

BCPL

Pe scurt, despre BCPL

BCPL (*Basic Combined Programming Language*) era un limbaj de programare procedural, imperativ și structurat. Limbajul a fost imaginat și implementat pentru prima dată de Martin Richards, de la Universitatea din Cambridge, în 1967. Destinat inițial scrierii de compilatoare pentru alte limbaje, a fost o adevărată bijuterie software a vremii. A avut o descendență ilustră: din el s-a derivat în 1969 limbajul B (Ken Thompson și Dennis Ritchie), din care s-a născut ulterior limbajul C. Richards, preocupat de soarta limbajului său, a utilizat multe concepte de informatică teoretică și programare apărute în anii '60 pentru a facilita implementarea limbajului pe cele mai diverse platforme.

BCPL a fost conceput astfel încât compilatoarele sale să fie mici și ușor de portat. Richards a implementat compilatorul chiar în BCPL, ceea ce îl făcea *self-hosting* — o condiție necesară pentru o portabilitate reală. Se știe însă că un compilator este în mod inerent dependent de mașina-țintă (cea pentru care se scrie), deoarece o parte din el — cea care se ocupă de generarea codului — trebuie rescrisă pentru fiecare mașină diferită, plecând de la un bagaj de software comun pentru orice platformă (program memorat pe benzi perforate sau magnetice).

„Ciudățenia” compilatorului BCPL, în concepția lui Richards, constă într-o formă intermediară explicită, pe care compilatorul o generează ca interfață între partea independentă și partea dependentă de mașină, pentru ca apoi, din această formă intermediară, să se genereze codul-mașină efectiv. Această formă intermediară este scrisă într-un limbaj numit OCODE și era privită ca limbajul de asamblare al unei mașini abstracte simple, orientate spre BCPL. Mașina abstractă avea o memorie liniară pentru salvarea variabilelor, constând din celule de dimensiuni egale, adresabile prin numere întregi, astfel încât numerele întregi consecutive să se refere la celulele adiacente. Această abordare a devenit o practică obișnuită ulterior (de exemplu, în Pascal, Java).

BCPL a introdus mai multe caracteristici ale multor limbaje moderne, inclusiv utilizarea acoladelor pentru delimitarea blocurilor de cod. Limbajul este neobișnuit și prin faptul că are un singur tip de date: un cuvânt — un număr fix de biți — ales de obicei pentru a se alinia cu cuvântul-mașină al arhitecturii platformei pe care se face implementarea și cu o capacitate adecvată pentru a reprezenta orice adresă de stocare validă.

Implementarea noastră a BCPL-ului

În 1974, Mariana Necula, colega mea de catedră, a adus din Anglia, pe o bandă perforată, sursa compilatorului limbajului BCPL, scrisă într-un limbaj de nivel intermediar numit INTCODE, care genera cod INTCODE. Această versiune fusese recent perfecționată de Martin Richards, autorul limbajului, pentru a simplifica procedura de „portare” a compilatorului BCPL pe diverse platforme.

Compilatorul original pentru BCPL fusese scris de Richards în însuși limbajul BCPL, în ideea de a-l face cât mai portabil. Dar soluția cu adevărat populară propusă de Richards pentru implementarea portabilă a lui BCPL a fost alegerea metodei de *bootstrapping*. Prin „bootstrapping” ne referim, de obicei, la procesul prin care un sistem simplu inițializează și/sau activează un sistem complex. Ca să fim mai preciși, există două tipuri de bootstrapping:

- **Whole bootstrap** implică scrierea unui compilator într-un limbaj de nivel înalt, compilarea lui cu un compilator deja existent, apoi folosirea noii versiuni pentru a se recompila singură — *self-hosting*.

- **Half-bootstrap** se referă, ca în cazul BCPL, la faptul că o parte fixă, minimă a sistemului (de exemplu, un interpretor pentru un cod intermediar) este scrisă direct pe mașina-țintă; apoi aceasta este folosită pentru a rula sau a genera restul sistemului.

În cazul BCPL, bootstrapping-ul presupunea:

1. un compilator BCPL scris în BCPL (deci *self-hosting*),
2. care genera cod intermediar: OCODE (*machine-independent*),
3. iar pentru „portare” pe o mașină nouă era nevoie doar de un translator OCODE → cod nativ sau, varianta mai simplă, de un interpretor OCODE, scris direct pentru acea mașină.

În versiunea noastră pentru bootstrapping-ul implementării compilatorului BCPL, Martin Richards introducea o mașină abstractă INTCODE, al cărei cod-mașină are un limbaj de asamblare de nivel intermediar, numit și el INTCODE: un limbaj compact și extrem de simplu, gândit a fi independent de orice mașină reală. INTCODE era deci un strat intermediar și mai simplu, pentru interpretare rapidă pe mașini noi, tocmai în scopul unui bootstrap rapid.

În versiunea primită de noi, pentru a transfera compilatorul BCPL pe o mașină nouă, Richards prevedea să se scrie mai întâi un mic interpretor INTCODE pentru mașina-țintă. Asamblorul și interpretorul de INTCODE erau împreună de aproximativ 4 până la 6 ori mai mici decât un generator de cod BCPL tipic, de la OCODE la limbajul de asamblare. Implementarea unui asamblor și a unui interpretor INTCODE se putea face în 2–3 zile. Scopul INTCODE era să permită dezvoltatorului să obțină o implementare interpretativă temporară pe mașina-țintă într-un timp minim. Cu interpretorul de INTCODE, compilatorul BCPL devenea executabil pe o mașină nouă: cu acest mic dispozitiv software se putea citi banda perforată conținând compilatorul BCPL scris în INTCODE și se putea executa compilatorul pe noua mașină pentru orice program BCPL, chiar înainte de existența unui *backend* pentru generarea de cod nativ (INTCODE → cod de mașină).

Cu alte cuvinte, INTCODE crea o implementare interpretativă temporară a compilatorului BCPL pe mașina-țintă. Ulterior, folosind chiar versiunea BCPL a compilatorului BCPL (*self-hosting*), se putea scrie primul generator nativ de cod și se putea renunța treptat la stratul interpretat.

Această ușurință de implementare se plătea cu încetinirea execuției programelor BCPL. Chiar dacă prin INTCODE se pierdea un factor de aproximativ 10:1 în viteza de execuție, interpretorul de INTCODE era totuși un instrument util pentru implementarea inițială a BCPL pe mașina-țintă, permițând ca apoi primul generator efectiv de cod să fie scris direct în BCPL, folosind versiunea BCPL a compilatorului BCPL.

În consecință, pentru a putea folosi limbajul BCPL la cursurile de programare, a trebuit să implementăm pe calculatorul FELIX un interpretor **INTCODE** capabil să execute codul original. Această fază, aparent modestă, a fost de fapt un moment decisiv: un cod scris într-un limbaj abstract, portabil, putea „încolți și să dea fructe” într-un mediu complet diferit prin simpla implementare a unui interpretor adaptat mediului. După rularea cu succes a interpretării, am obținut un translator BCPL funcțional, ceea ce ne-a permis să dezvoltăm și să rulăm programe în BCPL, liberi de constrângerile limbajelor de asamblare proprietate.

Lucrând cu limbajul BCPL, precursorul elegant și robust al C-ului, am început să schițez un nou vis: un program într-un limbaj care să supraviețuiască schimbării hardware-ului, să călătorească între sisteme și să-și păstreze identitatea în orice context. Două decenii mai târziu, James Gosling avea să imagineze limbajul Java pentru Sun Microsystems, folosind aceleași principii de portabilitate — deși nicio sursă nu menționează explicit influența lui Richards, analogia conceptuală mi se pare evidentă. Cunoștințele

academice ale celor doi au funcționat ca un arc extins în timp, având ca piloni același fundament al viitoarei inginerii software.

După ce am înțeles conceptual ce înseamnă portabilitatea și am prezentat BCPL ca limbaj-vehicul, a venit firesc un studiu comparativ: cum se comportă, în practică, diverse implementări ale aceluiași limbaj pe platforme diferite?

În acest articol am analizat mai multe compilatoare BCPL – unele obținute direct de la Richards, altele adaptate sau dezvoltate local. Ne interesa:

- cât din specificația limbajului era respectată uniform;
- ce diferențe de performanță și de resurse apăreau între implementări;
- în ce măsură arhitectura compilatorului (în special rolul codului intermediar) facilita sau frâna portabilitatea.

Studiul nostru avea o dublă miză. Pe de o parte, era o contribuție tehnică: oferă date și observații concrete despre o familie de compilatoare reale, în condițiile infrastructurii românești. Pe de altă parte, era și un exercițiu metodologic: arăta cum poate fi „măsurată” portabilitatea, nu doar invocată ca ideal. Această atitudine – de a pune în paralel implementări și de a extrage din ele reguli și modele – o voi relua, mai târziu, în contexte legate de procesul software și de interforme.

Concluzie la experiența BCPL

Privind în ansamblu, laboratorul nostru de BCPL devenise, fără să ne propunem neapărat, un mic „atelier al portabilității”. Banda perforată, interpretorul INTCODE pe FELIX, compilatoarele BCPL comparate între ele – toate acestea erau atât exerciții tehnice, cât și pretexte pentru a construi un cadru conceptual despre ce înseamnă, în practică, un program portabil.