

DE LA TURING LA GOOGLE

Calcul și programare

JORDI DELGADO PIN

Traducere de Ionela Polixenia Matei

CUPRINS

Capitolul 0. Turing	7
<i>Un auto-interpret</i>	10
<i>Un simplu exemplu de programare de ordin superior</i>	15
Capitolul 1. Despre notație: limbajul de programare WHILE	19
Datele: mulțimea $\mathbb{D}_A$	20
<i>Câteva exemple</i>	21
Limbajul WHILE	24
Programând, învățăm să programăm	34
Și programele sunt elemente ale lui $\mathbb{D}_A$	39
<i>Norme de traducere</i>	42
Auto-interpretul u	45
Simplificăm... din nou?	57
Asta vrem să povestim despre WHILE	63
Capitolul 2. Se poate calcula orice?	65
Funcții WHILE-calculabile și mulțimi WHILE-decidabile	65
<i>Demonstrație</i>	68
În WHILE-numerabilitatea nu este diferită de WHILE-semidecidabilitate	71
HALT (and catch fire)	79
<i>Metoda diagonală</i>	80
Indecidabili, indecidabili everywhere	86
const în limbajul de calcul	94
{autor , capitolul-2} ∈ HALT	97

Capitolul 3. Oglindă, oglindă	99
Teorema s-m-n	100
Proiecțiile lui Futamura	106
Teoremă de recursivitate	111
<i>Quines</i>	114
Cealaltă (a doua) teoremă de recursivitate	119
Teza lui Church-Turing	121
<i>Un model universal?</i>	124
<i>All good things...</i>	133
Lecturi recomandate	137

Capitolul 0

Turing

În zilele noastre, nimeni nu mai este surprins să audă că britanicul **Alan M. Turing** (1912-1954) este unul dintre cele mai mari genii ale tuturor timpurilor. Contribuțiile sale au fost excepționale în logică, matematică, inginerie, biologie... A fost și un pionier în informatica teoretică și în inteligența artificială, deși „pionier” poate fi considerat un eufemism, de fapt el practic le-a inventat. Se crede că, prin colaborarea sa ca expert în criptografie în timpul celui de-al Doilea Război Mondial, a scurtat conflictul cu doi ani și a salvat viețile a 14 milioane de oameni.



Figura 1. Alan M. Turing.

Viața sa a făcut obiectul a numeroase biografii, piese de teatru, filme și cântece și chiar apariția ca personaj într-unul din romanele *Doctor Who* scrise de Paul Leonard. Dezbaterile asupra evenimentelor din jurul morții sale este încă în desfășurare, dar în 2013 i s-a acordat „iertarea regală” (era și timpul, prea târziu), așadar, în 2017, a fost promulgată o lege care acorda achitarea tuturor persoanelor condamnate în Anglia și în Țara Galilor ca fiind homosexuali, cunoscută

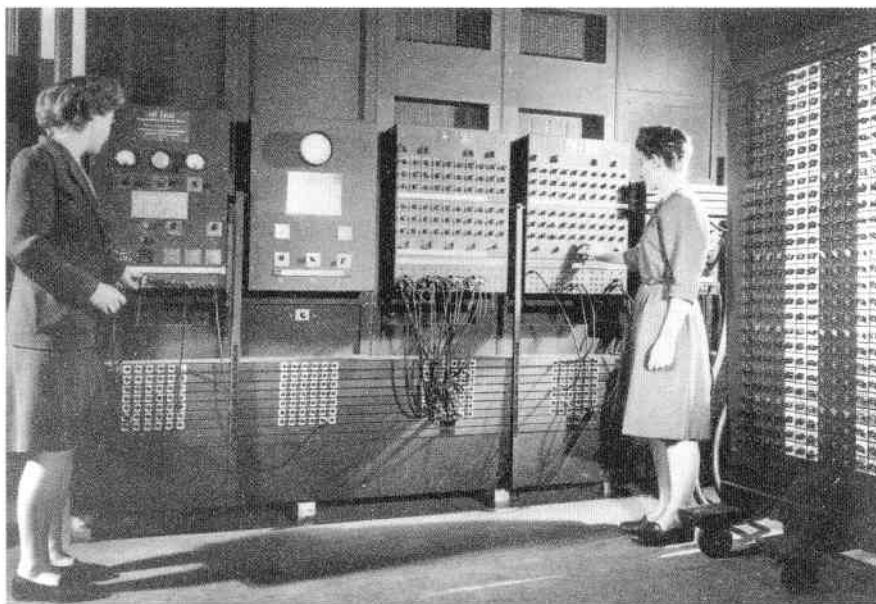


Figura 2. *Computer room* în cadrul NACA (astăzi NASA) în 1949.

și sub numele de legea lui Alan Turing. De remarcat, de asemenea, că în prezent, cel mai important premiu rezervat informaticienilor este Premiul Turing, un fel de Premiu Nobel pentru această disciplină, acordat de Asociația for Computer Machinery (ACM) din Statele Unite.

Nu este intenția noastră să scriem o biografie a lui Alan Turing, alții au făcut-o deja, dar vrem totuși să justificăm titlul acestei cărți spunând o anecdotă despre el, poate puțin cunoscută. Detaliile tehnice ale acestei anecdote, și altele, constituie tocmai conținutul restului cărții, așa că acum nu ne vom opri asupra clarificărilor formale. Cu puțină răbdare, totul va merge la locul lui.

În 1936, Turing a scris un articol de importanță fundamentală pentru logica matematică: „On Computable Numbers, with an Application to the Entscheidungsproblem”, publicat în 1937, în care își descrie

mașina lui Turing (MT), un fel de computer *teoretic*, schițat, rezultatul reflecției sale asupra sensului de „a calcula” (rețineți că la acea vreme un calculator era o persoană care se ocupa cu calculul; vezi figura 2). Definită ca un MT foarte special, mașina universală Turing (MTU) primește ca intrări (codificarea) unei alte MT, să-i spunem M, împreună cu datele sale de intrare, să le numim D, și trebuie să simuleze execuția lui M pe datele D. Cu alte cuvinte, o mașină Turing face parte din datele pe care o altă mașină Turing le primește pentru a funcționa. Este vorba despre o intuiție cu adevărat genială pentru acea vreme. În zilele noastre, oricine se pricepe la programare va ști să recunoască acest proces și ar putea spune că o MTU este un *interpret* al MT. Și, întrucât o MTU este și o mașină Turing, vom spune că este un „auto-interpret”. Scrierea unui auto-interpret este practic o tradiție în textele referitoare la familia de limbaje de programare legate de Lisp, încă de la primul *Lisp 1.5 Programmer's Manual*, datat 1962, deși nu este neobișnuit să găsești auto-interpreți în texte despre alte limbaje de programare precum Smalltalk (în *Smalltalk-80: The Language and its Implementation*, 1983) sau Prolog (în *The Art of Prolog*, din 1986).

Aproape concomitent cu articolul lui Turing, în 1937, inginerul german **Konrad Zuse** (1910-1995) a prezentat la rândul său două brevete care cuprindeau și propunerea de a stoca atât programul cât și datele în aceeași memorie. (Apropo, brevetele au fost respinse). Această idee este cunoscută sub numele de „calculator cu program stocat” (stored-program computer), a cărui configurație facilitează implementarea și realizarea intuiției geniale a lui Turing, respectiv, un program se încadrează în datele furnizate unui alt program.

Cele două relații decisive din istoria noastră au apărut abia la sfârșitul celui de-al Doilea Război Mondial. Pe de o parte, la sfârșitul anului 1945, Turing a scris „Proposed Electronic Calculator”, care conține proiectul său pentru ACE, prima specificație relativ completă a unui computer electronic cu program stocat. Deși ar fi oferit mai puține detalii, raportul „First Draft of a Report on the EDVAC”, scris de **John von Neumann**, (1903-1957), publicat de **Herman Goldstine**

UN AUTO-INTERPRET

Un *auto-interpret* nu trebuie să fie neapărat ceva foarte complicat. Totul depinde de până la ce punct limbajul interpretat utilizează (reciclează) implicit mecanismele deja prezente în limbajul în care interpretul este scris (care este același, de aici „auto”, de la auto-interpret). Uneori, limbajul interpretat este o submulțime restrânsă a limbajului în cauză. Un exemplu: scrieți în limbajul de programare Prolog un interpret pentru submulțimea Prolog cunoscută drept Prolog *pure* și respectiv simplă (evident odată ce o stăpânești). În capitolul 17 din *The Art of Prolog* găsim acest mic program:

```
solve(true).  
solve((A,B)) :- solve(A),solve(B).  
solve(A) :- clause(A,B),solve(B).
```

(1913-2004) și bazat pe idei originale ale echipei de ingineri responsabili din cadrul ENIAC, în special J. Presper Eckert și John Mauchly, de la Școala Moore de la Universitatea din Pennsylvania, a avut un impact mai mare. Datorită raportului menționat mai sus, nu fără controverse, arhitectura calculatoarelor cu program stocat este cunoscută sub denumirea de „arhitectura von Neumann” (vezi figura 3 de la p 12).).

Care a fost primul computer autentic bazat pe arhitectura von Neumann nu este atât de clar. Unii spun că este SSEM (Small Scale Experimental Machine, mai bine cunoscut sub numele de „Manchester Baby”), construit la Universitatea Victoria din Manchester de **Frederic C. Williams** (1911-1977), **Tom Kilburn** (1921-2001) și **Geoff Tootill** (1922-2017) în 1948. Alții susțin că Manchester Baby nu a fost altceva decât o dovadă a conceptului și că primul computer

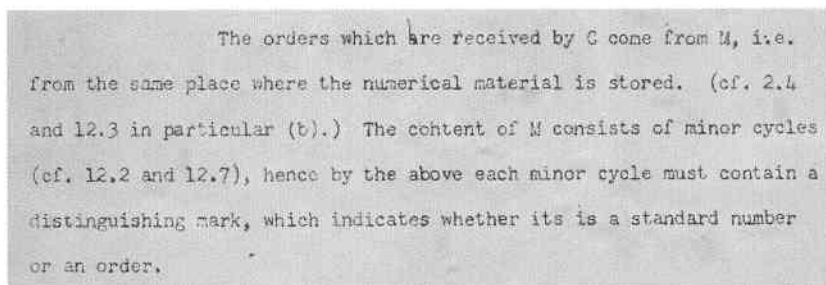


Figura 3. Imaginea textului „First Draft of a Report on the EDVAC” în care von Neumann sugerează ce program și ce date ar trebui să împartă memoria (secțiunea 14.1).

Înzestrat cu această arhitectură a fost fratele său mai mare, Manchester Mark 1, construit de aceeași echipă căreia i s-au alăturat doi doctoranzi, D.B.G. Edwards și G.E. Thomas. Primul program rulat în Mark 1, în iunie 1949, a fost folosit pentru o căutare a numerelor prime Mersenne în care computerul a funcționat fără probleme timp de nouă ore consecutive. EDSAC (Electronic Delay Storage Automatic Calculator), construit de Maurice Wilkes și echipa sa de la Mathematical Laboratory de la Universitatea din Cambridge, este un alt candidat pentru a fi primul computer adevărat cu program stocat. În acest sens, propunerile lui Turing și ale lui von Neumann nu ar fi văzut lumina imediat: prima în 1950 (era vorba despre Pilot ACE, o versiune redusă a ceea ce specifica Turing; vizibil în figura 4 de la p. 12) și a doua în 1952.

Dar nu vom intra în întrebarea cine a fost primul care a propus, a proiectat sau a construit un computer cu arhitectura von Neumann, pentru a ne concentra în schimb asupra modului în care sunt proiectate computerele utilizate în mod obișnuit: o unitate centrală de procesare care gestionează instrucțiunile, se execută succesiv după achiziționarea lor din memorie conform unui contor de program, și solicită datele aceleiași memorii în funcție de nevoile instrucțiunii în execuție. Desigur, toate acestea sunt fantezie, o simplificare având în

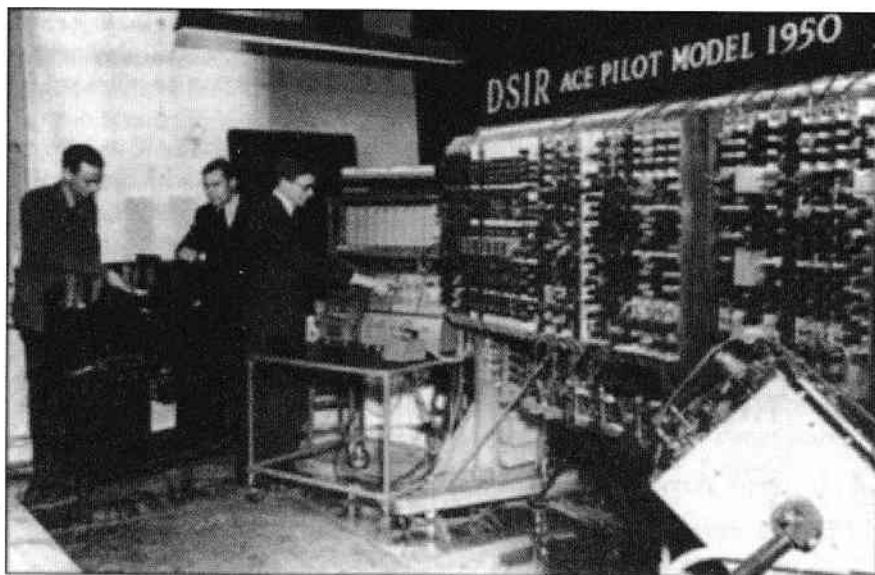


Figura 4. Pilotul ACE, bazat pe proiectul original al lui Turing.

vedere că, calculatoarele *multi-core* moderne dotate cu diferite memorii *cache* și alte accesorii utile pentru a câștiga viteză contemplă această arhitectură de departe, ca să spunem așa. Cu toate acestea, continuă să păstreze programele și datele în aceeași memorie, adică sunt întotdeauna computere cu program stocat. În acest moment este necesar să facem o precizare: microcontrolerele sunt de obicei proiectate cu arhitectură Harvard în care programul și datele sunt în memorii separate, iar acestea reprezintă inima a milioane de dispozitive automate care ne însoțesc și cu care interacționăm zilnic, așadar, nu se poate neglija importanța altor arhitecturi, altele decât cele care ne interesează.

Dar de ce este atât de important ca programul și datele să fie stocate în aceeași memorie? A fi în memorii separate ne împiedică să efectuăm o operațiune? Răspunsul la această ultimă întrebare este negativ. **Marvin L. Minsky** (1927-2016), în cartea sa extraordinară

din 1967, *Computation: Finite and Infinite Machines*, a afirmat următoarele:

Există un consens larg cu privire la faptul că progresele majore ale calculatoarelor moderne au fost realizate în urma acceptării ideii că programele ar putea fi stocate în aceeași memorie cu datele și că astfel de programe ar putea acționa asupra altor programe sau asupra lor însele, deși ca date. Poate că nu este pe deplin înțeles că același potențial teoretic (deși nu practic) poate fi obținut în mașini ale căror programe de nivel înalt nu pot fi modificate.

Cu alte cuvinte, nu există nimic care să se poată face cu un computer dotat cu un program stocat care să nu se poată face cu un computer fără program stocat. Și viceversa. Deci, întrebarea nu este *ce* putem realiza, ci mai degrabă *cum* se poate realiza. Există tehnici asociate programului stocat în aceeași memorie cu datele pe care nu le putem folosi dacă acest lucru nu se întâmplă, dar există tehnici alternative pentru a obține același efect. Acestea au de-a face cu *reflecția*, adică tehnicile care pot fi aplicate dacă un program are acces la propriul mediu de execuție, în special la propriul său cod, în timp ce rulează, ceea ce le include și pe cele care se modifică singure în timpul rulării. Și nu se poate nega puterea de evocare a faptului că se pot modifica singure, deși realitatea este mult mai banală, în condițiile în care, din punctul de vedere al programatorului, este foarte greu să întreținem și să purificăm aceste programe. Chiar și așa, în zilele noastre, codul cu auto-modificare continuă să fie folosit, de exemplu, în compilarea *Just-in-time*, mascarea virușilor de computer, ofuscarea codului și așa mai departe.

Dar povestea, dacă se poate, este și mai curioasă. Posibilitatea ca un program să primească un alt program ca parte a datelor sale a existat la nivel teoretic (în MT, dar și λ -calcul include această posibilitate și nu au fost singurele formalisme având această proprietate), precum și la nivel de mașină, deoarece arhitectura von Neumann a

inclus-o deja în proiectul său. Cu toate acestea, atunci când a apărut nevoia de a crea limbaje de programare de nivel înalt, mai aproape de programator decât de *hardware*, această funcționalitate a fost ignorată în unele cazuri importante. Fortran (1958) și Cobol (1959), limbaje folosite și astăzi în versiunile moderne corespunzătoare, în principiu nu permiteau unei funcții sau proceduri să primească o altă funcție sau procedură ca parametru și nici să returneze o funcție sau procedură ca rezultat. Programarea cu utilizarea acestei caracteristici ia denumirea de „programare de ordin superior” și implementarea acestei caracteristici nu este deloc ușoară. Între deceniile 1960 și 1970 au avut loc multe discuții pe tema așa-numitei probleme *funarg problem*. Limbaje precum C++ sau Java nu au permis acest tip de programare până relativ recent (respectiv C++ 11 și Java 8), și într-un mod nesatisfăcător. Cu toate acestea, alte limbaje de nivel înalt, mai apropiate de mediul academic, au inclus încă de la început programarea de ordin superior. Cel mai vechi exemplu este Lisp (1959), dar și Small-talk-76 (1976) avea și această caracteristică disponibilă.

Noțiunea de *algorithm* (și un *program de calculator* este chiar asta) nu a luat naștere cu Turing. **Euclid** (cca. 325 î.Hr.-cca. 265 î.Hr.) a propus algoritmi în urmă cu 2300 de ani pentru a calcula cel mai mare divizor comun al două numere întregi, iar matematicienii britanici **Charles Babbage** (1791-1871) și **Ada Lovelace** (1815-1852) în secolul al XIX-lea erau deja capabili să conceapă programe pentru mașina lor analitică, rămasă neterminată. În 1928, matematicianul german **David Hilbert** (1862-1943) a emis ipoteza existenței unui *algorithm* care să decidă dacă o propoziție dată de logica de ordinul întâi este demonstrabilă pornind de la axiome și folosind regulile logicii. În 1935-1936, matematicianul și logicianul american **Alonzo Church** (1903-1995) și la scurt timp, în 1936, însuși Turing, au dat în mod independent un răspuns negativ problemei presupuse de Hilbert. Una dintre consecințele interesante este că reflectarea asupra *Entscheidungsproblem* și încercarea de a da un răspuns i-a forțat pe Church și pe Turing să oficializeze noțiunea de *algorithm* și au luat naștere

UN SIMPLU EXEMPLU DE PROGRAMARE DE ORDIN SUPERIOR

Presupunem că dorim un program care, dată fiind o funcție oarecare f (astfel încât imaginea sa este inclusă în domeniul său), restituie o funcție care aplică de două ori f . Această funcție o vom numi *twice*. Așadar, dacă denumim g rezultatul de $twice(f)$, vom avea $g(x) = f(f(x))$ pentru oricare x din domeniul lui f .

Vedem *twice* în Javascript:

```
function twice (f) {
  return x => f(f(x));
}
```

În Python:

```
def twice(f):
  return lambda x: f(f(x))
```

sau în Common Lisp:

```
(defun twice (f)
  #'(lambda (x) (funcall f (funcall f x))))
```

λ -calcul și, respectiv, MT. Făcând acest lucru, au întemeiat o teorie despre cât de mult sau puțin poate fi calculat. Această carte servește ca o introducere la această ipoteză, *teoria computației*.

Vom încerca să o explicăm în raport cu intuițiile moderne în programare și, pentru a reuși, ideea-cheie originală a lui Turing, von Neumann și alții este fundamentală, și anume că un program poate consta din date pentru alt program. Decizia de a folosi concepte moderne de programare pentru a explica cele mai importante rezultate ale teoriilor menționate este justificată deoarece la aproape o sută de ani după lucrările lui Turing, dar și ale lui **Kurt Gödel** (1906-1978), Alonzo