Capitolul I

Introducere în λ-calcul (lambda-calcul)

λ-calculul (sau lambda-calculul) este o teorie a funcţiilor, care iniţial a fost dezvoltată de logicianul Alonzo Church ca fundament al matematicii. Această teorie a fost elaborată în anii 1930, cu mult înainte de a fi inventate computerele . Puţin mai devreme (prin anul 1920), Moses Schönfinkel dezvoltase o altă teorie a funcţiilor bazată pe ceea ce numim în zilele noastre teoria *combinatorilor*.

În anii 30, Haskell Curry a redescoperit şi extins teoria lui Schönfinkel şi a demonstrat (arătat) că era echivalentă cu λ-calculul. Referitor la acest lucru Kleene a arătat că λ-calculul este un sistem universal de calcul (în sensul că poate exprima toate *funcţiile intuitiv calculabile*); a fost primul sistem de genul acesta care a fost riguros analizat.

În anii 50, John McCarthy a fost inspirat de λ-calcul, inventând limbajul de programare LISP.

La începutul anilor 60, Peter Landin a arătat (demonstrat) cum pot fi descrise (declarate, specificate) semanticile limbajelor de programare imperativă, prin traducerea (transpunerea) lor în λ-calcul. De asemenea, el a inventat prototipul unui puternic limbaj de programare numit ISWIM [24]. Acesta a introdus principalele notaţii ale programării funcţionale şi a influenţat design-ul pentru ambele feluri de limbaje: funcţional şi imperativ.

Dezvoltând pe această teorie, Christopher Strachey a pus bazele domeniului important al semanticilor denotaţionale. Unele chestiuni tehnice din lucrările teoretice ale lui Strachey au condus pe matematicianul şi logicianul Dana Scott la descoperirea teoriei domeniilor, care acum este una dintre cele mai importante părţi din informatica teoretică.

În timpul anilor 70, Peter Henderson şi Jim Morris au pornit de la lucrarea lui Landin şi au scris un număr important de lucrări, argumentând că programarea funcţională ar constitui un instrument important pentru producătorii de software. Aproape în aceeaşi perioadă, David Turner a demonstrat despre *combinatorii* lui Schönfinkel şi Curry că ei ar putea fi folosiţi ca un fel de cod al calculatorului pentru a compila şi executa programe scrise în limbaje de programare funcţională.

Asemenea calculatoare pot beneficia de proprietăţile matematice ale lambda -calculului pentru a realiza o execuţie paralelă a programelor. În timpul anilor 80, mai multe grupuri de cercetare au preluat ideile lui Henderson şi Turner şi au început să lucreze pentru a implementa computere pentru programare funcţională prin conceperea unor arhitecturi speciale , unele dintre acestea având mai multe procesoare.

Noi astăzi, privind retrospectiv, vedem că o ramură obscură a logicii matematice stă la baza dezvoltării teoriei limbajelor de programare, din ea rezultând:

(i) Studiul chestiunilor fundamentale privind calculabilitatea
(ii) Designul limbajelor de programare
(iii) Semantica limbajelor de programare
(iv) Arhitectura computerelor

1.1. Sintaxa şi semantica lambda-calculului

Lambda-calcul este o notaţie pentru a defini funcţii. Expresiile notaţiei sunt numite λ-*expresii* şi fiecare asemenea expresie defineşte o funcţie. Se va vedea ulterior cum pot fi folosite funcţiile pentru a reprezenta o largă varietate de date şi structuri de date,

inclusiv numere, perechi, liste etc. De exemplu, va fi demonstrat mai departe cum o pereche arbitrară de numere (x,y), poate fi reprezentată ca lambda-expresie. Ca o convenţie de notaţie, mnemonicele sunt evidenţiate în **bold** (îngroşat) sau subliniat, în expresii lambda particulare, de exemplu: **1** este lambda-expresie (definită în subcapitolul 2.3), care este folosită pentru reprezentarea numărului unu (1).

Sintaxa. Sunt numai 3 feluri de λ-expresii:

(i) **variabilele**: x, y, z etc. Funcţiile ascunse în spatele unor nume de variabile sunt determinate de valorile (din context) de care sunt legate aceste variabile. Asemenea valori care denotă funcţi se numesc abstracţii (vezi 3 mai jos). Noi folosim V_1, V_3, V_3 etc. pentru variabile arbitrare.

(ii) **aplicarea de funcţii** sau **combinaţia**: dacă E1 şi E2 sunt λ-expresii, atunci şi (E1,E2) este tot o λ-expresie; indică rezultatul aplicării funcţiei indicate de E1, funcţiei indicate de E2. E1 este numită *rator* (de la ope*rator*) şi E2 este numită *rand* (de la ope*rand*). De exemplu, dacă (*m*, *n*) indică o funcţie (*De menţionat că în metalimbajul pe care îl folosim pentru a vorbi despre lambda-calcul* ***sum*** *este o lambda-expresie, în timp ce + este un simbol matematic !*) reprezentând o pereche de numere *m* şi *n* (vezi subcap. 2.2) iar *sum* (suma) indică o funcţie de adunare din λ-calcul, atunci aplicarea (**sum**(*m*, *n*)), indică (înseamnă) de fapt, suma m+n.

(iii) **formule abstracte (abstracţii)**: dacă V este o variabilă şi E este o λ-expresie, atunci şi λV. E este o o expresie numită abstracţie, având corpul E (*body E*) şi variabila legată V (the *bound variable V*) . Asemenea **abstracţii** indică, notează, funcţii ce iau ca argument pe *a* şi returnează ca rezultat funcţia indicată de E, într-un context în care legătura variabilei V, indică pe *a*. Mai specific, **abstracţia λV. E** indică o funcţie care dacă primeste un argument E' şi îl transformă în ceva notat cu E [E'/V] (notaţia înseamnă rezultatul substituirii lui V cu E' pentru toate apariţiile libere ale lui V în E, vezi subcap. 1.8). De exemplu, λx.**sum**(x,1) indică o funcţie de adunare cu o unitate.

Folosind BNF, sintaxa lambda-expresiilor este dată doar de 3 reguli:

<λ-expresia> ::= <variabila>
| (<λ-expression > <λ-expression >)
| (λ<variabila > . <λ-expresie >)

Dacă V depăşeşte sintaxa Notand cu V clasa <variabila> iar cu E, E1, E2, …elemente cu sintaxa din clasa <λ-expresie >, atunci notatia BNF o simplificăm scriind-o astfel:

E ::=V | (E1 E2) | λ V.E

unde: V – variabile, (E1E2) – aplicarea şi λ V.E – abstracţia

Descrierea a ceea ce înseamnă λ-expresie, aşa cum a fost exemplificată mai sus, este totuşi vagă şi intuitivă. Au fost necesari 40 de ani pentru logicieni (Dana Scott, în [32]), pentru a o face riguroasă şi folositoare. Nu vom intra în detalii privind acaestă muncă istorică.

Un exemplu: (λx. x) denotă, indică *funcţia identitate*: ((λx. x) E) = E.