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

Capitolul 2

Importanţa programării funcţionale ca nouă metodologie de programare

2.1. O nouă paradigmă de programare

Programarea funcţională este în primul rând o nouă paradigmă de programare, comparabilă în importanţă cu cea a promovării programării structurate. Este important de subliniat că programarea funcţională aduce cu sine o nouă disciplină de programare, utilizabilă chiar şi în absenţa unor limbaje de programare.

O abordare completă a problematicii programării funcţionale trebuie să conţină toate cele trei ipostaze sub care poate fi folosită aceasta, şi anume:

1. metodă pentru dezvoltarea de programe fiabile;
2. mijloc de analiză a eficienţei programelor;
3. metodă de analiză a corectitudinii programelor.

Ca principiu de lucru, metodologia programării funcţionale poate fi utilizată cu aceeaşi eficienţă chiar dacă limbajul de implementare nu este unul specific funcţional. De aceea spunem că programarea funcţională este o nouă paradigmă (principiu) de programare şi nu numai o nouă metodă de programare.

Programarea funcţională se mai cunoaşte şi sub denumirile de programare aplicativă sau programare orientată spre valoare. Importanţa ei pentru momentul actual rezultă din următoarele motive (care reprezintă totodată şi trăsături definitorii ale programării funcţionale):

- programarea funcţională renunţă la instrucţiunea de atribuire prezentă ca element de bază în cadrul limbajelor imperative; mai corect spus această atribuire este prezentă doar la un nivel mai scăzut de abstractizare (analog comparaţiei între **goto** şi structurile de control structurate **while**, **repeat** şi **for**);
- programarea funcţională încurajează gândirea la un nivel de abstractizare mai ridicat, permiţând prin intermediul funcţiilor de ordin superior (aşa-numitele forme funcţionale) modificarea comportamentului programelor existente precum şi combinarea acestora (practică numită *programming in the large* - lucrul cu unităţi mai mari decât instrucţiuni individuale).
- principiile de lucru ale programării funcţionale se potrivesc cu cerinţele programării masiv paralele: absenţa atribuirii, independenţa rezultatelor finale de ordinea de evaluare şi abilitatea de a opera la nivelul unor întregi structuri sunt trei motive pentru care se pare că limbajele funcţionale se vor impune în faţa celor imperative la momentul trecerii în masă spre programarea paralelă;
- studiul programării funcţionale este strâns legat de semantica denotaţională a limbajelor de programare; esenţa acestei semantici este translatarea programelor convenţionale în programe funcţionale echivalente.

Inteligenţa artificială a stat la baza promovării programării funcţionale. Obiectul acestui domeniu constă din studiul modului în care se pot realiza cu ajutorul calculatorului comportări care în mod obişnuit sunt calificate ca inteligente. Inteligenţa artificială, prin problematica aplicaţiilor specifice (simularea unor procese cognitive umane, traducerea automată, regăsirea informaţiilor) necesită, în primul rând, prelucrări simbolice şi mai puţin calcule numerice. Această "putere" sporită de exprimare a calculelor (simbolice) s-a putut obţine pe baza unor modele formale matematice logice cum ar fi calculul lambda (Lisp) sau algoritmii Markov (Snobol). Aceste modele au fost elaborate în 1941 şi respectiv 1954, independent de existenţa calculatoarelor, ca instrumente de lucru în studiul teoretic al calculabilităţii. Ele s-au dovedit extrem de generale şi utile, reprezentând împreună cu funcţiile general recursive, maşina Turing şi sistemele Post, modele formale mutual echivalente în definirea noţiunii de algoritm.

Caracteristica limbajelor de prelucrare simbolică a datelor constă în posibilitatea manipulării unor structuri de date oricât de complexe, structuri ce se construiesc dinamic (în cursul execuţiei programului). Informaţiile pe care le oferă aceste date evidenţiază în principal proprietăţi ale obiectelor precum şi relaţii între acestea. Datele sunt de obicei şiruri de caractere, liste sau arbori binari.

Nu se va insista prea mult asupra unor chestiuni de implementare a limbajelor functionale, din următoarele motive:

- pe arhitecturi convenţionale limbajele funcţionale sunt implementate folosind tehnicile utilizate pentru implementarea funcţiilor în limbaje structurate ce admit recursivitatea (Pascal, Modula, C);
- implementarea limbajelor funcţionale pe arhitecturi neconventionale se schimbă foarte rapid la ora actuală, fiind încă un domeniu de cercetare. Informatiile disponibile acum ar deveni în curând depăşite, ceea ce face de prisos abordarea acestora într-un curs de bază.

2.2. Scurt istoric al programării funcţionale

Georg Cantor (1845-1918) şi Leopold Kronecker (1823-1891) au afirmat că un obiect matematic există numai dacă el poate fi construit (cel putin în principiu). S-a pus astfel problema "ce înseamnă că un numar sau un obiect matematic este construibil?".

Giuseppe Peano (1858-1932) a arătat că numerele naturale pot fi construite printr-un număr mare de aplicări ale funcţiei succesor. Începând cu 1923, Thoralf Skolem (1887-1963) a arătat că aproape toată teoria numerelor naturale poate fi dezvoltată constructiv pe baza utilizării extensive a definirilor recursive. Astfel, începutul secolului 20 aduce cu sine o experienţă considerabilă în domeniul definiţiilor recursive a funcţiilor de numere naturale.

Pentru a evita utilizarea infinitului, s-a definit obiectul construibil ca un obiect ce poate fi construit într-un număr finit de paşi, fiecare pas cerând o cantitate finită de efort. Încercările de formalizare a unei astfel de definiţii au început în anii '30, ducând la apariţia termenului de calculabilitate.

Prima încercare a fost definiţia dată de Turing noţiunii de *clasă de maşini abstracte* (cunoscute de atunci sub numele de maşini Turing), maşini ce efectuează citiri şi scrieri simple pe o porţiune finită de bandă. Alte tentative:

- introducerea funcţiilor general recursive de către Kurt Godel (1906-1978);

- dezvoltarea calculului lambda de către Church şi Kleene;
- algoritmii Markov;
- sistemele de producţie Post.

Este foarte interesant de subliniat faptul că toate aceste noţiuni de calculabilitate formulate independent s-au dovedit a fi echivalente (Church, Kleene şi Turing au dovedit-o). Această echivalenţă l-a facut pe Church să propună teza ce-i poartă numele (Church thesis) şi anume ca noţiunea de funcţie calculabilă să fie identificată cu noţiunea de funcţie general recursivă (ideea acestora din urma apartine matematicianului francez Jacques Herbrand (1908-1931)).

Astfel, în perioada imediat următoare inventării primelor calculatoare electronice s-a aratat că orice funcţie calculabilă poate fi exprimată (deci şi programată) în termeni de funcţii recursive.

Următorul eveniment major pentru istoria programării funcţionale l-a constituit publicarea în 1960 de catre John McCarthy a unui articol despre limbajul Lisp. In 1958 McCarthy a investigat utilizarea operaţiilor cu liste la implementarea unui program de diferenţiere simbolică. Cum diferenţierea este un proces recursiv, McCarthy a fost condus spre utilizarea de funcţii recursive. Mai mult, a avut nevoie de posibilitatea de a transmite funcţii ca argumente ale altor funcţii. Calculul lambda l-a ajutat în acest sens şi astfel McCarthy a ajuns la folosirea notaţiilor lui Church pentru programele sale.

Aceasta este motivaţia demarării unui proiect în 1958 la MIT care avea ca scop implementarea unui limbaj ce încorpora astfel de idei. Rezultatul a fost Lisp 1, descris de McCarthy în articolul "Recursive Functions of Symbolic Expressions and Their Computation by Machine" (1960). Acest articol (care arată cum un număr foarte mare de programe importante pot fi exprimate ca funcţii pure ce operează asupra unor structuri de tip listă) este considerat ca punct de început al programării funcţionale.

Spre sfârşitul anilor '60 Peter Landin a încercat să definească limbajul non-funcţional Algol 60 prin intermediul calculului lambda. Această abordare a fost continuată de Strachey şi Scott şi a dus la o metodă de definire a semanticii limbajelor de programare cunoscută sub numele de semantica denotaţională. În esenţă, semantica denotaţională defineşte înţelesul unui program în termenii unui program funcţional echivalent.

Cercetarea pe scară largă în domeniul programării funcţionale a fost demarată în urma unui articol a lui John Backus, articol apărut în CACM în 1978: "Can Programming Be Eliberated of the Von Neumann Style?". In acest articol, inventatorul Fortranului a adus o critică severă limbajelor de programare convenţionale, chemând la dezvoltarea unei noi paradigme de programare pe care a denumit-o programare funcţională. Majoritatea formelor funcţionale propuse de Backus au fost inspirate de limbajul APL, un limbaj imperativ dezvoltat în anii '60 şi care prevedea operatori puternici de lucru la nivelul unor întregi structuri de date.

2.3. Limbaje funcţionale

Orice limbaj de programare poate fi împărţit în două componente:

- componentă **independentă** de domeniul de aplicaţii ales, numită **cadru**, care include mecanismele sintactice (lingvistice) de bază utilizate în construirea programelor. Pentru un limbaj imperativ cadrul conţine structurile de control, mecanismul de apel de procedură şi

operaţia de atribuire. Cadrul unui limbaj aplicativ include definiţia de funcţie şi mecanismul de aplicare a funcţiei.

- componentă **dependentă** de domeniul de aplicaţii ales, ce conţine componente utile acelui domeniu. Dacă domeniul este, de exemplu, programarea numerică, cadrul conţine numerele reale, operaţii (+,-) şi relaţii (=, < etc.).

Deci cadrul prevede forma iar componentele conţinutul pe baza căruia se construiesc programele. Alegerea cadrului determină tipul limbajului (aplicativ sau imperativ). Alegerea componentelor orientează limbajul către o clasă de probleme (de exemplu, numerice sau simbolice).

Un tip de dată se numeste abstract dacă el este specificat în termenii unei multimi (abstracte) de valori şi operaţii şi nu în termenii unei implementări concrete.

O componentă aflată în majoritatea limbajelor aplicative (şi lipsind din majoritatea limbajelor imperative) este structura de dată secvenţă. Secvenţa este un tip abstract de date. Mai mult, ea este şi un tip generic (secvenţe de întregi, secvenţe de string-uri, sunt tipuri particulare). Ea este implementată ca listă liniară simplu înlanţuită. Pentru a decide care trebuie să fie operaţiile primitive asupra secvenţei ne ajută recursivitatea (ţinând cont şi de faptul că dorim un numar cât mai mic de operaţii). Operaţia prefix (**cons**) a fost preferată celei postfix datorită modalităţii de implementare a listei liniare simplu înlănţuite.

Relativ la operaţiile de care avem nevoie la definirea unui tip abstract de date, acestea trebuie să fie:

(i) **constructori** - operaţii care construiesc un tip abstract de date într-un număr finit de paşi pe baza furnizării componentelor;
(ii) **selectori** - operaţii de selecţie de elemente conform unor criterii dorite;
(iii) **discriminatori** - operaţii capabile să compare diferite clase şi instanţe ale tipului abstract de date pentru a verifica dacă li se pot aplica selectorii.

În ceea ce priveşte limbajele de programare funcţionale actuale, ele suferă de lipsa unor notaţii sintactice standard. Acest lucru se întâmplă deoarece majoritatea acestor limbaje sunt încă limbaje experimentale, unul dintre scopuri fiind chiar experimentarea notaţiilor. Pe de altă parte, se remarcă la baza acestor limbaje un acelaşi fundament şi anume calculul lambda. Majoritatea sistemelor de programare funcţională acceptă comenzi de trei tipuri:

(1) Definiţii de funcţii
(2) Definiţii de date
(3) Evaluări de expresii

Programele funcţionale sunt mai apropiate de specificaţiile formale ale sistemelor de programe. Spre deosebire de limbajele convenţionale, limbajele funcţionale tratează funcţia ca un obiect fundamental (principiul regularităţii sau uniformităţii) ce poate fi tranmis ca parametru, returnat ca rezultat al unei prelucrări, constituit ca parte a unei structuri de date etc. Acest lucru sporeşte puterea de abstractizare a unui limbaj.

2.4. Scurt istoric al limbajului Lisp

În 1956 John McCarthy organizează la Dartmouth o întâlnire cu cercetători de inteligenţă artificială. S-a căzut de acord asupra necesităţii existenţei unui limbaj de programare destinat special cercetărilor de inteligenţă artificială.

Până în 1958 McCarthy elaborează o primă formă a unui limbaj (Lisp - LISt Processing) destinat prelucrărilor de liste, formă ce se baza pe ideea transcrierii în acel limbaj de programare a expresiilor algebrice.

Impunerea Lisp-ului ca limbajul de bază al programării aplicaţiilor de inteligenţă artificială s-a făcut însă numai în urma unor concesii din partea autorului, concesii destinate obţinerii unor implementări mai eficiente precum şi a unor exprimări mai concise. Ultima versiune elaborată de autor a fost Lisp 1.5 (1962). Implementările ulterioare succesive au remediat o parte din neajunsurile acelei versiuni, creându-se o serie de noi versiuni, majoritatea extensii. Nu există încă o versiune etalon, încercându-se abia în ultimii ani o standardizare. Mediile de programare Lisp sunt programe de mari dimensiuni (scrise de obicei tot în Lisp) ce oferă facilităţi de creare şi depanare interactivă a programelor Lisp şi de gestiune a tuturor funcţiilor unui program. În general un mediu Lisp este conceput având la bază un interpretor, existând însă versiuni ce conţin şi un compilator.

Cele mai cunoscute medii Lisp sunt

- InterLisp (MIT, 1978);
- MacLisp (MIT, 1970, DEC-10);
- Franz Lisp (Berkeley, sub UNIX).

MacLisp a permis dezvoltarea uneia dintre primele aplicaţii de reală utilitate: programul MACSYMA, destinat prelucrărilor simbolice în domeniul matematic.

Iată în continuare o prezentare succintă a Lisp-ului, făcută chiar de autorul lui:

- calcule cu expresii simbolice în loc de numere;
- reprezentarea expresiilor simbolice şi a altor informaţii prin structura de listă;
- compunerea funcţiilor ca instrument de formare a unor funcţii mai complexe;
- utilizarea recursivităţii în definiţia funcţiilor;
- reprezentarea programelor Lisp ca date Lisp;
- funcţia Lisp **eval** care serveşte deopotrivă ca definiţie formală a limbajului şi ca interpretor;
- colectarea spaţiului disponibil (*garbage collection*) ca mijloc de tratare a problemei dealocării;
- instrucţiunile Lisp se interpretează ca şi comenzi atunci când este folosit un mediu conversational.

Domeniul de utilizare al limbajului Lisp este cel al calculului simbolic, adică al prelucrărilor efectuate asupra unor şiruri de simboluri grupate în expresii. Una dintre cauzele forţei limbajului Lisp este posibilitatea de manipulare a unor obiecte cu structură ierarhizată.

Domeniile inteligenţei artificiale în care limbajul Lisp a găsit un larg câmp de aplicare sunt demonstrarea teoremelor, învăţarea, studiul limbajului natural, înţelegerea vorbirii, interpretarea imaginilor, sistemele expert, planificarea roboţilor etc.