Limbaje formale și tehnici de compilare

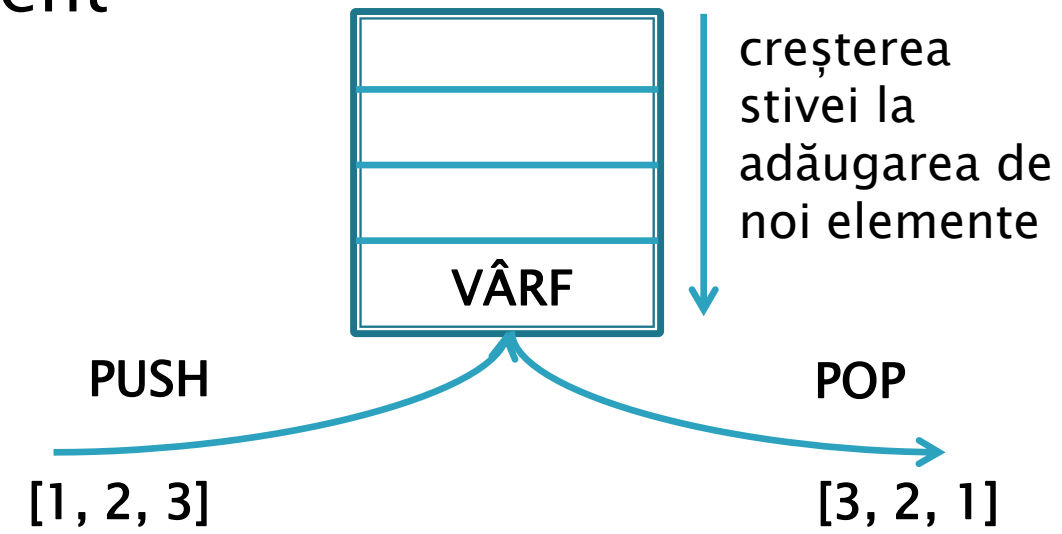
Curs 11: mașini virtuale bazate pe stivă: structură, generare de cod, implementare

Mașini virtuale (MV)

- O MV modelează un calculator virtual (CPU, memorie…)
- Dacă un compilator emite cod pentru o MV, atunci codul emis va fi independent de sistemul gazdă, existând doar necesitatea de a exista MV instalată pe el (compile once, run everywhere)
- Deoarece trebuie emulate instrucțiunile MV folosind CPU-ul gazdă, codul MV va fi mai lent decât codul mașină echivalent pentru CPU
- Se poate evita emularea traducând instrucțiunile MV în cod mașină atunci când e nevoie să se execute (JIT – Just In Time compiling) sau la instalarea/prima rulare a programului (AOT – Ahead Of Time). Pentru AOT se pot aplica mai multe optimizări codului mașină, deoarece nu există restricții de timp limită privind compilarea, ca la JIT, unde compilarea e inclusă în timpul de execuție

Stiva

- O structură de date care implementează o colecție de elemente, cu proprietatea că la extragerea unui element, întotdeauna se va extrage ultimul element introdus (LIFO – Last In, First Out)
- **Vârful stivei** – locul prin care se adaugă și se extrag elemente din stivă
- **PUSH** – adăugare element
- **POP** – extragere element



Implementarea unei stive

- În general stiva se implementează cu ajutorul unui vector de dimensiune fixă sau variabilă. O variabilă index se folosește pentru a indica ultimul element introdus (vârful stivei).
- Se pot implementa stive și folosind liste simplu înlănțuite, cu adăugare la început de listă (considerat vârful stivei)

Operație	Efect	Stivă inițială → finală (vârf stivă la dreapta)
PUSH x	adaugă valoarea x în stivă	… → … x
POP var	extrage ultima valoare din stivă și o depune în variabila dată	… x → … ; var=x
PEEK var	depune ultima valoare din stivă în variabila dată, fără a o elimina din stivă	… x → …x ; var=x
DROP	șterge ultima valoare din stivă	… x → …
DUP	duplică ultima valoare din stivă	… x → …x x

Mașini virtuale bazate pe stivă

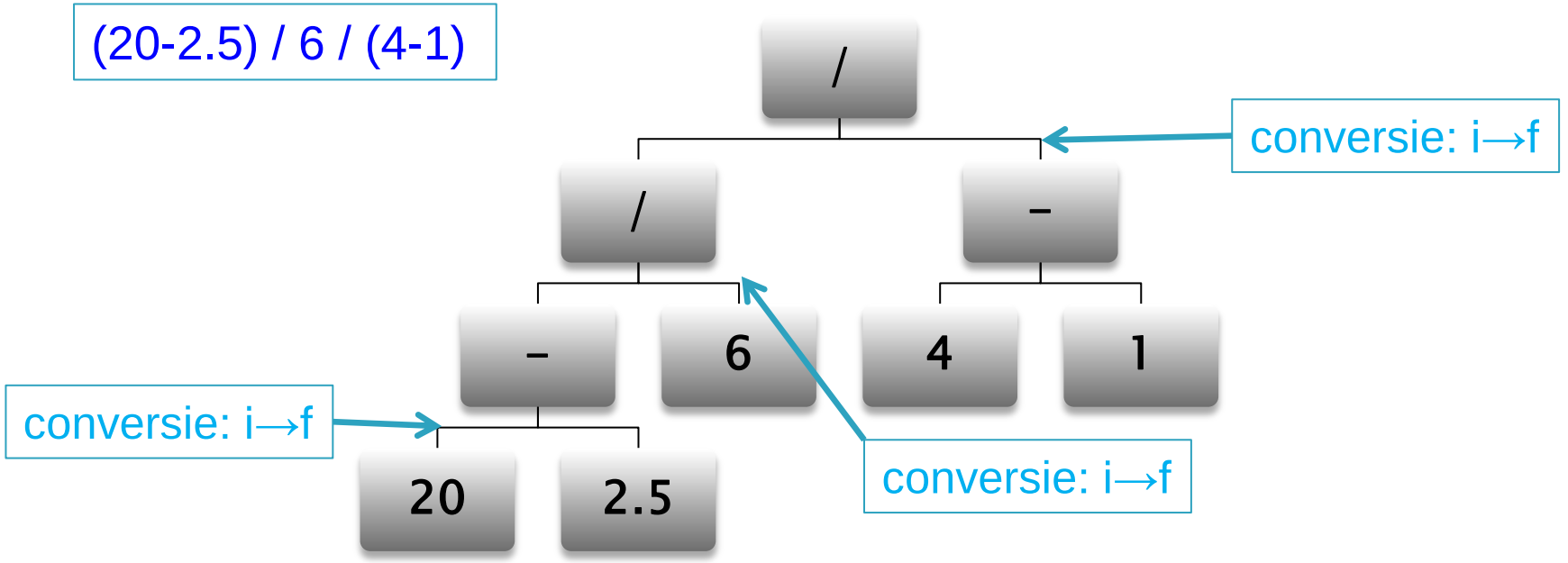
- Folosesc una sau mai multe stive pentru efectuarea operațiilor, păstrarea variabilelor locale și pentru apelurile de funcții
- Sunt simplu de implementat
- Generarea de cod pentru ele este simplă
- Exemple: JVM (Java Virtual Machine), CLR (Common Language Runtime – mașina virtuală pentru .NET)
- Operațiile (aritmetice, logice, conversii, …), apelurile de funcții, instrucțiunile (ex: if) întotdeauna vor avea argumentele pe stivă, iar rezultatul va rămâne tot pe stivă
- Valorile din stivă pot avea tipuri diferite, dar trebuie să ocupe un număr întreg de celule de stivă

Implementarea operațiilor aritmetice

- Deoarece se poate opera cu tipuri diferite, reprezentate diferit în memorie, trebuie să existe operații distincte pentru fiecare tip de date
- Convenții:
  - o literă mică cu punct după codul operației specifică tipul operanzilor (**i** – întreg, **f**–floating point)
  - constantele de un anumit tip, vor începe cu litera tipului

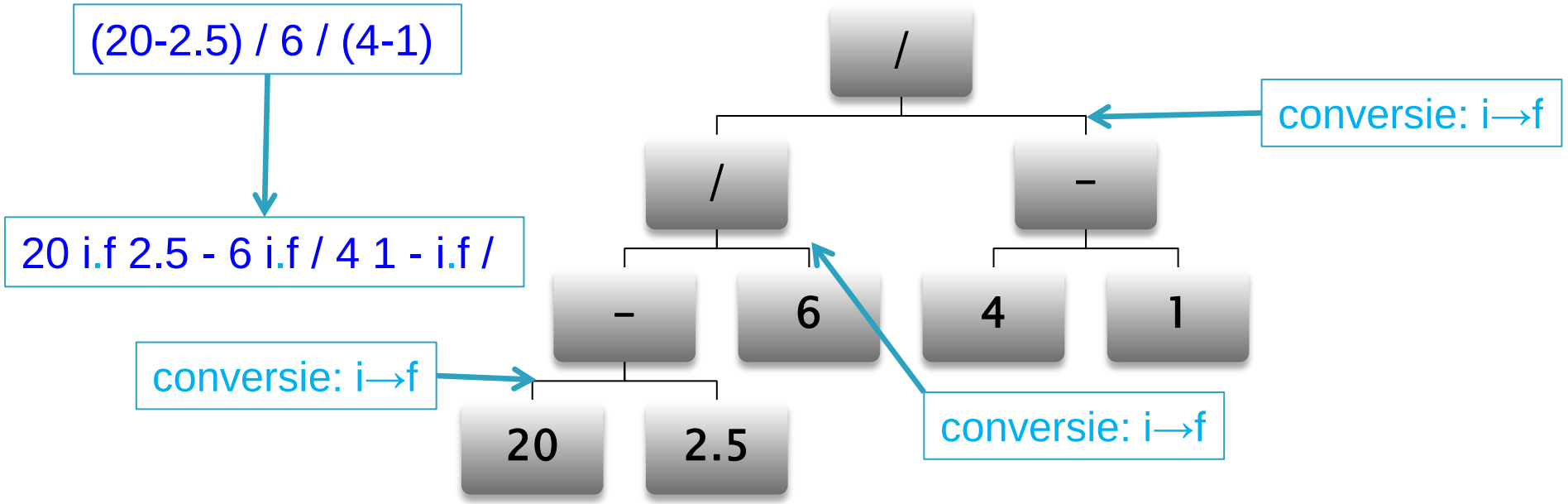
Operație	Stivă inițială → finală
ADD.i	… i_1 i_2 → … (i_1 + i_2)
DIV.f	… f_1 f_2 → … (f_1 / f_2)
CONV.i.f	… i_1 → … f_2
AND – doar pentru întregi	… i_1 i_2 → … (i_1 && i_2)
BITAND – doar pentru întregi	… i_1 i_2 → … (i_1 & i_2)

Construcția ASA pentru o expresie



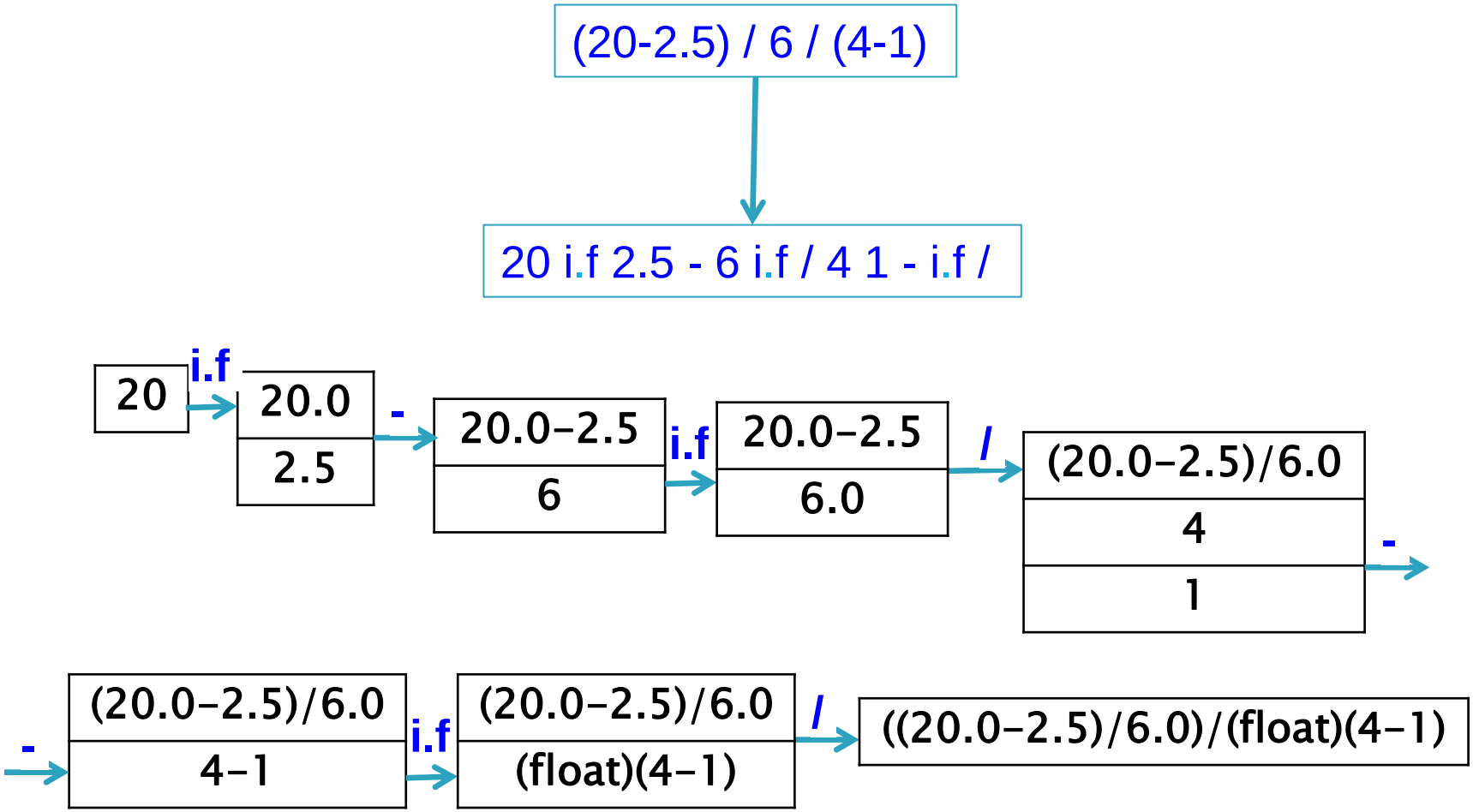
- Se consideră operația care se execută ultima și se pune ca nod curent
- Fii nodului curent vor fi subexpresiile stânga și dreapta
- Dacă subexpresiile sunt operanzi, aceștia vor deveni frunze
- Dacă subexpresiile sunt operatori, se repetă algoritmul până când se epuizează toate operațiile
- Unde este necesar, se inserează conversii de tipuri în arbore, astfel încât operațiile să aibă operanzi de același tip

Forma postfixată a unei expresii



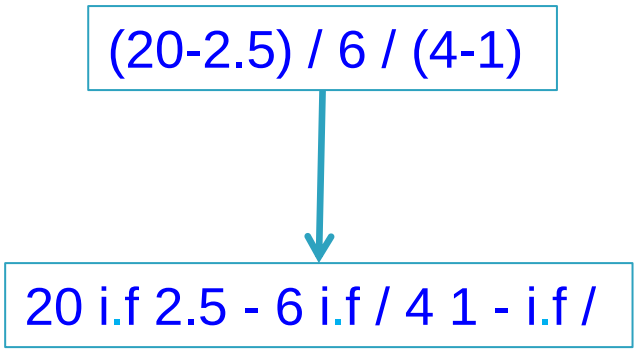
- Forma postfixată (poloneză), se obține prin parcurgerea în postordine a ASA
- O proprietate importantă a acestei forme este faptul că folosind o stivă, se poate evalua expresia de la stânga la dreapta, după următorul algoritm:
  - dacă se întâlnește un operand, se depune în stivă
  - dacă se întâlnește un operator, se extrag din stivă operanzii necesari, se face operația și rezultatul se depune înapoi în stivă

Evaluarea expresiei folosind o stivă



- vârful stivei este în partea de jos

Codul pentru expresii




```
PUSH.i    20
CONV.i.f
PUSH.f    2.5
SUB.f
PUSH.i    6
CONV.i.f
DIV.f
PUSH.i    4
PUSH.i    1
SUB.i
CONV.i.f
DIV.f
SHOW.f
```

- **PUSH.*tip*** **ct.*tip*** – depune pe stivă constanta cu tipul dat

Algoritm generare de cod pentru expresii

- Fiecare regulă gramaticală care generează o valoare (expr, factor, …):
  - va avea un atribut sintetizat ”**tip**”, care returnează tipul valorii generate (deoarece întotdeauna rezultatul expresiilor va fi pe stivă, nu este nevoie să se specifice și locația sa)
  - își va genera codul corespunzător atomilor consumați, astfel încât la terminarea regulii codul va fi deja generat
- **tipRezultat=combine(tip1,tip2)** – returnează tipul necesar pentru un operator care are operanzii de tipurile specificate (ex: int*float → float)
- **idx=posInstr()** – returnează indexul la care va fi generată următoarea instrucțiune

Generare de cod pentru expresii

```
enum Tip = { TipInt, TipFloat }

calculator = ( expr[t] {addShow(t);} ) * FINISH
expr[out tip:Tip] = expr[t1] {idx=posInstr();}
    ( ADD termen[t2] {/*1*/}
    | SUB termen[t2] {/*2*/} ) | termen[tip]
termen[out tip:Tip] = termen[t1] {idx=posInstr();}
    ( MUL factor[t2] {/*3*/}
    | DIV factor[t2] {/*4*/} ) | factor[tip]
factor[out tip:Tip] = INT[n] {addPush(TipInt,n.i);tip=TipInt;}
    | FLOAT[n] {addPush(TipFloat,n.f);tip=TipFloat;}
    | LPAR expr[tip] RPAR
```

```
// 1-ADD, 2-SUB, 3-MUL, 4-DIV
tip=combine(t1,t2);
setConversie(idx,t1,tip);
setConversie(posInstr(),t2,tip);
addDiv(tip);        // 1, 2, 3, 4
```

- **setConversie(idx,tipSrc,tipDst)** – dacă tipSrc!=tipDst, inserează la poziția idx conversia necesară (tipSrc→tipDst)

Codul pentru variabile și atribuiri

```
int n;
float x;
x=n*2.5;
```

```
LOAD.i    n
CONV.i.f
PUSH.f    2.5
MUL.f
STORE.f   x
```

- **LOAD.***tip addr* – ia valoarea de la adresa de memorie specificată și o depune în stivă
- **STORE.**tip addr – ia valoarea din vârful stivei și o depune la adresa de memorie specificată

Codul pentru instrucțiunea if

```
if(expr){
    // instructiuniTrue
    }
```

```
            // cod expresie
            JF L1
            // cod instructiuniTrue
L1:
```

```
if(expr){
    // instructiuniTrue
    }else{
    // instructiuniFalse
    }
```

```
            // cod expresie
            JF L1
            // cod instructiuniTrue
            JMP L2
L1:         // cod instructiuniFalse
L2:
```

- **JT *eticheta*** – ia valoarea de pe stivă și dacă e **true** sare la eticheta specificată (*Jump if True*)
- **JF *eticheta*** – ia valoarea de pe stivă și dacă e **false** sare la eticheta specificată (*Jump if False*)
- **JMP *eticheta*** – salt necondiționat la eticheta specificată
- **JT** și **JF** folosesc doar valori de tip **int**, deci rezultatul *expr* trebuie să fie de tip **int**

Generare cod pentru instrucțiunea if

```
instrIf = IF LPAR expr[tip] RPAR {
            addConv(tip,TipInt);
            idxJF=posInstr();
            addJF(0);          // se va seta cu indexul de la else sau de dupa if
            }
    block
    ( ELSE {
            idxJmp=posInstr();
            addJmp(0);                    // se va seta cu indexul de dupa if
            setJIdx(idxJF,posInstr());
            }

    block {setJIdx(idxJmp,posInstr());}
    | ε {setJIdx(idxJF,posInstr());}
    )
```

```
            // cod expresie
            JF L1
            // cod instructiuniTrue
            JMP L2
L1:         // cod instructiuniFalse
L2:
```

Codul pentru instrucțiunea while

```
while(expr){
    // instructiuni
    }
```

```
L1:     // cod expresie
        JF L2
        // cod instructiuni
        JMP L1
L2:
```

```
int n;
while(n>0){
    n=n-1;
    }
```

```
L1:     LOAD.i   n
        PUSH.i   0
        GT.i
        JF L2
        LOAD.i   n
        PUSH.i   1
        SUB.i
        STORE.i
        JMP L1
L2:
```

Apelurile de funcții

- O funcție poate avea simultan mai multe apelări ale ei în memorie
- Exemplu: `int fact(int n){return n<3 ? n : n*fact(n-1);}`
- Dacă apelăm *fact(4)*, în interiorul funcției trebuie să se calculeze prima oară *fact(3)*, deci se va face acest apel. La fel, pentru calculul lui *fact(3)*, trebuie calculat *fact(2)*. Rezultă că vor fi simultan în memorie apelurile pentru *fact(4)*, *fact(3)* și *fact(2)*.
- Deoarece pot exista simultan mai multe apeluri, fiecare cu valori distincte pentru argumente și variabile locale, nu este suficientă o singură locație pentru stocarea acestora, ca în cazul variabilelor globale
- Din acest motiv, valorile locale unei funcții (argumente și variabile locale) se implementează folosind stiva și un index special în stivă, numit **FP** (frame pointer)
- **Cadrul funcției** (function frame) – toate valorile locale, adresa de revenire și FP-ul funcției apelante
- Valorile din cadrul funcției se accesează folosind FP

Cadrul unei funcții

```
int min(int x, int y){
    int r;
    if(x<y)r=x;
        else r=y;
    return r;
    }
```

```
// g=min(k,108);
            LOAD.i   k
            PUSH.i   108
            CALL     min
ret_min:    STORE.i  g
```

Index față de FP	Valoare	Observații	
-3	x	primul arg (k)	
-2	y	al doilea arg (108)	
-1	ret_addr	adresa de revenire (ret_min)	
0	old_FP	vechiul FP	← FP
1	r	var locală	← SP

- **CALL *nume*** – apelează funcția cu numele dat
- La revenirea din funcție, execuția va continua cu instrucțiunea de după CALL
- Se respectă convenția de la operatori: toate argumentele trebuie puse pe stivă înainte de CALL, iar funcția le va înlocui pe stivă cu valoarea returnată:

$$arg_1 \ldots arg_N \rightarrow f(arg_1 \ldots arg_N)$$

Etapele unui apel de funcție

- Se depun toate argumentele pe stivă
- **Se apelează funcția** folosind **CALL**:
  - Depune pe stivă adresa de revenire (adresa următoarei instrucțiuni de după CALL)
  - Face JMP la codul funcției
- **La intrarea în funcție**:
  - Se salvează vechiul FP în stivă, apoi în FP se va pune indexul vârfului stivei
  - Se adună la vârful stivei numărul de variabile locale, pentru a se face loc pentru ele după vechiul FP
- **Revenirea din funcție** se face cu instrucțiunea **RET**:
  - Reface FP la valoarea anterioară apelului
  - Șterge din stivă tot cadrul funcției
  - În locul cadrului pune valoarea care era în vârful stivei. Aceasta va deveni valoarea returnată de funcție

Codul funcției

```
int min(int x, int y){
    int r;
    if(x<y)r=x;
        else r=y;
    return r;
    }
```

FP	Valoare
-3	x
-2	y
-1	ret_addr
0	old_FP
1	r

```
// funcția min
        ENTER     1          // nr var locale
        FPLOAD.i  -3         // PUSH.i FP[-3]
        FPLOAD.i  -2
        LESS.i
        JF        L1
        FPLOAD.i  -3
        FPSTORE.i 1          // POP.i FP[-3]
        JMP L2
L1:     FPLOAD.i  -2
        FPSTORE.i 1
L2:     FPLOAD.i  1
        RET       2          // nr argumente
```

- **ENTER** ***nr_var_locale*** – salvează vechiul FP în stivă; SP→FP; SP+=nr_var_locale
- **RET** ***nr_argumente*** – reface vechiul FP; șterge cadrul funcției; pune în vârful stivei de după ștergerea cadrului valoarea din vârful stivei de dinainte de ștergere; continuă execuția cu instrucțiunea de la adresa de revenire

Integrarea apelurilor de funcții în expresii

```
int a;
int max(int x, int y);
void show_i(int x);

show_i(3*max(a,7)-1.5);
```

```
PUSH.i   3
LOAD.i   a
PUSH.i   7
CALL     max
MUL.i
CONV.i.f
PUSH.f   1.5
SUB.f
CONV.f.i
CALL     show_i
```

- Funcțiile reprezintă o extindere a noțiunii de operator, acționând asupra a **n** argumente (de **aritate** n)
- La apelul funcției se depun pe stivă argumentele, de la stânga la dreapta, iar apoi se realizează apelul

Implementare MV stivă – structuri de date

```
typedef union _Val{
    int i;                  // nr int, index in instructiuni pentru CALL, RET
    float f;                // nr float
    struct _Val *addr;      // pentru salvarea adreselor in stiva
    }Val;
```

```
typedef enum{SUB_F, LOAD_I, STORE_I, JF,  CALL, ENTER, RET} Code;
typedef struct{
    Code code;
    union{
        Val v;
        Val *addr;       // pentru LOAD, STORE
        int idx;         // pentru JF, JT, FPLOAD, FPSTORE, CALL, RET, ENTER
        }arg;
    }Instr;
```

```
Val stack[];        // stiva
Val *SP;            // Stack Pointer - pointer la vârful stivei
Val *FP;            // Frame Pointer - pointer la locația unde s-a salvat vechiul FP
Instr instructions[];    // vectorul de instrucțiuni
Instr *IP;               // Instruction Pointer - pointer la instrucțiunea curentă
```

Implementare MV stivă – cod

```
switch(IP->code){
    case SUB_F:f2=popf();f1=popf();pushf(f1-f2);IP++;break;
    case LOAD_I:pushi(*IP->arg.addr);IP++;break;
    case STORE_I:*IP->arg.addr=popi();IP++;break;
    case JF:if(popi())IP++;else IP=instructions+IP->arg.idx;break;
    case CALL:
        pushi(IP-instructions+1);
        IP=instructions+IP->arg.idx;
        break;
    case ENTER:
        pusha(FP);      // pusha - pune pe stiva o adresa
        FP=SP;
        SP+=IP->arg.idx;
        IP++;
        break;
    case RET:
        v=popv();       // popv - ia de pe stivă o celulă (Val)
        SP=FP - IP->arg.idx - 2; // -(args, old_FP, ret_addr)
        IP=instructions+FP[-1].i;
        FP=FP[0].addr; pushv(v);
        break;
```

```
typedef struct{
    int i;
    float f;
    struct _Val *addr;
    }Val;

typedef struct{
    Cod cod;
    union{
        Val v;
        Val *addr;
        int idx;
        }arg;
    }Instr;
```