Introducerea de conversii implicite
 - Declaratii simplificate, tipuri ad-hoc

Inferenta de tipuri

- Function overloading
 - `void f(int) {...}`
`void f(char) {...}`
`f(3.14); // Se pot aplica conversii?`
- Generics / Templates
 - `template<T> f(T a, T* b) {...}`
`int x[]; f(x[0], x);`

Inferenta de tipuri

- Declaratii simplificate, tipuri ad-hoc
 - `map<int,list<string>> m;`
`map<int,list<string>>::iterator` `i = m.begin(); //C++`
`auto` `i = m.begin(); // C++11`
 - `Dictionary<int, string> d = new Dictionary<int, string>();`
`var d = new Dictionary<int, string>();`
 - `var p1 = new { Name = "Lawnmower", Price = 495.00 };`
`var p2 = new { Name = "Shovel", Price = 26.95 };`
`p1 = p2;`
- Se sintetizeaza tipul expresiei din dreapta, se infera tipul expresiei din stanga.

Inferenta functiilor polimorfice

- `fun lungime(lptr) = if null(lptr)
 then 0
 else lungime(tl(lptr)) + 1;`
- Limbaj functional – ML
- Ce tip intoarce functia lungime?
- Null si tl (“tail”) opereaza pe liste.

Expresii de tip

```
lungime:  $\beta$ ; //  $\beta, \gamma$  sunt variabile de tip
lptr :  $\gamma$ ;
if :  $\forall \alpha, \text{boolean} \times \alpha \times \alpha \rightarrow \alpha$ ; // functie polimorfica
null :  $\forall \alpha, \text{list}(\alpha) \rightarrow \text{boolean}$ ;
tl :  $\forall \alpha, \text{list}(\alpha) \rightarrow \text{list}(\alpha)$ 
0 : integer;
1 : integer;
+ : integer  $\times$  integer  $\rightarrow$  integer;
match :  $\forall \alpha, \alpha \times \alpha \rightarrow \alpha$ ;
match (
    lungime(lptr),
    if (null(lptr), 0, lungime(tl(lptr)) + 1)
) // pseudo-operator – sunt tipurile echivalente?
```

Inferenta functiilor polimorfice

Substitutie si unificare

lungime: $\gamma \rightarrow \delta$;

lptr : γ ;

if : $\forall \alpha, \text{boolean} \times \alpha \times \alpha \rightarrow \alpha$;

null : $\forall \alpha, \text{list}(\alpha) \rightarrow \text{boolean}$;

tl : $\forall \alpha, \text{list}(\alpha) \rightarrow \text{list}(\alpha)$

+ : $\text{integer} \times \text{integer} \rightarrow \text{integer}$;

match : $\forall \alpha, \alpha \times \alpha \rightarrow \alpha$;

match (
 lungime(γ),
 if (**boolean, integer**, lungime(tl(γ)) + **integer**)
)

Inferenta functiilor polimorfice

Substitutie si unificare

lungime: $\gamma \rightarrow \delta$;

lptr : γ ;

if : $\forall \alpha, \text{boolean} \times \alpha \times \alpha \rightarrow \alpha$;

tl : $\forall \alpha, \text{list}(\alpha) \rightarrow \text{list}(\alpha)$

+ : $\text{integer} \times \text{integer} \rightarrow \text{integer}$;

match : $\forall \alpha, \alpha \times \alpha \rightarrow \alpha$;

match (

 lungime(γ),

 if (boolean, integer, lungime(tl(γ)) + integer)

)

match(lungime(tl(γ)) + integer , integer)

match(tl(γ) , list (β))

Inferenta functiilor polimorfice

Substitutie si unificare

lungime: $\forall \beta, \text{list}(\beta) \rightarrow \text{integer}$; // Din *if(...)*

lptr : $\text{list}(\beta)$; // Din *tl(...)*

if : $\forall \alpha, \text{boolean} \times \alpha \times \alpha \rightarrow \alpha$;

tl : $\forall \alpha, \text{list}(\alpha) \rightarrow \text{list}(\alpha)$

+ : $\text{integer} \times \text{integer} \rightarrow \text{integer}$;

match : $\forall \alpha, \alpha \times \alpha \rightarrow \alpha$;

match (

 lungime(**list**(β)),

 if (boolean, integer, lungime(**list**(β)) + integer)

)

match(lungime(list(α)), integer) // Din ...+...

Unificare - algoritmul

- Unificare(s,t)
 - daca $(s==t) \rightarrow ok$
 - daca s, t sunt tipuri compuse similare, $s=f(s1,s2)$, $t=f(t1,t2)$
 - Inlocuieste s cu t
 - s si t vor face parte din aceeaasi clasa de echivalenta
 - Unificare(s1,t1) && Unificare(s2,t2);
 - daca s e o variabila \rightarrow inlocuieste s cu t; ok
 - daca t e o variabila \rightarrow inlocuieste t cu s; ok
 - altfel unificarea nu e posibila