Alte aspecte ale programării funcţionale

Cuprins

1. Parametrii Lisp în context “Dynamic Scoping”

Aşa cum am văzut până acum, toate apariţiile parametrilor formali sunt înlocuite cu valorile argumentelor corespunzătoare. Deci, avem o variantă de apel prin nume, apelul prin text, deoarece în Lisp avem de-a face cu determinarea dinamică a domeniului de vizibilitate (dynamic scoping) simultan cu înlocuirea textuală a valorilor evaluate.

Deşi parametrul se leagă de argument (în sensul apelului prin referinţă din limbajele imperative, adică numele argumentului şi cel al parametrului devin sinonime), nu este apel prin referinţă, deoarece modificările valorilor parametrilor formali nu afectează valoarea argumentelor (ceea ce aminteşte de apelul prin valoare).

De exemplu:

- (DEFUN FCT(X) (SETQ X 1) Y) se evaluează la FCT
- (SETQ Y 0) se evaluează la 0
- (FCT Y) se evaluează la 0

In secvenţa de mai sus Y se evaluează la 0, iar X se leaga la 0. Apelul FCT este astfel echivalent cu (FCT 0). X e parametru formal, Y e parametru actual, şi ca atare modificarea lui X nu afectează pe Y, deci NU avem de-a face cu apel prin referinţă.

Aceasta demonstrează că transmiterea de parametri în LISP nu se face prin referinţă. Cum justificăm însă că nu este nici apel prin valoare? Există totuşi situaţii în care valoarea parametrului actual poate fi modificată, de exemplu prin actiunea funcţiilor cu efect distructiv RPLACA şi RPLACD.

Să mai remarcăm faptul că în ciuda caracteristicii “Dynamic scoping”, variabila Y nu se leagă la execuţie de parametrul formal tocmai înlocuit cu Y, adica Y-ul “intern” funcţiei, ci îşi păstrează caracteristica de variabilă globală pentru FCT.
Corpul unei funcţii LISP este domeniul de vizibilitate a parametrilor săi formali, care se numesc variabile legate în raport cu funcţia respectivă. Celelalte variabile ce apar în definitia funcţiei se numesc variabile libere.

Contextul curent al execuţiei este alcătuit din toate variabilele din program împreună cu valorile la care sunt legate acestea în acel moment. Acest concept este necesar pentru stabilirea valorii variabilelor libere care intervin în evaluarea unei funcţii, evaluarea în LISP făcându-se într-

un context dinamic (dynamic scoping), prin ordinea de apel a funcţiilor, şi nu static, adică relativ la locul de definire.

Să reamintim în acest scop diferenţa între determinarea statică şi respectiv cea dinamica a domeniului de vizibilitate în cadrul limbajelor imperative. Fie programul Pascal:

```
var
        a:integer;

procedure P;
begin
        writeln(a);
end;

procedure Q;
var
        a: integer;
begin
        a := 7;
        P;
end;

begin
        a := 5;
        Q;
end.
```

Determinarea statică a domeniului de vizibilitate (DSDV, static scoping) presupune că procedura P acţionează în mediul de definire şi ca urmare ea va "vedea" întotdeauna variabila a globală. Este şi cazul limbajului Pascal, situatie în care P apelat în Q va tipari 5 şi nu 7.

Pe de alta parte, dacă am avea de-a face cu **determinarea dinamică a domeniului de vizibilitate** (DDDV, dynamic scoping), procedura P va tipări întotdeauna ultima (în ordinea apelurilor) variabilă *a* definită (sau legată, în terminologie Lisp). În acest caz se va tipări 7, deoarece ultima legare a lui *a* este relativ la valoarea 7.

Limbajul Lisp dispune de DDDV, această abordare fiind mai naturală decât cea statică pentru cazul sistemelor bazate pe interpretoare. O variantă Lisp a programului de mai sus care pune în evidenţă DDDV este:

```
(DEFUN P () A)
(DEFUN Q ()
        (SETQ A 7)
        (P)
)
(SETQ A 5)
(SETQ Y (LIST (P) (Q)))
(PRINT Y)
```

Se va tipări lista (5 7), valoarea întoarsă de P fiind de fiecare dată ultimul A legat.

Avantajul legării dinamice este adaptabilitatea, adică o flexibilitate sporită. În cazul argumentelor funcţionale însă pot să apară interacţiuni nedorite. Pentru a ilustra genul de probleme care pot apărea să luăm exemplul formei funcţionale twice, care aplica de două ori argumentul funcţie unei valori:

(DEFUN twice (func val) (funcall func (funcall func val)))

Exemple de aplicare:
- (twice 'add1 5) returnează 7
- (twice '(lambda (x) (* 2 x)) 3) returnează 12

Dacă se întâmplă însă să folosim identificatorul val şi în cadrul lui func, ţinând cont de DDDV, apare o coliziune de nume, cu următorul efect:

- (setq val 2) returnează 2
- (twice '(lambda (x) (* val x)) 3) returnează 27, nu 12

(cum probabil am dori să obţinem). Aceasta deoarece ultima legare (dinamică deci) a lui val s-a efectuat ca parametru formal al funcţiei twice, deci val a devenit 3.

Această problemă a fost denumită problema FUNARG (funcţional argument). Este o problemă de conflict între DSDV şi DDDV. Rezolvarea ei nu a constat în renunţarea la DDDV pentru LISP ci în introducerea unei forme speciale numite **function**, care realizeaza legarea unei lambda expresii de mediul ei de definire (deci tratarea acelui caz în mod static). Astfel,

- (twice (function (lambda (x) (* val x))) 3) se evaluează la 12

Concluzia este deci că LISP are două reguli de DDV: DDDV implicit şi DSDV prin intermediul constructiei function.

2. Universul expresiilor şi universul instrucţiunilor

Orice limbaj de programare poate fi împărţit în două aşa-numite universuri (domenii):
1. universul expresiilor;
2. universul instrucţiunilor.

Universul expresiilor include toate construcţiile limbajului de programare al căror scop este producerea unei valori prin intermediul procesului de evaluare.

Universul instrucţiunilor include instrucţiunile unui limbaj de programare. Acestea sunt de două feluri:

(i) instrucţiuni ce influenţează fluxul de control al programului:
- instrucţiuni conditionale;
- instrucţiuni de salt;
- instrucţiuni de ciclare;
- apeluri de proceduri şi funcţii.

(ii) instrucţiuni ce alterează starea memoriei:
- atribuirea (alterează memoria internă, primară);
- instrucţiuni de intrare/ieşire (alterează memoria externa, secundară)

Ca asemănare a acestor două universuri, putem spune că ambele alterează ceva. Există însă deosebiri importante. În universul instrucţiunilor ordinea în care se execută instrucţiunile este de obicei esenţială. Adică instrucţiunile

```
i := i + 1; a := a * i;
```

au efect diferit fata de instrucţiunile

```
a := a * i; i := i + 1;
```

Fie

```
z := (2 * a * y + b) * (2 * a * y + c);
```

Multe compilatoare elimină evaluarea redundantă a subexpresiei comune 2 * a * y prin următoarea substituţie:

```
t := 2 * a * y; z := (t + b) * (t + c);
```

Această înlocuire s-a putut efectua în universul expresiilor deoarece în cadrul unei expresii o subexpresie va avea întotdeauna aceeaşi valoare.

În cadrul universului instrucţiunilor situaţia se schimbă, datorită posibilelor efecte secundare. De exemplu, pentru secvenţa

```
y := 2 * a * y + b; z := 2 * a * y + c;
```

factorizarea subexpresiei comune furnizează secvenţa neechivalentă

```
t := 2 * a * y; y := t + b; z := t + c;
```

Deşi un compilator poate analiza un program pentru a determina pentru fiecare subexpresie în parte dacă poate fi factorizată sau nu, acest lucru cere tehnici sofisticate de analiză globală a fluxului (global flow analysis). Astfel de analize sunt costisitoare şi dificil de implementat. Concluzionăm deci că universul expresiilor prezintă un avantaj asupra universului instrcţiunilor, cel puţin referitor la acest aspect al eliminării subexpresiilor comune (există, oricum, şi alte avantaje care vor fi evidenţiate în continuare).

Scopul programării funcţionale este extinderea la nivelul întregului limbaj de programare a avantajelor pe care le promoveaza universul expresiilor faţă de universul instrucţiunilor.

3. Independenţa ordinii de evaluare

A evalua o expresie înseamnă a-i extrage valoarea. Evaluarea expresiei 6 * 2 + 2 furnizează valoarea 14. Evaluarea expresiei E = (2ax + b)(2ax + c) nu se poate face până nu precizăm valorile numelor a, b, c şi x. Deci, valoarea acestei expresii este dependentă de contextul evaluarii. Odată acest lucru precizat, mai este important să subliniem că valoarea expresiei E nu depinde de ordinea de evaluare (adică de înlocuire a numelor, mai întâi primul factor sau cel de-al doilea, etc). Este posibilă chiar evaluarea paralelă. Aceasta deoarece în cadrul expresiilor pure (aşa cum sunt cele matematice) evaluarea unei subexpresii nu poate afecta valoarea nici unei alte subexpresii.

Definiţie. O expresie pură este o expresie liberă de efecte secundare, adică nu conţine atribuiri nici explicit (gen C) şi nici implicit (apeluri de funcţii).

Această proprietate a expresiilor pure, şi anume independenţa ordinii de evaluare, se numeşte proprietatea Church-Rosser. Ea permite construirea de compilatoare ce aleg ordini de evaluare care fac uz într-un mod cât mai eficient de resursele masinii. Să subliniem din nou potenţialul de paralelism etalat de această proprietate.

Proprietatea de transparenţă referenţială presupune că într-un context fix înlocuirea subexpresiei cu valoarea sa este complet independentă de expresia înconjuratoare. Deci, odata evaluată, o subexpresie nu va mai fi evaluată din nou pentru că valoarea sa nu se va mai schimba. Transparenţa referenţială rezultă din faptul că operatorii aritmetici nu au memorie, astfel orice apel al unui operator cu aceleaşi intrări va produce acelaşi rezultat.

Una dintre caracteristicile noţatiei matematice este interfaţa manifestă, adică, conexiunile de intrare ieşire între o subexpresie şi expresia înconjuratoare sunt vizual evidente (de exemplu în expresia 3 + 8 nu există intrări "ascunse") şi ieşirile depind numai de intrări. Producătorii de efecte secundare, deci şi funcţiile în general, nu au interfaţă manifestă ci o interfaţă nemanifestă (hidden interface).

Să trecem în revistă proprietăţile expresiilor pure:

- valoarea este independentă de ordinea de evaluare;
- expresiile pot fi evaluate în paralel;
- transparenţa referenţiala;
- lipsa efectelor secundare;
- intrările şi efectele unei operatii sunt evidente.

Scopul programării funcţionale este extinderea tuturor acestor proprietăţi la întreg procesul de programare.

Programarea aplicativă are o singură construcţie sintactică fundamentală şi anume aplicarea unei funcţii argumentelor sale. Modurile de definire a funcţiilor sunt următoarele:

1. **Definire enumerativă**. Este posibilă doar când funcţia are un domeniu finit de dimensiune redusă.
2. **Definire prin compunere de funcţii deja definite.** Dacă e cazul unei definiri în termeni de un număr infinit sau neprecizat de compuneri, o astfel de metodă este utilă doar dacă se poate extrage un principiu de regularitate, principiu care să ne permită generarea cazurilor încă neenumerate pe baza celor specificate. Dacă un astfel de principiu există suntem în cazul unei definiţii recursive. În programarea funcţională recursivitatea este metoda de bază pentru a descrie un proces iterativ.

O altă distincţie ce trebuie făcută relativ la definirea de funcţii se referă la definiţii explicite şi implicite. O definiţie explicită ne spune ce este un lucru. O definiţie implicită statuează anumite proprietăţi pe care le are un anumit obiect.

Într-o definiţie explicită variabila definită apare doar în membrul stâng al ecuaţiei (de exemplu y = 2ax). Definiţiile explicite au avantajul că pot fi interpretate drept reguli de rescriere (reguli care specifică faptul că o clasă de expresii poate fi înlocuită de alta).

O variabilă este definită implicit dacă ea este definită de o ecuaţie în care ea apare în ambii membri (de exemplu 2a = a + 3). Pentru a-i găsi valoarea trebuie rezolvată ecuaţia.

Aplicarea repetată a regulilor de rescriere se termină întotdeauna. Este posibil însă ca anumite definiţii implicite să nu ducă la terminare (adică ele nu definesc nimic). De exemplu, ecuaţia fără soluţie a = a + 1 duce la un proces infinit de substituţie.

Un avantaj al programării funcţionale este că, la fel ca şi în cazul algebrei elementare, ea simplifică transformarea de definiţii implicite în definiţii explicite. Acest lucru este foarte important deoarece specificaţiile formale ale sistemelor soft au de obicei forma unor definiţii implicite, însă conversia specificaţiilor în programe are loc mai uşor în cazul definiţiilor explicite.

Să mai subliniem că definiţiile recursive sunt prin natura lor implicite. Totuşi, datorită formei lor regulate de specificare (adică membrul stâng e format numai din numele şi argumentele funcţiei) ele pot fi folosite ca reguli de rescriere. Condiţia de oprire asigură terminarea procesului de substituţii.

Faţă de limbajele convenţionale, limbajele funcţionale diferă şi în ceea ce priveşte modelele operaţionale utilizate pentru descrierea execuţiei programelor. Nu este proprie de exemplu pentru limbajele funcţionale urmărirea execuţiei pas cu pas, deoarece nu contează ordinea de evaluare a subexpresiilor. Deci nu avem nevoie de un depanator în adevăratul înţeles al cuvântului.

Totuşi, limbajul Lisp permite urmărirea execuţiei unei forme Lisp. Pentru aceasta există macrodefiniţia TRACE care primeşte ca argument numele funcţiei dorite, rezultatul întors de TRACE fiind T dacă funcţia respectivă există şi NIL în caz contrar.