

Capitolul 5

Macroinstrucțiuni

5.1 Scopul macroinstrucțiunilor. Definire și expandare

Macroinstrucțiunile permit programatorului să definească simbolic secvențe de program (instrucțiuni, definiții de date, directive etc.), asociate cu un nume. Folosind numele macroinstrucțiunii în program, se va genera întreaga secvență de program. În esență, este vorba de un proces de substituție (expandare) în textul programului sursă, care se petrece înainte de asamblarea programului. Un asamblor care dispune de macroinstrucțiuni se numește macroasamblor.

Sunt două etape de lucru cu macroinstrucțiuni: definirea macroinstrucțiunilor și utilizarea lor. Utilizarea se mai numește invocare sau chiar apel de macroinstrucțiune, dar ultima denumire poate produce confuzie, termenul utilizându-se în cazul procedurilor.

Spre deosebire de proceduri, macroinstrucțiunile sunt expandate la fiecare utilizare, deci programul nu se micșorează. Avantajul este că textul sursă scris de programator devine mai clar și mai scurt.

Macroinstrucțiunile pot fi cu parametri sau fără parametri. Din punct de vedere sintactic, ele sunt asemănătoare cu directivele de tip `DEFINE` din limbajele de nivel înalt, dispunând, în general, de mecanisme mai evolute decât acestea.

Definiția unei macroinstrucțiuni fără parametri se face în forma generală:

```
nume_macro MACRO
    ;
    ; Corp macroinstrucțiune
    ;
ENDM
```

Invocarea constă în scriere în textul sursă a numelui macroinstrucțiunii. Macroinstrucțiunea `init_ds_es` este definită în fișierul header `io.h` prin:

```
init_ds_es macro
    mov ax, DGROUP
    mov ds, ax
    mov es, ax
endm
```

Macroinstrucțiunea `exit_dos` este definită prin:

```
exit_dos macro
    mov ax, 4C00H
    int 21H
endm
```

O pereche de macroinstrucțiuni care salvează și refac registrele generale poate fi conceput astfel:

<pre>save macro push ax push bx push cx push dx push si push di</pre>	<pre>rest macro pop di pop si pop dx pop cx pop bx pop ax</pre>
---	---

```
endm                                endm
```

Putem utiliza aceste macroinstrucțiuni la intrarea și la ieșirea dintr-o procedură:

```
PROCEDURA  proc near
    save
    .
    .
    rest
    ret
PROCEDURA  endp
```

Din punct de vedere simbolic, este ca și cum setul de instrucțiuni al mașinii ar fi fost extins cu două noi instrucțiuni: SAVE și REST.

5.2 Macroinstrucțiuni cu parametri

Macroinstrucțiunile importante sunt cele cu parametri. Definiția unei macroinstrucțiuni cu parametri are forma generală:

```
nume_macro  MACRO P1, P2, ..., Pn
    ;
    ; Corp macroinstrucțiune
    ;
ENDM
```

În care P1, P2, ..., Pn sunt identificatori care specifică parametrii formali. Apelul (invocarea) unei macroinstrucțiuni cu parametri se face prin specificarea numelui, urmată de o listă de parametri actuali:

```
nume_macro  X1, X2, ..., Xn
```

La expandarea macroinstrucțiunii, pe lângă expandarea propriu- zisă, se va înlocui fiecare parametru formal cu parametrul actual respectiv.

Să considerăm câteva exemple. Apelurile de funcții DOS presupun numărul funcției în registrul AH. Putem deci defini o macroinstrucțiune de forma:

```
dosint macro N
    mov  ah, N
    int  21H
endm
```

și putem invoca macroinstrucțiunea prin linii de forma:

```
dosint 2
```

Similar, pentru deschiderea unui fișier disc pentru citire (operație realizată prin funcția DOS 3CH), putem scrie o macroinstrucțiune de forma:

```
o_read macro fname, hand
    mov  al, 0C0H
    lea  dx, fname
    dosint 3DH
    mov  hand, ax
endm
```

În care file_name conține numele fișierului iar handle este o variabilă de tip word în care se depune un indicator către fișierul deschis. Parametrul 0C0H codifică modul de acces. Se observă utilizarea macroinstrucțiunii dos_int în definiția lui o_read.

Citirea din fișier poate fi cofificată într-o macroinstrucțiune de forma:

```
f_read macro hand, buf, nr
    mov    bx, hand
    mov    cx, nr
    lea    dx, buf
    dosint 3FH
endm
```

În care hand este indicatorul către fișierul anterior deschis, nr este numărul de octeți care se citește iar buf este adresa unei zone de memorie în care se vor depune datele citite.

Închiderea unui fișier deschis se poate face cu o macroinstrucțiune de forma:

```
f_closemacro hand
    mov    bx, hand
    dosint 3EH
endm
```

Macroinstrucțiunile de mai sus sunt definite în fișierul io.h. Ne propunem acum să scriem un program executabil care să afișeze la consolă un fișier text. Beneficiem de macroinstrucțiunile definite în fișierul io.h:

```
.model      large
include     io.h
.stack 1024
.data
    file_name    db    30 dup (0)        ; Spatiu nume fișier
    hand         dw    ?                  ; Spațiu handler
    buf          db    1024 dup (?)       ; Buffer citire
.code
start:
init_ds_es
putsi        <'Nume fisier: '>          ; Mesaj
gets         file_name                  ; Citire nume
o_read       file_name, hand             ; Deschidere
jc           eroare                      ; Eroare deschidere ?
bucla:
f_read       hand, buf, 1024             ; Citire 1024 octeți
jc           eroare                      ; Eroare citire
mov          si, ax                      ; AX = câți octeți s-au
mov          byte ptr buf [si], 0        ; citit de fapt
puts         buf                         ; Afișare la consolă
cmp          ax, 1024                    ; S-au citit mai puțin
jbe          gata                        ; de 1024 ?
jmp          bucla                       ; Nu, reluare
gata:
f_close      hand                        ; Închidere
jmp          iesire                      ; Salt la ieșire
eroare:
putsi        <'Eroare fisier'>           ; Mesaj de
                                                    ; eroare
iesire:
exit_dos                                           ; Ieșire în DOS
end          start
```

În zona de date, se rezervă spațiu pentru numele fișierului, pentru indicator (handler) și pentru un buffer de citire de 1024 de octeți. Reamintim că operațiile cu perifericele sunt în esență transferuri între periferice și memorie.

Programul afișează un mesaj la consolă, după care citește un nume de fișier. Se face apoi operația de deschidere a fișierului specificat. Toate funcțiile DOS

de lucru cu fișiere întorc CF = 1 în caz de eroare. O eroare tipică la operația de deschidere pentru citire este un nume eronat de fișier. Se testează deci CF și, în caz de eroare, se afișează un mesaj adecvat și se iese în DOS.

Se trece acum la o buclă de citire-afișare. Funcția de citire întoarce în AX numărul de octeți efectiv citați (care este mai mic sau egal decât cel cerut). Dorim să afișăm numai ce s-a citit efectiv, așa că punem terminatorul 0 în buffer, după ultimul octet efectiv citit. E posibil ca, la ultima iterație, să se citească 0 octeți. Se afișează la consolă bufferul respectiv (cu macroinstrucțiunea puts).

Dacă numărul de octeți efectiv citați este mai mic strict decât cel cerut (1024) înseamnă că s-a ajuns la sfârșitul fișierului și bucla se termină. În caz contrar, se reia cu o nouă citire din fișier. În final, se închide fișierul și se iese în DOS.

Acest exemplu ilustrează foarte bine avantajele macroinstrucțiunilor. O acțiune destul de laborioasă în limbaj de asamblare (afișare fișiere text) a putut fi codificată prin câteva linii de program sursă (e drept că aproape toate sunt invocări de macroinstrucțiuni).

Morala fabulei este că, dacă reușim să concepem un set de macroinstrucțiuni adecvat unei probleme (în cazul de față, interfața cu sistemul DOS), scrierea programelor devine foarte comodă.

Cititorul este sfătuit să scrie un program similar, care să realizeze copierea unui fișier disc în alt fișier. Pentru operații de scriere, se pot utiliza macroinstrucțiunile:

```
o_write    macro fname, hand
            mov     al, 0C1H
            lea     dx, fname
            dosint  3DH
            move    hand, ax
        endm
f_write    macro hand, buf, nr
            mov     bx, hand
            mov     cx, nr
            lea     dx, buf
            dosint  40H
        endm
```

care deschid un fișier pentru scriere, respectiv scriu în fișier. Parametrii sunt asemănători cu cei din macroinstrucțiunile **o_read** și **f_read**. Bucla de tip **citire_din_fișier-afișare** din exemplul anterior se înlocuiește cu o buclă de tip **citire_din_fișier_sursă-scriere_în_fișier_destinație**.

În unele situații, substituția parametrilor formali cu cei actuali poate ridica unele probleme. Să presupunem că un tânăr programator care învață limbajul ASM nu a ajuns încă la instrucțiunea XCHG ci are cunoștință doar de instrucțiunile PUSH, POP și MOV. El își propune să scrie o macroinstrucțiune care să

interschimbe două cantități de 16 biți:

```
schimba macro X, Y
            push    ax
            push    bx
            mov     bx, X
```

```

        mov    ax, Y
        mov    X, ax
        mov    Y, bx
        pop    bx
        pop    ax
    endm

```

Aparent, totul e în ordine. Totuși, pot apare situații nedorite, ca în secvența:

```

    trans  AX, SI      ; Interschimbă AX cu SI (oare ?)

```

Această invocare de macroinstrucțiune se expandează în:

```

    push   ax
    push   bx
    mov    bx, AX
    mov    ax, SI
    mov    AX, ax
    mov    SI, bx
    pop    bx
    pop    ax

```

și este evident că registrul AX nu se modifică. Se poate însă și mai rău, ca în secvența:

```

    trans  SP, DI      ; Interschimbă SP cu DI (oare ?)

```

care se expandează în:

```

    push   ax
    push   bx
    mov    bx, SP
    mov    ax, DI
    mov    SP, ax      ; Aici se modifică SP
    mov    DI, bx      ; și POP-urile
    pop    bx          ; sunt
    pop    ax          ; compromise

```

Pericolul apare deci în situațiile în care parametrii actuali intră în conflict cu anumite variabile sau registre care sunt folosite în interiorul macroinstrucțiunii. Aceste situații trebuie evident evitate.

Utilitatea foarte mare a macroinstrucțiunilor devine evidentă la secvențele de apel ale procedurilor cu parametri (în acest caz, se vorbește despre macroinstrucțiuni de apel). Apelurile de funcții sistem codificate anterior sunt cazuri particulare de macroinstrucțiuni de apel.

Procedurile cu parametri utilizează diverse tehnici de transmitere a parametrilor. Parametrii trebuie plasați în anumite registre sau în stivă, într-o ordine specificată. Aceste detalii de apel sunt greu de ținut minte și nici nu interesează pe cel care apelează procedura. Putem însă dezvolta macroinstrucțiuni care să ascundă aceste detalii.

Să considerăm o procedură NEAR cu numele puts_proc, care afișează un șir de caractere (terminat cu 0). Adresa șirului (offset-ul în cadrul segmentului curent adresat prin DS) se specifică în registrul SI. O macroinstrucțiune de apel se poate scrie în forma:

```

    puts   macro X
           push   si
           lea     si, X
           call    puts_proc
    endm

```

```

        pop    si
    endm

```

Ca parametru actual se poate utiliza orice operand compatibil cu instrucțiunea LEA. Astfel, dacă există definiția de date:

```

.data
    STRING    db    "abcdefghijklmnopqrstuvwxyz", 0

```

se pot utiliza formele de invocare:

```

.code
    puts    STRING            ; Adresare directă
    lea     bx, STRING
    puts    [bx]              ; Adresare indirectă

```

În al doilea caz, nu putem scrie `puts bx`, pentru că acest apel ar conduce la o expandare de forma `lea si, bx`, ceea ce este o eroare de sintaxă. Forma `puts [bx]` se expandează în `lea si, [bx]`, care este corectă.

5.3 Controlul numărului de parametri actuali

O problemă importantă este controlul coincidenței dintre numărul parametrilor formali și cel al parametrilor actuali. Se vor utiliza următoarele directive specifice:

- Directiva `%OUT` (Mesaj la consolă la momentul asamblării). Are forma generală `%OUT TEXT`, efectul fiind afișarea `TEXT`-ului la consolă, la momentul asamblării; este utilizată pentru mesaje de eroare la asamblare.
- Directiva `.ERR` (Forțează eroare de asamblare. Această directivă forțează o eroare la asamblare, astfel încât să nu se mai genereze fișier obiect; este utilizată în cazul neconcordanței dintre numărul parametrilor actuali și cel al parametrilor formali.
- Directiva `EXITM` (Ieșire forțată din expandare)

În caz de eroare, stopăm expandarea macroinstrucțiunii curente și afișăm un mesaj de eroare la consolă.

Pentru controlul efectiv al parametrilor actuali, se mai utilizează directivele de asamblare condiționată `IFNB` (If Not Blank) și `IFB` (If Blank) care testează existența sau non- existența unui parametru actual.

Pentru forțarea unei erori, definim pentru început macroinstrucțiunea:

```

eroare    macro text
            %OUT text            ;; Mesaj la consolă
            .err                 ;; Forțare eroare la asamblare
    endm

```

Să considerăm macroinstrucțiunea `puts`, definită mai sus, care are un singur parametru formal. La invocare, s-ar putea să existe un parametru actual, nici unul sau mai mulți. Pentru a controla toate aceste situații, rescriem macroinstrucțiunea `puts` și o declarăm cu doi parametri: `X` și `IN_PLUS`. Dacă `IN_PLUS` este nevid, înseamnă că sunt cel puțin doi parametri actuali iar dacă `X` este vid, nu există nici un parametru actual. Codificarea va fi deci:

```

puts    macro X, IN_PLUS
    ifb    <X>
        eroare    <PUTS - lipsa parametru>
        exitm
    endif
    ifnb    <IN_PLUS>

```

```

                                eroare      <PUTS - parametri in plus>
                                exitm
                                endif
                                lea    si, X
                                call   puts_proc
endm

```

Testăm explicit dacă X este vid și dacă IN_PLUS este nevid, forțând mesaje adecvate de eroare. Scriere cu paranteze unghiulare <> înseamnă literalizarea textului respectiv, adică transmiterea lui ca un unic parametru. Dacă se consideră un fișier t_macro.asm cu secvențele de invocare:

```

.data
    sir    db 10 dup (0)
.code
    puts   [si]
    puts   [bx] [si]
    puts   [bx], [si]
    puts
    puts   [bx], sir
    puts   sir
end

```

atunci se vor obține mesajele de eroare:

```

PUTS - parametri in plus
PUTS - parametri in plus
PUTS - lipsa parametru
PUTS - parametri in plus
**Error** t_macro.asm(26) EROARE(2) User generated error
**Error** t_macro.asm(27) EROARE(2) User generated error
**Error** t_macro.asm(28) EROARE(2) User generated error
**Error** t_macro.asm(29) EROARE(2) User generated error

```

Numerele din paranteze indică liniile din fișierul sursă în care a fost forțată explicit eroarea de asamblare.

Putem acum da o regulă generală de scriere a unei macroinstrucțiuni cu controlul parametrilor. Dacă macroinstrucțiunea are N parametri utili, declarăm un al N+1-lea parametru suplimentar și testăm dacă al N-lea parametru este vid (cu directiva IFB), respectiv dacă al N+1-lea parametru este nevid (cu directiva IFNB).

5.4 Directive de control al listării. Etichete în macroinstrucțiuni

Există diverse posibilități de control al fișierului listing al unui program care conține macroinstrucțiuni. Controlul se face prin următoarele directive:

- Directiva .SALL (Suppress All - Suprimă tot). În textul sursă care urmează acestei directive, se va suprima listarea conținutului macroinstrucțiunilor invocate. În listing apare numai invocarea (numele) macroinstrucțiunii, exact ca în fișierul sursă.
- Directiva .LALL (List All - Listează tot). După această directivă, se listează atât invocarea macroinstrucțiunii cât și textul generat.
- Directiva .XALL - După această directivă, se listează textul generat efectiv de invocarea unei macroinstrucțiuni. Dacă macroinstrucțiunile conțin invocări ale altor macroinstrucțiuni, la al doilea nivel se listează numai invocarea.

Uneori e necesar ca macroinstrucțiunile să conțină etichete, de exemplu pentru instrucțiuni de salt. Apare însă următoarea problemă: dacă eticheta se definește în mod obișnuit, ea va fi generată la fiecare expandare a

macroinstrucțiunii, ceea ce va duce la mai multe etichete cu același nume. Acest fapt va provoca o eroare la asamblare, de tip "simbol multiplu definit".

Problema se rezolvă prin directiva LOCAL, care are forma generală:

```
LOCAL      simb_1, simb_2, ...
```

efectul fiind că simbolii care apar în directivă vor fi expandați în așa fel încât să nu apară două nume identice. Practic asamblorul generează niște nume complicate, de forma ??0000, ??0001, ??0002 etc. (care nu se repetă), pentru care există șanse minime să coincidă cu nume definite de utilizator.

Să considerăm macroinstrucțiunea:

```
test_local macro
local aici
    jmp    aici
aici:
endm
```

și secvența de invocare:

```
.code
.xall
    test_local
    test_local
    test_local
end
```

Se va genera următorul listing:

```
test_local
    jmp    ??0000
??0000:
test_local
    jmp    ??0001
??0001:
test_local
    jmp    ??0002
??0002:
```

Eticheta aici a fost înlocuită la expandare cu etichetele ??0000, ??0001 și ??0002.

Simbolii variabili (definiți cu operatorul de atribuire =) se dovedesc utili în macroinstrucțiuni, deoarece pot fi redefiniți chiar în macroinstrucțiune. Să considerăm definiția:

```
m_mesaj macro
    db    'Mesaj ', '0'+n, 0
    n = n + 1
endm
```

și secvența de program:

```
n = 1
.data
    m_mesaj
    m_mesaj
    m_mesaj
```

Se va genera textul sursă:

```
db    'Mesaj ', '0' + 1, 0
```



```
db    'Mesaj ', '0' + 2, 0
db    'Mesaj ', '0' + 3, 0
```

deci ca și cum s-ar fi scris:

```
db    'Mesaj 1', 0
db    'Mesaj 2', 0
db    'Mesaj 3', 0
```

Dacă dorim să includem comentarii in macroinstrucțiuni dar aceste să nu apară la fiecare expandare ci numai la definiția macroinstrucțiunii, putem utiliza secvența de două caractere ;; (început de comentariu).

5.5 Macroinstrucțiuni repetitive

Aceste macroinstrucțiuni sunt predefinite, deci nu trebuiesc definite de utilizator. Scopul lor este de a genera secvențe repetate de program.

Macroinstrucțiunea REPT (Repete - Repetă)

Forma generală de invocare este:

```
rept  n
    ;
    ; Corp macroinstrucțiune
    ;
endm
```

În care *n* este o constantă întreagă. Efectul este repetarea corpului macroinstrucțiunii de *n* ori. Secvența anterioară de generare a trei mesaje s-ar putea scrie acum:

```
rept 3
    m_mesaj
endm
```

Iată o secvență care generează un șir de caractere cu litere de la 'A' la 'Z':

```
n = 0
alfabet label byte
rept 26
    db 'A'+n
    n = n + 1
endm
```

Macroinstrucțiunea IRP (Indefinite Repeats - Repetă nedefinit)

Forma generală de invocare este:

```
irp p_formal, <listă_par_act>
    ;
    ; Corp macroinstrucțiune
    ;
endm
```

În care **p_formal** este un parametru formal iar **listă_par_act** este o listă de parametri actuali, separați prin virgulă. Efectul este repetarea corpului macroinstrucțiunii de atâtea ori câte elemente conține lista de parametri actuali. La fiecare repetare, se substituie parametrul formal cu câte un parametru actual. Invocarea:

```

    irp x, <'a', 'b', 'c'>
        db x
    endm

```

se va expanda în:

```

    db 'a'
    db 'b'
    db 'c'

```

5.6 Operatori specifici

Există o serie de operatori specifici macroinstrucțiunilor. Aceștia controlează în principall substituția parametrilor

actuali, fiind utili în special în macroinstrucțiunile repetitive.

5.6.1 Operatorul de substituie și concatenare (&)

Acest operator, aplicat unui parametru formal, realizează substituția și (eventual) concatenarea sa cu un text fix sau cu un alt parametru formal. Este necesar în contextul în care un parametru formal ar fi interpretat ca un simbol (de exemplu, într-un șir constant de caractere sau într-un identificator).

Să presupunem că dorim să definim automat liniile de program:

```

    mesaj_1    db 'Text 1', 0
    mesaj_2    db 'Text 2', 0
    mesaj_3    db 'Text 3', 0
    mesaj_4    db 'Text 4', 0

```

Vom apela evident la macroinstrucțiunea IRP. O prima încercare ar fi:

```

    irp x, < 1, 2, 3, 4 >
        mesaj_x    db 'Text x', 0
    endm

```

ceea ce este incorect, deoarece se va produce aleeași linie de program. Prima apariție a parametrului **x** nu poate fi distinsă de simbolul **mesaj_x** iar a doua nu poate fi distinsă de șirul constant **'Text x'**. Aici intervine operatorul **&**, definiția corectă fiind:

```

    irp x, < 1, 2, 3, 4 >
        mesaj_&x    db 'Text &x', 0
    endm

```

prin care, în primul caz, se concatenează textul fix **mesaj_** cu parametrul formal **x**, iar în al doilea, se substituie parametrul formal **x** chiar dacă apare într-un șir constant de caractere.

Să considerăm un exemplu înrudit, codificat prin macroinstrucțiunea următoare:

```

    genereaza macro fix, n
        X = 1
        rept n
            fix&&x    db    'Text &x', 0
            x = x + 1
        endm
    endm

```

Aici e necesară concatenarea a doi parametri formali (fix și x): fiecăruia i se aplică operatorul &, la stânga sau la dreapta, după locul în care are loc concatenarea. Un apel de forma:

```
genereaza mesaj_ , 4
```

va produce același text ca macroinstrucțiunea IRP de mai sus.

5.6.2 Operatorii de literalizare șir/caracter (<>, !)

Operatorul <> (literalizare șir)

Se utilizează atunci când se dorește ca un text în care apar eventuali separatori (spații albe, virgule etc.) să fie considerat ca un unic parametru (să fie literalizat). Operatorul se utilizează atât în definiții cât și în invocări de macroinstrucțiuni. De exemplu, în definiția:

```
init macro x
    irp y, <x>
        db y
    endm
endm
```

dorim ca parametrul x de la nivelul exterior să fie transmis ca atare către macroinstrucțiunea IRP. Invocarea se va face în forma:

```
init <'A', 'B', 'C', 'D'>
```

ceea ce are ca efect transmiterea listei 'A', 'B', 'C', 'D' ca un unic parametru.

Operatorul ! (literalizare caracter)

Se aplică unui singur caracter, efectul fiind de a trata acel caracter ca un caracter obișnuit (fără a-l interpreta). Se utilizează împreună cu caractere care au semnificații speciale în macroinstrucțiuni. Să considerăm o macroinstrucțiune care definește mesaje de eroare:

```
err_gen macro N, X
    err_&N      db 'Eroare &N : &X', 0
endm
```

O invocare de forma:

```
err_gen      23, <Parametru ilegal>
```

se va expanda în:

```
err_23      db 'Eroare 23 : Parametru ilegal', 0
```

Dacă dorim însă să generăm un mesaj de forma 'par_1 > par_2', intrăm în conflict cu semnificația specială a caracterului '>', care intră în componența operatorului de literalizare șir. Soluția este să folosim operatorul ! și să scriem:

```
err_gen      24, <par_1 !> par_2>
```

ceea ce va genera mesajul corect:

```
err_24      24, 'Eroare 24 : par_1 > par_2', 0
```

5.6.3 Operatorul de evaluare expresie (%)

Acest operator se aplică unei expresii oarecare, efectul fiind evaluarea acelei expresii. Dintr-un punct de vedere, operatorul % este inversul operatorilor de literalizare. Să considerăm macroinstrucțiunea:

```
def macro a, b
    db    '&a', 0
    db    '&b', 0
endm
```

care generează date. O invocare de forma:

```
alfa    equ 100
beta    equ 200
def <alfa + beta>, %(alfa + beta)
```

produce liniile de program:

```
db    'alfa + beta', 0
db    '300', 0
```

În invocarea lui def, primul parametru este literalizat iar al doilea evaluat, ceea ce explică textul generat.

5.7 Invocare recursivă de macroinstrucțiuni

O macroinstrucțiune se poate invoca recursiv, adică pe ea însăși. Ca și la proceduri recursive, cel mai important lucru este oprirea recursivității, care se poate face cu directivele de asamblare condiționată IFB sau IFNB. Să considerăm o macroinstrucțiune de salvare de registre în stivă. Vrem ca aceasta să permită un număr variabil de parametri. Soluția recursivă este să fixăm un număr maximal de parametri și să testăm explicit dacă primul parametru este vid:

```
push_all_1 macro r1, r2, r3, r4, r5, r6
    ifnb r1
        push r1
        push_all_1 r2, r3, r4, r5, r6
    endif
endm
```

Dacă primul parametru formal nu este vid, se generează instrucțiunea PUSH și apoi se invocă aceeași macroinstrucțiune, cu restul de parametri. Această formă poate fi folosită cu un număr oarecare de registre de la 1 la 6, de exemplu:

```
push_all_1    ax, bx, cx, dx
```

O altă variantă este cea repetitivă:

```
push_all_2 macro X
    irp y, <x>
        push y
    endm
endm
```

Invocarea necesită însă operatorul de literalizare:

```
push_all_2 <ax, bx, cx, dx>
```

5.8 Definirea macroinstrucțiunilor în macroinstrucțiuni

Se poate spune că macroinstrucțiunile automatizează oarecum procesul de definire a datelor și instrucțiunilor. Mai mult decât atât, chiar definirea macroinstrucțiunilor se poate face prin intermediul altor macroinstrucțiuni.

Să considerăm un asemenea caz. În cazul procesorului 8086, instrucțiunile de deplasare și rotație cu un număr de biți mai mic sau egal cu 3, se execută mai

rapid ca secvențe de rotații de câte un bit, în comparație cu instrucțiunile care utilizează registrul CL. Dorim să scriem câte o macroinstrucțiune pentru cele 8 instrucțiuni de deplasare și rotație, fiecare cu doi parametri, sursa și numărul de biți, care să se expandeze în secvența optima ca timp de execuție.

De exemplu, o invocare de forma:

```
m_shr ax, 5
```

să se expandeze în secvența:

```
mov    cl, 5
shr    ax, cl
```

iar o invocare de forma:

```
m_shr bx, 3
```

în secvența:

```
shr    bx, 1
shr    bx, 1
shr    bx, 1
```

Dorim ca generarea macroinstrucțiunilor (având numele de forma m_XXX, unde XXX este numele instrucțiunii corespunzătoare) să fie făcută automat.

Începem prin a defini o macroinstrucțiune care primește (în parametrul formal operation) numele instrucțiunii respective și generează macroinstrucțiunea corespunzătoare:

```
gen macro operation
    m_&operation macro operand, nr
        if nr lt 4
            rept nr
                operation operand, 1
            endm
        else
            mov cl, nr
            operation operand, cl
        endif
    endm
endm
```

La o invocare de forma:

```
gen    shl
```

se va genera automat macroinstrucțiunea m_shl, conform definiției echivalente:

```
m_shl macro operand, nr
    if nr lt 4
        rept nr
            shl operand, 1
        endm
    else
        mov cl, nr
        shl operand, cl
    endif
endm
```

adică exact ce ne-am propus. Generăm acum toate cele 8 macroinstrucțiuni, observând că numele operațiilor (instrucțiunilor) respective se compun din

secvențele RO, RC, SH, SA, la care se adaugă sufixele R sau L. Exploatăm acest fapt prin două macroinstrucțiuni repetitive:

```
irp X, <RO, RC, SH, SA>
    irp Y, <R, L>
        gen X&&Y
    endm
endm
```

Această secvență va defini macroinstrucțiunile m_ror, m_rol, m_rcr, m_rcl, m_shr, m_shl, m_sar și m_sal.

5.9 Tehnici avansate de utilizare a macroinstrucțiunilor

5.9.1 Macroinstrucțiuni care se autotransformă (se redefinesc) în proceduri

Un dezavantaj al utilizării intensive a macroinstrucțiunilor este consumul de memorie: dacă invocăm de 100 de ori o macroinstrucțiune, textul respectiv se va duplica de 100 de ori. Acest lucru nu deranjează în situația în care textul respectiv trebuia oricum scris (de exemplu, în secvențele de apel ale procedurilor).

Sunt însă situații în care corpul macroinstrucțiunii reprezintă o secvență oarecare de instrucțiuni. Dorim ca invocările repetate să nu conducă la repetarea corpului macroinstrucțiunii ci la apeluri ale unei proceduri, acest lucru fiind transparent pentru utilizator. Soluția este următoarea:

```
macsub macro
    local gata
    call subr
    jmp gata
    subr proc near
        ;
        ; Corpul macroinstrucțiunii
        ;
        ret
    subr endp
gata:
    macsub macro
        call subr
    endm
endm
```

Se înscrie corpul macroinstrucțiunii într-o procedură cu numele subr și se generează un apel al acestei proceduri, urmat de un salt peste procedura respectivă. Se redefinesc apoi macroinstrucțiunea macsub, în așa fel încât să se genereze doar apeluri ale procedurii subr.

La prima invocare a macroinstrucțiunii macsub, se va genera textul:

```
call    subr
jmp     ??0000
subr    proc near
    ;
    ; Corp macroinstrucțiune
    ;
    ret
subr endp
??0000:
```

La următoarele invocări, se va genera doar textul:

```
call subr
```

Acest exemplu ne arată că într-o macroinstrucțiune putem chiar redefini aceeași macroinstrucțiune.

5.9.2 Macroinstrucțiuni care generează atât date cât și cod

În descrierea interfeței cu sistemul DOS, realizată prin fișierul header io.h (vezi 2.7 și Anexa A), s-a prezentat macroinstrucțiunea puts, care permite afișarea (la momentul execuției) a unor șiruri constante "immediate", fără a le defini explicit într-un segment de date. Puteam deci scrie:

```
puts <'Acesta este un mesaj la consola'>
```

fără să ne punem problema unde se memorează șirul constant respectiv. Pentru afișarea propriu-zisă, utilizăm procedura puts_proc din fișierul io.asm (vezi Anexa B), care primește în DS:SI adresa de memorie a unui șir terminat cu 0.

Problema care se pune este că definiția șirului în memorie trebuie făcută chiar în macroinstrucțiunea puts. Vom beneficia de faptul că directivele simplificate de definire a segmentelor (în speță .code și .data) pot alterna în cuprinsul unui program. Definiția macroinstrucțiunii este următoarea:

```
puts macro X
    local string
    .data
        string db X, 0
    .code
    push si
    lea si, string
    call puts_proc
    pop si
endm
```

Se comută pe segmentul de date și se definește șirul transmis prin parametrul formal X, la care se adaugă terminatorul 0. Identificatorul string se declară local, pentru a nu fi duplicat în apeluri succesive. Se comută apoi pe segmentul de cod și se generează secvența de apel a procedurii puts_proc, cu salvarea și restaurarea registrului SI. Un apel de forma:

```
puts <'Acesta este un mesaj la consola'>
```

se va expanda într-o secvență de forma:

```
.data
??0001 db 'Acesta este un mesaj la consola', 0
.code
    push si
    lea si, ??0001
    call puts_proc
    pop si
```

În mod similar, în 2.7 a fost introdusă o macroinstrucțiune geti care citește un întreg cu semn pe 16 biți de la consolă, întorcând valoarea citită în registrul AX. Problema care apare este că de la consolă se pot citi date numai la nivel de caractere.

Presupunem că dispunem de o procedură gets_proc care citește un număr limitat de caractere de la consolă. Parametrii acestei proceduri sunt adresa

(near) unde se depun caracterele citite, transmisă prin SI și numărul maxim de caractere citite, transmis în CX. Citirea încetează fie la apăsarea pe Enter, fie la atingerea numărului maxim de caractere iar după ultimul caracter se depune terminatorul 0. Dacă CX = 0, atunci se poate introduce un număr nelimitat de caractere.

După ce s-au citit caracterele care formează numărul, acestea trebuie convertite de la șir de caractere ASCII la un întreg cu semn. Această conversie se realizează cu o procedură atoi_proc, care primește în SI adresa șirului de caractere și întoarce valoarea calculată în AX.

Macroinstrucțiunea geti se definește astfel:

```
geti    macro
local   string
.data
        buffer db      8 dup (0)
.code
        push    si
        lea     si, buf
        mov     cx, 7
        call    gets_proc
        lea     si, buf
        call    atoi_proc
        pop     si
endm
```