Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp şi Prolog. Aplicaţii în Inteligenţa Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

# Capitolul 1

# Aspecte introductive în programarea declarativă

Operarea unui computer tradiţional se bazează pe execuţia secvenţială a unor instrucţiuni încărcate dintr-un mediu de stocare unic, posibil ierarhic. Acest model de calcul durează de mult timp, este aproape universal şi are o influenţă puternică asupra naturii limbajelor de programare, până la punctul în care chiar astăzi suntem forţaţi să vedem programele sub formă de codificări de nivel înalt ale unor secvenţe de instrucţiuni convenţionale. Deşi varietatea limbajelor de programare din ultimii ani a arătat că multe dintre detaliile de nivel scăzut ale arhitecturii maşinii sunt ascunse, permiţând programatorului să se concentreze asupra problemei la nivele de abstractizare mai înalte, rămâne faptul că limbajele de programare prezintă un stil de programare bazat pe oferirea de reţete explicând **cum** trebuie rezolvată o problemă. Prin urmare, programatorul trebuie să aibă tot timpul în minte cum se evaluează programul, deoarece numai astfel el poate produce secvenţa de operaţii corectă pentru rezolvarea problemei. Prin urmare, filosofia din spatele procesului de programare este **"îţi spun eu cum"**. Cu alte cuvinte, se bazează pe descrierea soluţiilor problemelor şi nu pe descrierea problemelor însele. Astfel de limbaje sunt numite **imperative** pentru a reflecta faptul că fiecare instrucţiune dintr-un program este o reţetă depsre ceea ce trebuie făcut în continuare pentru a rezolva problema.

În ciuda statutului curent al limbajelor de programare, tendinţa continuă a fost de a oferi moduri din ce în ce mai abstracte de a rezolva problemele, creând un echilibru între simplitatea programului şi viteza sa de execuţie, fiecare dezvoltare majoră depărtând limbajul tot mai mult de modelul de execuţie secvenţială de instrucţiuni. Pare o abordare naturală să separăm activitatea de programare de modelul computaţional care se acunde în spate. Astfel ne putem depărta de imaginea unui program ca o reţetă pentru calculul unui răspuns şi putem dezvolta un program ca o afirmaţie clară şi concisă despre ce ar trebui să fie răspunsul, ignorând într-o măsură mare modul în care se determină acest răspuns. Filosofia din spatele procesului de programare este, astfel, **"îţi spun eu ce, tu vezi cum"** şi se bazează mai mult pe specificarea abstractă a problemei şi mai puţin pe o descriere a soluţiei.

În ultimul timp au apărut o serie de limbaje care au spart monopolul programării imperative convenţionale, o clasă de astfel de limbaje fiind limbajele **funcţionale** sau **aplicative**. Programele funcţionale sunt construite din funcţii pure, adică funcţii matematice, care sunt în contrast puternic cu funcţiile din limbajele de programare imperative prin aceea că sunt fără efect lateral, aceasta însemnând că evaluarea lor nu poate modifica mediul în care au loc calculele. Aceasta înseamnă că nu putem programa prin efecte, adică valoarea determinată de un program şi programul însuşi sunt unul şi acelaşi lucru. Execuţia unui program devine un proces de alterare a formei valorii finale cerute, exact în modul în care putem altera forma '8+1' la '9' cu conştiinţa faptului că ambele semnifică acelaşi lucru.

## 1.1. Limbaje procedurale

Pentru început vom da două definiţii ale noţiunii de **program**:
(1) un program **în sens restrictiv** este o secvenţă de instrucţiuni pentru un calculator;
(2) un program **în sens larg** este o descriere de valori, proprietăţi, metode şi soluţii. Rolul maşinii este de a mări viteza manipulării şi evaluării acestor descrieri, pentru a oferi soluţii unor probleme particulare. **Un limbaj de programare** este o convenţie pentru scrierea de descrieri care pot fi evaluate.

Limbajele de programare convenţionale (ca de exemplu Fortran, Cobol, Algol, Pascal, C, Modula, Ada) folosesc în mod esenţial instrucţiunile de atribuire ca şi construcţie de bază, în jurul căreia sunt construite structurile de control al execuţiei secvenţiale, ramificării şi ciclării. Astfel de limbaje de programare se numesc **imperative** sau **procedurale**, din cauza modului în care programele se bazează pe ideea de instrucţiune care se poate executa ca o reţetă. Aceste instrucţiuni transformă în mod incremental o zonă de celule prin actualizarea valorii celulelor, pentru a realiza un anumit efect global. De exemplu, atribuirea

$$X := X + 1$$

este gândită ca o instrucţiune de actualizare a conţinutului celulei de memorie asociată cu variabila de stocare X, prin adăugarea lui 1 valorii curente.

Limbajele procedurale au evoluat ca abstracţiuni de la modelul von Neumann al maşinilor secvenţiale şi mecanismele lor de control s-au ridicat ca abstracţiuni de la utilizarea unui contor de program pentru localizarea următoarei instrucţiuni şi pentru executarea de salturi (prin actualizarea contorului de program).

Consecinţa principală a existenţei variabilelor care se referă direct la celule de stocare este aceea că programatorul are sarcinile:
(a) de a organiza utilizarea şi reutilizarea acestora pentru a memora valori diverse;
(b) de a distribui valorile mari de-a lungul mai multor celule (de exemplu, cazul vectorilor).

Programatorul trebuie să realizeze mai multe lucruri în acelaşi timp, şi anume:
(1) să descrie ce anume este de calculat;
(2) să organizeze secvenţa de calcul în paşi mici;
(3) să organizeze managementul memoriei în timpul calculului.

### 1.2. Separarea preocupărilor

Ideal, programatorul ar trebui să se poată concentra asupra primei preocupări din lista de mai sus (aceea de a descrie ce anume este de calculat) fără să fie distras de celelalte două preocupări, mai administrative. In mod clar, administrarea este importantă, dar prin separarea acesteia de preocuparea principală vom obţine rezultate mai bune şi putem uşura activitatea de programare prin automatizarea cât de mult posibil a administrării.

Separarea preocupărilor mai are şi alte avantaje. De exemplu, **demonstrarea corectitudinii programelor** devine mult mai fezabilă când detaliile de secvenţionare şi management al memoriei lipsesc din program. Mai mult, descrierea a ceea ce trebuie determinat nu trebuie să conţină o descriere detaliată pas-cu-pas a modului în care trebuie să se procedeze, dacă problema trebuie evaluată pe maşini cu mai multe arhitecturi (de exemplu, maşina puternic paralelă, cu mii de procesoare şi facilităţi de stocare locală).

Automatizarea aspectelor administrative înseamnă că implementatorul limbajului trebuie să se gândească la ele, dar acesta are mai multe posibilităţi de a utiliza diferite mecanisme de calcul cu diferite arhitecturi. De exemplu, implementatorul ar putea dori să utilizeze cât de mult paralelism se poate când acesta este disponibil, şi să traseze un echilibru între accesul partajat şi copierea datelor. Există un echilibru între abilitatea programatorului de a controla utilizarea eficientă a spaţiului de stocare pentru un anumit program şi posibilităţile de descoperire automată a altor eficienţe ale programelor. Uneori un compilator scris bine poate lucra mai mult decât un programator la localizarea optimizărilor, în special când nu este frânat de detalii din program nenecesare şi necorespunzătoare.

Din acest punct de vedere se susţine că instrucţiunea de **atribuire** este considerată periculoasă în limbajele de nivel înalt, tot aşa cum instrucţiunea **goto** a fost considerată periculoasă pentru programarea structurată în anii '68. De notat că instrucţiunea de atribuire preia majoritatea blamărilor pentru faptul că nu permite separarea preocupărilor în programare. Chiar în majoritatea limbajelor procedurale un anumit grad de libertate faţă de detaliile explicite de control şi memorare se poate vedea în modul în care expresiile sunt utilizate (de obicei în partea dreaptă a instrucţiunilor de atribuire). De exemplu să considerăm următoarea expresie care, întâmplător, descrie suma întregilor dintre două numere întregi date, *m* şi *n*:

$$((m + n) * \text{abs}\ (m - n) + 1))\ \text{div}\ 2$$

Expresia descrie aplicarea mai multor funcţii şi referirea la valorile întregi *m, n*, 1 şi 2. Funcţiile implicate sunt adunarea, înmulţirea, scăderea şi împărţirea întreagă, precum şi valoarea absolută. În expresie remarcăm absenţa anumitor detalii, ca de exemplu locul unde sunt memorate rezultatele intermediare. De asemenea, putem vedea potenţialul pentru câteva ordini de evaluare diferite, incluzând evaluarea paralelă a subexpresiilor ***m* + *n*** şi **abs (*m* - *n*) + 1**. Un alt punct important este că apariţia lui *m* şi *n* în expresie semnifică valori întregi, nu celule de memorie în care se poate depozita o valoare întreagă. In contrast cu aceasta, variabilele din partea stângă a unei instrucţiuni de atribuire sunt utilizate ca identificatori pentru variabilele de stocare. De exemplu, în:

$$X := X + 1$$

prima apariţie a lui X denotă o celulă de memorare, şi nu conţinutul întreg al acesteia, iar a doua apariţie a lui X semnifică un întreg (conţinutul aceleiaşi celule). In expresii uzuale accentul este pe valorile datelor şi nu pe administrarea conceptului de locaţie (celulă).

### 1.3. Limbaje declarative (aplicative)

Limbajele de nivel foarte înalt pe care le vom vedea în continuare se bazează mult mai strâns pe expresii şi sunt denumite cu diverse nume: **limbaje aplicative** sau **descriptive** sau **declarative**. Proiectarea lor nu este atât de mult influenţată de detalii referitoare la o anumită maşină de calcul, ci mai degrabă de o înţelegere matematică clară a descrierilor. Desigur, limbajele declarative nu sunt doar limbaje imperative cu anumite facilităţi înlăturate, ci implică metode alternative pentru descrierea datelor şi calculelor.

Să considerăm dezideratele unui limbaj care trebuie să fie respectate de programarea descriptivă de nivel înalt şi nu programarea controlată de imaginea asupra maşinii.

(1) Limbajul trebuie să fie **expresiv**, astfel încât descrierile problemelor, situaţiilor, metodelor şi soluţiilor să nu fie prea dificil de scris.

(2) Trebuie să aibă o **bază simplă** şi uniformă, pentru a nu fi dificil de înţeles.
(3) Aceasta sugerează că trebuie să fie **extensibil** şi să permită utilizatorului să extindă limbajul cu uşurinţă, plecând de la o bază simplă, pentru a i se potrivi nevoilor sale particulare, mai degrabă decât să aibă o colecţie masivă de construcţii primitive.
(4) Limbajul ar trebui, de asemenea, ca în măsura posibilului să **protejeze** utilizatorii de la a face prea multe erori (de exemplu prin interzicerea utilizării inconsistente a descrierilor).
(5) Ar trebui să fie posibil să se scrie programe care rulează eficient pe maşinile disponibile actualmente.
(6) Finalmente, ar trebui să fie matematic **elegant** pentru a permite un suport matematic în activităţile de programare majore. Mai specific, analiza, proiectarea, specificarea, implementarea, abstractizarea şi raţionarea (deducţii ale consecinţelor şi proprietăţilor) devin activităţi din ce în ce mai formale. Limbajele declarative sunt de obicei generate din principii matematice. In ultimă instanţă, o parte substanţială a activităţii matematice a fost centrată pe descrierea şi raţionarea cu obiecte complexe şi ar fi fost neînţelept să ignorăm în programare această moştenire de notaţii şi concepte.

Deplasarea programării spre o abordare mai declarativă umple golul dintre noţiunea de program şi cea de specificaţie. Cele două noţiuni încep să se suprapună într-un singur spectru, unde orice specificaţie executabilă este un program şi unde există grade variate de eficienţă asociate cu specificaţiile. Devine din ce în ce mai fezabil să generăm programe eficiente prin transformarea specificaţiilor pentru a obţine gradul de eficienţă dorit. Se poate construi o colecţie de instrumente generale de transformare pentru utilizare alături de un compilator, pentru a asista programatorul.

Există două clase de limbaje declarative:
(1) **limbajele funcţionale** (de exemplu, Lisp), care se focalizează pe valori ale datelor descrise prin expresii (construite prin aplicări ale funcţiilor şi definiţii de funcţii), cu evaluare automată a expresiilor;
(2) **limbaje logice** (de exemplu, Prolog), care se focalizează pe aserţiuni logice care descriu relaţiile dintre valorile datelor şi derivări automate de răspunsuri la întrebări, plecând de la aceste aserţiuni.

In ambele cazuri programele pot fi văzute ca descrieri care declară informaţii despre valori, mai degrabă decât instrucţiuni pentru determinarea valorilor sau efectelor.

Uneori termenul **funcţional** este folosit doar pentru acele limbaje cu o absenţă totală a facilităţilor procedurale explicite (controlul memorării). In acest caz, cele mai multe variante de Lisp nu ar fi considerate funcţionale deoarece suportă unele facilităţi procedurale (atribuiri). Multe din beneficiile abordării funcţionale sunt pierdute odată cu introducerea acestor facilităţi procedurale. **Ca atare recomandăm utilizarea facilităţilor procedurale în Lisp (atribuiri, structuri nerecursive de ciclare) doar când este absolut necesar sau când problema o cere.** In aceste situaţii se va justifica necesitatea utilizării facilităţilor procedurale.

### 1.4. Recursivitate

Recursivitatea este una din noţiunile fundamentale ale informaticii, fiind un mecanism general de elaborare a programelor.

Două aspecte sunt esenţiale în abordarea recursivităţii şi anume:

- recursivitatea a apărut din necesităţi practice (transcrierea directă a formulelor matematice recursive; vezi funcţia lui Ackermann);
- recursivitatea este acel mecanism prin care un subprogram (funcţie, procedură) se autoapelează.

Din punct de vedere al modului în care se realizează autoapelul, există două tipuri de recursivitate: directă sau indirectă. Dacă subalgoritmul se autoapelează explicit, vorbim despre **recursivitate directă**. Daca, în schimb, subalgoritmul apelează un alt subalgoritm, care la rândul său îl apelează pe acesta, vorbim despre **recursivitate indirectă**.

Când scriem un algoritm recursiv trebuie să avem în vedere două lucruri. Pe de o parte, trebuie să realizăm **regula recursivă**: este suficient să ne gândim la ceea ce se întâmplă la un anumit nivel, pentru că la orice nivel se întâmplă exact acelaşi lucru. Pe de altă parte, trebuie să realizăm **condiţia de ieşire din recursivitate**: un algoritm recursiv corect, ca orice algoritm, trebuie să se termine, în caz contrar programul va intra într-o buclă infinită şi nu vom obţine rezultatul aşteptat.

Unul din avantajele recursivităţii este obţinerea unui text sursă extrem de scurt şi foarte clar. Bineînţeles că dezavantajul major al recursivităţii este umplerea segmentului de stivă în cazul în care numărul apelurilor recursive, respectiv ai parametrilor formali şi locali ai subprogramelor recursive este mare. Acest lucru va putea fi prevenit folosind anumite mecanisme specifice limbajului de implementare (vezi mecanismul recursivităţii de coadă în Prolog).