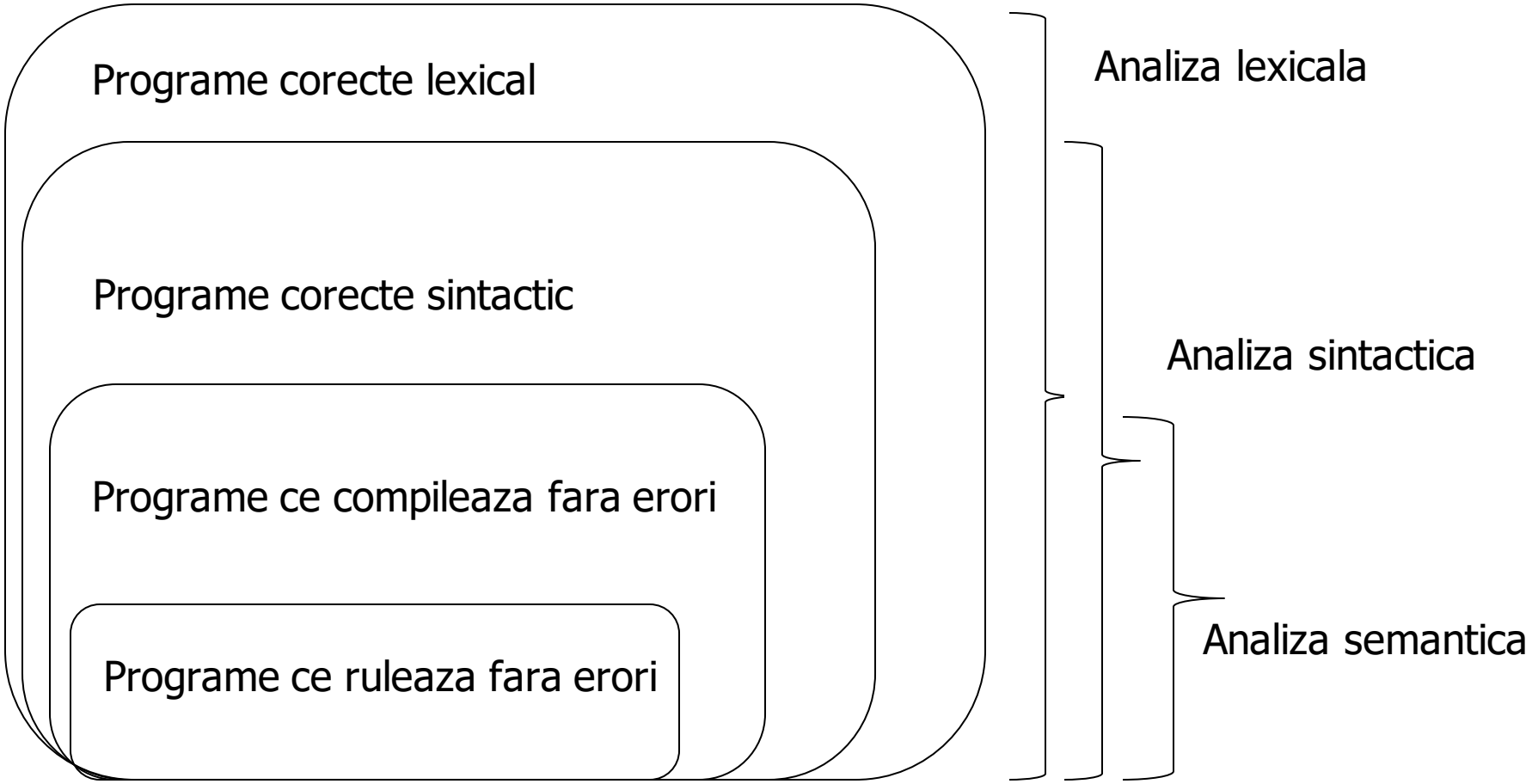
Compilatoare

Curs 4

Analiza semantica

ANALIZA SEMANTICA



CPL
IR crunch

ANALIZA SEMANTICA

- Calculeaza toate atributele asociate nodurilor din arborele sintactic
  - Exemple de atribute: Valoarea unei constante, numele unei variabile, tipul unei expresii
  - Atributele 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)

CPL
IR crunch

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

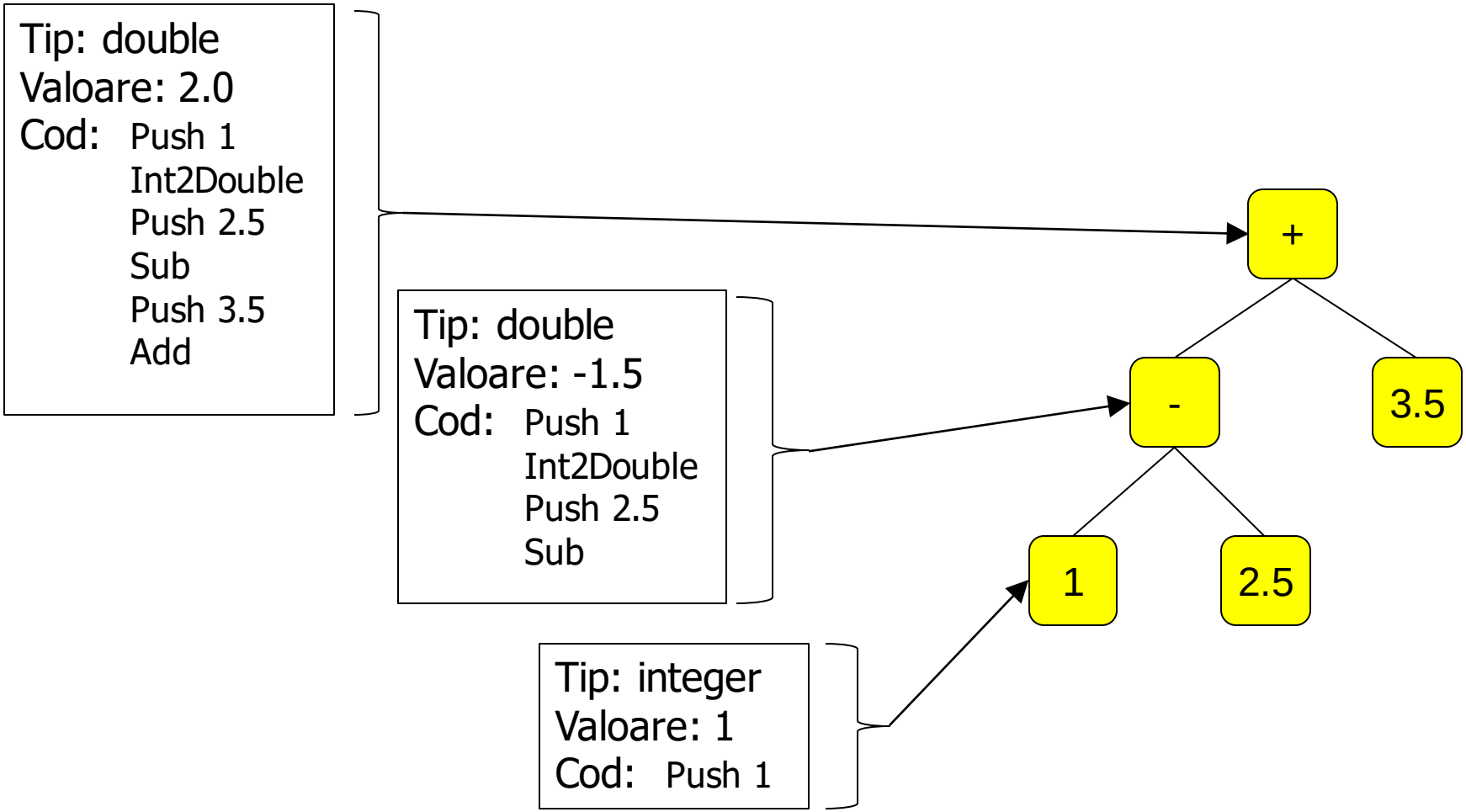
CPL
IR crunch

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.

Exemple de atribute

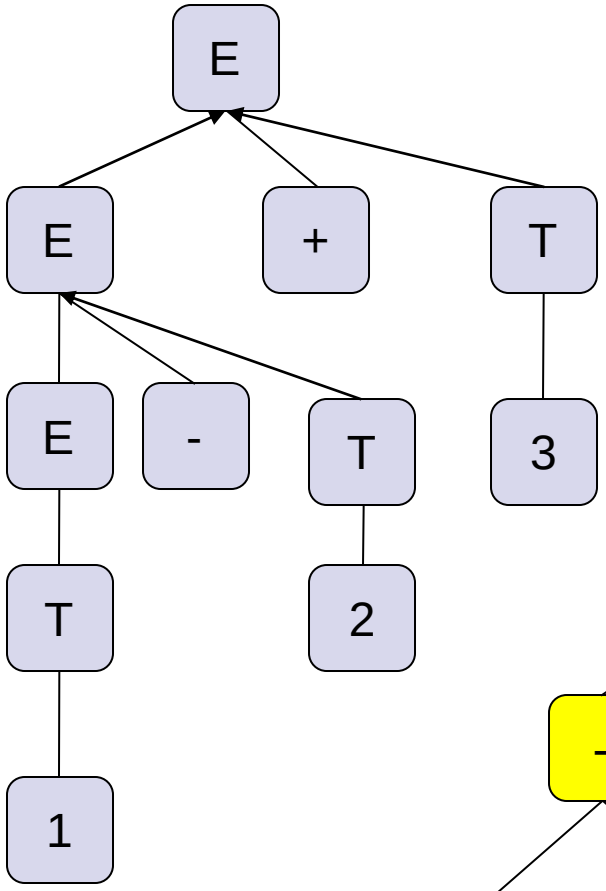


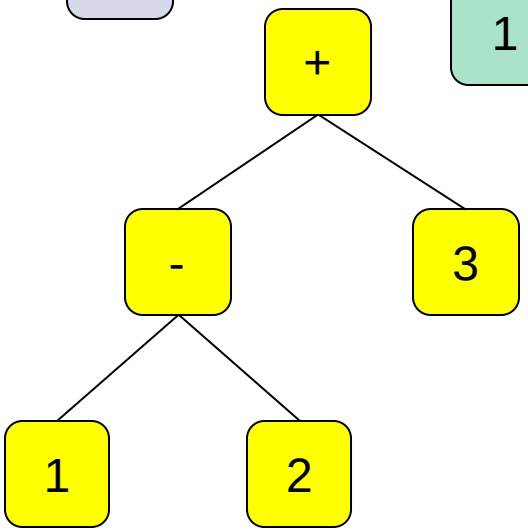
CPL
IR crunch

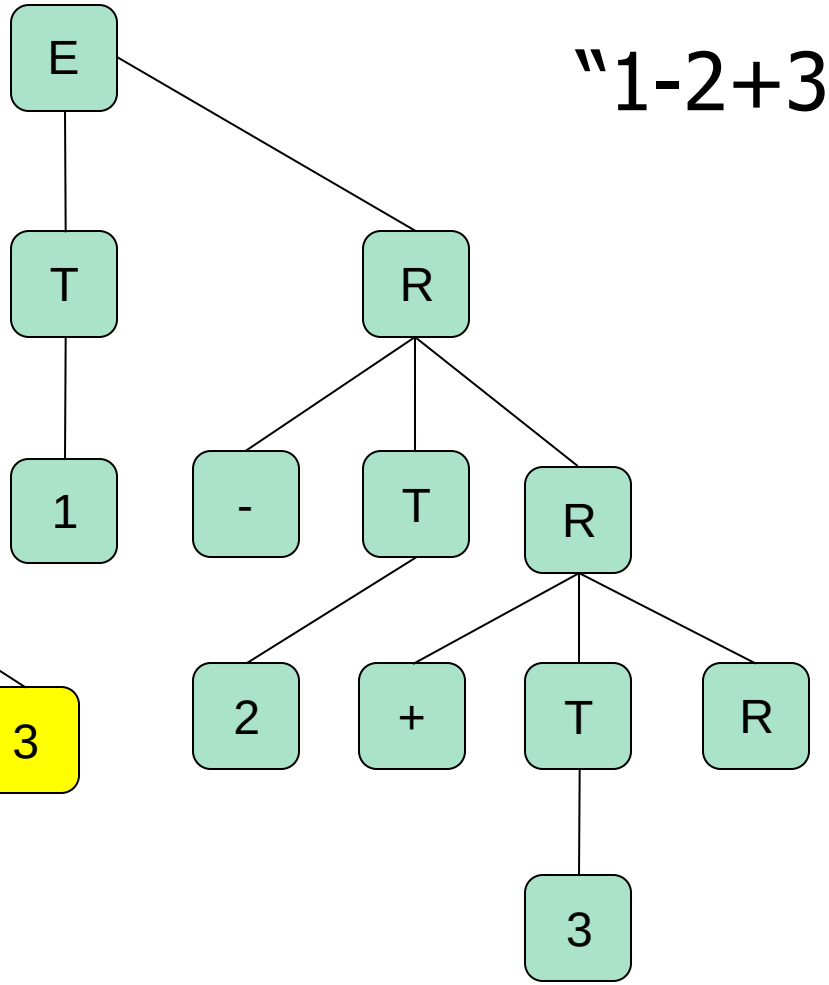
Arbori: derivare vs. sintactic

- 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





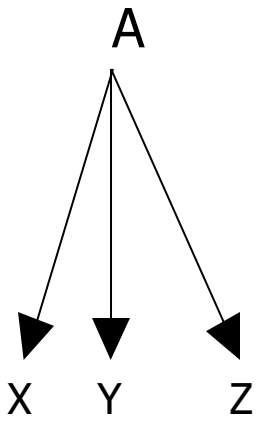


"1-2+3"

Cateva definitii

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

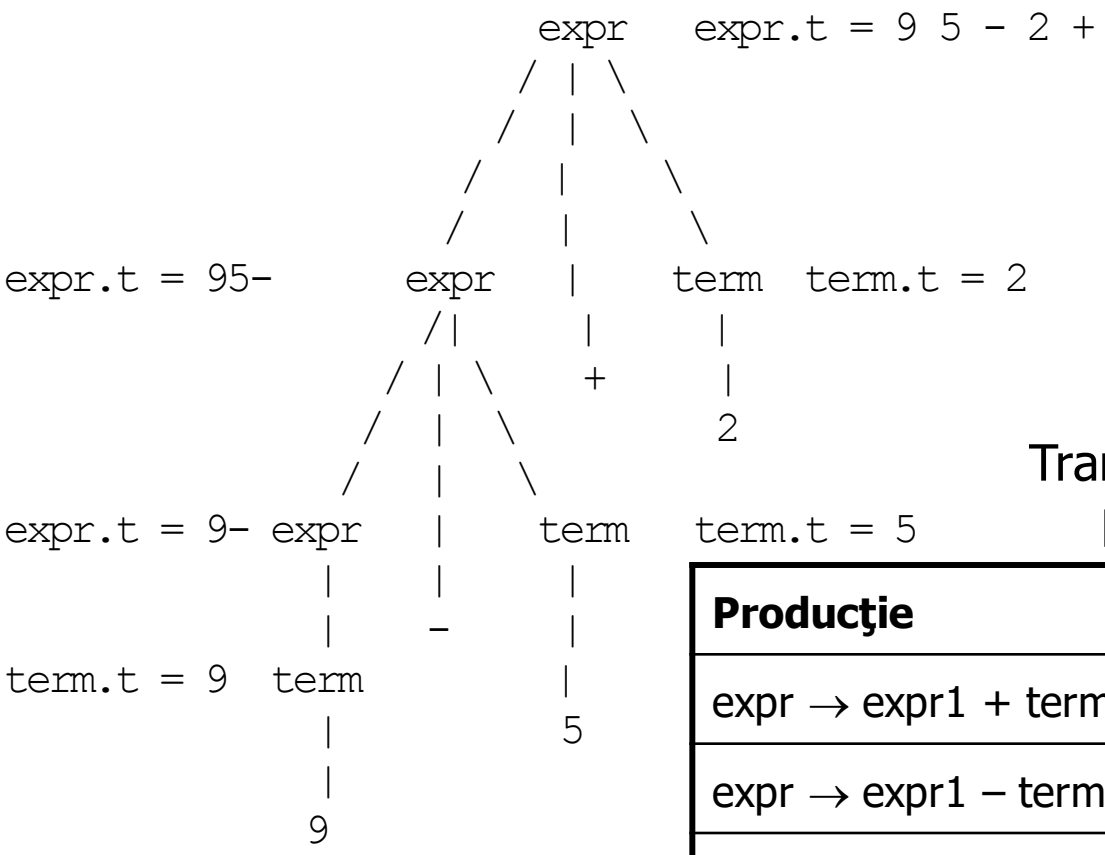
- Fie urmatorul arbore de derivare:
  - A.a = f(X.a, Y.a, Z.a) – atribut sintetizat
  - Y.a = F(A.a, X.a, Z.a) –atribut mostenit



Syntax directed definition

- Dupa ce am stabilit gramatica limbajului
  - Pentru fiecare simbol din gramatică se asociază un atribut (eventual cu mai multe campuri)
  - Pentru fiecare producţie se asociază o mulţime de reguli semantice (cum calculam valoarea atributelor)
- Gramatica + reguli semantice => definitie orientata sintaxa
- Pt o productie A $\rightarrow$ X1... Xk regulile semantice sunt de forma:
  - A.a:= f(X1.a, ..., Xk.a)
  - Xi.a:= f(A.a, X1.a, ..., Xk.a), cu Xi neterminal

Gramatica de atribute



Translatarea expresiilor in forma postfixata

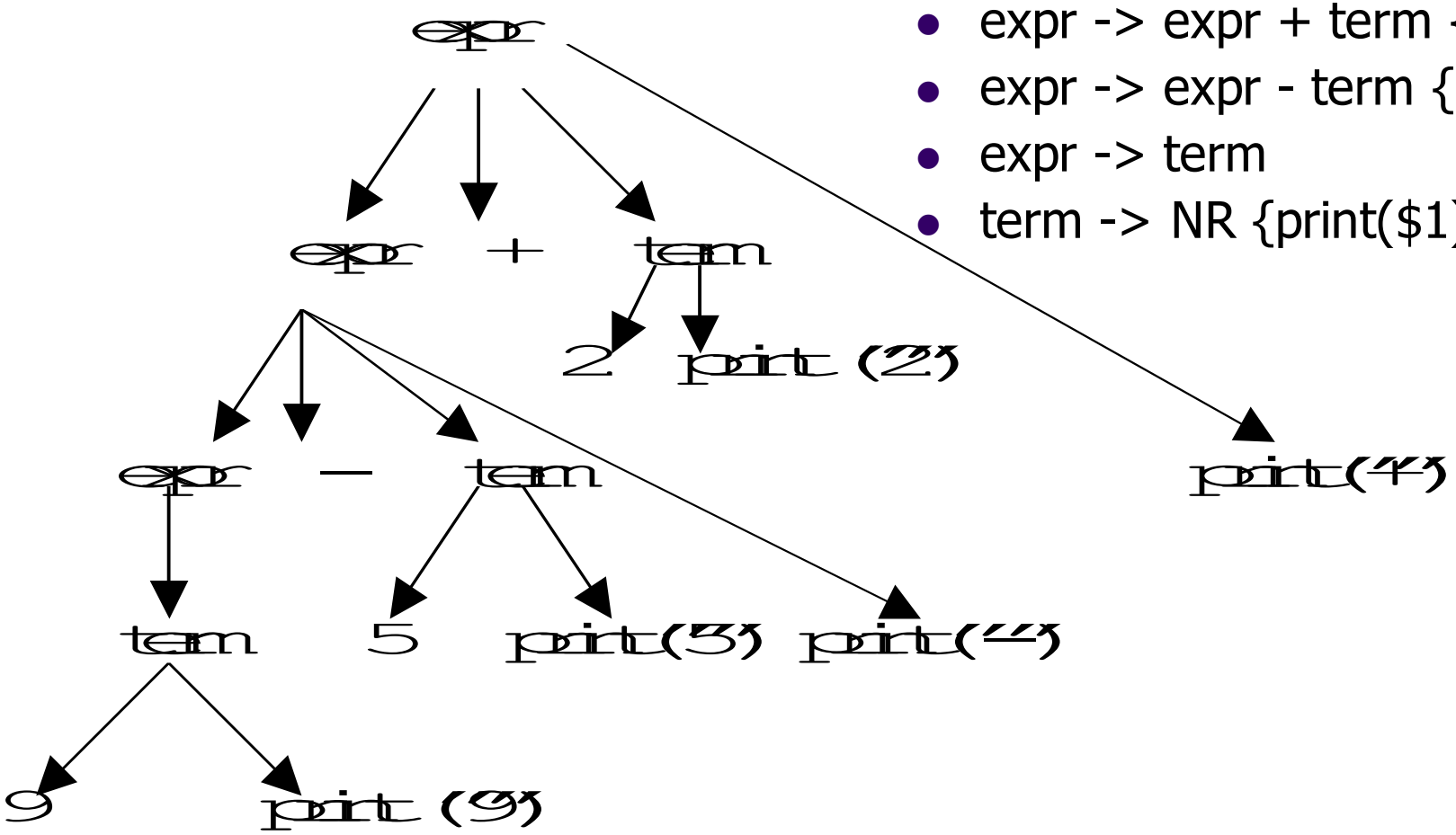
Producţie	**Acţiune**
expr → expr1 + term	expr.t:= expr1.t term.t '+'
expr → expr1 – term	expr.t:= expr1.t term.t '-'
expr → term	expr.t:= term.t
term → 0	term.t:= '0'
…	…

9 – 5 + 2 ➔ 9 5 – 2 +

Scheme de traducere

- O schema de traducere
  - GIC + actiuni semantice (definitie orientata sintaxa)
  - Momentul in care act. semantice sunt executate in timpul parsarii
- O specificare posibilă utilizează fragmente de program reprezentând acţiuni semantice intercalate între simbolii care apar în partea dreaptă a producţiilor:
  - A -> α { print('x')} β
  - se va afişa caracterul 'x' după ce se vizitează subarbo-rele α şi înainte de traversarea subarborelui β.
  - Un nod care reprezintă o acţiune semantica nu are descendenţi iar acţiunea semantică se execută atunci când este întâlnită în parcurgerea arborelui.

Exemplu: 9-5+2



- expr -> expr + term {print('+')}
- expr -> expr - term {print('-')}
- expr -> term
- term -> NR {print($1)}

Graful de dependenta

- Definitiile orientate sintaxa nu precizeaza cand se aplica regulile semantice
  - Dar se precizeaza 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 = atribute
  - Arc n1->n2 : n2 se calculeaza pe baza n1

Calculul atributelor

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

Exemplu

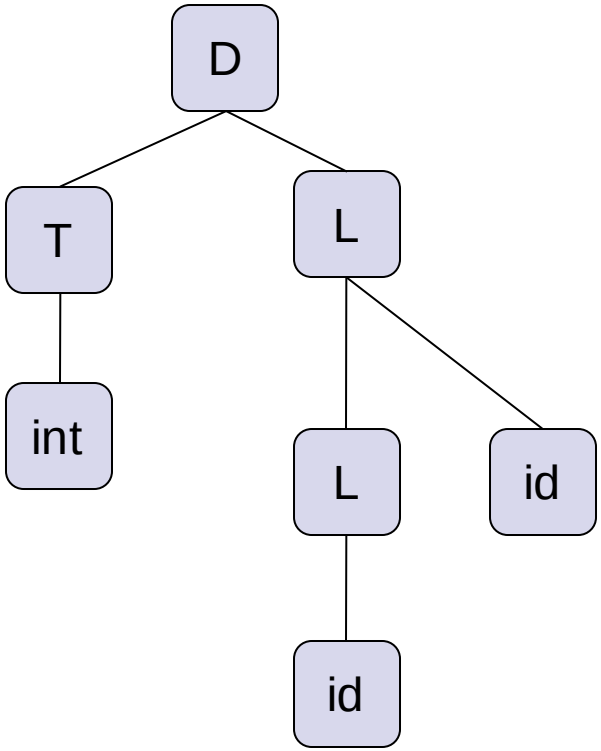
- float a, b, c;
- L.tip depinde de T.tip, addVar() depinde de L.tip
- L.tip, id.nume sunt mostenite
- T.tip e sintetizat

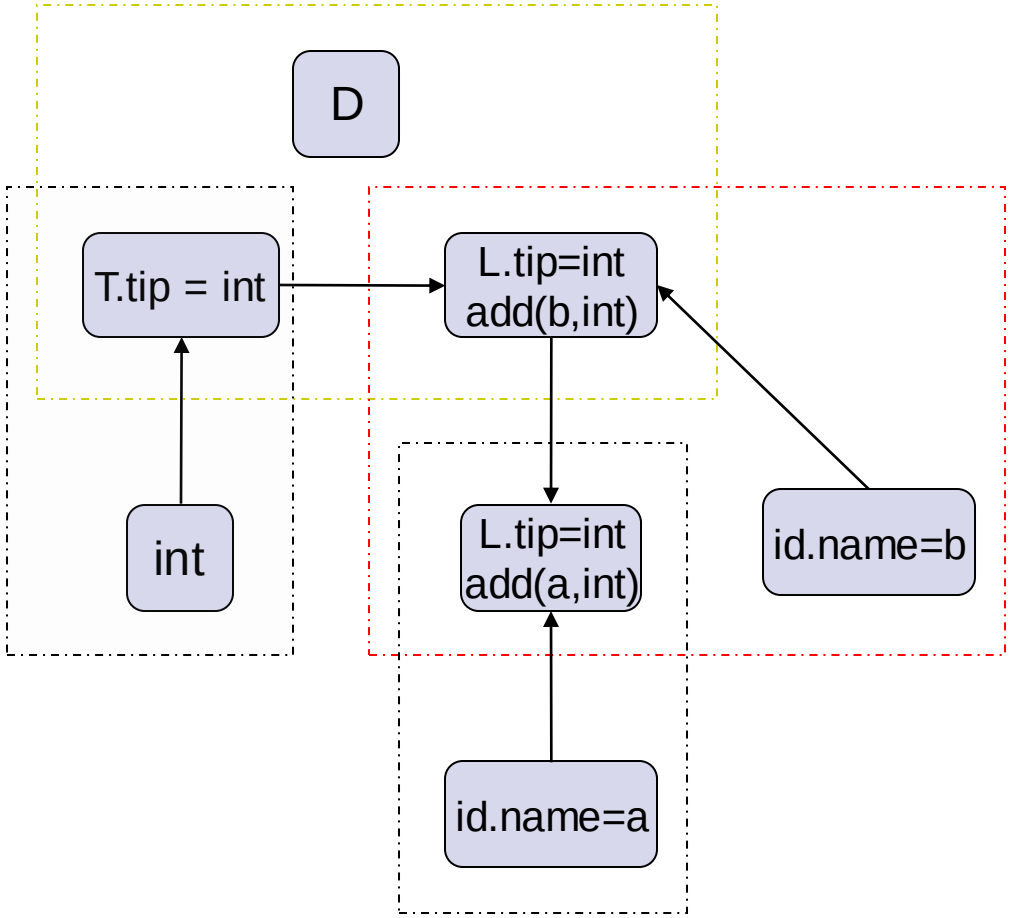
- Definitie orientata sintaxa, nu gramatica de atribute (addVar)

Producţie	Regula semantică (acţiune)
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;"





CPL
IR crunch

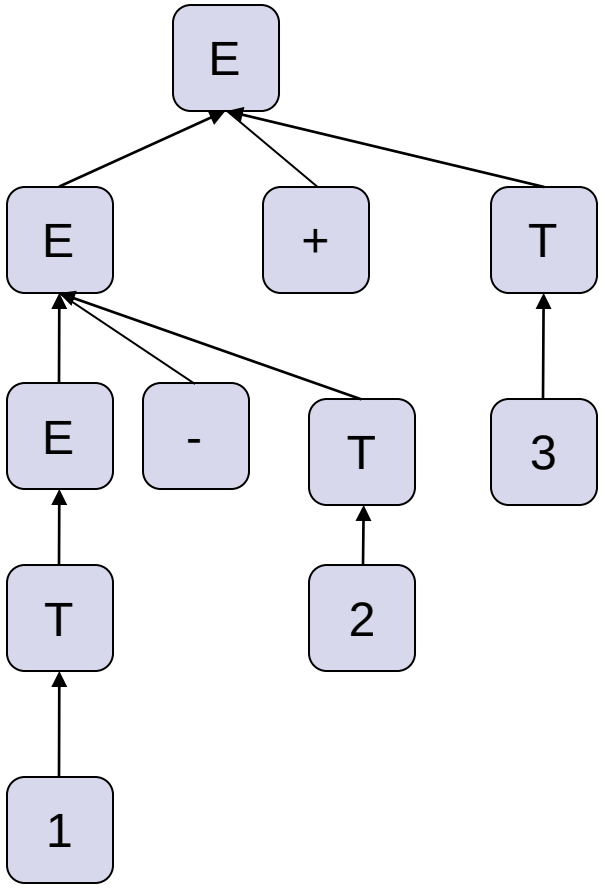
Calculul atributelor

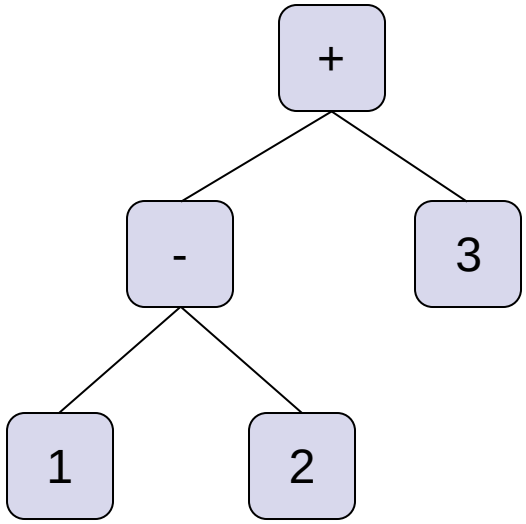
- Dandu-se o definitie orientata sintaxa, este graful necircular pentru orice arbore de derivare?
  - Algoritm exponential in cazul general
  - Se restrictioneaza regulile de calcul ale atributelor
- Evaluare in timpul parsarii
- Algoritmi care garanteaza ordinea de evaluare

Definitii S-atributate

Producţie	Regula semantică (acţiune)
$E \rightarrow E_1 + T$	E.s:= Nod('+', E_1.s, T.s);
$E \rightarrow E_1 - T$	E.s:= Nod('-', E_1.s, T.s);
$E \rightarrow T$	E.s:=T.s
$T \rightarrow num$	T.s:= Nod(num.val)

Definitii S-atributate (cont.)





“1-2+3”

Definitii S-atributate (cont.)

- Doar atribute sintetizate
  - Stiva e 'imbogatita' cu informatii legate de atributele neterminalilor recunoscuti
  - De câte ori se face o reducere, valorile atributelor sintetizate sunt calculate pornind de la atributele care apar în stivă pentru simbolii din partea dreaptă a producţiei.
  - 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:

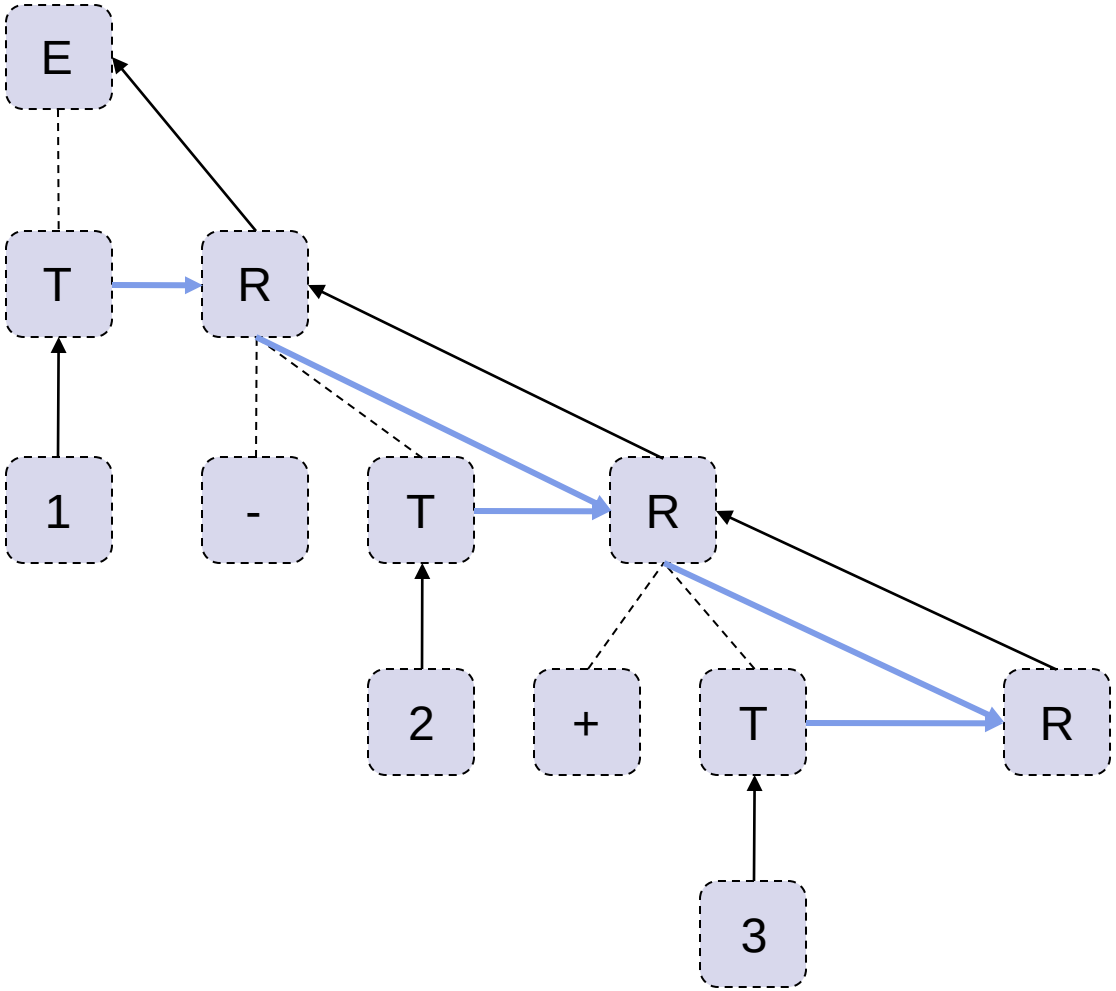
```
switch (state)    {
   case 3: value = stack[top - 2] + stack[top]; break;
   case 4: value = stack[top]; break;
   }
pop(stack, 3); push(stack, value);
```

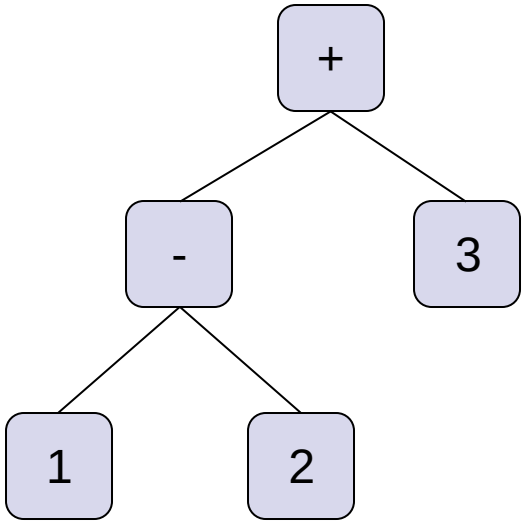
- Care e continutul stivei?
- Ce cod se genereaza pentru shift?

Definitii L-atributate

Producţie	Regula semantică (acţiune)
$E \rightarrow T\ R$	R.m:= T.s; E.s = R.s
$R \rightarrow -\ T\ R_1$	R_1.m:= Nod('- ', R.m, T.s); R.s = R_1.s
$R \rightarrow +\ T\ R_1$	R_1.m:= Nod('+', R.m, T.s); R.s = R_1.s
$R \rightarrow \lambda$	R.s:= R.m
$T \rightarrow$ num	T.s:= Nod(num.val)

Definitii L-atributate (cont.)





"1-2+3"

Definitii L-atributate (cont.)

- Orice atribut calculat printr-o regulă semantică asociată producţiei $A \rightarrow X_1 X_2 \ldots X_n$ este
  - fie sintetizat,
  - fie este un atribut moştenit pentru neterminalul X_j care depinde numai de atributele simbolilor X_1, X_2, ... X_{j-1} şi de atributele moştenite pentru A
- Includ definitiile S-atributate
- Naturale in analiza descendenta

Definitii L-atributate
Analiza descendent recursiva

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

```
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 -> TR

R -> +T { print('+') } R ¦ -T { print('-') } R ¦ λ

T -> numar { print(numar.val) }

- Putem să rescriem schema de traducere sub forma:

E -> TR

E -> +T M R ¦ -T N R ¦ λ

T -> numar {print(numar.val)}

M -> λ {print('+')}

N -> λ {print('-')}

- Ambele scheme de traducere reprezintă aceeaşi gramatică şi toate acţiunile sunt executate în aceeaşi ordine. Prin introducerea unor simboli neterminali suplimentari am reuşit să îndeplinim condiţia de a avea acţiunile semantice la sfârşitul producţiei

Implementare (cont.)

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

Producţie	Regula semantică (acţiune)
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)

Implementare (cont.)

- Solutia anterioara nu e generica.
- T nu mai apare intotdeauna in stiva inaintea lui L
- Putem modifica gramatica:
  - D -> T : X L
  - X -> λ
  - X.tip = T.tip
  - L.tip = T.tip

Producţie	Regula semantică (acţiune)
D-> T : L;	L.tip = T.tip
D-> TL;	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)

Implementare (cont.)

- Probleme daca gramatica nu e LL(1)

A_1->A_2x {A_2.m = f(A_1.m);} | y {y.i=f(A_1.m);}

Introducem neterminali:

A->M1 Ax | M2 y

M1-> λ

M2-> λ

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

CPL
IR crunch

Ce poate contine un atribut?

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

Atribute folosite in translatare

- T → var

```
T.Cod = ϵ
T.Res = var
```

- E → T

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

- E → E + T

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

Atribute folosite in translatare

- I → if E then I_1 else I_2

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

- I → L = E

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

- L → var

```
L.Cod = ∈
L.Address = addr(var)
```

- L → var [E]

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

Atribute folosite in translatare

- I → if C then I_1 else I_2

```
I.cod =
    C(x_a, x_b).cod;
x_a: I_1; goto x_c
x_b: I_2
x_c:
```

- C → E

```
C(x_true, x_false).cod =
if E==true goto x_true
   goto x_false
```

- C → C_1 and E

```
C(x_true, x_false).cod =
   C_1(x_n, x_false).cod;
x_n:if E==true goto x_true
   goto x_false
```

- C → C_1 or E

```
C(x_true, x_false).cod =
   C_1(x_true, x_n).cod;
x_n:if E==true goto x_true
   goto x_false
```

Analiza semantica, in practica

CPL
IR crunch

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

CPL
IR crunch

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

CPL
IR crunch

- 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 !!!

CPL
IR crunch

Implementarea contextelor

- Se face cu ajutorul tablelor de simboli
- Actiuni pentru tabela de simboli:
  - 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)

CPL
IR crunch

- Varianta 1: cu stiva. In fiecare context avem cate o tabela de simboli. 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 simboli, in care avem nume_identificator + nr. context. La inchiderea unui context, se sterg toti identificatori cu numarul respectiv.

Contexte statice sau dinamice

CPL
IR crunch

- Contexte statice – apartenenta unui simbol la un context e decisa la compilare
  - Natural in C/C++/Java
  - Pascal - o functie imbricata in alta functie poate accesa variabilele locale ale functiei ‘mama’.
- Contexte dinamice – decizie la rulare
  - LISP : defvar - se acceseaza variabile din functia apelanta

Tipuri

- Un tip e 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

CPL
IR crunch

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)

CPL
IR crunch

- 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’).

CPL
IR crunch

Verificarea de tip

- Verifica daca operatiile executate respecta sistemul de tipuri al limbajului
- Orice nerespectare – eroare de tip
  - Daca toate erorile de tip pot fi verificate la compilare – limbajul este ‘strongly typed’.
  - Erori minore – conversii implicite
- Verificare de tip
  - Statica – la compilare – C, Pascal
  - Dinamica – la runtime – Perl, Python, Ruby

CPL
IR crunch

Verificarea de tip

- Sinteza
  - Determinarea tipului unei constructii (e.g. expresie) pornind de la tipurile membrilor (subexpresii)
  - Daca f are tipul S_X x S_Y x … → T
    si x are tipul S_X , y are tipul S_Y
    atunci f(x,y…) are tipul T
  - Overloading – pentru functii si operatori
- Inferenta
  - Determinarea tipului unei constructii din context.

CPL
IR crunch

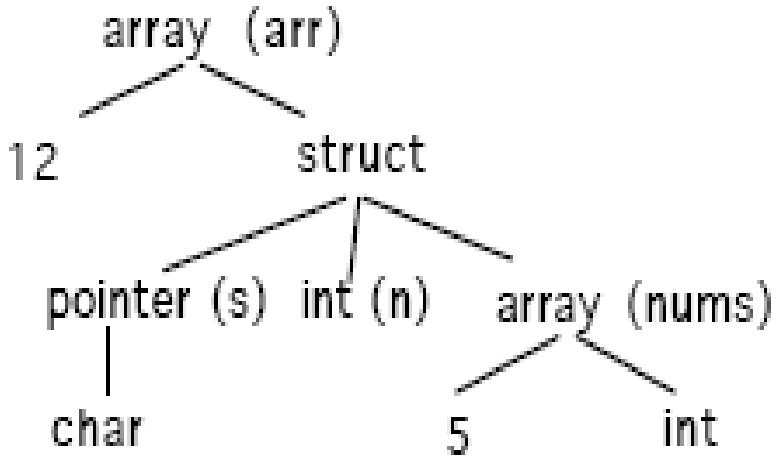
Actiuni din analiza semantica

- Declaratii -> adauga info. in tabela de simboli; daca nu gaseste tipul, raporteaza eroare
  - Declaratii array – pot produce tipuri noi
- Instructiuni/constructii: verifica regulile specifice fiecarei 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];
```



CPL
IR crunch

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
```

Constructii care au tip asociat

CPL
IR crunch

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

CPL
IR crunch

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);

CPL
IR crunch

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: ß; // ß, γ *sunt variabile de tip*
lptr : γ;
if : $\forall\alpha$, boolean x α x $\alpha \rightarrow\alpha$; //*functie polimorfica*
null : $\forall\alpha$, list(α)$\rightarrow$ boolean;
tl : $\forall\alpha$, list(α)$\rightarrow$ list(α)
0 : integer;
1 : integer;
+ : integer x integer $\rightarrow$ integer;
match : $\forall\alpha$, α x $\alpha \rightarrow \alpha$;
match (
lungime(lptr),
if (null(lptr), 0, lungime(tl(lptr)) + 1)
) // *pseudo-operator – sunt tipurile echivalente?*

Inferenta functiilor polimorfice
Substitutie si unificare

CPL IR crunch

lungime: $\gamma \rightarrow \delta$;
lptr : γ;
if : $\forall\alpha$, boolean x α x $\alpha \rightarrow\alpha$;
null : $\forall\alpha$, list(α)$\rightarrow$ boolean;
tl : $\forall\alpha$, list(α)$\rightarrow$ list(α)
+ : integer x integer $\rightarrow$ integer;
match : $\forall\alpha$, α x $\alpha \rightarrow \alpha$;
match (
lungime(γ),
if (**boolean**, **integer**, lungime(tl(γ)) + **integer**)
)

Inferenta functiilor polimorfice
Substitutie si unificare

lungime: $\gamma \rightarrow \delta$;
lptr : γ;
if : $\forall\alpha$, boolean x α x $\alpha \rightarrow\alpha$;
tl : $\forall\alpha$, list(α)$\rightarrow$ list(α)
+ : integer x integer $\rightarrow$ integer;
match : $\forall\alpha$, α x $\alpha \rightarrow \alpha$;
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

CPL
IR crunch

lungime: $\forall \beta$, list(β) $\rightarrow$ integer ; *// Din if(...)*
lptr : list(β); *// Din tl(...)*
if : $\forall\alpha$, boolean x α x $\alpha \rightarrow\alpha$;
tl : $\forall\alpha$, list(α)$\rightarrow$ list(α)
+ : integer x integer $\rightarrow$ integer;
match : $\forall\alpha$, α x $\alpha \rightarrow \alpha$;
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) → 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 aceeasi clasa de echivalenta
    - Unificare(s1,t1) && Unificare(s2,t2);
  - daca s e o variabila → inlocuieste s cu t; ok
  - daca t e o variabila → inlocuieste t cu s; ok
  - altfel unificarea nu e posibila