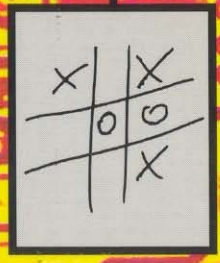
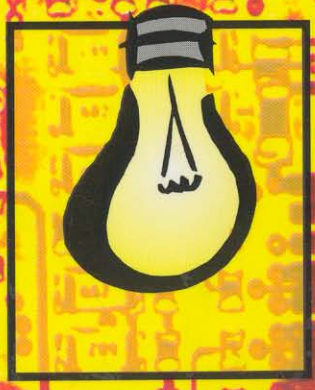
W. Daniel Hillis

WZORY

NA KRZEMOWEJ PŁYTCE

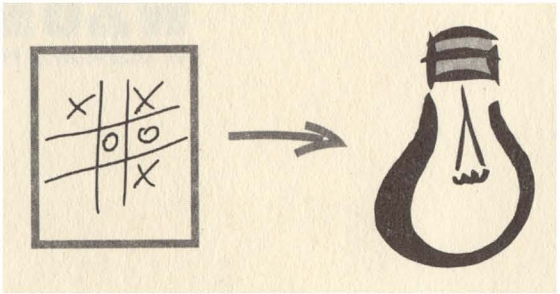




science masters

SCIENCE MASTERS

WZORY
NA KRZEMOWEJ PŁYTCE



W. Daniel Hillis

WZORY
NA KRZEMOWEJ PŁYTCE

Tłumaczył Piotr Rączka

Wydawnictwo CiS
Warszawa 2000

Tytuł oryginału: *The Pattern on the Stone. The Simple Ideas that Make Computers Work*

Przekład: Piotr Rączka

Projekt graficzny serii i okładki: Hanka Milanowicz
Redakcja: A. i J. Sanin
Skład: AG

Piętnasty tom serii SCIENCE MASTERS

Wydawnictwo CiS
Warszawa 2000
Wydanie pierwsze

ISBN: 83–85458–71–9

Druk: Drukarnia Wydawnictw Naukowych SA, Łódź

SPIS TREŚCI

Przedmowa

MAGICZNY KAMIEŃ

Rysuję na kamieniu geometryczne wzory. Dla niewtajemniczonych są one skomplikowane i wyglądają tajemniczo, ale ja wiem, że jeśli rozmieszczę je prawidłowo, obdarzają kamień szczególną mocą, pozwalającą mu reagować na zaklęcia, wypowiadane w języku, którym nie mówił nigdy żaden człowiek. W tym języku będę zadawał pytania, a kamień odpowie mi, roztaczając przede mną wizję świata powstałego w wyniku moich czarów, świata wykreowanego z rysunku na kamieniu.

Kilkaset lat temu, w mej rodzinnej Nowej Anglii, taki opis mojej pracy zaprowadziłby mnie niechybnie na stos. A jednak to, co robię, nie ma nic wspólnego z czarami — projektuję i programuję komputery. Kamieniem jest krzemowa płytka, a zaklęcia to oprogramowanie. Wzory nanoszone na płytkę i programy sterujące komputerem mogą wydawać się skomplikowane i tajemnicze, ale tworzone są w oparciu o kilka podstawowych zasad, które łatwo wyjaśnić.

Komputery to najbardziej złożone urządzenia, jakie my, ludzie, do tej pory stworzyliśmy, ale w istocie

są one niezwykle proste. Pracując z zespołami liczącymi jedynie kilkadziesiąt osób zaprojektowałem i zbudowałem komputery, składające się z miliardów funkcjonujących elementów. Gdyby kiedykolwiek narysowano schemat połączeń dla jednej z tych maszyn, wypełniłby on wszystkie książki w sporej bibliotece publicznej i nikomu nie starczyłoby cierpliwości, aby go w całości przejrzeć. Na szczęście, ze względu na regularności występujące w konstrukcji komputerów, takie schematy nie są potrzebne. Komputery są zbudowane w sposób hierarchiczny, przy czym każdy element powtarza się wiele razy. Aby je zrozumieć, wystarczy zrozumieć tę hierarchię.

Kolejna zasada, która sprawia, że zrozumienie komputera jest tak łatwe, to sposób, w jaki oddziałują ze sobą jego elementy. Oddziaływania te są proste i dobrze określone. Są też zwykle jednokierunkowe i dlatego w pracy komputera łatwo oddzielić przyczyny i skutki. Dzięki temu łatwiej opisać funkcjonowanie komputera, niż na przykład zasady pracy silnika samochodowego czy radia, od których ma on o wiele więcej części. Zawdzięczamy to temu, iż sposób w jaki jego elementy współpracują ze sobą jest znacznie prostszy, a ich praca w większym stopniu opiera się na koncepcjach niż na technice. Co więcej, koncepcje te nie mają prawie nic wspólnego z elektroniką, używaną przy budowie komputerów. Współczesne komputery zbudowane są z tranzystorów i przewodów, ale równie dobrze mogłyby być — według tych samych zasad — zbudowane z zaworów i rur z wodą lub drążków i linek. Istotą tego, co sprawia, że komputery przeprowadzają obliczenia, są koncepcje, których

ważną własnością jest to, że wykraczają poza technikę — i im właśnie poświęcona jest ta książka.

Taką książkę powinienem przeczytać, gdy po raz pierwszy zacząłem interesować się informatyką. W odróżnieniu od większości książek o komputerach — mówiących o tym, jak z nich korzystać, lub o technicznych szczegółach ich budowy (ROM, RAM, napędy dysków itd.) — ta książka dotyczy **koncepcji.** Wyjaśniam w niej, lub przynajmniej opisuję, większość koncepcji istotnych w informatyce, włącznie z algebrą Boole'a, maszynami o skończonej liczbie stanów, językami programowania, kompilatorami i interpreterami, uniwersalnością w sensie Turinga, teorią informacji, algorytmami i złożonością algorytmów, heurystykami, funkcjami nieobliczalnymi, przetwarzaniem równoległym, komputerami kwantowymi, sieciami neuronowymi, uczeniem się przez maszyny i systemami samoorganizującymi się. Osoba zainteresowana komputerami na tyle, aby przeczytać tę książkę, z pewnością już wcześniej zetknęła się z wieloma spośród tych koncepcji, ale poza formalnym kształceniem informatycznym niewiele jest możliwości, aby zobaczyć, w jaki sposób się one zazębiają. Ja zaś staram się opisać właśnie ich wzajemne powiązania — począwszy od prostych procesów fizycznych, takich jak zmiana ustawienia wyłącznika, aż po uczenie się i zdolności adaptacyjne, wykazywane przez samoorganizujące się komputery o architekturze równoległej.

Dobra prezentacja istoty komputerów musi zawierać kilka wątków ogólnych: pierwszym jest zasada **funkcjonalnej abstrakcji**, która prowadzi do wspomnianej wyżej hierarchii przyczyn i skutków.

Struktura komputera jest przykładem zastosowania tej zasady — wiele razy, na wielu poziomach. Komputery można zrozumieć, ponieważ można się skoncentrować na tym, co dzieje się na jednym poziomie hierarchii, bez martwienia się o szczegóły i o to, co dzieje się na niższych poziomach. Dzięki funkcjonalnej abstrakcji koncepcje stają się niezależne od technologii.

Drugim wątkiem zespalającym jest idea **komputera uniwersalnego** — zgodnie z nią tak naprawdę istnieje tylko jeden typ komputera, lub bardziej precyzyjnie, wszystkie rodzaje komputerów są do siebie podobne pod względem tego, co mogą i czego nie mogą zrobić. Na ile nam wiadomo, każda maszyna licząca, niezależnie od tego, czy zbudowana jest z tranzystorów, drążków i linek czy neuronów, może być symulowana przez komputer uniwersalny. Jest to niezwykła hipoteza: jak wyjaśnię, można z niej wnioskować, że stworzenie komputera, który potrafiłby myśleć jak mózg, jest tylko kwestią właściwego oprogramowania.

Trzeci wątek w tej książce, który nie zostanie w pełni rozwinięty aż do ostatniego rozdziału, jest w pewnym sensie antytezą pierwszego. Mogą istnieć całkowicie nowe metody projektowania i programowania komputerów — sposoby, które nie wykorzystują standardowych metod inżynierskich. Byłoby to bardzo podniecające, ponieważ sposób, w jaki zwykle projektujemy układy, zaczyna się załamywać w momencie, kiedy stają się one zbyt skomplikowane. Zasady, na których opiera się projektowanie komputerów, w nieunikniony sposób prowadzą do pewnego braku odporności i ograniczeń w wydajności. Ta sła-

bość nie ma nic wspólnego z jakimikolwiek fundamentalnymi ograniczeniami dla maszyn przetwarzających informację — jest to ograniczenie samej metody projektowania hierarchicznego. Co by się jednak stało, gdybyśmy zamiast tego posłużyli się procesem projektowania analogicznym do ewolucji biologicznej — a więc procesem, w którym zachowania układu **wyłaniają się** z nagromadzenia wielu prostych oddziaływań, bez żadnej „odgórnej" kontroli? Maszyna licząca zaprojektowana przez taki proces ewolucyjny mogłaby w pewnym stopniu wykazywać odporność i elastyczność organizmu biologicznego — a przynajmniej takie są nadzieje. Podejście to nie jest jeszcze dobrze zrozumiane i może okazać się niepraktyczne, ale ono właśnie jest przedmiotem moich obecnych badań.

Nasze rozważania o istocie komputerów zaczniemy od podstaw, z którymi trzeba się zapoznać, nim można będzie przejść do rzeczy bardziej interesujących. Pierwsze dwa rozdziały dotyczą algebry Boole'a, bitów i maszyn o skończonej liczbie stanów. Korzyść z nich jest taka, że dotarłszy do końca rozdziału 3. będziecie — od początku do końca — rozumieli, jak działają komputery. Przygotuje to grunt dla podniecających koncepcji dotyczących uniwersalnych maszyn liczących, które znajdziecie w rozdziale 4.

Filozof Gregory Bateson zdefiniował kiedyś informację jako „różnicę, która robi różnicę". Inaczej mówiąc, informacja polega na rozróżnieniach, które uznajemy za istotne. Na przykład, w prymitywnym arytmometrze elektrycznym informacja oznaczana jest za pomocą żaróweczek, które świecą się lub nie,

w zależności od tego, czy płynie przez nie prąd. Napięcie sygnału nie ma znaczenia, nieistotny jest też kierunek przepływu prądu. Znaczenie ma tylko to, że przewód przenosi któryś z dwóch możliwych sygnałów, a jeden z nich powoduje zapalenie się żarówki. Rozróżnieniem, które decydujemy się uznać za istotne — czyli różnicą, która robi różnicę, by użyć zwrotu Batesona — jest to, czy prąd płynie, czy nie. Definicja Batesona jest bardzo dobra, ale zawsze oznaczać będzie dla mnie jeszcze coś więcej. Otóż, w ciągu czterdziestu lat mojego życia świat uległ transformacji. Większość zmian w biznesie, polityce, nauce i filozofii, których byliśmy w tym czasie świadkami, wynikała z postępów w technice informatycznej. Wiele rzeczy jest w dzisiejszym świecie innych, ale tą różnicą, która sprawiła istotną różnicę, były komputery.

Dziś komputery uważa się powszechnie za urządzenia multimedialne, zawierające w sobie i integrujące wszystkie wcześniejsze postaci mediów — tekst, grafikę, ruchomy obraz i dźwięk. Myślę, że ten punkt widzenia przyczynia się do niedoceniania ich możliwości. Z pewnością prawdą jest, że komputer może zawierać i przekształcać wszystkie inne media, ale prawdziwa jego siła polega na tym, że może on manipulować nie tylko **sformułowaniami** koncepcji, ale również samymi koncepcjami. Zdumiewającą dla mnie rzeczą jest nie to, że w komputerze można zapisać zawartość wszystkich książek w bibliotece, ale że dostrzega on związki między pojęciami przedstawionymi w książkach; nie to, że potrafi wyświetlić obraz ptaka w locie lub wirującej galaktyki, ale że potrafi zobaczyć i przewidzieć konsekwencje praw fizycz-

nych, które stworzyły te cuda. Komputer nie jest jedynie zaawansowanym kalkulatorem, kamerą czy pędzlem; jest raczej urządzeniem, które przyspiesza i poszerza nasze procesy myślowe. Jest wehikułem wyobraźni, maszyną, która wychodzi od koncepcji, jakie w niej zawarliśmy i rozwija je znacznie dalej, niż to kiedykolwiek potrafilibyśmy zrobić sami.

Rozdział pierwszy

ŚRUBY I MUTRY, CZYLI ISTOTA RZECZY

Kiedy byłem mały, przeczytałem opowiadanie o chłopcu, który z części znalezionych na złomowisku zbudował robota. Robot potrafił się poruszać, mówić i myśleć — zupełnie jak istota ludzka — i został jego przyjacielem. Z jakiegoś powodu pomysł budowania robota wydał mi się bardzo atrakcyjny, zdecydowałem więc, że ja też coś takiego skonstruuję. Pamiętam, jak zbierałem części kadłuba — rury na ręce i nogi, silniki na mięśnie, żarówki na oczy i dużą puszkę po farbie na głowę — bez zastrzeżeń żywiąc optymistyczne przekonanie, że gdy części te połączone zostaną w jedną całość i podłączę ją do prądu, to otrzymam w efekcie działającego mechanicznego człowieka.

Poraziwszy się kilkakrotnie prądem, co omal nie skończyło się moją śmiercią, osiągnąłem w końcu tyle, że zgromadzone przeze mnie elementy zaczęły się poruszać, świecić i wydawać dźwięki. Miałem poczucie, że robię postępy. Zaczynałem rozumieć, jak należy konstruować ruchome przeguby rąk i nóg. Ale zacząłem też uświadamiać sobie coś jeszcze waż-

niejszego: że nie mam najmniejszego pojęcia, w jaki sposób sterować silnikami i światłami. Zorientowałem się, że w mojej wiedzy o tym, jak działają roboty, jest luka. Teraz potrafię nazwać to coś, czego mi brakowało: nazywa się to **obliczaniem.** Wtedy nazywałem to „myśleniem" i zdawałem sobie sprawę, że nie mam żadnego pomysłu, jak spowodować, aby jakieś urządzenie myślało. Teraz wydaje mi się oczywiste, że zdolność do przeprowadzania obliczeń jest najtrudniejszym elementem przy konstruowaniu mechanicznego człowieka, ale, gdy byłem dzieckiem, była to dla mnie niespodzianka.

LOGIKA BOOLE'A

Szczęśliwie złożyło się tak, że pierwszą przeczytaną przeze mnie książką, dotyczącą przeprowadzania obliczeń, było dzieło należące do klasyki. Mój ojciec był epidemiologiem i mieszkaliśmy wówczas w Kalkucie. Książki w języku angielskim były trudno dostępne, ale w bibliotece w konsulacie brytyjskim znalazłem zakurzony egzemplarz pracy napisanej przez dziewiętnastowiecznego logika George'a Boole'a. Zaintrygował mnie jej tytuł: *An Investigation of the Laws of Thought* („Badania nad prawami rządzącymi myśleniem"). To pobudziło moją wyobraźnię. Czy to możliwe, że myślenie rzeczywiście podlega jakimś prawom? W książce tej Boole próbował zredukować logikę ludzkiego myślenia do operacji matematycznych. Chociaż nie wyjaśnił tak naprawdę fenomenu inteligencji człowieka, Boole zademonstrował zadziwiającą potęgę i ogólność kilku prostych typów operacji logicznych. Wynalazł też język, za pomocą któ-

rego można opisywać stwierdzenia logiczne i operować nimi, a także orzekać o ich prawdziwości. Język ten nazywa się obecnie **algebrą Boole'a.**

Algebra Boole'a podobna jest do algebry, której naucza się w szkole średniej, tyle że zmienne w równaniach odpowiadają zdaniom logicznym, a nie liczbom. Zmienne Boole'a oznaczają zdania, które są prawdziwe lub fałszywe, a symbole $\wedge$, $\vee$ i $\neg$ reprezentują operacje logiczne AND (i), OR (lub) i NOT (nie). Na przykład, następujący wzór jest równaniem w algebrze Boole'a

$$\neg(A \vee B) = (\neg A) \vee (\neg B).$$

To akurat równanie, nazywane twierdzeniem De Morgana (od nazwiska Augustusa De Morgana, współpracownika Boole'a), mówi że jeśli nieprawdą jest, iż A jest prawdziwe lub B jest prawdziwe, to zarówno A jak i B muszą być fałszywe. Zmienne A i B mogą oznaczać jakiekolwiek stwierdzenia logiczne (czyli takie, o których można powiedzieć, że są prawdziwe lub fałszywe). To równanie jest w oczywisty sposób prawdziwe, ale algebra Boole'a pozwala na wypisywanie znacznie bardziej złożonych twierdzeń logicznych i orzekanie o ich prawdziwości.

Koncepcje Boole'a trafiły do informatyki dzięki pracy magisterskiej studenta inżynierii w Massachusetts Institute of Technology, Claude'a Shannona. Shannon jest najbardziej znany jako twórca działu matematyki zwanego **teorią informacji**, w którym zdefiniowana została miara informacji — **bit**. Wynalezienie bitu było imponującym osiągnięciem, ale to, co Shannon zrobił z logiką Boole'a, miało dla nauki o przeprowadzaniu obliczeń co najmniej taką samą wagę. Tymi dwoma dokonaniami Shannon

stworzył podstawy postępów, jakie miały się dokonać w dziedzinie przeprowadzania obliczeń w ciągu następnych pięćdziesięciu lat.

Shannon interesował się możliwością zbudowania maszyny, która potrafiłaby grać w szachy, a bardziej ogólnie, budowaniem urządzeń, które mogłyby imitować myślenie. W roku 1940 opublikował swoją pracę magisterską, zatytułowaną *A Symbolic Analysis of Relay Switching Circuits* („Analiza symboliczna przekaźnikowych obwodów przełączających"). W pracy tej pokazał, że możliwe jest zbudowanie układów elektrycznych odpowiadających wyrażeniom w algebrze Boole'a. W obwodach Shannona zamknięte lub otwarte wyłączniki odpowiadały różnym wartościom zmiennych logicznych algebry Boole'a. Shannon pokazał, w jaki sposób dowolne wyrażenie w algebrze Boole'a można przedstawić w postaci układu wyłączników. Jeśli stwierdzenie jest prawdziwe, to przez obwód płynąć będzie prąd, a jeśli jest fałszywe, to połączenie w obwodzie będzie przerwane. W konsekwencji, dowolna funkcja, którą potrafimy przedstawić jako precyzyjne wyrażenie logiczne, może zostać zrealizowana w postaci odpowiedniego układu wyłączników.

Zamiast przedstawiać w szczegółach formalizm rozwinięty przez Boole'a i Shannona, podam kilka przykładów jego zastosowania w projekcie bardzo prostego urządzenia liczącego, mianowicie maszyny do gry w kółko i krzyżyk. Urządzenie to jest znacznie prostsze od uniwersalnego komputera, ale ilustruje dwie zasady ważne dla komputerów dowolnego typu. Pokazuje, jak dowolne zadanie może zostać zredukowane do **funkcji logicznych** i w jaki sposób mogą

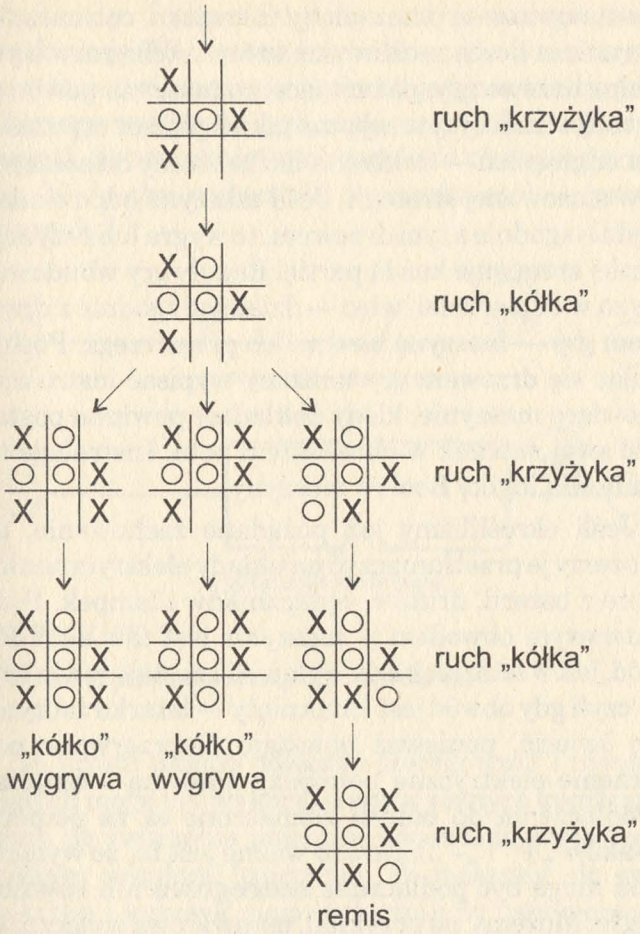
one zostać zrealizowane w postaci obwodu złożonego z połączonych wyłączników. Ja sam zbudowałem maszynę do gry w kółko i krzyżyk z lampek i wyłączników wkrótce po tym, jak przeczytałem książkę Boole'a w Kalkucie — i było to moje wprowadzenie do logiki komputerowej. Później, kiedy byłem studentem MIT, Claude Shannon stał się moim przyjacielem i nauczycielem — odkryłem wtedy, że on również zbudował z lampek i wyłączników maszynę, która mogłaby grać w kółko i krzyżyk.

Jak większości czytelników wiadomo, gra ta rozgrywana jest na kwadratowej planszy z siatką 3 na 3 pola. Gracze kolejno zaznaczają pola, jeden używając X, a drugi O. Ten, który pierwszy umieści trzy znaki w jednym rzędzie (pionowo, poziomo lub na ukos) wygrywa partię. Małe dzieci lubią grać w kółko i krzyżyk, ponieważ liczba strategii prowadzących do wygranej w tej grze wydaje się być nieskończona. W końcu jednak nawet one dostrzegają, że może się pojawić tylko bardzo niewiele kombinacji i wtedy gra traci swój urok; kiedy obaj gracze je poznają, każda partia nieodmiennie kończy się remisem. Gra w kółko i krzyżyk to dobry przykład obliczeń komputerowych właśnie z tego powodu, że znajduje się na granicy pomiędzy tym, co proste, a tym, co złożone, podczas gdy istotą obliczeń jest właśnie przekraczanie tej granicy. Komputerowe obliczenia polegają na rozwiązywaniu zadań, które wydają się być złożone (na przykład wygrana w kółko i krzyżyk), poprzez rozbicie ich na proste operacje (zamknięcie wyłącznika).

W grze w kółko i krzyżyk liczba mogących się pojawić sytuacji jest na tyle mała, że dogodnie jest wszystkie je wypisać i wbudować w maszynę prawid-

łową odpowiedź dla każdego przypadku. Przy projektowaniu maszyny możemy zastosować prostą, dwustopniową procedurę: po pierwsze, zredukować grę do zestawu przypadków, określając prawidłową odpowiedź dla każdej sekwencji ruchów; po drugie, przekształcić te przypadki w obwody elektryczne przez takie połączenie wyłączników, aby rozpoznawały one sekwencję ruchów i dawały właściwą odpowiedź.

Jednym ze sposobów postępowania mogłoby być wypisanie wszystkich możliwych do pomyślenia układów krzyżyków i kółek w kratkach, a potem zdecydowanie, jak komputer powinien zagrać w każdym przypadku. Ponieważ każdy z dziewięciu kwadratów może znajdować się w jednym z trzech stanów (X, O i pusty), to istnieje 3^9 (czyli 19 683) sposobów zapełnienia kratek. Ale większość z tych układów nigdy nie pojawi się w trakcie gry. Lepszą metodą skatalogowania możliwych układów jest narysowanie **drzewa gry** — struktury, w której zapisane są wszystkie możliwe sekwencje posunięć. Drzewo gry zaczyna się pustą planszą w miejscu korzenia i ma odgałęzienia, odpowiadające każdej z możliwych alternatywnych sekwencji posunięć, w zależności od ruchów wykonywanych przez człowieka. (Drzewo nie musi mieć odgałęzień odpowiadających ruchom maszyny, ponieważ odpowiedź maszyny na każdy ruch jest z góry określona.) Mały fragment takiego drzewa gry pokazany jest na rysunku 1. Dla każdego ruchu, jaki może wykonać X, czyli gracz będący człowiekiem, istnieje z góry określona odpowiedź O, generowana przez maszynę. (Z jakiegoś dziwnego powodu informatycy zawsze rysują drzewa „do góry nogami", z „korzeniem" na górze.)



RYSUNEK 1.

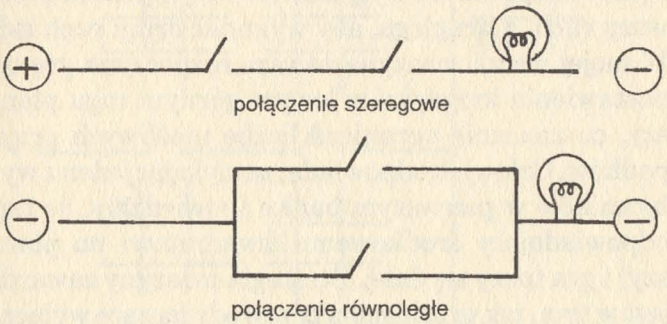
Fragment drzewa gry dla gry w kółko i krzyżyk

Drzewo na rysunku 1. ilustruje strategię, którą zawsze stosuję przy grze w kółko i krzyżyk: gram na środek, kiedy to tylko możliwe. Ruchy maszyny zde-

terminowane są przez ruchy człowieka, co znacznie ogranicza liczbę możliwości, które trzeba rozważyć. Pełne drzewo gry, pokazujące, co maszyna powinna zrobić w każdej sytuacji, ma jakieś pięćset czy sześćset odgałęzień — dokładna liczba zależy od szczegółów stosowanej strategii. Jeśli maszyna odpowiadać będzie zgodnie z tym drzewem, to wygra lub przynajmniej zremisuje każdą partię. Reguły gry wbudowane są w odpowiedzi, więc — działając zgodnie z drzewem gry — maszyna zawsze ich przestrzega. Posługując się drzewem gry możemy wypisać instrukcje mówiące maszynie, kiedy dokładnie powinna postawić swój znaczek w określonym polu. Instrukcje te stanowią logikę Boole'a maszyny.

Jeśli określiliśmy już pożądane zachowanie, to możemy je przetłumaczyć na układy elektryczne złożone z baterii, drutów, wyłączników i lampek. Podstawowym obwodem w maszynie jest taki sam obwód, jak w latarce: kiedy wyłącznik zostaje wciśnięty — czyli gdy obwód jest zamknięty — latarka zaczyna się świecić, ponieważ powstaje nieprzerwane połączenie elektryczne pomiędzy żarówką a baterią. (Podłączenia do baterii zaznaczone są za pomocą znaków „+" i „–".) Bardzo ważne jest to, że wyłączniki mogą być podłączane **szeregowo** lub **równolegle**. Możemy, na przykład, ustawić dwa wyłączniki jeden za drugim, uzyskując w ten sposób lampkę, która działa tylko wtedy, kiedy oba wyłączniki są zamknięte. Taki obwód realizuje jedną z podstawowych funkcji przełączających komputera — „blok logiczny" znany jako funkcja **AND**, nazywany tak, ponieważ żarówka zapala się tylko wtedy, gdy pierwszy wyłącznik jest zamknięty i drugi wyłącznik jest

zamknięty. Wyłączniki połączone równolegle tworzą funkcję **OR**, która zamyka obwód (a tym samym powoduje świecenie się żarówki) jedynie wtedy, gdy pierwszy wyłącznik jest zamknięty **lub** drugi wyłącznik jest zamknięty (obejmuje to również sytuację, kiedy oba wyłączniki są zamknięte) — patrz rysunek 2.



RYSUNEK 2.

Wyłączniki połączone szeregowo i równolegle

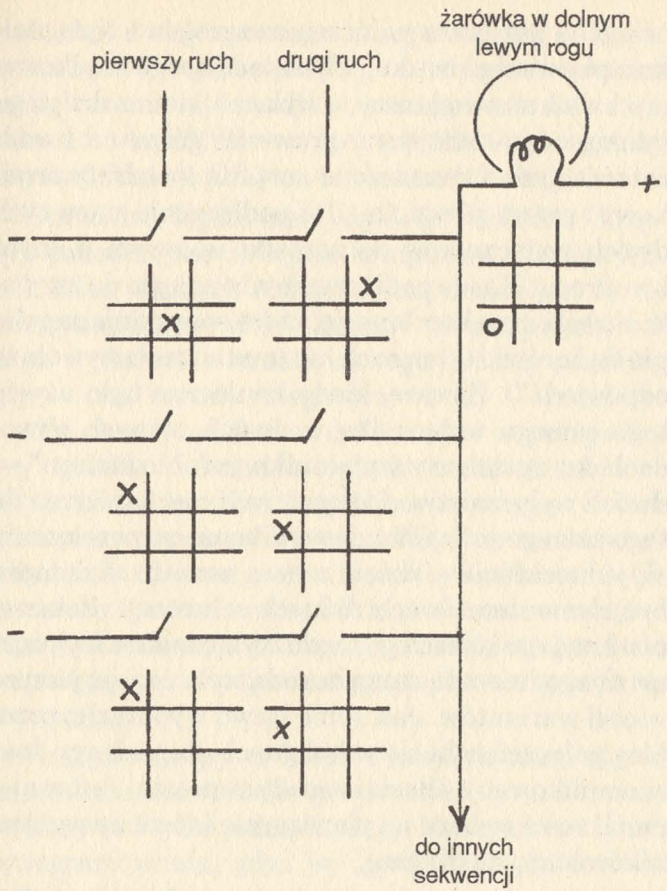
Te proste układy połączeń szeregowych i równoległych mogą być wykorzystane w różnych kombinacjach do tworzenia połączeń odpowiadających rozmaitym regułom logicznym. W maszynie do gry w kółko i krzyżyk ciągi wyłączników ustawionych **szeregowo** używane są do rozpoznawania sekwencji, podłączone są zaś **równolegle** do lampek. W ten sposób kilka sekwencji może spowodować zapalenie się tej samej lampki — czyli wywołać tę samą reakcję maszyny.

Zbudowana przeze mnie maszyna do gry w kółko i krzyżyk posiada cztery rzędy — tak zwane banki —

wyłączników, każdy po dziewięć wyłączników, a każdy wyłącznik odpowiada jednemu z dziewięciu pól na planszy. W jej skład wchodzi również dziewięć żarówek, ułożonych zgodnie ze schematem planszy gry. Maszyna, do której zawsze należy pierwszy ruch, wykonuje go zapalając żarówkę. Człowiek wykonuje swój ruch zamykając wyłącznik — korzystając z pierwszego banku wyłączników, aby wykonać pierwszy ruch, z drugiego, aby wykonać drugi ruch itd. W mojej wersji maszyna zawsze rozpoczyna grę od postawienia krzyżyka w lewym górnym rogu planszy, co znacznie ogranicza liczbę możliwych przypadków. Człowiek odpowiada, zamykając jeden z wyłączników w pierwszym banku (powiedzmy, że ten odpowiadający środkowemu kwadratowi na planszy) i gra toczy się dalej. Strategia maszyny zawarta jest w tym, jak przebiegają przewody łączące wyłączniki i lampki.

Układ przewodów generujący pierwszą odpowiedź maszyny jest prosty (patrz rysunek 3.). Każdy wyłącznik z pierwszego banku podłączony jest do lampki, która reprezentuje odpowiedź maszyny. Na przykład, na zagranie w środek maszyna odpowiada zagraniem w dolny prawy róg, a więc środkowy wyłącznik jest połączony przewodem z lampką w prawym dolnym rogu. Ponieważ moja maszyna zawsze, kiedy to tylko możliwe, odpowiada zaznaczeniem środkowego pola, to większość wyłączników połączona jest równolegle ze środkową lampką.

Każde posunięcie maszyny w drugiej rundzie zależy od pierwszego i drugiego posunięcia człowieka. Dla rozpoznania układu posunięć wykonanych przez człowieka odpowiednie wyłączniki połączone



RYSUNEK 3.

Kilka różnych sekwencji posunięć wywołujących tę samą odpowiedź

są szeregowo. Na przykład, jeżeli w pierwszym ruchu gracz zaznaczył środkowe pole, a w drugim prawe górne, to maszyna powinna odpowiedzieć, zaznaczając pole w lewym dolnym rogu. Taka sekwencja

osiągana jest przez połączenie szeregowe wyłącznika z pierwszego banku, odpowiadającego środkowemu kwadratowi planszy, z wyłącznikiem z drugiego banku, odpowiadającym prawemu górnemu kwadratowi („jeżeli zaznaczone zostaną kwadraty środkowy i prawy górny, to...") i podłączenie ciągu tych dwóch wyłączników do żarówki w lewym dolnym kwadracie. Każde podłączenie równoległe do żarówki określa inną kombinację, która spowoduje zapalenie się żarówki („ten ruch lub tamten ruch wywoła tę odpowiedź"). Zawsze, kiedy konieczne było użycie tego samego wyłącznika w dwóch różnych obwodach, korzystałem z wyłącznika „zdublowanego" — dwóch wyłączników podłączonych mechanicznie do tego samego przycisku, w wyniku czego przełączały się jednocześnie — dzięki czemu ten sam ruch może być elementem dwóch różnych sekwencji. Połączenia drugiego i trzeciego rzędu wyłączników wykonane są zgodnie z tą samą zasadą, tyle że jest jeszcze więcej wariantów. Jak sobie łatwo wyobrazić, przebieg połączeń robi się w kolejnych etapach gry dość skomplikowany, chociaż zasady są proste. Jest mniej możliwości wyboru na planszy, ale łańcuchy wyłączników stają się dłuższe.

Zbudowana przeze mnie maszyna do gry w kółko i krzyżyk miała około stu pięćdziesięciu wyłączników. W owym czasie wydawało mi się, że to bardzo dużo (wyłączniki zrobiłem z drewna i gwoździ), ale komputerowe układy scalone, które projektuję dzisiaj, mają miliony wyłączników; większość z nich połączona jest w układy bardzo podobne do tych, które zastosowałem w maszynie do gry w kółko i krzyżyk. W większości współczesnych komputerów używa się

wyłącznika elektrycznego innego rodzaju — tranzystora, który opiszę później — ale podstawowa zasada łączenia wyłączników szeregowo w celu uzyskania funkcji AND oraz łączenia ich równolegle, by uzyskać funkcję OR jest dokładnie taka sama.

Wprawdzie logika maszyny do gry w kółko i krzyżyk podobna jest do logiki komputera, istnieje jednak kilka ważnych różnic. Jedną z nich jest to, że maszyna do gry w kółko i krzyżyk nie zna pojęcia rozwoju wydarzeń w czasie; dlatego też cały przebieg gry — tzn. kształt całego drzewa gry — musi być z góry określony. Jest to dość trudne w przypadku gry w kółko i krzyżyk i praktycznie niemożliwe w przypadku bardziej skomplikowanych gier, takich jak szachy czy nawet warcaby. Współczesne komputery bardzo dobrze grają w warcaby i całkiem nieźle w szachy (patrz rozdział 9.), ponieważ w miejsce z góry określonego drzewa gry stosują inną metodę — polegającą na badaniu ciągu ruchów w sposób sekwencyjny.

Maszyna do gry w kółko i krzyżyk różni się od komputera uniwersalnego również i tym, że może wykonywać tylko jedną funkcję. Maszyna ta nie ma oprogramowania, gdyż jej „program" wbudowany jest w schemat konstrukcyjny.

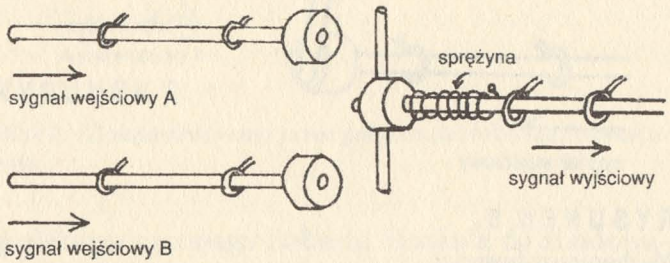
BITY I BLOKI LOGICZNE

Jak zaznaczyłem we wstępie, nie ma powodu, aby maszynę do gry w kółko i krzyżyk (lub jakikolwiek inny komputer) należało koniecznie budować z przełączników elektrycznych. Komputer może przedstawiać informację za pomocą prądu elektrycznego, ciś-

nienia płynu lub nawet reakcji chemicznych. Zasady, na których oparte jest działanie komputera, pozostają w dużym stopniu takie same, niezależnie od tego, czy buduje się go z tranzystorów, zaworów hydraulicznych, czy układów chemicznych. Podstawową koncepcją w maszynie do gry w kółko i krzyżyk jest realizowanie funkcji AND za pomocą szeregowego połączenia dwóch wyłączników, a funkcji OR za pomocą połączenia równoległego — istnieje jednak wiele innych sposobów realizowania funkcji AND i OR.

Tu muszę się zatrzymać, aby opowiedzieć trochę o **bitach**. Najmniejsza „różnica, która robi różnicę" (używając jeszcze raz sformułowania Batesona) jest różnicą, która dzieli wszystkie sygnały na dwie kategorie. W maszynie do gry w kółko i krzyżyk te dwie kategorie to „prąd płynie" i „prąd nie płynie". Na zasadzie konwencji nazywamy te dwie kategorie 1 i 0. To tylko nazwy, moglibyśmy je nazwać PRAWDA i NIEPRAWDA lub ALICJA i BOB. Nawet wybór tego, która kategoria nazywana jest 0, a która 1, jest arbitralny. Sygnał, który może przenosić jedną z dwóch możliwych informacji (np. 1 i 0) nazywany jest **sygnałem binarnym** lub **bitem**. Komputer używa kombinacji bitów do przedstawiania wszelkiego rodzaju zbiorów — zbioru różnych ruchów w grze w kółko i krzyżyk, lub, powiedzmy, zbioru różnych kolorów, które mają być wyświetlone na monitorze. Ponieważ zwykle bity oznacza się przez jedynki i zera, ludzie często traktują te układy bitów jak liczby — stąd często słyszane powiedzenie, że „komputer robi wszystko za pomocą liczb". Ale ta konwencja to tylko pewien sposób myślenia o tym, co się

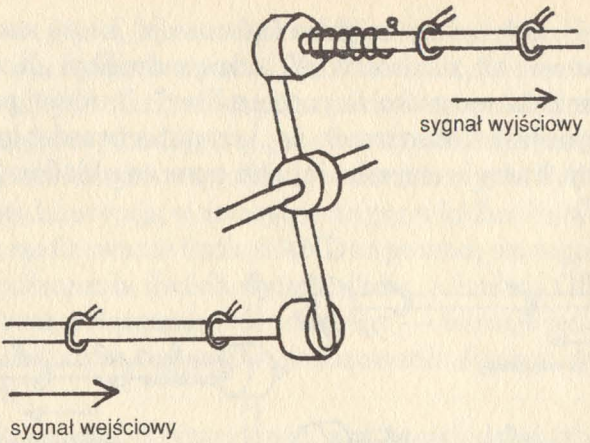
dzieje. Gdybyśmy te dwie informacje, które może przenosić bit, nazwali X i Y, ludzie mówiliby: „Komputer robi wszystko za pomocą liter". Bardziej precyzyjne jest stwierdzenie, że „komputer przedstawia liczby, litery i wszystko inne za pomocą układów bitów".



RYSUNEK 4.

Mechaniczna realizacja funkcji OR

Do przedstawienia bitu moglibyśmy zamiast prądu elektrycznego wykorzystać ruch mechaniczny. Na rys. 4. pokazane jest, jak można zrealizować funkcję OR za pomocą konstrukcji, w której bit 1 odpowiada przesunięciu drążka w prawo. Dopóki drążki A i B, przekazujące sygnał wejściowy, będą przesunięte w lewo, co odpowiada bitowi 0, to na skutek działania sprężyny drążek przekazujący sygnał wyjściowy również będzie przesunięty w lewo. Jeśli jednak oba drążki wejściowe przesuną się na prawo, to drążek wyjściowy także przesunie się na prawo. Układ przedstawiony na rys. 5 realizuje inną użyteczną funkcję, a mianowicie negację: **inwerter** zamienia każdy sygnał na przeciwny — zamienia na

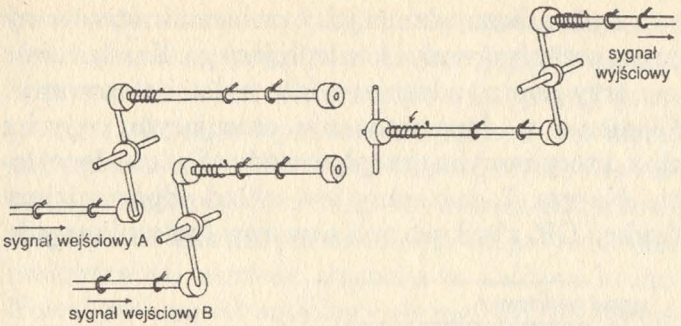


RYSUNEK 5.
Mechaniczny inwerter

przykład pchnięcie w prawo na pociągnięcie w lewo i na odwrót.

Funkcje AND, OR i NOT to **bloki logiczne**, które mogą być łączone w celu tworzenia innych funkcji. Na przykład, wyjście bloku OR może zostać podłączone do bloku NOT, tworząc w ten sposób funkcję NOT-OR: funkcja ta da na wyjściu 1, jeśli żaden z sygnałów na wejściu nie będzie 1. Możemy też utworzyć blok AND przez podłączenie dwóch bloków NOT do wejść bloku OR i podłączenie trzeciego bloku NOT do jego wyjścia (patrz rys. 6.). W tym układzie czterech bloków sygnał na wyjściu jest 1 tylko wtedy, gdy oba sygnały na wejściu to 1.

Wczesne maszyny liczące oparte były na układach mechanicznych. W siedemnastym wieku Blaise Pascal zbudował mechaniczny sumator, który zainspirował Gottfrieda Wilhelma Leibniza i wielkiego an-



RYSUNEK 6.

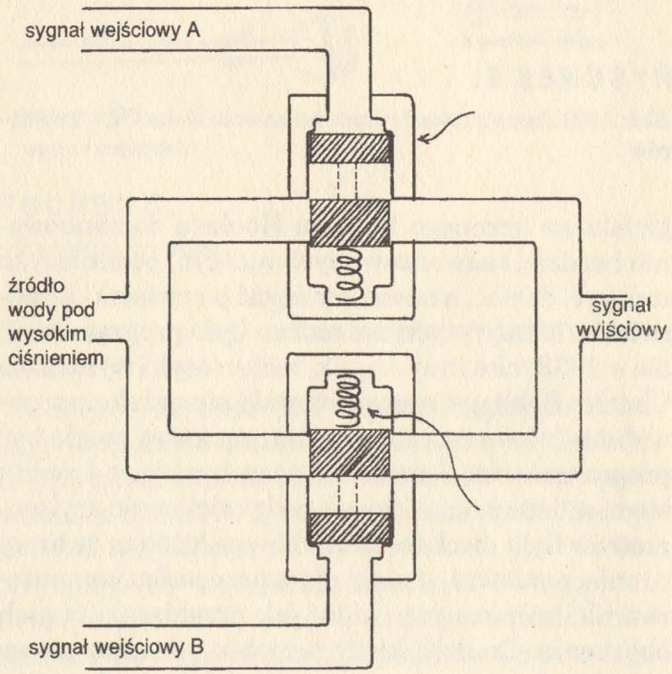
Blok AND skonstruowany przez połączenie bloku OR i inwerterów

gielskiego uczonego Roberta Hooke'a do zbudowania bardziej zaawansowanych maszyn, potrafiących mnożyć, dzielić, a nawet wyciągać pierwiastki kwadratowe. Maszyn tych nie można było programować, ale w 1833 roku inny Anglik, matematyk i wynalazca Charles Babbage, zaprojektował i niemal skonstruował mechaniczną maszynę liczącą, która mogła być programowana. Jeszcze w czasach mojego dzieciństwa, w latach sześćdziesiątych, większość arytmometrów była mechaniczna. Zawsze lubiłem te urządzenia, ponieważ, inaczej niż w przypadku komputerów elektronicznych, widać jak przebiegają w nich obliczenia. Do dziś, kiedy projektuję komputerowy układ scalony, wyobrażam sobie działanie obwodów w postaci poruszających się mechanicznych części.

KOMPUTER NA WODĘ

Obraz, jaki mam w myślach, kiedy projektuję obwód logiczny, to układ zaworów hydraulicznych. Za-

wór hydrauliczny działa jak przełącznik, sterowany przez przepływ wody i kontrolujący go. Każdy zawór ma trzy doprowadzenia: wlot, wylot i sterowanie. Ciśnienie w doprowadzeniu sterującym popycha tłok, który zamyka przepływ wody od wlotu do wylotu. Na rys. 7. pokazany jest układ odpowiadający funkcji OR, zbudowany z zaworów hydraulicznych.



RYSUNEK 7.

Blok OR zbudowany przy użyciu zaworów hydraulicznych

W tym układzie ciśnienie wody używane jest do rozróżnienia między dwoma możliwymi sygnałami. Należy podkreślić, że w zaworze hydraulicznym rur-

ka sterująca może wpłynąć na wylotową, ale rurka wylotowa nie może wpływać na sterującą. To ograniczenie ustala kierunek przepływu informacji przez przełącznik; w pewnym sensie wyznacza ono kierunek upływu czasu. Ponieważ zawór jest albo otwarty, albo zamknięty, spełnia on również dodatkową rolę **wzmocnienia**, pozwalającego na przywrócenie maksymalnej wartości sygnału w każdym kroku. Nawet jeśli sygnał wejściowy nie ma zbyt dużego ciśnienia — ponieważ przechodzi przez długą, cienką rurkę lub chociażby z powodu przecieku — to, ze względu na skokowy sposób działania zaworu, sygnał wyjściowy zawsze będzie miał ciśnienie maksymalne. Jest to zasadnicza cecha, odróżniająca układ **cyfrowy** od **analogowego**: „zawór" cyfrowy jest albo włączony, albo wyłączony, natomiast zawór analogowy, na przykład kuchenny kran, może przyjąć dowolne położenie pośrednie. W przypadku komputera hydraulicznego, od sygnału wejściowego wymaga się jedynie, aby był wystarczająco silny, by poruszyć zawór. W tym przypadku „różnicą, która robi różnicę" jest różnica ciśnień, wystarczająca do włączenia zaworu. Ponieważ osłabiony sygnał wejściowy również wytworzy sygnał wyjściowy o pełnej sile, to możemy łączyć tysiące warstw układów logicznych w taki sposób, że sygnał wyjściowy jednej warstwy steruje następną, i nie musimy martwić się o stopniowy spadek ciśnienia. Sygnał wyjściowy każdej bramki zawsze będzie miał właściwe ciśnienie.

Tego typu konstrukcja nazywa się **układem logicznym z regeneracją**, a przykład z hydrauliki jest szczególnie interesujący, ponieważ prawie dokładnie odpowiada układom logicznym, wykorzy-

stywanym we współczesnych komputerach. Ciśnienie wody w rurkach jest odpowiednikiem napięcia w przewodach, zawór hydrauliczny jest analogiem tzw. tranzystora polowego z izolowaną bramką, a doprowadzenia sterujące, wlotowe i wylotowe w zaworze odpowiadają trzem elektrodom (nazywanym **bramką, źródłem i drenem**) w tranzystorze. Analogia między zaworami hydraulicznymi i tranzystorami jest tak dokładna, że można by wręcz przenieść konstrukcję współczesnego mikroprocesora bezpośrednio na projekt komputera hydraulicznego. Aby tego dokonać, należałoby obejrzeć pod mikroskopem układ na płytce krzemowej, a potem wygiąć zestaw rurek w te same kształty, co obwody na płytce i połączyć je w dokładnie taki sam układ. W miejscu każdego tranzystora trzeba użyć zaworu hydraulicznego. Rurka odpowiadająca obwodowi doprowadzającemu napięcie do takiego mikroprocesora powinna zostać podłączona do źródła wody pod ciśnieniem, a rurka odpowiadająca uziemieniu powinna biec do odpływu.

Aby posłużyć się hydraulicznym komputerem, musielibyście podłączyć hydrauliczne odpowiedniki wejść i wyjść — potrzebna będzie hydrauliczna klawiatura, hydrauliczny ekran, hydrauliczne układy pamięci, itd. — ale jeśli to wszystko zrobicie, wasz komputer przechodzić będzie przez dokładnie takie same sekwencje przełączeń, jakie zachodzą w elektronicznym procesorze. Oczywiście, taki komputer będzie znacznie wolniejszy od waszego najnowszego mikroprocesora (nie mówiąc o tym, że będzie od niego znacznie większy), ponieważ ciśnienie wody przemieszcza się wzdłuż rurek znacznie wolniej niż syg-

nały elektryczne w obwodach. A co do wielkości — ponieważ współczesny mikroprocesor ma kilka milionów tranzystorów, to jego hydrauliczny odpowiednik wymagać będzie kilku milionów zaworów. Tranzystor w układzie scalonym ma rozmiary około milionowej części metra; zawór hydrauliczny — około 10 centymetrów. Jeśli rurki wykonane będą z zachowaniem tej samej skali, to obwody komputera hydraulicznego zajmą obszar około kilometra kwadratowego. Oglądane z samolotu, wyglądałyby mniej więcej tak samo, jak oglądany pod mikroskopem mikroprocesor.

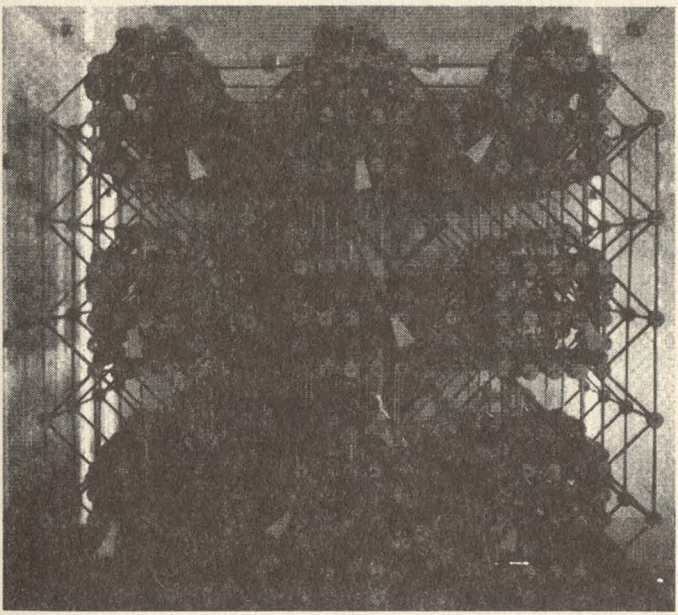
Kiedy projektuję komputerowy układ scalony, to najpierw rysuję linie na ekranie komputera, a potem rysunek jest pomniejszany (w procesie podobnym do zmniejszania zdjęcia fotograficznego) i nanoszony na płytkę krzemową. Linie na ekranie są moimi rurkami i zaworami. W istocie, większość projektantów komputerów nie zadaje sobie nawet trudu, aby rysować linie; określają jedynie połączenia między blokami logicznymi AND i OR i pozostawiają komputerowi decyzję co do szczegółów rozmieszczenia i względnego położenia przełączników. Przez większość czasu nie myślą oni o technice, lecz koncentrują się na zasadzie działania. Ja też niekiedy tak robię, choć na ogół wolę sam rysować układy. Zawsze, kiedy projektuję mikroukład, chcę go przede wszystkim obejrzeć pod mikroskopem — nie dlatego, bym uważał, że mogę się w ten sposób nauczyć czegoś nowego, ale dlatego, że fascynuje mnie, jak rysunek może kreować rzeczywistość.

Poza cudem miniaturyzacji nie ma specjalnego powodu, aby budować komputery w oparciu o technologię krzemową. Budowanie ich w jakiejkolwiek technologii wymaga jedynie posiadania znacznych ilości dwóch rodzajów elementów: **przełączników** i **łączników.** Przełącznik jest elementem sterującym (zawór hydrauliczny lub tranzystor), który może przekształcać wiele sygnałów w jeden. Najlepiej, aby przełącznik był asymetryczny — czyli by sygnał wejściowy mógł wpływać na sygnał wyjściowy, ale nie na odwrót — i powinien mieć też zdolność regeneracji sygnału, aby słaby lub zaburzony sygnał wejściowy nie dał w efekcie zniekształconego sygnału na wyjściu. Łącznik jest obwodem lub rurką, przenoszącą sygnał między przełącznikami. Najlepiej byłoby, aby łącznik dopuszczał możliwość dzielenia sygnału — to znaczy, by jeden sygnał wyjściowy mógł posłużyć za wiele sygnałów wejściowych. Są to jedyne elementy konieczne do zbudowania komputera. Później poznamy jeszcze jeden element — rejestr, służący do przechowywania informacji — który jednak może być skonstruowany z tych samych elementów sterujących i łączących.

Nigdy nie zbudowałem komputera hydraulicznego, ale raz, z grupą przyjaciół, skonstruowałem komputer z drążków i linek. Elementy te pochodziły z dziecięcego zestawu zwanego Tinker Toy. Jak niektórzy jeszcze pamiętają, jest to zestaw cylindrycznych drążków, które wciska się w małe drewniane gniazda z otworami. Układ logiczny mojego komputera z elementów Tinker Toy działał bardzo podob-

nie do konstrukcji pokazanej na rys. 8. Podobnie jak komputer zbudowany z przełączników i żarówek, komputer Tinker Toy również grał w kółko i krzyżyk. Nigdy nie przegrywał. Zbudowanie go kosztowało nas wiele wysiłku i wymagało dziesiątek tysięcy części z ponad stu wielkich kompletów Tinker Toy, a rezultat końcowy (znajdujący się obecnie w Muzeum Komputerów w Bostonie, w stanie Massachusetts) wygląda na bardzo skomplikowany. A jednak jego działanie opiera się na prostej kombinacji omówionych wyżej funkcji AND i OR.

Projektując komputer z elementów Tinker Toy, popełniłem wielki błąd, który polegał na tym, że nie



RYSUNEK 8.

Komputer zbudowany z elementów Tinker Toy

użyłem **układów logicznych z regeneracją** — między kolejnymi krokami logicznymi nie było wzmocnienia. Realizacja logiki w tym komputerze opierała się na konstrukcji, w której pewne drążki naciskają na inne, według schematu podobnego do tego pokazanego na rys. 4. Przy takiej konstrukcji cała siła konieczna do poruszenia setek elementów w maszynie musiała być dostarczona przez naciśnięcie przełącznika na wejściu. Skutkiem działania tej skumulowanej siły było rozciąganie linek przenoszących ruchy, a ponieważ na kolejnych krokach nie było regeneracji, błędy spowodowane rozciąganiem kumulowały się przy przejściu od jednego układu logicznego do drugiego. Jeśli linki nie były ciągle regulowane, maszyna robiła błędy.

Skonstruowałem następną wersję komputera z elementów Tinker Toy, w której ten problem został rozwiązany, ale nigdy nie zapomniałem lekcji otrzymanej przy pierwszej maszynie: technologia użyta do realizacji układów logicznych musi dawać idealne sygnały na wyjściu nawet wtedy, gdy sygnały na wejściu zawierają zakłócenia W ten sposób można uniknąć kumulowania się małych błędów. Jest to istota techniki cyfrowej, w której na każdym kroku sygnały przywracane są do niemal idealnej postaci. Jest to jedyny do tej pory znany nam sposób na zachowanie kontroli nad skomplikowanym układem.

WYSTARCZY MARTWIĆ SIĘ JEDYNIE O TĘ RÓŻNICĘ, KTÓRA ROBI RÓŻNICĘ

Nazwanie dwóch sygnałów w logice komputerowej 0 i 1 jest przykładem abstrakcji funkcjonalnej. W ten sposób możemy manipulować informacją bez konieczności troszczenia się o szczegóły jej konkretnej reprezentacji. Kiedy tylko zrozumiemy, jak zrealizować daną funkcję, możemy zamknąć mechanizm w „czarnej skrzynce" lub „bloku" i przestać o nim myśleć. Funkcja realizowana przez blok może być używana wielokrotnie, bez zwracania uwagi na to, co jest w środku. Ten proces funkcjonalnej abstrakcji ma zasadnicze znaczenie przy projektowaniu komputerów — nie jest to wprawdzie jedyna metoda projektowania skomplikowanych układów, ale za to najczęściej stosowana (później opiszę metodę alternatywną). Komputery zbudowane są z hierarchii takich abstrakcji funkcjonalnych, z których każda ucieleśniana jest przez **blok**. Bloki realizujące funkcje łączone są razem w celu realizowania funkcji bardziej złożonych, a układy bloków z kolei stają się blokami dla następnego poziomu.

Ten hierarchiczny układ abstrakcji jest naszym najpotężniejszym narzędziem w analizie skomplikowanych układów, ponieważ pozwala nam na skoncentrowanie się na jednym tylko aspekcie problemu. Możemy na przykład mówić o funkcjach Boole'a, takich jak AND i OR, w sposób abstrakcyjny, bez troszczenia się o to, czy są one zbudowane z przełączników elektrycznych, z drążków i linek, czy też ze sterowanych strumieniem wody zaworów hydraulicznych. Przy większości zastosowań możemy zapo-

mnieć o technice. Jest to wspaniałe, ponieważ oznacza, że prawie wszystko, co mówimy o komputerach, pozostanie prawdziwe nawet wtedy, kiedy tranzystory i krzemowe płytki staną się przestarzałe.

Rozdział drugi

UNIWERSALNE ELEMENTY KONSTRUKCYJNE

Od tego momentu możemy zapomnieć o przewodach i przełącznikach i zająć się abstrakcyjnymi blokami logicznymi, operującymi sygnałami 1 i 0 — ten prosty krok pozwala nam przejść ze świata techniki do świata matematyki. Ten rozdział jest najbardziej abstrakcyjnym rozdziałem w książce; opisuję w nim jak — korzystając z metod użytych przy budowie maszyny do gry w kółko i krzyżyk — można zrealizować praktycznie dowolną funkcję. Teraz właśnie zdefiniujemy ważny zbiór elementów konstrukcyjnych; funkcje logiczne i maszyny o skończonej liczbie stanów. Korzystając z tych elementów łatwo jest zbudować komputer.

FUNKCJE LOGICZNE

Przy budowie maszyny do gry w kółko i krzyżyk zaczęliśmy od narysowania drzewa gry, co pozwoliło nam ustalić zbiór zasad dla przetwarzania sygnałów wejściowych w sygnały wyjściowe. Okazuje się, że taka metoda postępowania jest użyteczna w każdym

przypadku. Kiedy już spiszemy reguły określające, jakie sygnały chcemy uzyskiwać na wyjściu przy każdej z kombinacji sygnałów wejściowych, możemy skonstruować urządzenie, realizujące te reguły za pomocą funkcji AND, OR i NOT. Bloki logiczne AND, OR i NOT tworzą **uniwersalny zbiór podstawowych bloków logicznych**, przy użyciu których można zrealizować dowolny zbiór reguł. (Te elementarne typy bloków logicznych nazywane są czasem **bramkami logicznymi**.)

Koncepcja zbioru elementów konstrukcyjnych na tyle ogólnego, że można z nich zbudować dowolny układ logiczny, jest bardzo ważna. Moją ulubioną zabawką w dzieciństwie były klocki Lego, z których budowałem najrozmaitsze zabawki: samochody, domy, statki kosmiczne, dinozaury. Uwielbiałem bawić się tymi klockami, ale nie były one zupełnie uniwersalne, ponieważ wszystko, co można było z nich zbudować, miało charakterystyczny, kanciasty i schodkowy wygląd. Budowanie czegoś o innym kształcie — na przykład cylindra albo kuli — wymagałoby klocków innego rodzaju. Aby w końcu skonstruować to, co chciałem, musiałem posłużyć się innymi elementami. Natomiast bloki AND, OR i NOT w logice Boole'a tworzą uniwersalny zbiór bloków, przetwarzających sygnały wejściowe na sygnały wyjściowe. Najlepszym sposobem zobaczenia, na czym polega ich uniwersalność, jest zapoznanie się z ogólną metodą posługiwania się nimi w celu realizowania reguł logicznych. Rozważmy na początek reguły **binarne** — czyli takie, przy których sygnały wejściowe i wyjściowe można zapisać jako 1 lub 0. Maszyna do gry w kółko i krzyżyk jest dobrym przykładem funkcji

określanej przez reguły binarne, ponieważ wyłączniki na wejściu i żarówki na wyjściu są albo włączone, albo wyłączone — czyli albo 1, albo 0. (Później omówimy reguły umożliwiające przetwarzanie sygnałów innego typu: liter, liczb, a nawet obrazów i dźwięków.) Dowolny zestaw reguł binarnych jest jednoznacznie określony przez tabelę sygnałów wyjściowych dla każdej z możliwych kombinacji sygnałów 1 i 0 na wejściu. Na przykład, reguły dla funkcji OR określone są przez następującą tabelę:

sygnał na wejściu A	sygnał na wejściu B	wyjście
0	0	0
0	1	1
1	0	1
1	1	1

funkcja OR

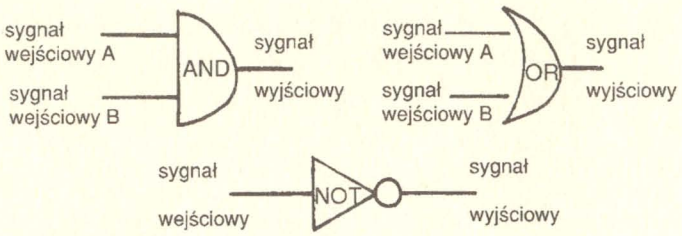
Funkcja NOT określana jest przez jeszcze prostszą tabelę:

sygnał na wejściu	sygnał na wyjściu
0	1
1	0

funkcja NOT

Dla funkcji binarnej z n sygnałami wejściowymi istnieje 2^n możliwych kombinacji tych sygnałów. Czasami nie musimy uwzględniać wszystkich, ponieważ pewne kombinacje na wejściu nie będą dla nas istotne. Na przykład, przy określaniu reguł kierujących ruchami maszyny do gry w kółko i krzyżyk, nie interesuje nas, co się stanie, kiedy gracz zaznaczy wszystkie kratki jednocześnie. Taki ruch będzie zaka-

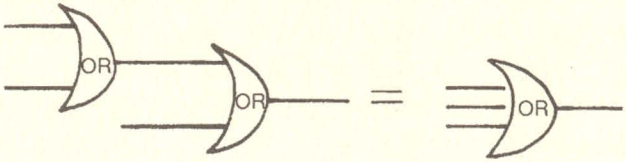
zany i nie trzeba określać sygnału wyjściowego naszej funkcji dla tej kombinacji sygnałów wejściowych.



RYSUNEK 9.

Symbole bloków AND, OR i NOT

Bardziej złożone bloki logiczne konstruowane są przez łączenie bloków AND, OR i NOT. Przyjęto konwencję, że na rysunkach przedstawiających schematy połączeń te trzy bloki przedstawiane są jako ramki o różnym kształcie (patrz rys. 9.); linie dochodzące do ramek od lewej strony oznaczają sygnały wchodzące do bloku, a linie z prawej strony oznaczają sygnały wychodzące. Na rys. 10. pokazano, jak można połączyć dwa bloki OR z dwoma wejściami, aby otrzymać funkcję OR z trzema wejściami; funkcja ta da na wyjściu 1 jeśli **którykolwiek** z trzech sygnałów wejściowych jest 1. Łącząc w podobny sposób kil-

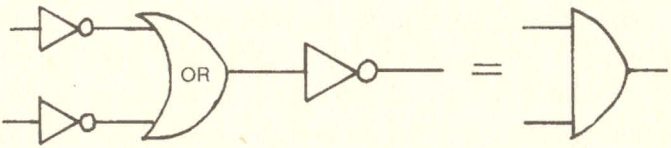


RYSUNEK 10.

Blok OR z trzema wejściami utworzony z dwóch bloków OR z dwoma wejściami

ka bloków AND można otrzymać blok AND z dowolną liczbą wejść.

Na rys. 11. pokazane jest, jak można skonstruować blok AND przez podłączenie inwertera do wejść i wyjścia bloku OR. (Znów kłania się prawo de Morgana.) Najlepszym sposobem, by zrozumieć działanie tego układu w sposób intuicyjny jest prześledzenie jedynek i zer dla każdej kombinacji sygnałów wejściowych. Proszę zauważyć, że ten rysunek jest w istocie identyczny z rys. 6. z poprzedniego rozdziału. Wynika stąd ciekawy wniosek: bloki AND nie są tak naprawdę niezbędne w naszym uniwersalnym zestawie konstrukcyjnym, ponieważ możemy je zawsze zbudować z bloków OR i inwerterów.



RYSUNEK 11.

Blok AND zbudowany z bloku OR i inwerterów

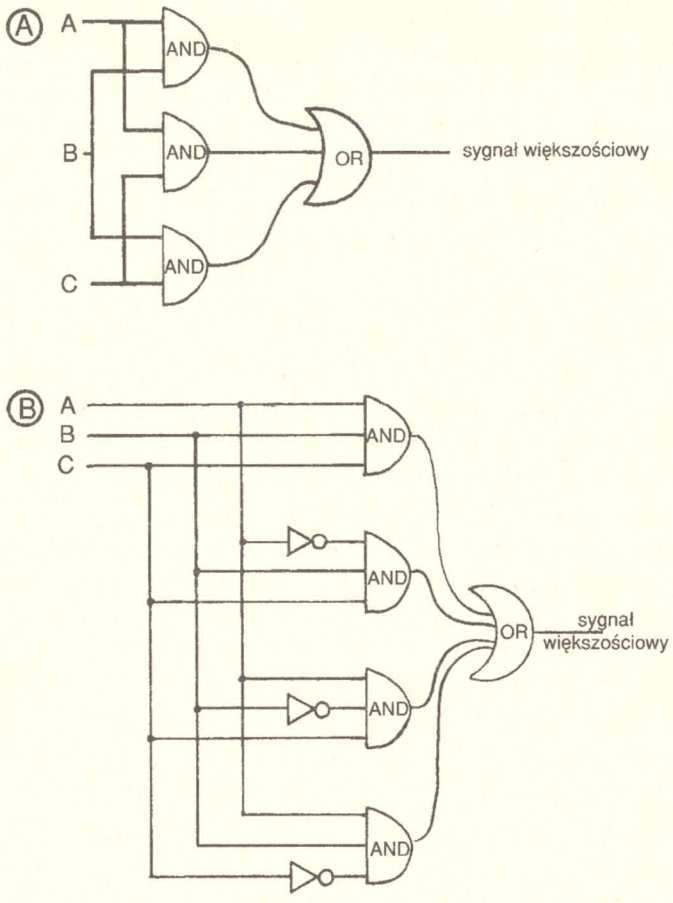
Podobnie jak w maszynie do gry w kółko i krzyżyk, bloki AND używane są do wykrywania tych kombinacji sygnałów wejściowych, dla których sygnał wyjściowy jest 1, podczas gdy bloki OR tworzą zestawienie tych kombinacji. Rozważmy na przykład prostą funkcję z trzema wejściami. Wyobraźmy sobie, że chcemy zbudować blok, który umożliwi trzem sygnałom wejściowym „głosowanie" nad sygnałem wyjściowym. W tym nowym bloku wygrywa większość — na wyjściu 1 pojawi się tylko wtedy, gdy przynajmniej dwa sygnały wejściowe odpowiadają 1.

sygnały na wejściu			większościowy sygnał wyjściowy
A	B	C	
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Na rys. 12A. pokazano, jak taka funkcja może zostać zrealizowana. Blok AND z odpowiednimi blokami NOT na wejściu wykorzystany jest do rozpoznania każdej kombinacji sygnałów wejściowych, dla których „głosowanie" daje 1; te bloki podłączone są do bloku OR, wytwarzającego sygnał wyjściowy. Strategia ta może zostać wykorzystana do zrealizowania dowolnej transformacji sygnałów wejściowych na wyjściowe.

Oczywiście, ta szczególna metoda, polegająca na wykorzystaniu osobnej bramki AND do rozpoznania każdej kombinacji sygnałów wejściowych, nie jest jedynym sposobem na zrealizowanie tej funkcji i często też nie jest sposobem najprostszym. Na rys. 12B. pokazano prostszy sposób zrealizowania funkcji większościowej. Zaletą opisanej metody nie jest to, że daje najlepszą realizację, ale że zawsze prowadzi do realizacji, która działa właściwie. Można stąd wyciągnąć ważny wniosek, że przez łączenie bloków AND, OR i NOT możliwe jest zrealizowanie **dowolnej** funkcji binarnej — czyli dowolnej funkcji, która

może być zdefiniowana za pomocą zawierającej jedynie zera i jedynki tabeli wejście/wyjście.



RYSUNEK 12

Funkcja większościowa zrealizowana za pomocą bloków AND, OR i NOT

Ograniczenie sygnałów wejściowych i wyjściowych do liczb w układzie dwójkowym nie jest tak naprawdę ograniczeniem istotnym, ponieważ kombinacje jedynek i zer mogą zostać użyte do reprezentowania wszelkich innych symboli — liter, liczb, czy wręcz dowolnej wielkości, jaka może zostać zakodowana. Aby dać przykład funkcji, która nie jest binarna, przypuśćmy, że chcemy zbudować mechanicznego sędziego, rozstrzygającego w dziecięcej grze Nożyczki/Papier/Kamień. Jest to gra dla dwóch graczy, z których każdy wybiera w tajemnicy jedną z trzech „broni" — nożyczki, papier lub kamień. Reguły są proste: nożyczki przecinają papier, papier przykrywa kamień, kamień niszczy nożyczki. Jeśli obaj gracze wybiorą tę samą broń, jest remis. Zamiast budować maszynę, która grałaby w tę grę (co wiązałoby się z odgadywaniem, którą broń wybierze przeciwnik), zbudujemy urządzenie orzekające kto wygrał. Poniżej przedstawiona jest tabela wejście/wyjście dla funkcji, która wybory graczy traktuje jako sygnały wejściowe, a na wyjściu daje sygnał, kto wygrał. Tabela odzwierciedla reguły gry:

sygnał na wejściu A	sygnał na wejściu B	wynik
nożyczki	nożyczki	remis
nożyczki	papier	wygrywa A
nożyczki	kamień	wygrywa B
papier	nożyczki	wygrywa B
papier	papier	remis
papier	kamień	wygrywa A
kamień	nożyczki	wygrywa A
kamień	papier	wygrywa B
kamień	kamień	remis

Funkcja orzekająca «Nożyczki–Papier–Kamień» jest funkcją kombinatoryczną, ale nie jest funkcją binarną, ponieważ sygnały na wejściu i wyjściu mogą przyjmować więcej niż dwie wartości. Aby zrealizować ją jako kombinatoryczny blok logiczny, musimy ją przekształcić w funkcję operującą na jedynkach i zerach. Wymaga to przyjęcia pewnej konwencji dla reprezentowania sygnałów na wejściu i wyjściu. Prostym sposobem jest użycie osobnego bitu dla każdej z możliwości. Można wprowadzić trzy sygnały wejściowe: jedynka na pierwszym wejściu reprezentuje Nożyczki, jedynka na drugim Kamień, a jedynka na trzecim wejściu — Papier. Podobnie moglibyśmy użyć osobnych linii wyjściowych do przedstawienia wygranej gracza A, wygranej gracza B lub remisu. Tak więc, skrzynka miałaby sześć wejść i trzy wyjścia.

Użycie po jednym sygnale wejściowym dla każdej „broni" jest zupełnie dobrym sposobem zrealizowania tej funkcji, ale gdybyśmy robili to w komputerze, to prawdopodobnie wykorzystalibyśmy jakiś sposób kodowania, wymagający mniejszej liczby wejść i wyjść. Moglibyśmy na przykład użyć dwóch bitów dla każdego wejścia i zastosować kombinację 01 do reprezentowania Nożyczek, 10 dla Papieru i 11 dla Kamienia. W podobny sposób moglibyśmy zakodować każdy z możliwych sygnałów wyjściowych. Kodowanie tego typu dałoby w efekcie prostszą tabelę trzy-wejścia/dwa-wyjścia, pokazaną na następnej stronie.

Komputery mogą używać kombinacji bitów do reprezentowania wszystkiego: liczba bitów zależy od liczby różnych komunikatów, jakie chcemy przekazać. Wyobraźmy sobie na przykład komputer, który

operuje na literach alfabetu. Pięciobitowy sygnał wejściowy może reprezentować 32 różne znaki (2^5 = 32). Funkcje w komputerze operujące na literach czasami wykorzystują taki kod, choć częściej używają kodu z siedmioma a nawet ośmioma bitami, co umożliwia im reprezentację dużych i małych liter, znaków interpunkcyjnych, cyfr itd. Większość współczesnych komputerów wykorzystuje standardową reprezentację liter alfabetu nazywaną ASCII (skrót od *American Standard Code for Information Interchange* — amerykański standardowy kod wymiany informacji). W ASCII sekwencja 1000001 przedstawia wielką literę A, 1000010 — wielką literę B i tak dalej. Konwencja jest oczywiście arbitralna, przyporządkowania mogłyby być zupełnie inne.

	sygnały na wejściu A	sygnały na wejściu B	sygnały na wyjściu
Nożyczki = 01	01	01	00
Papier = 10	01	10	01
Kamień = 11	01	11	10
Wygrywa A = 10	10	01	01
Wygrywa B = 01	10	10	00
Remis = 00	10	11	10
	11	01	10
	11	10	01
	11	11	00

Większość komputerów wykorzystuje kilka systemów reprezentowania liczb. Jednym z najbardziej rozpowszechnionych jest reprezentacja dwójkowa, w której sekwencja 0000000 odpowiada zeru, sekwencja 0000001 — 1, 0000010 to 2 itd. Mówiąc o komputerach, że są „64-bitowe" lub „32-bitowe",

mamy na myśli liczbę pozycji bitów w reprezentacji używanej w obwodach: komputer 32-bitowy wykorzystuje kombinację trzydziestu dwóch bitów do przedstawienia liczby w układzie dwójkowym. Układ dwójkowy jest powszechnie stosowaną konwencją, ale nie ma niczego, co narzucałoby jego stosowanie. Niektóre komputery nie wykorzystują go wcale, a większość tych, które z niego korzystają, przedstawia liczby na kilka różnych sposobów w zależności od potrzeb. Na przykład, wiele komputerów wykorzystuje nieco inną konwencję dla reprezentowania liczb ujemnych, a ułamki dziesiętne przedstawia za pomocą konwencji zwanej **zmiennoprzecinkową**. (Położenie przecinka w ułamku dziesiętnym jest zmienne, co pozwala opisać za pomocą ustalonej liczby cyfr szeroki zakres liczb.) Konkretne reprezentacje są często wybierane w taki sposób, aby uprościć schemat logiczny układów wykonujących operacje arytmetyczne lub ułatwić przechodzenie od jednej konwencji do innej.

Ponieważ dowolna funkcja logiczna może zostać zrealizowana jako blok logiki Boole'a, to możliwe jest tworzenie bloków wykonujących operacje arytmetyczne na liczbach w dowolnej reprezentacji. Wyobraźmy sobie na przykład, że chcemy zbudować blok, który będzie dodawał liczby na ośmiobitowym komputerze. Ośmiobitowy sumator musi mieć szesnaście sygnałów wejściowych (osiem dla każdej z dodawanych liczb) i osiem sygnałów wyjściowych dla sumy. Ponieważ każda liczba reprezentowana jest przez osiem bitów, to istnieje 256 możliwych kombinacji i każda może przedstawiać inną liczbę. Moglibyśmy na przykład użyć tych kombinacji do repre-

zentowania liczb między 0 a 255, lub -100 i +154. Definiowanie działania bloku polega na wypisaniu tabeli dodawania, a następnie przetłumaczeniu jej na jedynki i zera w wybranej reprezentacji. Tablicę jedynek i zer można następnie zamienić na układ bloków AND i OR za pomocą metod opisanych powyżej.

Dodając jeszcze do tego bloku dwa wejścia moglibyśmy zbudować w taki sposób blok, który nie tylko dodaje, ale też odejmuje, mnoży i dzieli. Te dwa dodatkowe wejścia sterujące określałyby, która z operacji ma być wykonana. Jeśli na przykład sygnałem sterującym byłoby 01, to sygnał wyjściowy w każdej linii tabeli byłby sumą liczb na wejściu; jeśli sygnałem sterującym byłoby 10, to za sygnał wyjściowy przyjęlibyśmy iloczyn liczb, itd. Bloki logiczne tego typu nazywane są **arytmometrami** — i wyposażona jest w nie większość komputerów.

Łącząc bloki AND i OR zgodnie z naszą strategią można zrealizować dowolną funkcję logiczną, ale nie zawsze jest to realizacja najbardziej efektywna. Wiele obwodów można zrealizować za pomocą znacznie mniejszej liczby bloków, niż tego wymaga opisana powyżej metoda. Czasami może wyniknąć potrzeba użycia bloków konstrukcyjnych innego typu lub zaprojektowania obwodów tak, aby zminimalizować opóźnienie sygnału wyjściowego w stosunku do sygnału wejściowego. A oto kilka typowych zagadek z dziedziny projektowania układów logicznych: Jak, za pomocą bloków AND i inwerterów, budować bloki OR? (Łatwe.) Jak, za pomocą zestawu bloków AND i OR i tylko dwóch inwerterów można zbudować układ trzech inwerterów? (Trudne, ale możliwe.) Zagadki tego typu pojawiają się ciągle przy projekto-

waniu komputerów i między innym dlatego sprawia ono tak wielką frajdę.

MASZYNY O SKOŃCZONEJ LICZBIE STANÓW

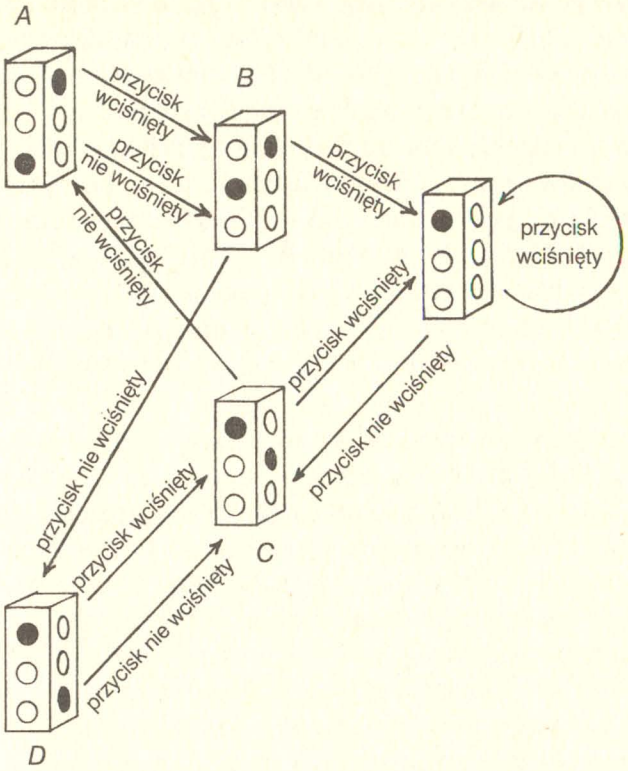
Opisana przeze mnie metoda może być użyta do zrealizowania dowolnej funkcji niezależnej od czasu, ale bardziej interesująca jest klasa funkcji związanych z ciągami czasowymi. Aby poradzić sobie z takimi funkcjami wykorzystujemy urządzenie nazywane **maszyną o skończonej liczbie stanów** (maszyną skończoną). Za pomocą maszyn skończonych można zrealizować funkcje zależne od czasu — takie, które zależą nie tylko od bieżącego sygnału wejściowego, ale też od historii sygnałów wejściowych. Kiedy nauczymy się rozpoznawać maszyny skończone, dostrzeżemy je wszędzie — w zamkach szyfrowych, długopisach, a nawet w umowach cywilnoprawnych. Koncepcja maszyny skończonej polega na połączeniu tabeli określającej sygnał wyjściowy w zależności od sygnału wejściowego z elementem pamięciowym. Pamięć używana jest do przechowywania zbioru danych o przeszłości, co określa **stan** maszyny skończonej.

Prostym przykładem maszyny skończonej jest zamek szyfrowy. Stan takiego zamka jest zbiorem sekwencji liczb wybranych w zamku. Zamek nie pamięta wszystkich liczb, jakie kiedykolwiek zostały w nim wybrane, ale jego pamięć o ostatnio wybranych liczbach jest wystarczająca do rozpoznawania, kiedy tworzą one ciąg, który go otwiera. Jeszcze prostszym przykładem maszyny skończonej jest długopis z chowanym wkładem. Ta maszyna skończona

ma dwa możliwe stany — wysunięty i wsunięty — i pamięta, czy końcówka została naciśnięta parzystą, czy nieparzystą liczbę razy. Wszystkie maszyny skończone mają ustalony zbiór możliwych stanów, zbiór dopuszczalnych sygnałów wejściowych, zmieniających stan (naciskanie końcówki długopisu lub wybór kombinacji w zamku szyfrowym) i zbiór możliwych sygnałów wyjściowych (chowanie lub wysuwanie wkładu długopisu albo otwieranie zamka). Sygnały wyjściowe zależą jedynie od stanu, który z kolei zależy wyłącznie od tego, jaka była sekwencja sygnałów wejściowych.

Innym prostym przykładem maszyny skończonej jest licznik — na przykład licznik z kołowrotkiem, rejestrujący liczbę ludzi, którzy przeszli przez jakąś bramkę. Za każdym razem, kiedy kolejna osoba przechodzi przez bramkę, stan licznika zwiększa się o jeden. Licznik jest **maszyną skończoną**, ponieważ liczba zliczeń jest ograniczona przez liczbę cyfr w liczniku. Kiedy osiąga stan maksymalny — powiedzmy, 999 — następny krok powoduje zerowanie. Tak właśnie działają liczniki w samochodach. Jechałem raz starą taksówką, w której licznik kilometrów pokazywał 70 000, ale nie wiedziałem, czy przejechała ona 70 000, 170 000 czy 270 000 km, ponieważ taki licznik ma jedynie 100 000 stanów; z jego punktu widzenia wszystkie te możliwości były równoważne. To właśnie powód, dla którego matematycy często definiują stan jako „zbiór równoważnych historii".

Inne dobrze znane przykłady maszyn skończonych to sygnalizatory świetlne na skrzyżowaniach ulic i tablice przycisków sterujących ruchem windy. W urządzeniach tych sekwencja stanów kontrolowa-



RYSUNEK 13.

Schemat stanów dla sygnalizatora ulicznego

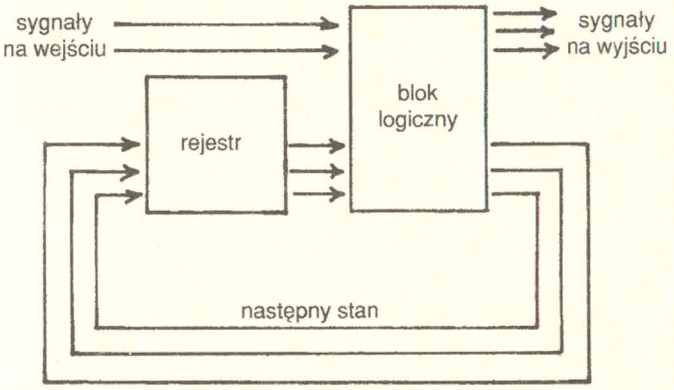
na jest przez pewną kombinację wewnętrznego zegara i przycisków wejściowych, takich jak guzik „przejście" na przejściu dla pieszych czy przyciski wzywające windę lub wybierające piętro, na którym winda ma się zatrzymać. Następny stan urządzenia zależy nie tylko od poprzednich stanów, ale też od sygnałów, które dochodzą z przycisków wejściowych.

Przejścia z jednego stanu do drugiego zachodzą według ustalonego zbioru zasad, który może być przedstawiony za pomocą prostego schemat stanów z zaznaczonymi przejściami pomiędzy nimi. Na rys. 13. pokazano schemat stanów dla sygnalizatora świetlnego na skrzyżowaniu ulic, na którym, po przyciśnięciu przycisku „przejście", światło zmienia się na czerwone dla obu kierunków jazdy. Każdy rysunek sygnalizatora reprezentuje pewien stan, a każda strzałka reprezentuje zmianę stanu. Zmiana zależy od tego, czy przycisk „przejście" został wciśnięty, czy nie.

Aby przechowywać stany maszyny skończonej, musimy wprowadzić ostatni element konstrukcyjny — urządzenie nazywane **rejestrem**, które wykorzystywane jest do przechowywania bitów. Rejestr n-bitowy ma n wejść, n wyjść i dodatkowe wejście, które określa, kiedy powinien zmienić stan. Przechowywanie nowej informacji nazywane jest „zapisywaniem" stanu w rejestrze. Kiedy sygnał kontrolny nakazuje rejestrowi zapisanie nowego stanu, ten zmienia swój stan, aby być w zgodzie z sygnałami na wejściu. Wyjścia rejestru zawsze wskazują jego aktualny stan. Rejestry mogą być realizowane na wiele sposobów; jednym z nich jest użycie układu logiki Boole'a do przesyłania informacji o stanie w pętli. Ten rodzaj rejestru jest często wykorzystywany w komputerach elektronicznych, co tłumaczy, dlaczego zaczynają działać nieprawidłowo, kiedy ich zasilanie zostanie zakłócone.

Maszyna skończona składa się z bloku logiki Boole'a połączonego z rejestrem, co pokazuje rys. 14. Zmienia ona swój stan, zapisując sygnał wyjściowy

bloku logiki Boole'a w rejestrze; blok logiczny wylicza potem stan następny, wynikający z sygnału wejściowego i stanu aktualnego. Ten nowy stan zapisany zostaje w rejestrze w kolejnym cyklu i proces ten powtarza się w każdym cyklu.



RYSUNEK 14.

Maszyna skończona, w której blok logiczny przesyła dane do rejestru

Sposób działania maszyny skończonej określony jest przez tabelę, która pokazuje — dla każdego stanu maszyny i każdego sygnału wejściowego — jaki powinien być następny stan. Na przykład, działanie układu sterującego sygnalizatorem ulicznym może być przedstawione za pomocą zamieszczonej na następnej stronie tabeli.

Pierwszym krokiem przy realizacji maszyny skończonej jest stworzenie takiej tabeli, a następnym — przypisanie każdemu stanowi innego układu bitów. Pięć stanów układu sterującego sygnalizatorem

ulicznym wymagać będzie trzech bitów. (Za pomocą n bitów można zarejestrować 2^n stanów, ponieważ każdy bit podwaja liczbę możliwości.) Zamieniając w konsekwentny sposób każde słowo w powyższej tabeli na układ bitów, możemy przekształcić tabelę w funkcję, która może być zrealizowana za pomocą logiki Boole'a.

przycisk „przejście"	stan układu	światło dla głównej ulicy	światło dla drogi podporządkowanej	następny stan
nie wciśnięty	A	czerwone	zielone	B
nie wciśnięty	B	czerwone	żółte	D
nie wciśnięty	C	żółte	czerwone	A
nie wciśnięty	D	zielone	czerwone	C
nie wciśnięty	„idź!"	„idź!"	„idź!"	D
wciśnięty	A	czerwone	zielone	B
wciśnięty	B	czerwone	żółte	„idź!"
wciśnięty	C	żółte	czerwone	„idź!"
wciśnięty	D	zielone	czerwone	C
wciśnięty	„idź!"	„idź!"	„idź!"	„idź!"

W ulicznych sygnalizatorach świetlnych zapisywanie rejestru sterowane jest przez mechanizm zegarowy, co powoduje zmianę stanu w regularnych odstępach czasu. Innym przykładem maszyny skończonej, zmieniającej stan w regularnych odstępach czasu, jest zegar cyfrowy. Zegar cyfrowy z sekundnikiem może znajdować się w $24 \times 60 \times 60 = 86\,400$ możliwych stanach wyświetlacza — na każdą sekundę dnia przypada jeden stan. Mechanizm odmierzający czas w zegarze powoduje zmiany jego stanu dokładnie raz na sekundę. Wiele innych typów cyfrowych urządzeń liczących, włącznie z większością komputerów uniwersalnych, również zmienia swój stan w regularnych odstępach czasu, a tempo tych zmian

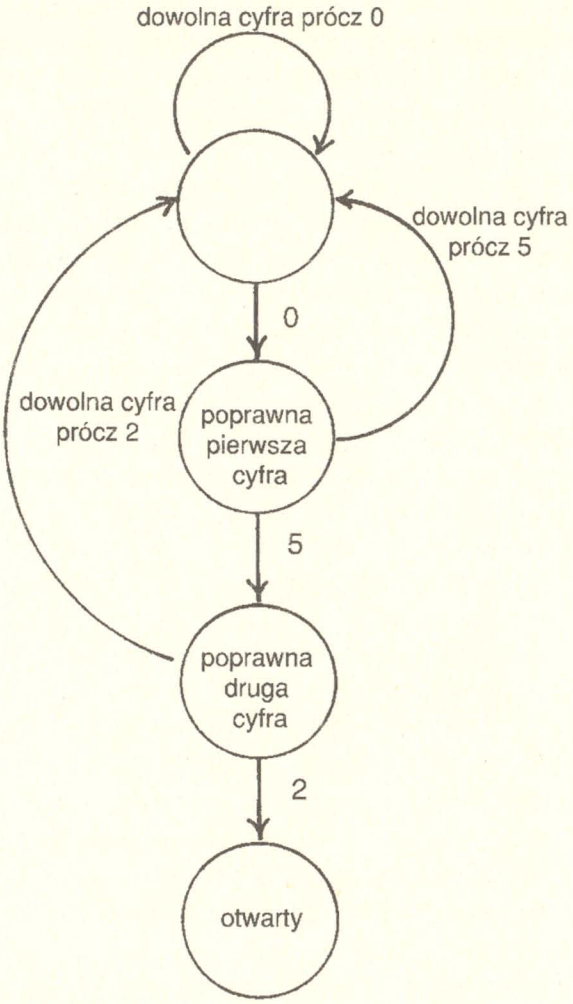
nazywane jest **częstotliwością cyklu zegarowego** danego urządzenia. W komputerze czas nie jest czymś ciągłym, ale ustaloną sekwencją przejść między stanami. Częstotliwość cyklu zegarowego komputera określa szybkość, z jaką przejścia te zachodzą, w ten sposób ustalając związek między czasem fizycznym a komputerowym. Na przykład, częstotliwość cyklu zegarowego laptopa, na którym teraz piszę tę książkę, wynosi 33 megaherce, co oznacza, że zmienia on swój stan w tempie 33 miliony razy na sekundę. Komputer jest szybszy, jeśli częstotliwość cyklu zegarowego jest wyższa, ale jego szybkość jest ograniczona czasem koniecznym do tego, aby informacja przeszła przez bloki logiczne w celu wyliczenia następnego stanu. W miarę jak udoskonalana jest technika, układy logiczne stają się coraz szybsze a częstotliwość cyklu zegarowego ulega zwiększeniu. Kiedy piszę te słowa, mój komputer należy do najnowocześniejszych w swojej klasie, ale w momencie, w którym wy będziecie czytać tę książkę, komputery z zegarem o częstotliwości 33 MHz prawdopodobnie uważane będą za powolne[1]. To jest jeden z cudów technologii krzemowej: w miarę jak uczymy się robić komputery coraz mniejsze i mniejsze, układy logiczne stają się coraz szybsze i szybsze.

Maszyny skończone są tak użyteczne między innymi z tego powodu, że potrafią rozpoznawać sekwencje. Rozważmy zamek szyfrowy, otwierający się jedynie wtedy, gdy wybrana zostanie w nim sekwen-

---

[1] Procesor komputera, na którym redagowany jest ten tekst, taktowany jest zegarem o częstotliwości 400 MHz. W chwili, gdy książka znajdzie się w rękach czytelników, komputery będą na pewno znacznie szybsze — przyp. red.

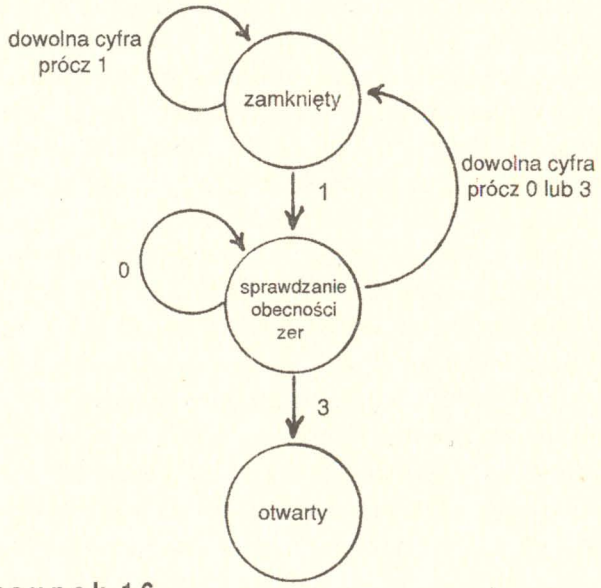
cja 0–5–2. Taki zamek, niezależnie od tego, czy jest mechaniczny, czy elektroniczny, jest maszyną skończoną ze schematem stanów pokazanym na rys. 15.



RYSUNEK 15.

Schemat stanów dla zamka z szyfrem 0–5–2

Można też skonstruować podobne urządzenie do rozpoznawania dowolnego skończonego ciągu. Maszyna skończona może też zostać zaprojektowana tak, że rozpoznaje ciągi wykazujące określone prawidłowości. Na rys. 16. pokazany jest schemat stanów maszyny rozpoznającej ciągi zaczynające się od 1, po której następuje dowolna liczba zer, a potem 3. Zamek z takim mechanizmem otworzy się przy zastosowaniu kombinacji 1–0–3 lub 1–0–0–0–3, ale **nie** kombinacji 1–0–2–3, która nie wykazuje wspomnianej prawidłowości. Bardziej złożone maszyny skończone mogą rozpoznawać bardziej skomplikowane prawidłowości, jak chociażby niepoprawnie napisane słowo w ciągu wyrazów.



Rysunek 16.

Schemat stanów przy rozpoznawaniu ciągów w rodzaju 1–0–3 i 1–0–0–0–3

Choć maszyny skończone mają duże możliwości, to jednak nie są w stanie rozpoznawać dowolnego typu prawidłowości. Na przykład, niemożliwe jest zbudowanie maszyny skończonej, która otwierałaby zamek zawsze wtedy, gdy wprowadzany jest palindrom — czyli ciąg, który wygląda tak samo, jeśli odwróci się w nim kolejność składników, w rodzaju 3–2–1–1–2–3. Wynika to z faktu, że palindromy mogą być dowolnej długości, a na to, by rozpoznać drugą połowę palindromu, należy zapamiętać wszystkie człony pierwszej połowy. Ponieważ takich pierwszych połówek może być nieskończenie wiele, zadanie mogłaby zrealizować tylko maszyna z nieskończoną liczbą stanów.

Podobna argumentacja pomaga wykazać, dlaczego niemożliwe jest zbudowanie maszyny skończonej, która rozpoznawałaby, czy dane zdanie jest poprawne pod względem gramatycznym. Rozważmy proste zdanie: „Psy gryzą". Jego znaczenie można zmienić przez dodanie zdania przydawkowego, na przykład: „Psy, które są drażnione przez ludzi, gryzą". To zdanie z kolei można zmodyfikować, wstawiając kolejny zwrot i będzie ono brzmiało: „Psy, które są drażnione przez ludzi z psami, gryzą". Choć sens takich zdań można wyrazić prościej, i choć stają się one coraz mniej zrozumiałe, to pozostają poprawne pod względem gramatycznym. W zasadzie proces wstawiania zdań podrzędnych może ciągnąć się w nieskończoność, dając w rezultacie takie absurdalne zdania, jak na przykład: „Psy, które drażnione są przez psy, które drażnione są przez psy, które zjadają bity, gryzą". Niemożliwe, aby maszyna skończona mogła rozpoznawać takie zdania jako gramatycznie poprawne.

Dokładnie z tego samego powodu jest to trudne dla ludzi: trzeba mieć dobrą pamięć, że nie pogubić się w tych wszystkich psach. Fakt, że istoty ludzkie mają ze zrozumieniem zdań pewnego typu problemy podobne, jak maszyny skończone, skłonił niektórych do przypuszczeń, że nasze rozumienie języka opiera się na czymś w rodzaju takich maszyn działających w naszych głowach. Jak zobaczymy w następnym rozdziale, istnieją inne rodzaje urządzeń, które w jeszcze bardziej naturalny sposób przystosowane są do rekursywnej struktury ludzkiej gramatyki.

Z maszynami skończonymi zapoznał mnie mój nauczyciel Marvin Minsky. Przedstawił mi słynną zagadkę, nazywaną **problemem plutonu egzekucyjnego**: Jesteś generałem, który dowodzi plutonem żołnierzy ustawionych w bardzo długi szereg. Szereg jest zbyt długi, aby wszyscy żołnierze usłyszeli twój rozkaz „strzelać", musisz więc przekazać swój rozkaz pierwszemu żołnierzowi w szeregu i poprosić go o powtórzenie następnemu z kolei... i tak dalej. Trudność polega na tym, że wszyscy żołnierze w szeregu powinni wystrzelić w tym samym momencie. W tle nieustannie słychać uderzenia bębna. Nie można nawet określić, kiedy żołnierze powinni wystrzelić, ponieważ nie wiadomo, ilu żołnierzy jest w szeregu. Problem można rozwiązać, wydając złożony zbiór rozkazów, które mówią każdemu żołnierzowi, co ma powiedzieć swoim sąsiadom. W tym zagadnieniu żołnierze stanowią odpowiednik szeregu maszyn skończonych, z których wszystkie zmieniają swoje stany w tym samym tempie (uderzenia w bęben) i każda otrzymuje sygnał wejściowy z wyjścia swoich bezpośrednich sąsiadów. Rozwiązanie zagadki spro-

wadza się do zaprojektowania szeregu identycznych maszyn skończonych, które w odpowiedzi na komendę podaną na końcu szeregu wytworzą jednocześnie na wyjściu sygnał „strzelać". (Maszyny skończone na obu końcach szeregu mogą różnić się od pozostałych.) Nie chciałbym zepsuć tej zagadki przez ujawnianie rozwiązania — powiem tylko, że można ją rozwiązać za pomocą maszyn skończonych, mających jedynie niewielką liczbę stanów.

Zanim pokażę, w jaki sposób za pomocą logiki Boole'a i maszyn skończonych można stworzyć komputer, wybiegnę trochę do przodu i powiem wam, dokąd zmierzamy. Następny rozdział zaczyna się od przedstawienia najwyższego stopnia abstrakcji w funkcjonowaniu komputera, będącego jednocześnie tym poziomem, na którym oddziałuje z maszyną większość programistów.

Rozdział trzeci

PROGRAMOWANIE

Magia komputera polega na jego zdolności stawania się niemal wszystkim, co możemy sobie wyobrazić, jeśli tylko potrafimy dokładnie wyjaśnić, czym to ma być. Problem polega na wyjaśnieniu, czego oczekujemy. Dobrze zaprogramowany komputer może pełnić rolę teatru, instrumentu muzycznego, encyklopedii lub przeciwnika w grze w szachy. Żaden twór na świecie, z wyjątkiem człowieka, nie ma tak uniwersalnej i zdolnej do dostosowania się natury. Ostatecznie wszystkie te funkcje realizowane są za pomocą bloków logiki Boole'a i maszyn skończonych, opisanych w poprzednim rozdziale, ale programista rzadko myśli o tych elementach; programiści posługują się bardziej dogodnym narzędziem zwanym **językiem programowania.**

Tak jak logika Boole'a i maszyny skończone są podstawowymi elementami konstrukcyjnymi komputera, tak język programowania jest zbiorem elementów konstrukcyjnych, służących do tworzenia oprogramowania komputerowego. Język programowania — podobnie jak języki ludzkie — ma swój

słownik i gramatykę, ale — w odróżnieniu od dowolnego ludzkiego języka — każde słowo i zdanie w języku programowania ma ściśle określone znaczenie. Podobnie jak logika Boole'a, większość języków programowania jest uniwersalna; mogą być używane do opisania wszystkiego, co komputer może zrobić. Każdy, kto kiedykolwiek pisał program — lub usuwał błędy z programu — wie o tym, że powiedzenie komputerowi, co ma zrobić, wcale nie jest takie proste. Każdy szczegół oczekiwanej operacji musi być szczegółowo opisany. Na przykład, jeśli każesz programowi prowadzącemu księgowość obciążyć twoich klientów kwotą, jaką każdy z nich jest winien, to komputer wyśle tygodniowy rachunek na kwotę 0,00 złotych do klientów, którzy nic nie są winni. Jeśli polecisz wysłać list z ostrzeżeniem do klientów, którzy nie zapłacili, to ludzie, którzy nie są nic winni otrzymywać będą listy z ponagleniami dotąd, aż dokonają wpłaty w wysokości 0,00 złotych. Unikanie tego rodzaju nieporozumień to właśnie to, o co chodzi w programowaniu komputerowym. Sztuka programowania to sztuka mówienia, czego dokładnie chcesz. W tym przykładzie oznacza to dokonanie rozróżnienia między klientami, którzy nie przysłali pieniędzy, a klientami, którzy rzeczywiście są coś winni. Parafrazując Marka Twaina można powiedzieć, że różnica między dobrym programem a prawie dobrym programem jest taka, jak między światłem błyskawicy a świetlikiem.

Wprawny programista jest jak poeta — potrafi wyrazić słowami idee, które innym wydają się niemożliwe do wyrażenia. Jeśli jesteś poetą, zakładasz pewien zasób wspólnej wiedzy i doświadczeń u swo-

ich czytelników. Wiedzą i doświadczeniem, jakie programista i komputer mają wspólne, jest znaczenie języka programowania. Skąd komputer „wie", jakie jest znaczenie języka programowania, napiszę później; najpierw omówimy gramatykę, słownictwo i idiomy takich języków.

ROZMOWA Z KOMPUTEREM

Jest wiele różnych języków programowania. Ta różnorodność wynika z historii, przyzwyczajeń, upodobań, ale też i z tego, że różne języki programowania zostały stworzone do opisywania innych rzeczy. Każdy język ma swoją własną składnię. Trzeba ją poznać, aby móc pisać w określonym języku, ale — tak jak ortografia czy interpunkcja w językach naturalnych — składnia nie jest najważniejsza, jeśli chodzi o znaczenie czy możliwości opisowe języka. O możliwościach opisowych decyduje słownictwo — tak zwane wyrażenia pierwotne danego języka — i sposób, w jaki wyrażenia pierwotne mogą być łączone w celu definiowania nowych pojęć.

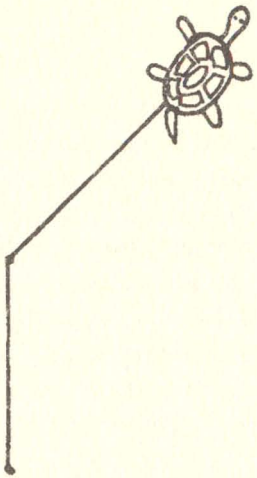
Języki programowania opisują operacje przeprowadzane na danych — tym, co je odróżnia, jest rodzaj danych jakimi mogą operować. Pierwsze języki komputerowe zostały zaprojektowane głównie do operowania liczbami i ciągami znaków. Współczesne języki programowania mogą operować słowami, obrazami, dźwiękami, a nawet innymi programami komputerowymi. Niezależnie jednak od tego, do pracy z jakim typem danych zaprojektowany został dany język komputerowy, zwykle umożliwia on wczytywanie danych do komputera, dzielenie ich na części, łącze-

nie, modyfikowanie ich, porównywanie i nadawanie im nazw.

Przypuszczalnie łatwiej nam będzie, jeśli zilustrujemy te abstrakcyjne rozważania opisem jakiegoś konkretnego języka komputerowego — wybrałem w tym celu Logo, zaprojektowany przez nauczyciela i matematyka Seymoura Paperta jako język komputerowy dla dzieci. Dzieci mogą w Logo pisać programy, potrafiące tworzyć i przetwarzać obrazy, słowa, liczby i dźwięki. Choć język ten jest na tyle prosty, że może go używać dziesięciolatek, dysponuje wieloma możliwościami najbardziej wyrafinowanych języków komputerowych, włącznie z możliwością pisania programów przetwarzających inne programy. Jest też językiem **rozszerzalnym** — to znaczy, można używać Logo do definiowania nowych słów w Logo.

Jednym z najprostszych programów, jakie można napisać w Logo, jest procedura do tworzenia rysunków. Realizujemy ją poprzez wydawanie rozkazów myślowemu żółwiowi, który żyje na ekranie. Żółw pełni funkcję pióra — porusza się po ekranie i pozostawia za sobą linie. Kiedy komputer zostaje włączony, żółw pokazuje się na środku ekranu, zwrócony głową do góry. Jeśli dziecko wystuka komendę FORWARD 10, to żółw wykona dziesięć kroków do przodu — czyli do góry — rysując linię o długości dziesięciu jednostek. Liczba 10, następująca po komendzie FORWARD, nazywana jest **parametrem**: w tym przypadku parametr mówi żółwiowi, ile kroków ma wykonać. Aby narysować linę w innym kierunku, dziecko musi obrócić żółwia. Komenda RIGHT 45 spowoduje obrócenie żółwia o 45 stopni w prawo (następny parametr) od jego ostatniej orientacji. Na-

stępna komenda FORWARD, z odpowiednim parametrem, spowoduje narysowanie linii w nowym kierunku.



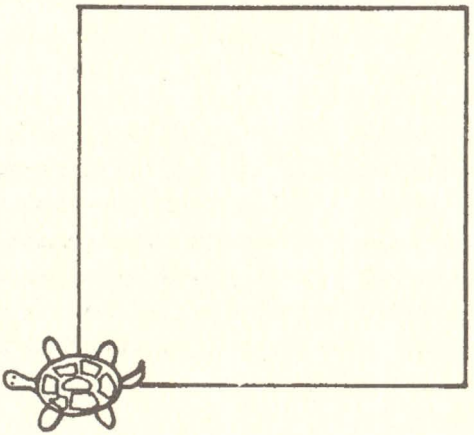
RYSUNEK 17.
FORWARD 10, RIGHT 45, FORWARD 10

Za pomocą takich komend jak FORWARD, BACKWARD, RIGHT i LEFT dziecko może przesuwać żółwia na ekranie i tworzyć rysunki, wymaga to jednak wiele pisaniny i szybko staje się męczące. Co jednak czyni ten język atrakcyjnym, to możliwość definiowania nowych słów: oto przykład, jak dziecko może nauczyć żółwia (zaprogramować komputer), jak rysuje się kwadrat:

```
TO KWADRAT
FORWARD 10
RIGHT 90
```

```
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
END
```

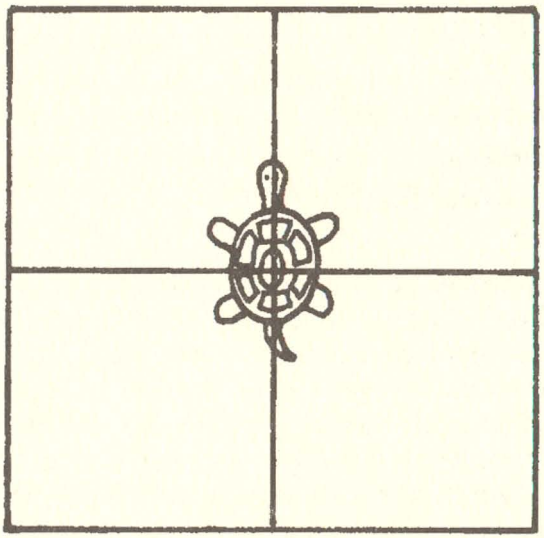
Zdefiniowawszy słowo „kwadrat", dziecko może następnie narysować kwadrat o boku długości dziesięciu jednostek po prostu wystukując nową komendę: KWADRAT. (Nie trzeba chyba podkreślać, że nazwa „kwadrat" jest całkowicie arbitralna; dziecko mogłoby równie dobrze nazwać procedurę RAMKA lub XYZ, a żółw dalej wykonywałby dokładnie to samo. Kiedy dzieci to odkrywają, często sprawia im przyjemność „oszukiwanie" komputera — na przykład nazywanie procedury rysującej kwadrat TRÓJKĄT, lub na odwrót.)



RYSUNEK 18.
Kwadrat

Kiedy już słowo „kwadrat" zostało zdefiniowane, to staje się częścią słownika komputera i może być następnie używane do definiowania innych słów. Oto przykład:

```
TO OKNO
KWADRAT
KWADRAT
KWADRAT
KWADRAT
END
```



RYSUNEK 19.

Okno utworzone z czterech kwadratów

Każdy kwadrat zostanie narysowany w innym miejscu, ponieważ procedura rysująca kwadrat pozostawia żółwia obróconego o 90 stopni. W termino-

logii komputerowej, KWADRAT jest **podprogramem** programu OKNO, który go **wywołuje**. Podprogram KWADRAT z kolei zdefiniowany jest za pomocą wyrażeń pierwotnych FORWARD i RIGHT. Zdefiniowane przez użytkownika słowa w Logo mogą także zawierać parametry. Na przykład, dziecko może określić wielkość kwadratu podając parametr, który określa długość każdego boku.

```
TO KWADRAT :BOK
FORWARD :BOK
RIGHT 90
FORWARD :BOK
RIGHT 90
FORWARD :BOK
RIGHT 90
FORWARD :BOK
END
```

Dwukropek przed komendą BOK jest przykładem składni. W Logo dwukropek oznacza, że słowo po nim następujące jest nazwą parametru, przedstawiającego coś innego — w tym przypadku liczbą, która ma być podawana przy każdym „wywołaniu" podprogramu KWADRAT. Kiedy KWADRAT jest zdefiniowany w ten sposób, to komenda KWADRAT 15 nakaże komputerowi narysowanie kwadratu, którego bok będzie miał długość piętnastu jednostek. Nazwa BOK, określająca parametr, jest również i w tym przypadku nazwą dowolną i ma znaczenie jedynie wewnątrz definicji procedury KWADRAT; jeśli wszystkie pięć powtórzeń słowa BOK zamieniono by na, powiedzmy, X, to podprogram działałby dokładnie w ten sam sposób.

Istnieją inne sposoby pisania podprogramów rysujących kwadrat. Można na przykład nakazać żółwiowi obrót w lewo cztery razy lub pójście w przeciwną stronę cztery razy. Ciekawe jest, że nie ma znaczenia, jak zdefiniowany jest KWADRAT; istotne jest jedynie, co ten podprogram rysuje i w którym miejscu pozostawia żółwia. Inne programy mogą wywoływać KWADRAT niezależnie od tego, jak jest zdefiniowany i czy jest słowem zdefiniowanym przez użytkownika, czy też wyrażeniem pierwotnym języka. Rozszerzając język, programista wykorzystuje potęgę abstrakcji funkcjonalnej w celu stworzenia nowych elementów konstrukcyjnych.

Jedna ze sztuczek, jakie odkrywają dzieci używające Logo, polega na wstawianiu jakiegoś słowa do jego własnej definicji; taki sposób postępowania nazywa się **rekurencją**. Oto przykład, jak dziecko może wytworzyć kolisty wzór, składający się z wielu obróconych kwadratów:

```
TO WZÓR
KWADRAT
RIGHT 10
WZÓR
END
```

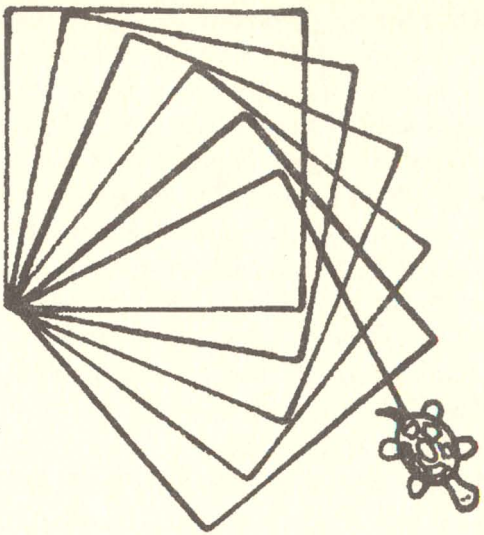
Komputer realizuje komendę WZÓR rysując kwadrat, następnie obracając żółwia o 10 stopni i rysując kolejny element w ten sam sposób. W tym przypadku, rekurencyjna definicja wzoru ma pewną wadę: rekurencja ciągnie się w nieskończoność. Za każdym razem, kiedy komputer zaczyna realizować komendę WZÓR, rysuje kwadrat, a potem przechodzi do następnego wzoru, i tak *ad infinitum*. Przypomina to przypowieść o pewnym mędrcu, który twier-

dził, że Ziemia spoczywa na grzbiecie olbrzymiego żółwia. „A na czym stoi ten żółw?" — spytał uczeń. „Na innym żółwiu" — odparł mędrzec. „A na czym z kolei stoi ten żółw?" — zapytał znów uczeń, stając się coraz bardziej sceptyczny. „Nie ma sensu o to pytać — rzekł mędrzec. — Aż do samego końca są żółwie."

Komputer rysując deseń przechodzi przez ten sam proces, co istoty ludzkie, kiedy próbują wyobrazić sobie nieskończoną piramidę olbrzymich żółwi, ale komputer nie jest na tyle mądry, aby zauważyć, że zmierza donikąd. Nie zatrzyma się, dopóki mu się nie przerwie — przykład częstego zachowania programów, nazywanego **nieskończoną pętlą**. Programiści często powodują powstanie nieskończonych pętli w sposób przypadkowy; jak zobaczymy, przewidzenie, czy takie pętle powstaną, może być bardzo trudne. Tej konkretnej nieskończonej pętli łatwo uniknąć przez napisanie programu z parametrem określającym, ile kwadratów ma zostać narysowanych.

```
TO WZÓR :LICZBA
KWADRAT
WZÓR 10
IF :LICZBA = 1 STOP ELSE WZÓR :LICZBA-1
END
```

Tak zdefiniowany podprogram WZÓR wykonywać będzie jedną z dwóch rzeczy, zależnie od tego, czy parametrem jest 1 czy jakaś większa liczba. WZÓR 1 narysuje tylko jeden kwadrat, ale na przykład WZÓR 5 narysuje kwadrat, wykona obrót, a potem narysuje WZÓR 4, która narysuje kwadrat i potem wykona WZÓR 3, aż dojdzie do WZÓR 1, kiedy to narysuje kwadrat i się zatrzyma.



RYSUNEK 20.
Rysowanie wzoru

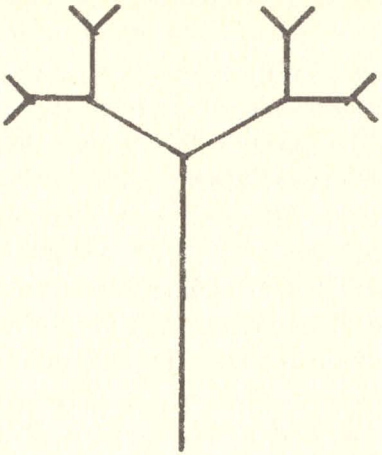
Ten rodzaj definicji rekurencyjnej ze zmiennym parametrem jest przydatny przy tworzeniu czegokolwiek, co ma strukturę samopodobną. Przykładem struktury rekurencyjnej samopodobnej jest rysunek, który jako element zawiera swój obraz; takie struktury znane są powszechnie jako **fraktale**. W realnym świecie struktury samopodobne nie są nieskończone: na przykład każda gałąź drzewa wygląda w podobnie jak małe drzewo, a każda z mniejszych gałęzi ma odgałęzienia, które wyglądają jak jeszcze mniejsze drzewa. Ta rekurencja przebiega przez kilka poziomów, w końcu jednak odgałęzienia stają się tak małe, że nie mają już dalszych odgałęzień.

Poniżej pokazany jest rekurencyjny program rysujący w Logo drzewo. Być może da to wam pewne pojęcie o tym, jak program komputerowy może zawierać elementy poezji — choć w tym przypadku główny motyw przesłaniają szczegóły dotyczące położenia żółwia i sprowadzania go z powrotem do punktu wyjściowego. A oto w dużym przybliżeniu to, co wyraża program: „Duże drzewo to patyk z dwoma mniejszymi drzewami na czubku, ale małe drzewo to tylko patyk". Rysunek stworzony przez program do rysowania drzewa pokazany jest poniżej.

```
TO DRZEWO :ROZMIAR
FORWARD :ROZMIAR
IF :ROZMIAR<1 STOP ELSE DWA-DRZEWA
ROZMIAR/2
BACK :ROZMIAR
END

TO DWA-DRZEWA :ROZMIAR
LEFT 45
DRZEWO :ROZMIAR
RIGHT 90
DRZEWO :ROZMIAR
LEFT 45
END
```

Okazuje się, że ta technika definiowania rzeczy w sposób rekurencyjny ma bardzo duże możliwości. Wiele rodzajów danych, którymi lubimy operować — w szczególności same programy komputerowe — ma strukturę rekurencyjną. Definicje rekurencyjne są bardzo wygodne przy specyfikowaniu operacji na danych o strukturze rekurencyjnej. Typowa definicja



RYSUNEK 21.
Drzewo

rekurencyjna składa się z dwóch części — pierwsza opisuje, co stanie się w szczególnym prostym przypadku, druga zaś, jak bardziej złożony przypadek może zostać zredukowany do czegoś prostszego. W rekurencyjnym drzewie, na przykład, prostym przypadkiem jest drzewo mniejsze od 1, a przypadkiem bardziej złożonym jest drzewo składające się z pnia i dwóch mniejszych drzew.

Kolejnym przykładem jest definicja palindromu, którą można sformułować w następujący sposób: słowo jest palindromem, jeśli ma mniej niż dwie litery (prosty przypadek) lub gdy jego pierwsza litera i ostatnia litera są takie same, a litery między nimi tworzą palindrom (krok rekurencyjny). Najprostszym sposobem na napisanie programu w Logo, któ-

ry rozpoznawałby palindromy, byłoby użycie tej definicji.

Istnieje wiele innych języków komputerowych: LISP, Ada, FORTRAN, C, ALGOL i inne; większość nazw to mało przejrzyste skróty (np. FORTRAN to skrót od *FORmula TRANslation*, LISP to skrót od *LISt Processing*). Choć języki te różnią się od Logo szczegółami słownika i składni, wszystkie mogą opisywać procedury tego samego rodzaju. Niektóre, jak na przykład FORTRAN, mają ograniczone możliwości definiowania operacji w sposób rekurencyjny lub manipulowania danymi nie mającymi charakteru numerycznego. Inne, jak C czy LISP, pozwalają programiście bezpośrednio manipulować bitami przedstawiającymi dane, co daje większe możliwości — i więcej okazji do popełnienia błędu. Na przykład w C jest zupełnie możliwe mnożenie dwóch znaków alfabetu; wynik tej nonsensownej operacji zależeć będzie od binarnej reprezentacji używanej przez maszynę. Języki takie jak LISP oferują zarówno funkcje abstrakcyjne, jak i funkcje niskiego stopnia. Mój przyjaciel, informatyk Guy Steele, wyraził to kiedyś w ten sposób: „LISP jest językiem wysokiego poziomu, ale ciągle czuć bity prześlizgujące się między palcami".

Ostatnio zaczęły się pojawiać języki nowej generacji. Języki te — Small-Talk, C++, Java — to tak zwane **języki obiektowe**. Traktują one strukturę danych — na przykład rysunek, jaki ma być narysowany na ekranie — jako „obiekt", który posiada swój własny stan wewnętrzny, na przykład to, gdzie ma być narysowany lub jakiego jest koloru. Obiekty te mogą otrzymywać rozkazy od innych obiektów. Aby

zrozumieć, dlaczego jest to użyteczne, wyobraźmy sobie, że piszemy program dla gry wideo, w której występują skaczące piłki. Każda piłka na ekranie zdefiniowana jest jako odrębny obiekt. Pogram określa reguły zachowania, które mówią obiektowi, jak ma być odwzorowany na ekranie, jak ma się poruszać i oddziaływać z innymi obiektami w tej grze. Każda piłka zachowywać się będzie w podobny sposób, ale każda będzie w nieco innym stanie, ponieważ będzie zajmowała inne położenie na ekranie i będzie miała charakterystyczny dla siebie kolor, prędkość, rozmiary itd.

Największą zaletą obiektowych języków programowania jest to, że obiekty — na przykład rozmaite obiekty w grze wideo — mogą być definiowane niezależnie od siebie i następnie łączone, w celu tworzenia nowych programów. Czasami można odnieść wrażenie, że pisanie nowego programu obiektowego jest jak wpuszczenie do klatki gromady zwierząt i przyglądanie się, co z tego wyniknie. Zachowanie programu **wyłania się** właśnie jako skutek oddziaływań zaprogramowanych między obiektami. Z tego powodu, jak też ze względu na to, że języki obiektowe pojawiły się dość niedawno, należałoby się dobrze zastanowić, zanim napisze się w takim języku program do pilotowania samolotu.

Uczenie się języka programowania nie jest tak trudne, jak uczenie się języka naturalnego. Ogólnie biorąc, jeśli znasz już dwa albo trzy języki, kolejnego możesz nauczyć się w kilka godzin, ponieważ składnia jest względnie prosta, a słowniki rzadko liczą więcej niż kilkaset słów. Ale, tak jak w przypadku języków naturalnych, istnieje duża różnica między

posiadaniem zdolności rozumienia języka, a umiejętnością posługiwania się nim. Każdy język komputerowy ma swoich Szekspirów; czytanie stworzonych przez nich programów jest wielką przyjemnością. Dobrze napisany program komputerowy ma swój styl, swoją finezję, a nawet humor — i przejrzystość, która może konkurować z najlepszą prozą.

JAK SIĘ TO WSZYSTKO ZE SOBĄ WIĄŻE?

W jaki sposób maszyny skończone mogą zostać użyte do wykonywania rozkazów napisanych w takim języku jak Logo? Aby odpowiedzieć na to pytanie, wróćmy do bardziej szczegółowego poziomu naszej dyskusji, wiążącego się z logiką Boole'a. Przejście od maszyn skończonych do Logo wymaga trzech zasadniczych kroków: trzeba wiedzieć po pierwsze, w jaki sposób można rozbudować maszyny skończone dodając im **pamięć** (czyli urządzenie pozwalające przechowywać definicje tego, co kazano im zrobić); po drugie, jak te rozbudowane maszyny mogą wykonywać rozkazy zapisane w **języku maszynowym** — prostym języku, który określa działania maszyny; po trzecie wreszcie, jak język maszynowy może dać maszynie rozkaz, aby zinterpretowała język programowania — na przykład Logo. W dalszej części rozdziału opiszę dość dokładnie, jak to wszystko działa — jest tam znacznie więcej szczegółów, niż potrzeba do zrozumienia pozostałej części tej książki. Czytelnik nie powinien czuć się zmuszony do zrozumienia każdego kroku. Ważną rzeczą jest zrozumienie, jak stopnie abstrakcji funkcjonalnej opierają się jeden

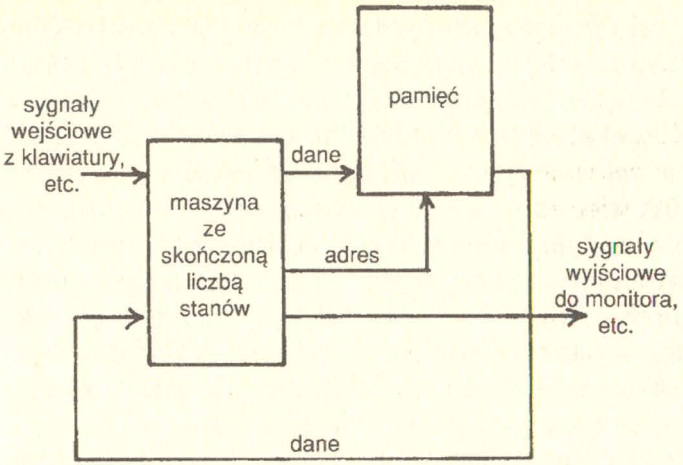
na drugim, co podsumuję w ostatnim paragrafie tego rozdziału.

Komputer to po prostu specjalny rodzaj maszyny skończonej, podłączonej do pamięci. Pamięć komputera — w rzeczywistości zestaw szufladek do przechowywania danych — zbudowana jest z rejestrów, podobnych do rejestrów, które przechowują stany maszyny skończonej. Każdy rejestr przechowuje układ bitów nazywany **słowem**, który może być przeczytany (lub zapisany) przez maszynę skończoną. Liczba bitów w słowie zmienia się w zależności od komputera, ale we współczesnych mikroprocesorach (w chwili, kiedy to piszę) wynosi zwykle osiem, szesnaście lub trzydzieści dwa bity. (Wielkość słowa będzie prawdopodobnie rosnąć wraz z postępem techniki.) Typowa pamięć ma miliony, a nawet miliardy takich rejestrów, z których każdy zawiera tylko jedno słowo. W danej chwili maszyna ma dostęp tylko do jednego rejestru — oznacza to, że w każdym cyklu maszyny skończonej dane mogą zostać przeczytane lub zapisane tylko w jednym rejestrze pamięci. Każdy rejestr w pamięci ma inny **adres** — czyli układ bitów, za pomocą którego można uzyskać do niego dostęp — tak więc rejestry określane są jako **lokalizacje w pamięci**. Pamięć zawiera bloki logiki Boole'a, odczytujące adres i wybierające miejsce, w którym informacja ma zostać odczytana lub zapisana. Jeśli w tym miejscu w pamięci dane mają zostać zapisane, to bloki logiczne przesyłają nowe dane do wybranego rejestru. Jeśli dane z tego rejestru mają zostać odczytane, to bloki logiczne kierują dane z wybranego rejestru do wyjścia z pamięci, które jest połączone z wejściem do maszyny skończonej.

Niektóre ze słów zapisanych w pamięci reprezentują dane, na których mają zostać wykonane operacje, takie jak liczby i litery. Inne przedstawiają **rozkazy**, mówiące maszynie, jaką sekwencję operacji ma wykonać. Rozkazy zapisane są w języku maszynowym, który, jak zaznaczyliśmy, jest znacznie prostszy od typowego języka programowania. Język maszynowy interpretowany jest bezpośrednio przez maszynę skończoną. W tym typie komputerów, które opiszemy, każdy rozkaz języka maszynowego jest przechowywany w pojedynczym słowie pamięci, a sekwencja rozkazów przechowywana jest w bloku kolejno ponumerowanych lokalizacji w pamięci. Te sekwencje rozkazów w języku maszynowym są najprostszym rodzajem programu w komputerze.

Maszyna skończona wielokrotnie wykonuje następujący ciąg operacji: (1) **przeczytaj** rozkaz z pamięci, (2) **wykonaj** operację opisaną przez ten rozkaz i (3) **oblicz** adres następnego rozkazu. Sekwencja stanów niezbędnych do wykonania tego zadania wbudowana jest w logikę Boole'a maszyny, a same rozkazy są szczególnymi układami bitów — układami, które powodują, że maszyna skończona wykonuje rozmaite operacje na danych z pamięci. Na przykład rozkaz DODAJ jest jednoznacznym układem bitów określających, które dwa rejestry w pamięci mają zostać zsumowane. Po rozpoznaniu tego układu maszyna skończona przejdzie przez sekwencję stanów, dzięki czemu przeczyta dane z tych lokalizacji w pamięci, które mają zostać dodane, doda liczby do siebie i zapisze sumę z powrotem w pamięci.

W większości komputerów stosuje się dwa podstawowe typy rozkazów: rozkazy przetwarzania i roz-



RYSUNEK 22.

Maszyna skończona połączona z pamięcią

kazy sterujące. **Rozkazy przetwarzania** przesyłają dane do pamięci, pobierają dane z pamięci lub łączą dane w celu wykonywania operacji arytmetycznych i logicznych. Rozkazy przetwarzania określają też adresy lokalizacji w pamięci, czyli rejestry. Zwykle rozkazy te dotyczą bezpośrednio jedynie kilku rejestrów; inne rejestry zaangażowane są w sposób pośredni, ponieważ ich adresy są przechowywane w rejestrach. Na przykład rozkaz PRZESUŃ może przesuwać dane z rejestru 1 do adresu określonego w rejestrze 2. Jeśli rejestr 2 zawiera układ bitów odpowiadający liczbie 1234, to dane zostaną przesunięte do rejestru 1234. Inne rozkazy przetwarzania łączą dane z różnych rejestrów pamięci. Istnieją również rozkazy wykonujące na układach bitów w rejestrze funkcje Boole'a — AND, OR i NOT.

Rozkazy sterujące określają adres następnego rozkazu, który ma zostać pobrany; adres ten przechowywany jest w specjalnym rejestrze nazywanym **licznikiem rozkazów**. Zwykle rozkazy pobierane są sekwencyjnie z kolejnych lokalizacji w pamięci, tak więc adres w liczniku rozkazów zwiększa się o 1 po każdym kolejnym rozkazie. Rozkazy sterujące pozwalają na wprowadzenie do licznika rozkazów innej liczby, wpływając w ten sposób na sekwencję, jaka zostanie zrealizowana. Najprostszym rozkazem sterującym jest rozkaz JUMP („skocz"), przechowujący w liczniku rozkazów adres, z którego ma zostać pobrany następny rozkaz. Pewną odmianą rozkazu JUMP jest warunkowy rozkaz JUMP, wprowadzający do licznika rozkazów nowy adres jedynie wtedy, gdy spełniony jest określony warunek — na przykład układ bitów w dwóch rejestrach jest taki sam. Jeśli warunek nie został spełniony, to warunkowy rozkaz JUMP nie zadziała i zostanie pobrany następny rozkaz.

Jeśli zachodzi potrzeba, aby ten sam ciąg rozkazów został wykonany kilkakrotnie, raz za razem, to można umieścić warunkowy rozkaz JUMP na końcu sekwencji, aby przestawić licznik rozkazów do stanu początkowego tyle razy, ile to jest konieczne. Taka operacja nazywana jest **pętlą** — poznaliśmy już przykład pętli przy opisie programowania w Logo. Wykonywanie sekwencji będzie powtarzane dotąd, aż warunek, od którego zależy skok, nie będzie już dłużej spełniony. Jeśli zestaw rozkazów ma być powtarzany, powiedzmy, dziesięć razy, to jeden z rejestrów pamięci może zostać użyty do liczenia liczby iteracji pętli.

Zbiór rozkazów rozpoznawanych przez maszynę zmienia się w zależności od komputera. Projektanci komputerów mogą (i robią to) latami dyskutować o tym, jaki jest optymalny zestaw rozkazów. Typowym przykładem jest kontrowersja co do zalet i wad komputerów ze zredukowaną listą rozkazów (RISC — od ang. *reduced instruction set computer*), które używają prostego, minimalnego zbioru rozkazów, w porównaniu z komputerami z rozbudowaną listą rozkazów (CISC — od ang. *complex instruction set computer*), korzystającymi z bogatej, złożonej listy rozkazów o dużych możliwościach. Dyskusja ta ma jednak dla programistów niewielkie znaczenie, ponieważ każdy rozsądny zbiór rozkazów może symulować dowolny inny zbiór. Historia pokazuje, że sukces handlowy tego czy innego typu komputerów nie ma praktycznie nic wspólnego ze złożonością zastosowanego zbioru rozkazów lub jakimkolwiek innym szczegółem wewnętrznej konstrukcji. W istocie, niektóre z komputerów cieszących się największym powodzeniem — chociażby mikroprocesory używane w komputerach osobistych — mają według przekonania większości projektantów kiepsko zaprojektowane zbiory rozkazów. Najwyraźniej szczegóły konstrukcyjne maszyny mają niewielkie znaczenie dla jej użytkowników.

Stopień złożoności zbioru rozkazów komputera nie ma wielkiego znaczenia między innymi z powodu istnienia podprogramów. Podprogramy umożliwiają wielokrotne odwoływanie się do pewnych sekwencji rozkazów z wielu miejsc w programie. Możliwość ich wywoływania pozwala programiście na definiowanie nowych rozkazów za pomocą sekwencji innych

rozkazów — program główny wywołuje podprogram, wprowadzając adres podprogramu do licznika rozkazów za pomocą rozkazu JUMP; zanim jednak komputer tego dokona, zachowuje uprzednią zawartość licznika rozkazów w specjalnej lokalizacji w pamięci. Na końcu podprogramu inny rozkaz odczytuje ten adres powrotny i przeskakuje z powrotem do lokalizacji, z której podprogram został wywołany.

Ten proces wywoływania podprogramów może mieć charakter rekurencyjny — ciągi rozkazów w podprogramach mogą zawierać przeskoki do innych podprogramów, a te z kolei do jeszcze innych, itd. W rekurencyjnie zdefiniowanej funkcji podprogram może nawet wywoływać sam siebie. W celu śledzenia tych zagnieżdżonych podprogramów komputer potrzebuje systematycznego sposobu przechowywania ich adresów zwrotnych, aby było wiadome, w które miejsce należy powrócić po zakończeniu każdego z podprogramów. Nie może po prostu przechowywać ich wszystkich w tym samym miejscu, ponieważ podprogramy są zagnieżdżone i komputer musi pamiętać więcej niż jeden adres powrotny. Zwykle komputer przechowuje adresy powrotne w sekwencyjnym zbiorze lokalizacji zwanym **stosem**. Ostatni adres powrotny przechowywany jest na „szczycie stosu". Stos w pamięci działa tak jak stos talerzy: przedmioty są zawsze dodawane i usuwane ze szczytu. System przechowywania — „ostatni-włożony/pierwszy-wyjęty" bardzo dobrze nadaje się do przechowywania adresów powrotnych zagnieżdżonych podprogramów, ponieważ podprogram nigdy nie jest zakończony, dopóki wszystkie zagnieżdżone w nim podprogramy nie zostaną zakończone.

Niektóre podprogramy są tak użyteczne, że są zawsze wprowadzane do komputera. Ten zbiór podprogramów nazywa się **systemem operacyjnym**. System operacyjny zawiera między innymi podprogramy wpisujące lub odczytujące znaki wystukiwane na klawiaturze, rysujące linie na ekranie lub w inny sposób oddziałujące z użytkownikiem. To system operacyjny komputera odpowiada w przeważającej mierze za to, jak komputer wygląda i jakie budzi odczucia w użytkowniku. Kieruje on również interfejsem między komputerem a dowolnym wykonywanym programem, ponieważ podprogramy systemu operacyjnego wyposażają program w zbiór operacji bardziej złożonych niż rozkazy języka maszynowego.

Programista nie musi wiedzieć, czy funkcje realizowane są przez układy logiczne komputera, czy przez programy systemu operacyjnego, póki ten sam układ bitów wywołuje taki sam efekt. Ten sam program, działający na dwóch rożnych typach komputerów, może w jednym przypadku dokonywać operacji arytmetycznych w układach logicznych komputera, a w drugim wykonywać je poprzez podprogram zawarty w systemie operacyjnym. Podobnie, system operacyjny jednego typu komputerów może pozwalać na emulowanie całego zbioru rozkazów na innym typie komputera. Producenci komputerów czasami wykorzystują takie emulacje, aby spowodować, że nowsze modele zachowują się jak modele wcześniejsze, dzięki czemu stare oprogramowanie może być używane bez modyfikacji.

System operacyjny zawiera zwykle wszystkie podprogramy, które wykonują operacje wejścia i wyjścia, czyli operacje pozwalające programowi na oddziały-

wanie ze światem zewnętrznym. To oddziaływanie jest realizowane przez połączenie pewnych lokalizacji w pamięci komputera z urządzeniami wejściowymi, takimi jak klawiatura lub mysz, i urządzeniami wyjściowymi, jak ekran monitora. Na przykład klawisz spacji na klawiaturze może być podłączony do rejestru 23 w pamięci, tak że dane odczytane pod adresem 23 to 1, jeśli klawisz spacji jest wciśnięty, i 0 w przeciwnym przypadku. Inny rejestr w pamięci może sterować kolorem wyświetlanym w pewnym punkcie na ekranie. Jeśli każda kropka na ekranie pokazuje dane przechowywane w innej lokalizacji w pamięci, to komputer może narysować na ekranie dowolny wzór, zapisując po prostu odpowiedni wzór w pamięci.

Poza urządzeniami wejścia/wyjścia opisany przez nas komputer jest po prostu maszyną skończoną podłączoną do pamięci. Oba te elementy mogą zostać w całości zbudowane z rejestrów i bloków logiki Boole'a, za pomocą technik opisanych w rozdziałach 1. i 2. Maszyna skończona, sterująca komputerem, jest skomplikowana, ale jeśli chodzi o zasadę działania nie różni się od urządzenia, które steruje ulicznym sygnalizatorem. Projektowanie takiej maszyny to po prostu sprawa prześledzenia szczegółów — danych do zapamiętania, adresów i ciągów stanów dla każdego rozkazu, jaki ma zostać wykonany, a następnie zamiana tabeli stanów w układ logiki Boole'a. Należy pamiętać, że zarówno maszyna skończona jak i pamięć wykonane są z rejestrów i bloków logiki, mogą więc zostać zrealizowane w wielu technikach: elektronicznej, hydraulicznej lub mechanicznej (przesuwające się drążki).

Ustaliliśmy już ciąg powiązań między technologią a rozkazami. Jak jednak rozkazy realizują program napisany na przykład w języku Logo, kiedy ten język operuje słowami, a rozkazy są układami bitów? Okazuje się, że niezbędnego przekładu dokonuje sam komputer.

Proces tłumaczenia przeprowadzany przez komputer podobny jest do postępowania cierpliwego i skrupulatnego tłumacza, który musi przełożyć dokument napisany w nieznanym mu języku, mając do dyspozycji wyłącznie słownik. Tłumacz może odszukać znaczenie dowolnego nieznanego słowa w słowniku, a jeśli słowa w definicji słownikowej również nie są mu znane, to ich znaczenie również musi odnaleźć. Proces ten ciągnąć się będzie dotąd, aż tłumacz otrzyma definicję wyrażoną w słowach, których znaczenie rozumie. W tej analogii słownikiem tłumacza (czyli komputera) jest program, a słowa zrozumiałe dla komputera to wspomniane wcześniej wyrażenia pierwotne w języku programowania. Wyrażenia te są zdefiniowane bezpośrednio jako proste ciągi rozkazów maszynowych. Na przykład, kiedy komputer wyszukuje definicje wyrażenia pierwotnego FORWARD, występującego w Logo, natrafia na sekwencję rozkazów maszynowych, które spowodują narysowanie odpowiedniej linii na ekranie.

Dla zrozumienia, w jaki sposób komputer tłumaczy wyrażenia pierwotne z Logo na język maszynowy, pomocna jest wiedza o konwencjach, z jakich komputer korzysta, aby reprezentować programy napisane w Logo w swojej pamięci. Jednym ze sposo-

bów na przechowywanie programu w pamięci komputera jest ciąg znaków w sąsiednich lokalizacjach w pamięci, przy czym każdy znak przechowywany jest w jednej lokalizacji. Komputer przechowuje w swojej pamięci spis adresów ciągów rozkazów, odpowiadających każdej nazwie komendy. Spis ten przechowywany jest w pamięci jako lista nazw z przyporządkowanymi im adresami. Komputer odnajduje lokalizację obiektu o danej nazwie, przeszukując spis w celu odnalezienia tej nazwy i odszukując odpowiadający jej adres — kiedy otrzymuje rozkaz wykonania jakiejś komendy, wyszukuje jej nazwę w spisie, aby dowiedzieć się, gdzie przechowywana jest jej definicja.

Niektóre z tych procesów wyszukiwania obiektów i odnajdywania odpowiadających im ciągów wyrażeń w języku maszynowym można przeprowadzić przed rozpoczęciem wykonywania programu. Oszczędza to czas, ponieważ wielokrotne wyszukiwanie tych samych rzeczy nie ma sensu, jeśli program ma być wykonywany więcej niż jeden raz. Jeśli większość pracy nad konwersją zrobiono zanim program zaczął działać, to proces ten nazywa się **kompilacją**, a program przeprowadzający kompilację nazywa się **kompilatorem**. Jeśli zaś większość pracy wykonywana jest w trakcie działania programu, to proces taki nazywa się **interpretacją**, a program — **interpreterem.** Nie ma sztywnego rozgraniczenia między tymi dwoma typami programów.

Podsumujmy teraz naszą wiedzę o działaniu komputera. Większość czytelników straciła zapewne orientację w tylu szczegółach, ale **nie jest istotne, aby pamiętać, jak działa każdy kolejny poziom!** Ważne jest to, że istnieje hierarchia abstrakcji funkcjonalnych.

Praca wykonywana przez komputer określana jest przez **program**, napisany w pewnym **języku programowania**. Język ten zamieniany jest na ciąg rozkazów w **języku maszynowym** przez **interpretery** lub **kompilatory**, za pomocą ustalonego wcześniej zbioru podprogramów nazywanego **systemem operacyjnym**. Rozkazy przechowywane są w **pamięci** komputera i określają operacje, jakie mają zostać przeprowadzone na danych, które również przechowywane są w pamięci komputera. **Maszyna skończona** pobiera te rozkazy i wykonuje je. Rozkazy i dane reprezentowane są przez układy **bitów**. Maszyny skończone i pamięć składają się z **rejestrów** i **bloków logiki Boole'a**, a te ostatnie zbudowane są z prostych **funkcji logicznych**, takich jak **AND, OR i NOT**. Funkcje logiczne są realizowane za pomocą **przełączników**, połączonych **szeregowo** lub **równolegle**, a przełączniki sterują fizyczną materią, taką jak woda czy elektryczność, której użyto do przesyłania od jednego przełącznika do drugiego któregoś z dwóch możliwych sygnałów: **1** lub **0**. Taka jest hierarchia abstrakcji, dzięki której działa komputer.

Rozdział czwarty

NA ILE UNIWERSALNE SĄ MASZYNY TURINGA?

Jakie są granice możliwości komputerów? Czy wszystkie komputery muszą zawierać logikę Boole'a i rejestry, czy też może istnieje jakiś inny, potężniejszy rodzaj komputerów? Te pytania prowadzą nas do najbardziej interesujących pod względem filozoficznym zagadnień w tej książce: maszyn Turinga, obliczalności, układów chaotycznych, twierdzenia Gödla o niezupełności i komputerów kwantowych — zagadnień, na których koncentruje się większość współczesnych dyskusji o tym, co komputery mogą, a czego nie mogą robić.

Ponieważ komputery potrafią robić pewne rzeczy bardzo przypominające ludzkie myślenie, ludzie często niepokoją się, że zagrożą one ich wyjątkowej pozycji jako istot rozumnych; są tacy, którzy szukają uspokojenia w matematycznych dowodach ograniczeń możliwości komputerów. W naszej historii pojawiały się już kontrowersje podobnego typu. Kiedyś uważano za istotne, aby Ziemia była w centrum Wszechświata i to wyobrażenie o naszej centralnej pozycji symbolizowało naszą wartość. Odkrycie, że

nie zajmujemy tak wyróżnionego miejsca, a nasza planeta jest tylko jedną z wielu planet krążących dokoła Słońca, było w swoim czasie dla wielu ludzi bardzo niepokojące, zaś filozoficzne implikacje astronomii stały się przedmiotem zaciętych dyskusji. Podobna kontrowersja pojawiła się w przypadku teorii ewolucji, która również wydawała się zagrażać ludzkiej wyjątkowości. Podłożem tych wcześniejszych kryzysów filozoficznych była niewłaściwa ocena istoty ludzkiej wartości. Jestem przekonany, że większość obecnych dyskusji filozoficznych o granicach możliwości komputerów wynika z podobnie błędnej oceny.

MASZYNY TURINGA

Podstawową koncepcją w nauce o obliczeniach jest idea **komputera uniwersalnego** — komputera wystarczająco potężnego, aby symulować zachowanie jakiegokolwiek innego urządzenia liczącego. Wielofunkcyjny komputer opisany w poprzednich rozdziałach jest przykładem komputera uniwersalnego; w istocie większość komputerów, jakie spotykamy w życiu codziennym, to komputery uniwersalne. Przy odpowiednim oprogramowaniu i wystarczającej ilości czasu i pamięci, każdy komputer uniwersalny może symulować każdy inny rodzaj komputera, lub (przynajmniej na ile nam wiadomo) dowolne inne urządzenie przetwarzające informacje.

Z tej zasady uniwersalności wynika między innymi, że jedyną istotną różnicą między dwoma komputerami jest szybkość i wielkość pamięci. Komputery mogą różnić się rodzajami podłączonych urządzeń

wejściowych i wyjściowych, ale te tak zwane urządzenia peryferyjne nie są istotnymi charakterystykami komputera, przynajmniej nie bardziej niż jego rozmiary, cena czy kolor obudowy. Pod względem swoich możliwości wszystkie komputery (i wszystkie inny typy uniwersalnych maszyn liczących) tak naprawdę są identyczne.

Koncepcja komputera uniwersalnego została sformułowana i opisana w roku 1937 przez brytyjskiego matematyka Alana Turinga. Turing, jak wielu innych pionierów informatyki, zajmował się problemem stworzenia maszyny, która potrafiłaby myśleć i odkrył schemat dla wielofunkcyjnej maszyny liczącej. Nazwał swoją myślową konstrukcję „maszyną uniwersalną", ponieważ w tamtych czasach słowo *computer* ciągle jeszcze oznaczało „osobę, która przeprowadza obliczenia".

Aby zrozumieć, czym jest maszyna Turinga, wyobraźmy sobie matematyka przeprowadzającego obliczenia na papierowym zwoju. Załóżmy też, że zwój ma nieskończoną długość, a więc nigdy nie będziemy musieli się martwić, że zabraknie miejsca, aby coś zapisać. Matematyk jest w stanie rozwiązać każdy rozwiązywalny problem obliczeniowy, niezależnie od tego, ile operacji będzie z tym związanych, choć może mu to zająć bardzo dużo czasu. Turing pokazał, że jakiekolwiek obliczenie, które może być wykonane przez inteligentnego matematyka, może również zostać wykonane przez głupiego, ale skrupulatnego urzędnika, który przy odczytywaniu i zapisywaniu informacji na zwoju postępuje według prostego zbioru zasad. W istocie Turing pokazał, że urzędnik może zostać zastąpiony maszyną skończoną. Ta-

ka maszyna odczytuje ze zwoju za każdym razem tylko po jednym symbolu, więc zwój najlepiej wyobrażać sobie jako wąską taśmę, na której w każdej linii może być zapisany jeden znak.

Dziś połączenie maszyny skończonej z nieskończenie długą taśmą nazywamy **maszyną Turinga**. Taśma w maszynie to odpowiednik pamięci we współczesnym komputerze. Maszyna skończona jedynie odczytuje i zapisuje znaki na taśmie i przesuwa ją w jedną lub w drugą stronę, zgodnie ze stałym i prostym zbiorem zasad. Turing pokazał, że dowolny problem obliczalny może zostać rozwiązany przez zapisywanie znaków na taśmie maszyny Turinga — znaków określających nie tylko sam problem, ale też i metodę jego rozwiązania. Maszyna Turinga oblicza odpowiedź, przesuwając się tam i z powrotem wzdłuż taśmy, odczytując i zapisując znaki, aż na taśmie zostanie zapisane rozwiązanie.

Prawdę mówiąc, trudno mi sobie wyobrazić działanie konstrukcji zaproponowanej przez Turinga. Dla mnie konwencjonalny komputer, który ma pamięć zamiast taśmy, jest łatwiejszym do zrozumienia przykładem maszyny uniwersalnej. Łatwiej mi sobie wyobrazić, jak konwencjonalny komputer może zostać zaprogramowany w celu symulowania maszyny Turinga, niż na odwrót. Zdumiewa mnie nie tyle myślowa konstrukcja Turinga, co jego hipoteza, że istnieje tylko jeden typ komputera uniwersalnego. Na ile nam wiadomo, żadne urządzenie zbudowane w realnym świecie nie może mieć większej mocy obliczeniowej od maszyny Turinga. Mówiąc bardziej precyzyjnie, każde obliczenie, jakie może zostać wykonane przez dowolne realne urządzenie liczące,

może zostać wykonane prze komputer uniwersalny, o ile ma on wystarczająco wiele czasu i pamięci. Jest to niezwykłe stwierdzenie, sugerujące, że komputer uniwersalny z odpowiednim oprogramowaniem powinien potrafić symulować funkcjonowanie ludzkiego mózgu.

POZIOMY MOŻLIWOŚCI

Co sprawia, że hipoteza Turinga jest prawdziwa? Czy nie jest tak, że niektóre typy komputerów mogą być lepsze od tych, które opisaliśmy. Po pierwsze, omawiane przez nas dotychczas komputery były binarne, czyli reprezentowały wszystko w kategoriach 1 i 0. Czy komputer nie byłby silniejszy, gdybyśmy mogli reprezentować obiekty za pomocą logiki z trzema stanami, na przykład TAK, NIE, BYĆ MOŻE. Nie, nie byłby. Wiemy, że komputer z trzema stanami nie potrafiłby zrobić niczego, czego nie może zrobić komputer z dwoma stanami, ponieważ umiemy symulować jeden za pomocą drugiego. W komputerze z dwoma stanami możemy odtworzyć każdą operację, jaka może zostać dokonana na komputerze z trzema stanami, przez zakodowanie każdego z tych trzech stanów jako pary bitów — na przykład 00 dla TAK, 11 dla NOT, i 01 dla BYĆ MOŻE. Dla każdej możliwej funkcji w logice z trzema stanami istnieje odpowiednia funkcja w logice z dwoma stanami, która działa na tej reprezentacji. Nie oznacza to, że komputer z trzema stanami może nie mieć pewnych praktycznych przewag nad maszyną z dwoma stanami: na przykład, mógłby korzystać z mniejszej liczby połączeń i dzięki temu być mniejszy lub

tańszy w produkcji. Ale z pewnością możemy powiedzieć, że takie komputery nie byłyby w stanie zrobić niczego nowego. Byłyby tylko jeszcze jedną wersją maszyny uniwersalnej.

Podobny argument stosuje się do komputerów o czterech czy pięciu stanach, lub do maszyn z dowolną skończoną liczbą stanów. A co w takim razie z komputerami operującymi sygnałami analogowymi — czyli sygnałami z nieskończoną liczbą możliwych wartości? Wyobraźmy sobie na przykład komputer, który do oznaczania liczb używa ciągłego zakresu napięć. Zamiast jedynie dwóch, trzech lub pięciu, każdy sygnał mógłby przenosić nieskończoną liczbę możliwych komunikatów, odpowiadających ciągłemu zakresowi napięć. Na przykład, komputer analogowy mógłby przedstawiać liczbę między 0 a 1 za pomocą napięcia między zero a jednym woltem. Ułamek mógłby być reprezentowany na dowolnym poziomie precyzji, niezależnie od liczby miejsc dziesiętnych, przez użycie dokładnie odpowiadającego mu napięcia.

Komputery wykorzystujące do reprezentowania wielkości takie sygnały analogowe już istnieją, a w istocie pierwsze komputery oparte były właśnie na tej zasadzie. Nazywane są one **komputerami analogowymi**, dla odróżnienia od komputerów cyfrowych, które tu omawialiśmy, operujących dyskretną liczbą możliwych komunikatów w każdym sygnale. Można by przypuszczać, że komputery analogowe będą miały większe możliwości, ponieważ mogą reprezentować continuum wartości, podczas gdy komputery cyfrowe reprezentują dane jedynie za pomocą dyskretnych liczb. Jednak, gdy przyjrzymy się bliżej,

ta przewaga znika. Prawdziwe continuum nie może być zrealizowane w realnym świecie.

Kłopot z komputerami analogowymi polega na tym, że sygnały, którymi operują, osiągają jedynie ograniczony stopień dokładności. Każdy rodzaj sygnału analogowego — elektryczny, mechaniczny czy chemiczny — zawiera jakieś zakłócenia; oznacza to, że przy pewnej określonej rozdzielczości sygnał będzie w istocie losowy. Sygnał analogowy skazany jest na wpływy licznych nieistotnych i nieznanych źródeł zakłóceń; na przykład sygnał elektryczny może zostać zakłócony przez przypadkowe ruchy molekuł w przewodzie lub przez pole magnetyczne wytwarzane, gdy w pokoju obok włączane jest światło. W bardzo dobrym obwodzie elektrycznym taki szum można zminimalizować — ale zawsze będzie istniał. Wprawdzie istnieje nieskończona liczba możliwych poziomów sygnału, ale jedynie skończona liczba poziomów umożliwia sensowne rozróżnienia — a zatem reprezentuje informację. Jeśli szumem jest jedna milionowa sygnału, to w sygnale jest jedynie około miliona sensownych rozróżnień; tak więc informacja, którą niesie, może być reprezentowana przez sygnał cyfrowy używający dwudziestu bitów ($2^{20} = 1\,048\,578$). Podwajanie liczby sensownych rozróżnień w komputerze analogowym wymagałoby wykonania wszystkiego z dwukrotnie większą precyzją, podczas gdy w komputerze cyfrowym można podwoić liczbę sensownych rozróżnień przez dodanie pojedynczego bitu. Najlepsze komputery analogowe mają mniej niż trzydzieści bitów dokładności. Ponieważ komputery cyfrowe często reprezentują liczby za pomocą trzydziestu dwóch lub sześćdziesię-

ciu czterech bitów, to w praktyce często mogą generować znacznie większe liczby sensownych rozróżnień niż mogą to zrobić komputery analogowe.

Ktoś mógłby argumentować, że choć szum w komputerze analogowym może nie nieść informacji, nie oznacza to, że jest on całkiem bezużyteczny. Z pewnością można sobie wyobrazić obliczenia, którym sprzyja obecność szumu. Przykładem mogą być opisane dalej obliczenia wymagające liczb losowych. Ale komputer cyfrowy też może wygenerować losowy szum, jeśli do obliczeń potrzebna jest przypadkowość.

LICZBY LOSOWE

W jaki sposób komputer cyfrowy może wytworzyć coś losowego? Czy taki ściśle deterministyczny układ jak komputer może wygenerować w pełni losowy ciąg liczb? W sensie formalnym odpowiedź jest negatywna, ponieważ wszystko, co zrobi komputer cyfrowy, jest jednoznacznie określone przez jego strukturę i dane wejściowe. Ale to samo można powiedzieć o kole ruletki — ostatecznie, miejsce, w którym zatrzyma się kulka, też jest jednoznacznie określone przez właściwości fizyczne kulki (jej masę i prędkość) i koła ruletki. Jeśli znalibyśmy wszystkie szczegóły konstrukcji ruletki i dokładne dane wejściowe, określające ruch koła ruletki i rzut kulki, moglibyśmy przewidzieć pole, na którym kulka się zatrzyma. Wynik wydaje się losowy, ponieważ nie widać w nim żadnych prostych prawidłowości i w praktyce jest trudny do przewidzenia.

Podobnie jak koło ruletki, komputer również może wytwarzać ciągi liczb, które są losowe w tym sa-

mym sensie. W istocie, może wytworzyć ciąg liczb losowych, symulując fizyczną ruletkę za pomocą modelu matematycznego i rzucając wirtualną kulką za każdym razem pod nieco innym kątem. Nawet jeśli kąty, pod którymi komputer ją wyrzuca, podlegają jakiejś prawidłowości, to symulowana dynamika koła przetransformuje te maleńkie różnice na coś, co odpowiada ciągowi nieprzewidywalnych liczb. Taki ciąg liczb nazywany jest ciągiem liczb **pseudolosowych**, ponieważ jest on losowy tylko dla obserwatora, który nie wie, w jaki sposób go otrzymano. Ciągi wytworzone przez generatory liczb pseudolosowych spełniają wszystkie zwykłe kryteria statystyczne rozkładów losowych.

Koło ruletki jest przykładem układu, który fizycy nazywają **układem chaotycznym** — układu, w którym mała zmiana w warunkach początkowych (prędkość rzutu, masa kulki, średnica koła itd.) może spowodować dużą zmianę stanu, będącego wynikiem ewolucji układu (czyli wygenerowanej liczby). Koncepcja układu chaotycznego pomaga wyjaśnić, w jaki sposób deterministyczny zbiór oddziaływań może wytworzyć nieprzewidywalne rezultaty. W komputerze istnieją sposoby generowania ciągów pseudolosowych prostsze niż symulacja koła ruletki, ale koncepcyjnie wszystkie są podobne do tego modelu.

Komputery cyfrowe są przewidywalne i nieprzewidywalne dokładnie w tym samym sensie, co cała reszta fizycznego świata. Podlegają prawom deterministycznym, ale te prawa mają skomplikowane konsekwencje, bardzo trudne do przewidzenia. Często nie opłaca się zgadywanie, co komputery będą mogły zrobić, zanim tego nie zrobią. Podobnie

jak w układach fizycznych, nie trzeba wiele, aby obliczenia stały się złożone. W komputerach układy chaotyczne — czyli układy, których zachowanie jest bardzo czułe na warunki początkowe — są normą.

OBLICZALNOŚĆ

Nawet jeśli komputer uniwersalny może obliczyć wszystko, cokolwiek może być obliczone na jakimkolwiek innym urządzeniu liczącym, są pewne rzeczy, których po prostu obliczyć się nie da. Oczywiście niemożliwe jest obliczenie odpowiedzi na nieprecyzyjnie sformułowane pytanie, chociażby „Co to jest życie?", lub pytania, dla których brakuje danych, na przykład „Jaka liczba zostanie wylosowana w jutrzejszej loterii?". Ale istnieją również poprawnie sformułowane problemy obliczeniowe, których nie można rozwiązać. Takie problemy nazywamy **nieobliczalnymi.**

Powinienem was ostrzec, że problemy nieobliczalne w praktyce pojawiają się niezwykle rzadko. W istocie trudno jest znaleźć przykład dobrze określonego problemu nieobliczalnego, który ktokolwiek chciałby rozwiązać. Rzadkim przykładem dobrze zdefiniowanego oraz użytecznego choć nieobliczalnego problemu jest **problem zakończenia pracy (stopu).** Wyobraźmy sobie, że próbuję napisać program komputerowy, który będzie badał inny program komputerowy i określał, czy ten program się zatrzyma. Jeśli badany program nie zawiera pętli i rekurencyjnych wywołań podprogramów, to musi się on po jakimś czasie zatrzymać, ale jeśli takie struktury w tym programie istnieją, to może równie

dobrze działać w nieskończoność. Okazuje się, że nie ma algorytmu dla badania programu w celu określenia, czy nie zawiera on nieszczęsnej nieskończonej pętli. Co więcej, nie tylko nikt jeszcze nie odkrył takiego algorytmu, ale wiadomo, że taki algorytm nie może istnieć. Problem zakończenia pracy jest nieobliczalny.

Aby to zrozumieć, wyobraźmy sobie przez moment, że mamy taki program, nazwany Test-dla-Stopu, i że traktuje on badany program jako dane wejściowe. (Traktowanie programu jako danych może wydawać się dziwne, ale jest zupełnie możliwe, ponieważ program, jak wszystko inne, może być reprezentowany w postaci bitów.) Mogę wstawić program Test-dla-Stopu jako podprogram w innym programie — nazwijmy go Paradoks — który będzie przeprowadzać Test-dla-Stopu na samym Paradoksie. Wyobraźmy sobie, że program Paradoks został tak napisany, iż dla dowolnego rezultatu dawanego przez Test-dla-Stopu da wynik przeciwny. Jeśli Test-dla-Stopu stwierdzi, że Paradoks się zatrzyma, to Paradoks zostaje zaprogramowany tak, aby wpadł w nieskończoną pętlę. Jeśli zaś Test-dla-Stopu stwierdzi, że Paradoks będzie działać w nieskończoność, to Paradoks zostaje zaprogramowany tak, aby się zatrzymał. Ponieważ Paradoks zaprzecza programowi Test-dla-Stopu, to Test-dla-Stopu nie działa w przypadku Paradoksu, a tym samym nie jest prawdą, że działa dla wszystkich możliwych programów. Zatem program obliczający funkcję stopu nie może istnieć.

Problem zakończenia pracy, wymyślony przez Alana Turinga, jest ważny głównie jako przykład

problemu nieobliczalnego; większość problemów nieobliczalnych, które pojawiają się w praktyce, jest podobna do niego lub mu równoważna. Niezdolność komputera do rozwiązania takiego problemu nie jest jego słabością, ponieważ jest on ze swej istoty nierozwiązywalny. Nie można skonstruować maszyny, która mogłaby go rozwiązać. I, na ile wiemy, nie istnieje urządzenie, które mogłoby wykonać jakiekolwiek inne obliczenie, którego nie potrafi wykonać maszyna uniwersalna. Klasa problemów, które może rozwiązać komputer cyfrowy, najwyraźniej zawiera wszystkie problemy, które mogą zostać obliczone za pomocą jakiegokolwiek urządzenia. (To ostatnie stwierdzenie nazywane jest czasami twierdzeniem Churcha, od nazwiska jednego ze współczesnych Turingowi uczonych, Alonzo Churcha. Matematycy myśleli o obliczeniach i logice przez setki lat, ale — w jednym z bardziej spektakularnych przykładów synchroniczności w nauce — Turing, Church i jeszcze jeden brytyjski matematyk, Emil Post, niezależnie od siebie wynaleźli koncepcję uniwersalnych obliczeń mniej więcej w tym samym czasie. Opisywali ją w bardzo różny sposób, ale wszyscy opublikowali swoje rezultaty w 1937 roku, przygotowując grunt pod rewolucję komputerową, która wkrótce miała nastąpić.)

Inną nieobliczalną funkcją, blisko związaną z przedstawionym problemem, jest problem określenia, czy dane twierdzenie matematyczne jest prawdziwe czy fałszywe. Okazuje się, że nie istnieje algorytm, który byłby w stanie rozwiązać ten problem — wynika to z twierdzenia Gödla o niezupełności, udowodnionego przez Kurta Gödla w roku 1931, tuż przed tym, jak

Turing opisał problem zatrzymania pracy. Twierdzenie Gödla zaszokowało wielu matematyków, którzy do tego momentu generalnie zakładali, że dla **każdego** twierdzenia matematycznego można pokazać, iż jest albo prawdziwe, albo fałszywe. Twierdzenie Gödla mówi zaś, że w każdym niesprzecznym systemie matematycznym zawierającym arytmetykę istnieją twierdzenia, co do których nie można udowodnić, że są prawdziwe, ani też, że są fałszywe. Matematycy rozumieli swoją pracę jako dowodzenie lub obalanie twierdzeń, a tymczasem Gödel pokazał, że w pewnych przypadkach taka „praca" jest niemożliwa do wykonania.

Niektórzy matematycy i filozofowie przypisywali twierdzeniu Gödla o niezupełności niemal mistyczne własności. Kilku z nich wierzy nadal, że twierdzenie to pokazuje, iż ludzka intuicja wykracza poza możliwości komputera — że istoty ludzkie mogą dzięki intuicji dochodzić do prawd, których maszyny nie mogą udowodnić ani obalić. Jest to argument atrakcyjny pod względem emocjonalnym i czasami posługują się nim filozofowie, którzy nie lubią być porównywani z komputerami. Ale argument ten jest błędny. Niezależnie od tego, czy ludziom udają się intuicyjne skoki nieosiągalne dla komputerów, twierdzenie Gödla o niezupełności nie daje powodu, aby wierzyć, że istnieją twierdzenia matematyczne, które mogą być udowodnione przez matematyka, ale nie mogą być udowodnione przez komputer. Przynajmniej na ile nam wiadomo, dowolne twierdzenie, które może zostać udowodnione przez istotę ludzką, może również zostać udowodnione przez maszynę.

Ludzie wcale nie radzą sobie z problemami nieobliczalnymi lepiej niż komputery.

Choć trzeba się trochę wysilić, aby podać przykłady problemów nieobliczalnych, to łatwo można udowodnić, że większość funkcji matematycznych jest nieobliczalna. Jest tak, ponieważ każdy program może być określony za pomocą skończonej liczby bitów, podczas gdy funkcja z reguły wymaga ich nieskończonej liczby, istnieje więc znacznie więcej funkcji niż programów. Rozważmy ten rodzaj funkcji matematycznych, które zamieniają jedną liczbę w inną — na przykład cosinus czy logarytm. Matematycy mogą definiować wszelkiego rodzaju dziwaczne funkcje tego typu: na przykład funkcję, która każdemu ułamkowi dziesiętnemu przyporządkowuje sumę występujących w nim cyfr. O ile wiem, funkcja ta jest bezużyteczna, ale matematycy uważają ją za zupełnie dobrą, po prostu dlatego, że zamienia każdą liczbę na dokładnie jedną inną liczbę. Można podać matematyczny dowód, że funkcji jest nieskończenie więcej niż programów. Tak więc, dla większości funkcji nie ma programu, który mógłby je obliczyć. Faktyczne zliczanie wiąże się z różnego rodzaju trudnościami (włącznie z koniecznością przeliczania zbiorów nieskończonych i rozróżniania między różnym stopniami nieskończoności!), ale konkluzja jest poprawna: statystycznie rzecz biorąc, większość funkcji matematycznych jest nieobliczalna. Na szczęście, prawie wszystkie te nieobliczalne funkcje są bezużyteczne, a prawie wszystkie funkcje, które chcielibyśmy obliczyć, są obliczalne.

Jak już zauważyliśmy wcześniej, ciągi liczb pseudolosowych generowane przez komputer wyglądają na przypadkowe, ale kryje się za nimi algorytm, który je generuje. Jeśli wiemy, w jaki sposób generowany jest ciąg, to z definicji jest on przewidywalny i tym samym nie jest losowy. Jeśli kiedykolwiek potrzebowalibyśmy naprawdę nieprzewidywalnego ciągu liczb losowych, musielibyśmy uzupełnić naszą maszynę uniwersalną o niedeterministyczne urządzenie do generowania przypadkowości.

Można sobie wyobrażać takie urządzenie do generowania przypadkowości jako coś w rodzaju elektronicznej ruletki, ale, jak już widzieliśmy, ze względu na prawa fizyki takie urządzenie nie jest naprawdę losowe. Jedynym sposobem na osiągnięcie rzeczywiście nieprzewidywalnych efektów jest wykorzystanie mechaniki kwantowej. W odróżnieniu od fizyki klasycznej, rządzącej kołem ruletki, w której skutki określane są przez przyczyny, mechanika kwantowa dostarcza efektów w pełni probabilistycznych. Nie można przewidzieć, na przykład, kiedy jakiś atom uranu rozpadnie się, tworząc ołów. Można zatem posłużyć się licznikiem Geigera do wytworzenia naprawdę losowego ciągu danych — czego komputer uniwersalny z zasady nie potrafi zrobić.

Prawa mechaniki kwantowej spowodowały postawienie wielu pytań dotyczących komputerów uniwersalnych, na które nikt do tej pory nie odpowiedział. Na pierwszy rzut oka wydawałoby się, że mechanika kwantowa powinna dobrze pasować do komputerów cyfrowych, ponieważ termin „kwanto-

wy" niesie w zasadzie to samo znaczenie co „cyfrowy". Tak jak efekty cyfrowe, zjawiska kwantowe istnieją jedynie w stanach dyskretnych. Z kwantowego punktu widzenia na pozór ciągła, analogowa istota świata fizycznego — na przykład przepływ prądu — jest iluzją spowodowaną faktem, że widzimy rzeczy w skali znacznie większej niż skala atomowa. Dobrą stroną mechaniki kwantowej jest, że w skali atomowej wszystko jest dyskretne, wszystko jest cyfrowe. Ładunek elektryczny jest zawsze pewną wielokrotnością ładunku elektronu i nie ma czegoś takiego, jak pół elektronu. Niestety, reguły rządzące oddziaływaniami obiektów w tej skali kłócą się z naszą intuicją.

Na przykład, nasze zdroworozsądkowe intuicje mówią nam, że przedmiot nie może być w dwóch miejscach naraz. W kwantowym świecie nie jest to do końca prawdą, ponieważ, ogólnie biorąc, zgodnie z mechaniką kwantową nic nie może znajdować się w dokładnie określonym miejscu. Pojedyncza cząstka subatomowa istnieje wszędzie naraz, mamy tylko większe szanse na zaobserwowanie jej w jakimś miejscu. Zwykle możemy myśleć, że cząstka jest tam, gdzie ją zaobserwowaliśmy, ale do wyjaśnienia wszystkich obserwowanych efektów musimy uznać, że jest ona w więcej niż jednym miejscu naraz. Prawie wszyscy, łącznie z niektórymi fizykami, mają trudności ze zrozumieniem tej koncepcji.

Czy możemy wykorzystać efekty kwantowe do zbudowania sprawniejszych komputerów? W chwili obecnej pytanie to pozostaje bez odpowiedzi, ale pojawiły się sugestie, że być może da się to zrobić. Wydaje się, że atomy potrafią z łatwością rozwiązywać pewne problemy — na przykład, jak trzymać się

razem — skądinąd bardzo trudne do obliczenia za pomocą konwencjonalnego komputera. Kiedy dwa atomy wodoru łączą się z atomem tlenu tworząc cząsteczkę wody, to w jakiś sposób wyliczają, że kąt między dwoma wiązaniami powinien wynosić 107 stopni. W przybliżeniu można ten kąt obliczyć, wychodząc z zasad mechaniki kwantowej i posługując się komputerem cyfrowym, ale zajmuje to wiele czasu, tym więcej, im dokładniejszy chcemy mieć wynik. A jednak każda cząsteczka w szklance wody jest w stanie przeprowadzić to obliczenie w mgnieniu oka. W jaki sposób jedna cząsteczka może być o tyle szybsza niż komputer cyfrowy?

Zwykłemu komputerowi ten kwantowomechaniczny problem zajmuje tak wiele czasu, ponieważ w celu obliczenia dokładnej odpowiedzi musi on wziąć pod uwagę nieskończoną liczbę możliwych konfiguracji cząsteczek wody i to przy założeniu, że atomy tworzące cząsteczkę mogą być we wszystkich konfiguracjach naraz. Oto powód, dla którego komputer może w skończonym czasie otrzymać jedynie odpowiedź przybliżoną. Jednym ze sposobów wytłumaczenia, jak cząsteczka wody może wykonać to samo obliczenie, polega na wyobrażeniu sobie, że próbuje ona wszystkich możliwych konfiguracji naraz — innymi słowy, korzysta z przetwarzania równoległego. Czy moglibyśmy zaprząc tę zdolność obiektów kwantowych do jednoczesnych obliczeń do stworzenia silniejszego komputera? Nikt nie wie tego na pewno.

Ostatnio pojawiły się intrygujące wskazówki, że być może uda się nam zbudować komputer kwantowy, który wykorzystuje zjawisko znane jako **stany**

splątane. Kiedy w układzie kwantowym oddziałują dwie cząstki, ich losy złączone są w sposób niepodobny do niczego, co znamy ze świata klasycznego: gdy mierzymy jakąś własność jednej z nich, wpływa to na wynik pomiaru dla drugiej cząstki, nawet jeśli cząstki są od siebie znacznie oddalone. Einstein nazwał ten efekt — w którym nie występuje żadne opóźnienie w czasie — „tajemniczym oddziaływaniem na odległość" i powszechnie znane jest jego niezadowolenie z faktu, że świat może działać w ten sposób.

Komputer kwantowy wykorzystywałby splątanie: jednobitowy kwantowy rejestr pamięci przechowywałby nie tylko 1 czy 0; przechowywałby superpozycję wielu jedynek i zer. Jest to analogiczne do cząsteczki znajdującej się w wielu miejscach naraz: bit znajdujący się w wielu stanach (1 lub 0) naraz. Różni się to od bycia w stanie pośrednim między 1 i 0, ponieważ każda z będących w superpozycji cząstek może zostać splątana z innym bitami w komputerze kwantowym. Kiedy dwa takie kwantowe bity połączone zostaną w kwantowy blok logiczny, każdy z ich stanów może oddziaływać w inny sposób, tworząc jeszcze bogatszy zbiór splątań. Ilość obliczeń, jaka może zostać wykonana przez taki pojedynczy kwantowy blok logiczny jest bardzo duża, być może nawet nieskończona.

Teoria będąca podstawą komputerów kwantowych jest dobrze ugruntowana, ale ciągle są problemy z wykorzystaniem jej w praktyce. Przede wszystkim, jak możemy je wykorzystać do obliczenia czegoś użytecznego? Peter Shor odkrył ostatnio sposób na wykorzystanie efektów kwantowych — przynajmniej w teorii — do pewnych ważnych i trudnych

obliczeń, na przykład do rozkładu dużych liczb na czynniki pierwsze, a jego praca wywołała nową falę zainteresowania komputerami kwantowymi. Ciągle jednak pozostaje wiele trudności. Jednym z problemów jest, że na to, by można było przeprowadzać obliczenia, bity w komputerze kwantowym muszą pozostawać splątane, a nawet najmniejsze zaburzenie — powiedzmy, promieniowanie kosmiczne czy nawet nieuniknione fluktuacje samej próżni — mogą ten stan zakłócić. (Tak, w mechanice kwantowej nawet próżnia robi dziwne rzeczy.) Ta utrata splątania, zwana **dekoherencją**, może okazać się piętą Achillesową komputerów kwantowych. Co więcej, metoda Shora wydaje się działać jedynie dla szczególnej klasy obliczeń, wykorzystujących szybką operację zwaną uogólnioną transformacją Fouriera. Nie można wykluczyć, że problemy, które należą do tej kategorii, mogą okazać się łatwe do obliczenia za pomocą klasycznej maszyny Turinga; jeśli by tak było, to kwantowe koncepcje Shora byłyby równoważne pewnemu programowi, który można wykonać na konwencjonalnym komputerze.

Jeśli komputery kwantowe będą w stanie przeszukiwać nieskończoną liczbę możliwości naraz, to będą jakościowo i fundamentalnie silniejsze od komputerów konwencjonalnych. Większość naukowców byłaby zdziwiona, gdyby mechanice kwantowej udało się stworzyć komputer efektywniejszy niż maszyna Turinga, ale postęp w nauce polega na niespodziankach. Jeśli macie nadzieję, że zostaniecie zaskoczeni przez nowy rodzaj komputerów, to mechanika kwantowa jest tą dziedziną, na którą dobrze jest mieć oko.

Prowadzi nas to z powrotem do zagadnień filozoficznych, którymi zajmowaliśmy się na początku rozdziału — na przykład związku między komputerem a ludzkim mózgiem. Z pewnością można sobie wyobrazić — jak spekulował przynajmniej jeden znany fizyk (ku niezadowoleniu jego kolegów) — że ludzki mózg wykorzystuje efekty kwantowe. Nie ma jednak żadnych dowodów na to, że tak się dzieje. Z pewnością mechanika kwantowa ma wpływ na neurony, tak, jak to się dzieje przypadku fizyki tranzystora, ale nie ma dowodów na to, że funkcjonowanie neuronów przebiega na poziomie kwantowym, a nie klasycznym; mówiąc inaczej, nic nie wskazuje na to, aby mechanika kwantowa była niezbędna do wyjaśnienia ludzkiego myślenia. Na ile nam wiadomo, wszystkie istotne własności obliczeniowe neuronu mogą być symulowane przez konwencjonalny komputer. Jeśli tak jest rzeczywiście, to możliwe jest również symulowanie sieci dziesiątek miliardów takich neuronów, co z kolei oznacza, że mózg może być symulowany przez maszynę uniwersalną. Nawet jeśli okaże się, że mózg korzysta z obliczeń kwantowych, to prawdopodobnie nauczymy się budować urządzenia wykorzystujące te same efekty — a wtedy ciągle możliwe będzie symulowanie umysłu ludzkiego przy użyciu maszyny.

Teoretyczne ograniczenia możliwości komputerów nie dają nam użytecznej linii podziału na istoty ludzkie i maszyny. Wszystko wskazuje, że mózg jest czymś w rodzaju komputera, a myśli to po prostu złożone obliczenia. Być może, ten wniosek nie jest dla was przyjemny, ale według mnie w niczym nie ujmuje wspaniałości i wartości ludzkiej myśli. Twierdze-

nie, że myśl jest złożonym procesem obliczeniowym, brzmi jak teza wygłaszana czasem przez biologów, że życie jest złożoną reakcją chemiczną: oba zdania są prawdziwe, a jednak ciągle mogą być uznawane za niekompletne. Wskazują one na prawdziwe elementy, ale ignorują tajemnicę. Dla mnie zarówno życie, jak i myśl, stają się czymś jeszcze wspanialszym, jeśli uświadomimy sobie, że wyłaniają się z prostych, możliwych do zrozumienia elementów. Nie czuję się poniżony moim pokrewieństwem z maszyną Turinga.

Rozdział piąty

ALGORYTMY I HEURYSTYKI

Kiedy studiowałem na MIT, mieszkałem z kolegą, który miał kilkadziesiąt par skarpet, każda w trochę innym kolorze i z innym wzorem. Często zwlekał z praniem do momentu, aż wszystkie były brudne, zawsze więc po praniu stawał przed niełatwym zadaniem połączenia skarpet z powrotem w pary. A oto jego metoda: wybierał przypadkową skarpetę ze stosu upranej bielizny, potem w przypadkowy sposób wybierał drugą i porównywał z pierwszą sprawdzając, czy są takie same. Jeśli nie były, rzucał tę drugą z powrotem i wybierał następną. Postępował tak, aż trafił na skarpetę pasującą do pary, a następnie powtarzał całą procedurę z kolejną skarpetą. Ponieważ miał do posortowania znaczną ilość upranej bielizny, cały proces przebiegał bardzo wolno — szczególnie na początku, ponieważ liczba skarpet, na jakie można było trafić, zanim znalazło się tę właściwą, była wtedy znaczna.

Mój kolega studiował matematykę, ale najwyraźniej uczęszczał też na jakiś wykład o komputerach. Pewnego dnia, kiedy przyniósł kosz z praniem do na-

szego pokoju, oznajmił: „Postanowiłem użyć do sortowania skarpet lepszego algorytmu". Chodziło mu o to, że teraz postępował w zupełnie inny sposób. Wyciągał pierwszą skarpetę i kładł ją na stole, potem wyciągał następną i porównywał ją z pierwszą; jeśli nie pasowała, kładł ją obok pierwszej. Za każdym razem, kiedy wyciągał skarpetę, porównywał ją z wydłużającym się szeregiem skarpet na stole. Jeśli potrafił dobrać skarpetę do pary, zwijał je i wrzucał do szafki. A jeśli nie, to dokładał skarpetę do szeregu na stole. Posługując się tą metodą mógł dobrać skarpetki do pary w czasie będącym zaledwie ułamkiem tego, jaki zajmowało mu to przedtem. Rodzice, którzy drogo płacili za jego studia, byliby zapewne dumni, gdyby dowiedzieli się, jak wykorzystywał w praktyce dopiero co nabytą wiedzę.

ALGORYTMICZNA GWARANCJA

Ze skarpetami czy bez, **algorytm** jest procedurą, która nie może zawieść, procedurą, która gwarantuje osiągnięcie określonego celu. Słowo „algorytm" pochodzi od imienia arabskiego matematyka al-Khwarizmiego, który w dziewiątym wieku stworzył duży zbiór algorytmów. W istocie słowo „algebra" pochodzi od zwrotu **al-jabr** („przestawienie"), który pojawił się w tytule jednej z jego książek. Wiele algorytmów al-Khwarizmiego używanych jest do dziś. Opisał je oczywiście w języku arabskim, co być może jest powodem, dlaczego język ten uznany został za język magicznych zaklęć. (Sugerowano nawet, że zaklęcie „abrakadabra" jest przekręconym pełnym imieniem

al-Khwarizmiego — Abu Abdullah abu Jafar Muhammad ibn Musa al-Khwarizmi.)

Algorytmy komputerowe formułowane są zwykle w postaci programów. Ponieważ określenie to odnosi się bardziej do ciągu operacji niż do szczególnego sposobu, w jaki są opisane, możliwe jest wyrażenie tego samego algorytmu w wielu różnych językach komputerowych, a nawet wbudowanie go w konstrukcję urządzenia liczącego, poprzez połączenie odpowiednich rejestrów i bramek logicznych. Zwykle ten sam rezultat można otrzymać za pomocą wielu algorytmów. Tak jak w przykładzie ze skarpetami, różne algorytmy wymagają różnych ilości czasu na wykonanie zadania. Niektóre mogą też mieć inne zalety: mogą używać bardzo małej części pamięci komputera lub wymagać jedynie szczególnie prostych sposobów komunikacji, łatwych do wbudowania w obwody urządzenia liczącego przez zastosowanie odpowiednich połączeń. Różnica w wymaganiach co do szybkości i pamięci między dobrym a złym algorytmem to często czynnik rzędu tysięcy, a nawet milionów. Niekiedy odkrycie nowego algorytmu pozwala na rozwiązanie problemów, które wcześniej znajdowały się poza zasięgiem możliwości.

Ponieważ algorytm może być zrealizowany na wiele różnych sposobów i może być stosowany do problemów o różnej wielkości, nie można ocenić, jak jest szybki, mierząc po prostu czas, jaki upływa, zanim osiągnięte zostanie rozwiązanie konkretnego problemu. Czas ten zależy od metody realizacji i skali problemu. Zamiast tego zwykle opisujemy szybkość algorytmu przez podanie, na ile czas potrzebny do wykonania zdania rośnie przy zwiększaniu proble-

mu. W przykładzie z parowaniem skarpet większość czasu zajmuje wyciąganie skarpetek z koszyka, możemy więc porównywać dwa algorytmy pytając, jak liczba skarpet wyciągana w każdym z nich ma się do całkowitej liczby skarpet. Przyjmijmy, że w koszu z praniem jest n skarpet. W pierwszym algorytmie odnalezienie dwóch pasujących do siebie skarpet wymaga wyciągnięcia i włożenia z powrotem przeciętnie połowy pozostałych skarpet, tak więc liczba wyciągniętych skarpet proporcjonalna jest do kwadratu liczby skarpet. Przy analizowaniu algorytmów zwykle nie troszczymy się o wyliczenie dokładnej stałej proporcjonalności; po prostu mówimy, że algorytm jest **rzędu** n^2, co oznacza, że dla dużych problemów czas wymagany na wykonanie zadania tym algorytmem rośnie jak kwadrat wielkości problemu. Oznacza to, że jeśli mamy dziesięć razy więcej skarpet, to pierwszy algorytm zajmie sto razy więcej czasu, nie jest więc zbyt dobrym algorytmem do parowania dużej liczby skarpet. Jednak w drugim algorytmie każda z n skarpet wybierana jest tylko jeden raz, algorytm jest więc rzędu n. Kiedy użyjemy tego drugiego algorytmu do sortowania dziesięciokrotnie większej ilości skarpet, zadanie zajmie tylko dziesięć razy więcej czasu.

Jedną z największych przyjemności w programowaniu komputerowym jest odkrywanie nowych, szybszych i bardziej wydajnych algorytmów do wykonywania jakichś zadań — szczególnie wtedy, gdy wielu znanych ludzi wymyśliło gorsze rozwiązania. Informatycy mogą osiągnąć sławę i uznanie — przynajmniej wśród innych informatyków — odkrywając szybsze algorytmy dla ogólnie znanych problemów.

Ponieważ złe algorytmy mogą tygodniami rozwiązywać problem, z którym dobry algorytm upora się w kilka minut, to klasycznym sposobem demonstrowania swojej wyższości jest napisanie nowego programu i obliczenie prawidłowej odpowiedzi w czasie, gdy gorszy program twojego kolegi ciągle jeszcze pracuje nad rozwiązaniem.

Najlepsze algorytmy często wydają się mało naturalne. Rozważmy problem sortowania talii kart według rosnących numerów. Jednym ze sposobów jest rozpoczęcie od przeszukiwania całej talii, w celu znalezienia karty o najniższym numerze. Karta ta jest usuwana i staje się pierwszą kartą posortowanego pliku wyjściowego. Następnie, wśród pozostałych kart szukamy karty o najniższym numerze. Proces jest powtarzany, aż zostaną wyczerpane nieposortowane karty i talia zostanie uporządkowana. Procedura ta wymaga przejrzenia całej talii za każdym razem, kiedy wyjmowana jest karta. Ponieważ jest n kart, z których każda wymaga n porównań, to czas realizacji tego algorytmu będzie rzędu n^2.

Jeśli wiemy, że karty są ponumerowane kolejno od 1 do n, to możemy posortować je za pomocą innej metody — takiej, która wykorzystuje definicję rekurencyjną, podobnie jak procedura w Logo rysująca drzewo, opisana w rozdziale 3. Aby posortować talię kart w sposób rekurencyjny, przeglądamy ją jeden raz, przesuwając karty z numerem niższym niż średnia do dolnej połowy talii i pozostawiając karty z wyższym niż średni numerem w górnej połowie. Następnie, używając tego samego algorytmu, sortujemy dwie połówki talii. Rekurencyjne stosowanie tego algorytmu do każdej połówki talii wiązać się bę-

dzie z rekurencyjnym stosowaniem go do każdej połowy połówki itd. Każdy krok tej rekurencji zmniejsza o połowę liczbę kart do posortowania; rekurencja kończy się, kiedy pozostaje tylko jedna karta — która oczywiście jest już posortowana. Ponieważ algorytm wiąże się z powtarzającym dzieleniem talii kart, aż bada się tylko jedną kartę, to wymaga czasu proporcjonalnego do ilości pracy niezbędnej do podzielenia talii złożonej z n kart na kolejne połówki — czyli do logarytmu o podstawie 2 z liczby kart. Tak więc algorytm jest rzędu $n\log n$. (Jeśli zapomnieliście, co to są logarytmy, nie martwcie się. Logarytmy to zwykle małe liczby i spokojnie możemy je pominąć.)

Istnieje jeszcze bardziej elegancki algorytm rekurencyjny, który nie wymaga, aby karty były kolejno ponumerowane; byłby on użyteczny na przykład przy porządkowaniu alfabetycznym dużej liczby wizytówek. Algorytm ten, nazywany **sortowaniem przez łączenie,** jest nieco trudniejszy do zrozumienia, ale jest tak piękny, że nie mogę się oprzeć, aby go nie opisać. Algorytm sortowania przez łączenie opiera się na fakcie, że łatwo jest łączyć dwa już posortowane stosy w jeden posortowany stos, zbierając kolejno karty z najwyższymi numerami ze szczytu jednego lub drugiego stosu; ta procedura łączenia jest podprogramem algorytmu, a sam algorytm działa w następujący sposób: jeśli twój stos składa się tylko z jednej karty, to ta karta już jest posortowana. W innym przypadku podziel stos na połowy i rekurencyjnie użyj algorytmu sortowania przez łączenie do posortowania każdej połowy, a potem połącz je za pomocą procedury łączenia opisanej powyżej. I to wszystko. (Jeśli brzmi to zbyt prosto, aby mogło być

skuteczne, to spróbujcie to zrobić z kilkoma kartami. Zacznijcie od ośmiu.) Algorytm sortowania przez łączenie jest dobrym przykładem na niemal mistyczną potęgę i elegancję rekurencji.

Algorytm sortujący, wymagający tylko $n\log n$ kroków, taki jak sortowanie przez łączenie, jest dość szybki. W istocie jest to najszybszy algorytm z możliwych. Udowodnienie tego twierdzenia wykracza poza ramy tej książki, ale rozumowanie będące podstawą dowodu jest interesujące. Dowód można przeprowadzić, licząc możliwe uporządkowania n kart. Z tego wyliczenia można wywnioskować, że trzeba znać $n\log n$ bitów informacji, aby ułożyć karty we właściwym porządku. Ponieważ każde porównanie dwóch kart tworzy tylko jeden bit informacji (albo pierwsza karta ma wyższy numer niż druga, albo nie), to do posortowania n numerów potrzeba przynajmniej $n\log n$ porównań, tak więc algorytm sortowania przez łączenie jest co najmniej nie gorszy niż jakikolwiek inny. Napisano całe książki na temat właściwego wyboru odpowiedniego algorytmu sortującego. Dla wielu przypadków, kiedy na sortowanie nakłada się pewne dodatkowe ograniczenia lub dostępna jest dodatkowa wiedza o sortowanych obiektach, najszybszy algorytm sortujący pozostaje nieznany. Ciągle jednak, w skali zadań, dla których chcielibyśmy projektować algorytmy, problem sortowania uważany jest za względnie prosty.

Przykładem zagadnienia trudnego jest **problem komiwojażera.** Wyobraźmy sobie, że komiwojażer ma odwiedzić n miast. Znając odległości między nimi, spróbujmy powiedzieć, w jakiej kolejności komiwojażer powinien je odwiedzać, aby zminimalizo-

wać całkowitą przebytą drogę? Nikt nie zna algorytmu umożliwiającego rozwiązanie tego problemu, który składałby się z liczby kroków rzędu n, n^2, n^3 lub wręcz rzędu n do jakiejkolwiek potęgi. Najlepszy znany algorytm jest rzędu 2^n, co oznacza, że czas wymagany do rozwiązania zadania rośnie wykładniczo z wielkością problemu. Jeśli dodamy dziesięć miast do planu podróży komiwojażera, to problem stanie się tysiąc razy trudniejszy ($2^{10}=1024$). Jeśli dodamy trzydzieści, to stanie się około miliarda razy trudniejszy (2^{30} to w przybliżeniu 10^9). W przypadku dużych problemów algorytmy wykładnicze nie są zbyt przydatne, ale dla problemu komiwojażera to najlepsze algorytmy, jakie znamy. Najszybszy komputer na świecie, pracujący miliardy lat, nie miałby wystarczająco dużo czasu, aby znaleźć optymalną drogę dla zaledwie kilku tysięcy miast.

Problem komiwojażera może wydawać się mało ważny, ale okazuje się równoważny wielu innym problemom — tak zwanym **problemom N-P zupełnym**[1] — których rozwiązanie byłoby bardzo przydatne. Szybkie rozwiązanie dla problemu komiwojażera prowadziłoby natychmiast do ich rozwiązania: na przykład pewne szyfry, używane do przekazywania tajnych informacji, stałyby się łatwe do złamania. Każdy, kto posługuje się takimi szyframi zakłada, że szybki algorytm dla rozwiązywania problemu komiwojażera nigdy nie zostanie odkryty — i najprawdopodobniej jest to bezpieczne założenie.

---

[1] N-P oznacza problemy, które mają niedeterministyczne algorytmy o czasie wielomianowym (ang. *nondeterministic polynomial*) — przyp. tłum.

Żaden możliwy do przewidzenia przełom w technice komputerowej nie pomoże w rozwiązaniu problemu komiwojażera, ponieważ nawet komputer miliard razy szybszy ciągle można zapędzić w kozi róg przez dodanie kilku kolejnych miast. Algorytmy wykładnicze są po prostu zbyt wolne, by ich używać do dużych problemów. Postępem byłoby stworzenie nowego algorytmu: nikt nigdy nie pokazał, że szybki algorytm dla problemu komiwojażera **nie może** istnieć. Badania nad algorytmami znacznie posunęły się do przodu w ciągu ostatnich kilkudziesięciu lat, lecz szybki algorytm dla problemu komiwojażera — lub dowód, że nie można go opracować — pozostaje jednym ze świętych Graali informatyki.

ZGODA NA „PRAWIE ZAWSZE"

Choć problem komiwojażera jest trudny, nie jest najtrudniejszym problemem rozwiązywanym za pomocą komputera. O pewnych problemach wiadomo, że ich rozwiązanie wymaga czasu jeszcze dłuższego, niż czas rosnący wykładniczo ze skalą problemu. Jak to zostało omówione w poprzednim rozdziale, znamy problemy nieobliczalne, o których wiadomo, że nie istnieje dla nich rozwiązanie algorytmiczne. Jednak, nawet gdy dla jakichś problemów istnieją rozwiązania algorytmiczne, to ich stosowanie wcale nie musi być najlepszym sposobem rozwiązania. Zgodnie z definicją algorytm gwarantuje, że zadanie zostanie wykonane, ale za tę gwarancję sukcesu często przychodzi zapłacić zbyt wysoką cenę. W wielu przypadkach bardziej praktyczne jest użycie procedury, która tylko **prawie zawsze** daje prawidłową odpo-

wiedź. Często „prawie zawsze" jest wystarczająco dobre. Reguła, która daje zwykle dobrą odpowiedź, ale tego nie gwarantuje, nazywana jest **heurystyką**. Nieraz użycie heurystyki jest bardziej praktyczne niż zastosowanie algorytmu: na przykład dla problemu komiwojażera istnieje wiele skutecznych heurystyk — procedur dających w krótkim czasie rozwiązania prawie optymalne. W istocie te heurystyki zwykle znajdują najlepszą trasę, choć nie ma absolutnej gwarancji, że tak będzie. W realnym życiu komiwojażer prawdopodobnie byłby bardziej zadowolony z dobrej, szybkiej heurystyki, niż z powolnego algorytmu.

Prostym przykładem użycia heurystyki jest gra w szachy. Utalentowany programista, będący tylko przeciętnym szachistą, może napisać program grający w szachy, który systematycznie będzie z nim wygrywał. Taki program nie jest jednak algorytmem, ponieważ nie ma gwarancji, że wygra. Heurystyki dają uzasadnione naukowo przypuszczenia co do prawdopodobnego rozwiązania; dobre heurystyki zwykle dokonują właściwego wyboru. Niektóre z najbardziej spektakularnych zachowań komputerów są raczej wynikiem działania heurystyk niż algorytmów. (Filozofowie napisali wiele bzdur o „ograniczonych możliwościach komputerów", kiedy tak naprawdę myśleli o ograniczonych możliwościach algorytmów.)

Aby napisać dobry program do gry w szachy, wystarczy posłużyć się następującymi regułami heurystycznymi:

1. Oceń względną siłę pozycji każdego z partnerów, licząc figury i piony znajdujące się na planszy.

2. Wykonaj taki ruch, aby po kilku ruchach twoja pozycja była możliwie najmocniejsza.
3. Przyjmij, że twój przeciwnik stosuje strategię podobną do twojej.

Każda z tych reguł jest jedynie przybliżeniem idealnej strategii i można sobie wyobrazić sytuacje, w których prowadzić będzie do błędu. Na przykład, względna siła pozycji gracza zależy nie tylko od liczby figur i pionów, ale też od ich rozmieszczenia. Dobra pozycja może być często korzystniejsza niż dodatkowy pion. Niezależnie od tego, pierwsza reguła jest zwykle poprawna; w większości sytuacji posiadanie przewagi jest lepsze. Jeszcze przed epoką komputerów szachiści opracowali proste metody liczbowego wyrażania względnej siły pozycji graczy, przez przypisanie jednego punktu pionkowi, trzech gońcowi, pięciu punktów wieży itd. — siła pozycji gracza wyrażana jest przez sumę punktów wszystkich jego figur, znajdujących się na szachownicy.

Posługując się tymi heurystykami można napisać program szachowy, który będzie śledził wszystkie możliwe do przyjęcia linie gry dla kilku następnych ruchów. Oczywiście dobrze byłoby, gdyby program brał pod uwagę **wszystkie** linie gry, prawdopodobne czy nie, aż do samego końca gry. Było to proste w grze w kółko i krzyżyk, ale w przypadku szachów jest to niepraktyczne nawet dla najszybszych komputerów. W typowej pozycji w grze środkowej gracz ma do wyboru około trzydziestu sześciu dopuszczalnych ruchów, z których każdy prowadzi do trzydziestu sześciu odpowiedzi. Ponieważ przeciętna partia szachów trwa dłużej niż osiemdziesiąt ruchów, to komputer musiałby przeanalizować w przybliżeniu 36^{80},

czyli około 10^{124} możliwości. Nawet na najszybszych współczesnych komputerach takie przeszukiwanie trwałoby setki lat. Problem polega na tym, że liczba możliwych linii gry rośnie wykładniczo z liczbą ruchów; niepraktyczne jest więc szukanie dalej niż pięć czy dziesięć ruchów do przodu — dlatego właśnie komputery do wyliczania swoich ruchów używają heurystyk.

Przyjmijmy na chwilę, że druga reguła heurystyczna jest poprawna i zgódźmy się, że najlepszą strategią gry jest taka, która optymalizuje pozycję gracza po kilku ruchach w przyszłości. Przyjmijmy dalej, że program sięgać będzie sześć ruchów do przodu. Wtedy, zgodnie z pierwszą regułą heurystyczną, program obliczy siłę każdej ze stron po szóstym ruchu, licząc pozostające na planszy figury i przypisując im wartości zgodnie z opisanym powyżej systemem punktowym. Względna siła obu stron w dowolnej pozycji oceniana będzie na podstawie różnicy między tymi wynikami.

Jaki jest przy tych założeniach najlepszy sposób dla programu na wybranie następnego ruchu? Nie wystarczy, aby komputer wybrał ruch prowadzący do najkorzystniejszej sekwencji sześciu przyszłych ruchów, ponieważ odpowiedzi na te ruchy określane będą przez przeciwnika. Zamiast tego musimy przyjąć, że przeciwnik zawsze będzie próbował wybrać taką linię gry, która będzie faworyzować jego względną pozycję; to założenie zawarte jest w trzeciej regule heurystycznej. Aby przewidzieć ruchy przeciwnika, komputer stawia siebie w pozycji oponenta. Ocenia każdy ruch dopuszczany przez reguły gry, jaki może być przez niego wykonany — powiedzmy, białymi.

Procedura oceny możliwych ruchów białymi zależy od wywołania procedury dla oceny możliwych ruchów, jakich dokonać mogą czarne i vice versa. W efekcie, komputer śledzi wszystkie możliwe sekwencje posunięć przez sześć ruchów, na przemian stawiając się w położeniu czarnych i białych. Program wypróbowuje ruchy na wirtualnej szachownicy w pamięci komputera, w dużym stopniu tak, jak mistrzowie szachowi wyobrażają sobie sekwencje posunięć „w głowie". Programy oceniające pozycje białych i czarnych wywołują siebie nawzajem jako podprogramy w sposób rekurencyjny. Rekurencja kończy się po sześciu krokach, kiedy komputer oblicza wynik, licząc pozostałe na szachownicy figury.

Większość programów szachowych korzysta z dodatkowych reguł heurystycznych, aby przerywać analizę mało prawdopodobnych sekwencji ruchów i głębiej badać sekwencje, które wiążą się z wymianą. Istnieją również bardziej skomplikowane systemy oceny pozycji bez przeszukiwania — na przykład systemy przyznające punkty za posiadanie kontroli nad centrum szachownicy lub za ochronę króla. Każda z tych reguł heurystycznych jest tylko dodatkowym przypuszczeniem i każda może usprawnić przeszukiwanie w pewnych sytuacjach — za cenę możliwości popełniania błędu w innych. Podstawowa procedura przeszukiwania, uzupełniona rozmaitymi udoskonaleniami, leży u podstaw prawie każdego programu szachowego. Jest to skuteczne, ponieważ w ten sposób wykorzystywana jest szybkość komputera w przeszukiwaniu milionów dopuszczalnych sekwencji ruchów. Między tymi wieloma milionami możliwości często trafia się wariant, który za-

skoczy programistę, a czasem nawet doświadczonego szachistę. Ta właśnie zdolność sprawiania niespodzianek pozwala maszynie grać w szachy lepiej od programisty.

Maszyny do gry w szachy mają w informatyce długą i czasami niezbyt chwalebną historię. Osiemnastowieczny węgierski wynalazca Wofgang von Kempelen ściągnął na siebie uwagę całego świata, kiedy zaprezentował automat grający w szachy, mający postać mechanicznego Turka w turbanie. Okazało się jednak, że maszyna działała dlatego, że w środku ukryty jest karzeł-szachista. W roku 1914 hiszpański inżynier Luis Torres y Quevedo zbudował mechaniczne urządzenie, które potrafiło grać w uproszczoną wersję szachów bez pomocy ukrytego człowieka, a w późnych latach czterdziestych Claude Shannon opisał, jak można zaprogramować komputer za pomocą zbioru szachowych reguł heurystycznych, podobnych do reguł przedstawionych powyżej. Upłynęło jednak wiele lat, zanim komputery stały się wystarczająco szybkie, aby przyzwoicie grać w szachy, co zresztą było na rękę wielu filozofom, którzy argumentowali, że gra w szachy jest przykładem unikalnych możliwości ludzkiego umysłu. Współczesne komputery, wykorzystujące te same heurystyki, mogą teraz pokonać najlepszych szachistów na świecie (czego przykładem może być zwycięstwo w 1997 roku komputera IBM Deep Blue nad Garym Kasparowem), filozofowie musieli więc poszukać innych przykładów.

Proste heurystyki dla przeszukiwań dają dobre wyniki, ponieważ dla każdego ruchu trzeba rozważyć względnie niedużą liczbę odpowiedzi. W warca-

bach, gdzie jest jeszcze mniej odpowiedzi na każde posunięcie, maszyny oparte na heurystykach zaczęły pokonywać najlepszych graczy już w latach sześćdziesiątych. Z drugiej strony, w chińsko-japońskiej grze w go ludzie ciągle wygrywają z maszynami, ponieważ większa plansza (19×19) dopuszcza znacznie więcej możliwych ruchów. (Wolę go niż szachy, dokładnie z tego względu, że przeszukiwania możliwych ruchów są w tym przypadku mniej użyteczne; przez to moja niecierpliwość staje się mniej dokuczliwą wadą.)

ROZKŁADY ZDATNOŚCI

Wykorzystywanie heurystyk do przeszukiwania zbioru możliwości jest bardzo rozpowszechnione w programowaniu komputerowym i ma zastosowania znacznie poważniejsze niż gry. Często właśnie w ten sposób komputery znajdują „twórcze" rozwiązania problemów — zwykle problemów, o których wiadomo, że ich rozwiązaniach mieszczą się w dużym, ale skończonym zbiorze możliwości, zwanym **przestrzenią możliwości.** Przestrzeń możliwości w szachach jest zbiorem wszystkich możliwych sekwencji ruchów; przestrzeń możliwości w przypadku problemu komiwojażera składa się ze wszystkich możliwych tras, łączących miasta z listy komiwojażera. Ponieważ przestrzenie te są zbyt duże, by je całkowicie przeszukać, do ograniczenia zakresu poszukiwań używa się heurystyk. W przypadku małych przestrzeni poszukiwań, takich jak w grze w kółko i krzyżyk, lepsze jest dokładne prze-

szukiwanie, ponieważ gwarantuje znalezienie właściwej odpowiedzi.

Ogólnie biorąc, przestrzeń możliwości jest duża, ponieważ możliwe warianty otrzymywane są przez tworzenie kombinacji prostszych elementów — pojedynczych ruchów w szachach, dróg z jednego miasta do drugiego w problemie komiwojażera. Te kombinacje elementów prowadzą do **kombinatorycznej eksplozji** możliwości — wykładniczego wzrostu liczby możliwości wraz ze wzrostem liczby elementów podlegających łączeniu. Ponieważ możliwe warianty budowane są z kombinacji elementów, w przestrzeni możliwości istnieje pewien odpowiednik odległości; kombinacje posiadające elementy wspólne są „bliższe" niż kombinacje, które takich wspólnych elementów nie posiadają. Z tego też powodu mówi się o „przestrzeni", a nie tylko zbiorze możliwości. Aby rozszerzyć tę analogię, wyobraźmy sobie, że możliwym rozwiązaniom przyporządkowana jest powierzchnia, zwana czasami **rozkładem zdatności**. Ocena atrakcyjności każdego możliwego rozwiązania reprezentowana jest przez odległość punktu na powierzchni od poziomu odniesienia. Jeśli podobne możliwości mają zbliżone oceny, to sąsiadujące punkty leżeć będą na podobnej wysokości, tak więc na powierzchni tej występować będą wyraźne szczyty i doliny. W tej analogii znajdowanie najlepszego rozwiązania podobne jest do poszukiwania najwyższego szczytu. Biorąc jako przykład problem komiwojażera, możemy wyobrazić sobie, że każdy punkt na powierzchni reprezentuje pewien plan podróży. Wysokość, na jakiej znajduje się punkt, odpowiada odległości, jaką komiwojażer musi pokonać, przy

czym punkty odpowiadające bardziej ekonomicznym podróżom znajdują się na większej wysokości. Najlepszy plan podróży odpowiadać będzie szczytowi najwyższego wzgórza.

Jednym z najprostszych sposobów przeszukiwania takiej przestrzeni jest porównywanie losowo wybranych punktów i zapamiętywanie najlepszego punktu ze zbadanych. Liczba punktów, które mogą zostać przeszukane w ten sposób, jest ogólnie biorąc ograniczona tylko przez maksymalny czas, jaki trwać mogą poszukiwania, a cała procedura może zostać zastosowana do przestrzeni dowolnego typu. Jest ona równoważna zrzuceniu skoczków spadochronowych w różnych miejscach na powierzchni i poproszeniu ich o przekazanie raportu o wysokości, na jakiej się znaleźli. Nie jest to bardzo wydajny sposób znajdowania szczytu góry. Jeśli przestrzeń jest duża, to jedynie skromny ułamek wszystkich możliwości może zostać zbadany, a zatem niewielka jest szansa, że najlepszy spośród zbadanych punktów będzie tym leżącym najwyżej.

W przypadku, kiedy przestrzeń możliwości podobna jest do przestrzeni w problemie komiwojażera, gdzie jest bardzo prawdopodobne, że leżące blisko siebie punkty mają zbliżone właściwości, zwykle lepiej jest używać procedury przeszukującej ścieżkę w przestrzeni metodą przemieszczania się od jednego punktu do innego punktu, leżącego w pobliżu. Najlepszą metodą na znalezienie szczytu w górzystym otoczeniu jest maszerowanie pod górę — odpowiadającą temu heurystyką jest wybranie najlepszego z rozwiązań znalezionych w bliskim otoczeniu w przestrzeni możliwości. Na przykład, w problemie

komiwojażera komputer może zmieniać najlepsze z dotychczas znalezionych rozwiązań przez modyfikację w planie podróży kolejności odwiedzin dwóch miast. Jeśli ta zmiana prowadzi do bardziej efektywnego rozkładu podróży, to jest akceptowana jako lepsze rozwiązanie (krok pod górę); w innym przypadku jest odrzucana i wypróbowywany jest kolejny wariant. W tej metodzie przeszukiwania będziemy wędrować po przestrzeni, zawsze zmierzając w górę, aż osiągniemy szczyt wzgórza. W tym punkcie rozwiązanie nie może być już ulepszone przez zamianę kolejności odwiedzin dla jakiejkolwiek pary miast.

Słabością tej metody, zwanej **metodą wspinaczki**, jest to, że choć zawsze osiąga się szczyt, nie jest to koniecznie najwyższa góra w otoczeniu. Metoda wspinaczki jest heurystyką, a nie algorytmem. Istnieją inne heurystyki, podobne do metody wspinaczki, przy których jest mniej prawdopodobne, że utknie się na jakimś mniejszym wzgórzu. Można na przykład powtarzać proces wspinaczki wiele razy, startując z losowo wybranych położeń (możemy poprosić spadochroniarzy, aby szli pod górę). Można też od czasu do czasu robić krok w dół, aby nie utknąć w miejscu. Istnieje wiele odmian tej metody, każda ma jakieś zalety i wady.

Takie heurystyki jak metoda wspinaczki działają dobrze w przypadku problemu komiwojażera i dają dobre odpowiedzi w krótkim czasie. Nawet gdy w grę wchodzą tysiące miast, zwykle możliwe jest znalezienie dobrego rozwiązania problemu przez rozpoczęcie od rozsądnej hipotezy i poprawianie jej za pomocą metody wspinaczki. Dlaczego więc problem komiwojażera uważany jest za trudny? Używając heurystyk,

możemy **prawie** zawsze uzyskać **prawie** najlepszy plan podróży. Ale metoda, która działa **prawie** zawsze nie jest algorytmem. Co pewien czas dużo szumu robi kolejny badacz, który „rozwiązał" problem komiwojażera; jak dotąd, wszystko, co kiedykolwiek udało się uzyskać, to nowe heurystyki. Nietrudno jest wymyślić szybkie rozwiązanie heurystyczne dla problemu komiwojażera; prawdziwym problemem jest znalezienie szybkiego algorytmu.

Istnieje wiele problemów, dla których nie potrzebujemy za każdym razem najbardziej poprawnej odpowiedzi; problemów, dla których możemy zaakceptować rozwiązania odległe od ideału. A nawet wtedy, kiedy potrzebujemy idealnego rozwiązania, nie zawsze nas na nie stać. W przypadku takich problemów komputer może dostarczyć naukowo uzasadnione i dobrze przebadane hipotezy. Ponieważ komputer może zbadać gigantyczne liczby kombinacji i możliwości, takie hipotezy często zaskakują programistę. Kiedy maszyna używa heurystyk, to może sprawiać nam niespodzianki, ale może również popełniać błędy — co czyni komputer choć trochę podobnym do człowieka i mniej podobnym do automatu.

Rozdział szósty

PAMIĘĆ: INFORMACJA I TAJNE SZYFRY

Do tej pory pomijaliśmy na ogół ograniczenia komputera wynikające z wielkości pamięci. Wyidealizowany komputer uniwersalny ma pamięć nieskończenie wielką, ale pamięć komputera rzeczywistego jest ograniczona, najczęściej ze względu na koszty. Dopóki pamięć jest wystarczająca do wykonania danego zadania, możemy to ograniczenie pominąć, ale pewne algorytmy i programy użytkowe intensywnie ją wykorzystujące przechowują tak duże ilości danych, że wielkość dostępnej pamięci staje się ważnym czynnikiem. Programy użytkowe, operujące reprezentacjami świata fizycznego — obrazami, dźwiękami czy trójwymiarowymi modelami — często wymagają dużej pamięci. Aby określić, czy komputer jest wystarczająco mocny, by poradzić sobie z danym zadaniem i oszacować czas potrzebny na przetworzenie informacji, trzeba wiedzieć, ile pamięci potrzebuje dana aplikacja.

Bit — jednostka miary informacji — nadaje się zarówno do przesyłania informacji, jak i jej przechowywania. W pewnym sensie, przesyłanie i przechowy-

wanie to dwa aspekty tej samej operacji: można powiedzieć, że przechowując informację „przesyłamy" ją w czasie. Ta równoważność między przesyłaniem i przechowywaniem może wydawać się dziwna, o ile nie przywykliście do myślenia w kategoriach czterowymiarowej czasoprzestrzeni, ale pomyślcie o wysyłaniu listu jako sposobie komunikowania się, który wiąże się z oboma aspektami. Wysyłanie listu do kogoś jest sposobem przemieszczania informacji w przestrzeni; wysłanie listu do siebie samego jest sposobem na przechowanie informacji. Jeśli bliżej się temu przyjrzeć, każda forma komunikowania się zawiera zarówno aspekt przestrzenny, jak i czasowy. Jednym ze sposobów przechowywania informacji przez komputery elektroniczne jest nieustanne przesyłanie jej — jest to elektroniczny odpowiednik wysyłania listów do samego siebie.

Wiemy, że komputery z n bitami pamięci mogą przechowywać do n bitów informacji, ale jak określimy, ile bitów potrzeba do reprezentowania danej porcji informacji? Na przykład, ile bitów zawartych jest w słowach tej książki? Obliczenie odpowiedzi okazuje się niełatwe; w istocie, jest kilka poprawnych odpowiedzi. Rozważania nad tym pytaniem prowadzą nas do takich koncepcji jak kompresja danych, wykrywanie i poprawiane błędów, liczby losowe i tajne szyfry.

Liczba bitów, potrzebnych do tego, aby przesłać lub przechować jakikolwiek fragment danych zależy od sposobu ich zakodowania. Jednym ze sposobów reprezentowania złożonych zbiorów danych (jak chociażby tekst tej książki), jest przedstawianie ich jako ciągu prostszych elementów: na przykład, ciągu

znaków pojawiających się w tekście. W tej powszechnie stosowanej reprezentacji liczba bitów w zbiorze danych równa jest liczbie znaków pomnożonej przez liczbę bitów przypadającą na znak. Tekst tej książki zawiera około 250 000 znaków, zaś mój komputer stosuje systemu kodowania, wymagający 8 bitów (jeden bajt) do zapisania jednego znaku, tak więc wielkość pliku, w którym komputer przechowuje tekst, to około 2 milionów bitów. Nasuwa się wniosek, że ta książka zawiera około 2 milionów bitów informacji, ponieważ tyle pamięci potrzebuje komputer, aby przechować tekst, ale jest to tylko jedna z miar informacji — miara, która zależy od sposobu reprezentacji przekazu. Jest użyteczna, ponieważ mówi nam nie tylko, ile pamięci potrzebuje komputer, aby przechować informacje, ale również ile czasu zajmie ten proces. Jeśli na przykład wiem, że mój komputer może zapisywać informacje na dysku z prędkością 20 milionów bitów na sekundę, i wiem też, że korzysta z reprezentacji tej książki o wielkości 2 milionów bitów, to mogę obliczyć, że zapisanie tej książki w pliku na dysku zajmie około 1/10 sekundy.

Mierzenie liczby bitów za pomocą liczby znaków pomnożonej przez 8 ma tę wadę, że zależne jest od reprezentacji zastosowanej w komputerze. Inny komputer lub inny program użytkowy, działający na tym samym komputerze, może do przechowywania dokładnie tego samego ciągu znaków wymagać innej liczby bitów. Na przykład, przy 8 bitach na znak możliwe jest przedstawienie 256 różnych znaków, ale w angielskim tekście tej książki pojawiają się mniej niż 64 różne znaki — po 26 dużych i małych liter oraz cyfry i znaki interpunkcyjne. Tak więc, bardziej wy-

dajny kod mógłby reprezentować każdy znak za pomocą 6 bitów informacji ($2^6 = 64$), a tym samym zredukować reprezentację tekstu do jedynie 1,5 miliona bitów.

Byłoby bardzo dobrze mieć taką miarę informacji, która nie zależałaby od sposobu reprezentacji. Bardziej fundamentalną miarą informacji byłaby **minimalna** liczba bitów, za pomocą której można reprezentować przekaz. Łatwo to pojęcie zdefiniować, ale trudniej jest je obliczyć.

KOMPRESJA

Na ile możemy skompresować dany tekst bez utraty informacji? Prostą formą kompresji jest zredukowanie liczby bitów na znak z 8 do 6. Inne metody kompresji wykorzystują rozmaite prawidłowości występujące w tekście. Na przykład litery T i E w tekstach angielskich występują znacznie częściej niż litery Q i Z. Bardziej wydajny kod wykorzystywałby krótsze ciągi jedynek i zer dla reprezentowania częściej używanych liter. Zapisywanie znaków za pomocą kodu o zmiennej długości (kodu nierównomiernego) w celu osiągnięcia bardziej zwartej reprezentacji było chwytem wykorzystywanym przez pierwszych telegrafistów i radioamatorów. W alfabecie Morse'a litera E reprezentowana jest przez pojedynczą kropkę, a T przez pojedynczą kreskę. Rzadziej występujące litery, takie jak Q i Z, reprezentowane są ciągami liczącymi do czterech kropek i kresek. Ponieważ trzeci rodzaj sygnału — przerwa — używany jest do oznaczania końca liter, to kropki

i kreski alfabetu Morse'a nie odpowiadają dokładnie 1 i 0, ale zasada jest podobna.

Aby stworzyć kod nierównomierny za pomocą sygnałów 1 i 0, konieczny jest staranny wybór symboli kodu, by można było jednoznacznie wydzielać z niego znaki. Jest to możliwe, dopóki żaden ciąg bitów używany do reprezentowania znaku nie zaczyna się sekwencją sygnałów 1 i 0, reprezentującą inny znak. Na przykład, wszystkie często występujące znaki mogą być reprezentowane przez 4 bity i odpowiadające im ciągi powinny zaczynać się od 1, podczas gdy rzadziej występujące znaki mogłyby mieć 7 bitów i zaczynać się od 0. To umożliwiłoby jednoznaczny podział ciągu bitów na krótkie i długie znaki. Wybór kodu nierównomiernego, wykorzystującego fakt występowania różnych liter z różnymi częstościami, pozwoli na znaczną kompresję tekstu. W przypadku tekstu tej książki zredukowałoby to liczbę bitów z początkowych 2 milionów do około 1 miliona, czyli o około 50%.

Każda metoda kompresji wykorzystuje prawidłowości w danych. Opisany powyżej kod bazuje na częstości występowania pojedynczych znaków, ale są też i inne prawidłowości, którymi można się posłużyć. Na przykład, w języku angielskim nie wszystkie pary sąsiadujących liter pojawiają się z tą samą częstością. Po literze Q prawie zawsze następuje U, a po literze Z nigdy nie pojawia się K. Tworząc kod nierównomierny raczej dla par, a nie dla pojedynczych liter, możemy wykorzystać fakt, że kombinacje dwuliterowe występują z różną częstością. Kod taki może wykorzystywać krótsze sekwencje bitów dla bardziej powszechnych par i dłuższe dla par pojawiających się

rzadziej. Jeśli posłużymy się tą metodą, to liczba bitów potrzebna do zapisania tekstu tej książki może zostać zredukowana o dalsze 10 procent, do przeciętnie 3,5 bitu na literę.

Jeszcze bardziej wydajnym kodem byłoby wykorzystanie prawidłowości pojawiających się w dłuższych ciągach liter. Na przykład słowo „*the*" występuje w angielskim tekście tej książki około 3000 razy. Byłoby korzystne posłużenie się kodem, który używa względnie krótkiej sekwencji bitów do reprezentowania całego tego słowa. Jest też wiele innych słów, jak „*computer*" i „*bit*", które pojawiają się w mojej książce tak często, że również warte są specjalnego zakodowania. Istnieją także prawidłowości wykraczające poza statystyczne związki w ciągach znaków, na przykład gramatyczne, w strukturze zdań i interpunkcji — pozwalają one na dalszą kompresję tekstu. Ale od pewnego momentu nasze zyski będą maleć. Ostatecznie, kompresja wykorzystująca najlepsze dostępne metody statystyczne, prawdopodobnie osiągnie przeciętną wielkość reprezentacji mniejszą niż 2 bity na znak — około 25% standardowej ośmiobitowej reprezentacji.

Kompresja daje dość dobre wyniki w przypadku tekstu, ale działa jeszcze lepiej dla sygnałów, które są reprezentacjami realnego świata, jak dźwięki czy obrazy. Sygnały te są zwykle wczytywane do komputera poprzez proces znany jako **konwersja analogowo-cyfrowa.** Dane wejściowe — natężenie dźwięku lub, powiedzmy, natężenie światła — to zwykle sygnały zmieniające się w sposób ciągły. Na przykład punkt — lub inaczej **piksel** — w czarno-białej fotografii może być biały, czarny lub mieć

dowolny odcień szarości pomiędzy tymi skrajnościami. Ponieważ komputer nie może stworzyć reprezentacji dla nieskończonej liczby możliwości, upraszcza sygnał, redukując każdy piksel do jednego ze skończonej liczby poziomów szarości. Zwykle liczba gradacji jest dokładną potęgą 2, aby dobrze pasowała do liczby bitów w kodzie. Na przykład, jasność kropki w obrazie czarno-białym może być reprezentowana przez 8 bitów, co pozwala na przedstawienie 256 odcieni szarości. Obraz wyższej jakości reprezentowany będzie za pomocą kodu 12-bitowego, który daje 4096 odcieni szarości. Obraz kolorowy może wykorzystywać 24 bity na kropkę — 8 bitów dla intensywności każdej z trzech barw podstawowych.

Innym parametrem określającym jakość obrazu fotograficznego jest **rozdzielczość** — czyli liczba pikseli użyta do jego reprezentacji. Obraz o wysokiej rozdzielczości wytworzony przez układ 1000×1000 kropek będzie bardziej wierną reprezentacją niż obraz o rozdzielczości 100×100. Ponieważ jednak reprezentacja obrazu o wysokiej rozdzielczości wykorzystuje 1 000 000 pikseli zamiast 10 000, to wasz komputer może potrzebować 100 razy więcej pamięci do jego przechowania, a przetwarzanie obrazu trwać będzie 100 razy dłużej. Jakość kosztuje.

Ponieważ obrazy o wysokiej rozdzielczości zawierają dużą liczbę bitów, często pożądane jest ich skompresowanie w celu zredukowania kosztów przechowywania i przesyłania. Jest to szczególnie istotne w przypadku obrazów ruchomych, które zwykle zawierają od 24 do 100 klatek na sekundę. Na szczęście, obrazy stosunkowo łatwo poddają się kompresji, ponieważ występuje w nich wiele prawidłowości.

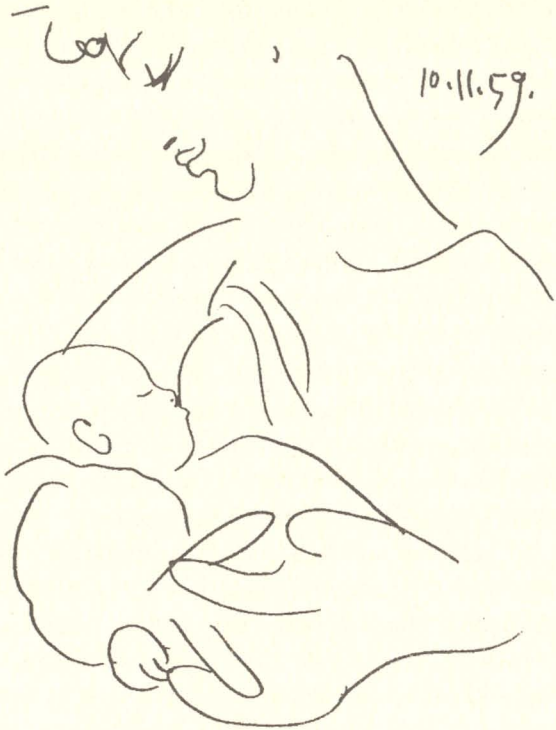
W większości przypadków natężenie i barwa określonego piksela są niemal identyczne z natężeniem i barwą sąsiedniego. Dwa piksele w obrazie twarzy, reprezentujące sąsiadujące ze sobą fragmenty (na przykład policzka), z dużym prawdopodobieństwem będą bardzo podobne pod względem jasności i barwy. Większość algorytmów do kompresowania obrazów wykorzystuje to podobieństwo. Algorytm do kompresowania obrazów może reprezentować obszary o takiej samej jasności i barwie za pomocą kilku zaledwie liczb opisujących barwę i rozmiar. Inne metody kompresji obrazów wykorzystują bardziej złożone prawidłowości: na przykład podobne tekstury w różnych częściach obrazu. W przypadku obrazów ruchomych, takich jak audycje telewizyjne, metody kompresji zwykle wykorzystują podobieństwo następujących po sobie klatek. Za pomocą tej techniki często można skompresować reprezentację fotografii o czynnik 10, a obrazu ruchomego o czynnik 100. Podobne metody kompresji mogą zostać zastosowane do dźwięków.

Tego typu metody kompresji prowadzą do mało intuicyjnego wyobrażenia o ilości informacji zawartej w obrazie. Jeśli minimalna liczba bitów niezbędna do reprezentowania obrazu uznana zostanie za miarę ilości informacji, oznaczałoby to, że obraz, który łatwo skompresować, zawiera mniej informacji. Obraz twarzy, na przykład, zawierałby mniej informacji niż obraz stosu kamyków na plaży, ponieważ w przypadku sąsiednich pikseli w obrazie twarzy bardziej prawdopodobne jest, że będą podobne. Sterta kamieni wymaga zarejestrowania i przechowania większej ilości informacji, mimo że oglądający ją

człowiek mógłby uznać, że jest wprost przeciwnie. Według tej miary, obrazem zawierającym najwięcej informacji byłby obraz zupełnie losowego zbioru pikseli, takich jak zakłócenia na ekranie zepsutego telewizora. Jeśli kropki w obrazie nie są ze sobą skorelowane, to nie ma prawidłowości, którą można skompresować. Takie obrazy wydają się nam zupełnie bezsensowne — i mogą naprawdę być bezsensowne — ale ich komputerowa reprezentacja wymaga największej ilości informacji.

Miara informacji oparta na minimalnej reprezentacji nie odzwierciedla dobrze naszej intuicji, ponieważ komputer nie odróżnia informacji sensownej od bezsensownej. Komputer musi reprezentować barwę każdego piksela lub położenie każdego kamyka na plaży, nawet jeśli te szczegóły nie są istotne dla obserwatora. Decydowanie, jaka informacja jest sensowna, a jaka nie, jest subtelną sztuką; zależy to od sposobu, w jaki wykorzystywany jest obraz, i od tego, kto z niego korzysta. Występowanie maleńkiej plamki na zdjęciu rentgenowskim może być trudno zauważalne dla niewprawnego obserwatora, ale bardzo istotne dla lekarza. Wielki artysta, na przykład Picasso, potrafi „skompresować" obraz złożonej sceny do kilku prostych linii, ale to skomplikowany proces decyzyjny prowadzi do wyboru najbardziej znaczących aspektów obrazu (patrz rys. 23).

Gdyby komputer potrafił skompresować obraz poprzez wybór jedynie znaczącej informacji, to liczba bitów w reprezentacji byłaby bliższa naszemu zdroworozsądkowemu odczuciu co do jej ilości. Komputer mógłby na przykład reprezentować losowy układ pikseli, zaznaczając, że ten obraz nie wykazuje żad-



RYSUNEK 23.
Szkic Picassa

nej prawidłowości i tym samym nie niesie żadnej sensownej informacji. Poproszony o odtworzenie obrazu, generowałby po prostu inny równie losowy układ pikseli. Szczegóły — chociażby odcień poszczególnych punktów — w odtworzonym obrazie byłyby inne niż w oryginale, ale dla ludzkiego oka te różnice nie miałyby znaczenia.

Wiele algorytmów do kompresji obrazów i dźwięków eliminuje pewne nieistotne informacje, w celu

zredukowania rozmiarów reprezentacji. W tych tak zwanych **algorytmach kompresji ze stratami** przyjmuje się, że pewne szczegóły w obrazie i dźwiękach są nierozpoznawalne dla oka czy ucha. Ogólnie biorąc, metody kompresji ze stratami stosowane są wtedy, kiedy wiadomo, że odtworzona informacja służyć będzie jakimś szczególnym potrzebom. Jeśli na przykład pewien szczegół pojawia się jedynie na pojedynczej klatce filmu, to można go bez ryzyka wyrzucić, ponieważ i tak nie zostanie dostrzeżony.

Istnieje jeszcze jedna ważna forma reprezentacji obrazu, dzięki której można osiągnąć nawet większy stopień kompresji, niż za pomocą metod opisanych powyżej. Jeśli proces generujący oryginalne obrazy jest znany, to korzystniejsze może być zachowanie raczej zapisu tego procesu niż samego obrazu. Jeśli na przykład obraz jest rysunkiem, który powstał w wyniku narysowania ciągu linii, to taki obraz może być reprezentowany przez zapisanie listy linii — ten schemat reprezentacji jest często wykorzystywany w programach komputerowych wykonujących proste szkice.

Idea reprezentowania jakiegoś obiektu przez zapisanie procedury lub programu, który dany obiekt wygenerował, stosuje się też do innych typów danych, takich jak dźwięki. W przypadku dźwięków może się wydawać, że technika ta nie jest niczym bardziej wyrafinowanym niż zapisywanie utworów muzycznych za pomocą nut, ale w komputerze odpowiedni znak może zawierać wszystkie szczegóły niezbędne do odtworzenia oryginału — strój instrumentów, technikę użycia smyczków, a nawet nastrój orkiestry. Jeśli jakiś obiekt może być wygenerowany

przez komputer, to z definicji istnieje precyzyjna procedura, pozwalająca na wygenerowanie go — jej opis może posłużyć jako reprezentacja obiektu.

Prowadzi to nas do kolejnej miary informacji: **Ilość informacji w układzie bitów równa jest długości najmniejszego programu komputerowego, który jest w stanie wygenerować ten układ.** Tę definicję informacji można stosować niezależnie od tego, czy układ bitów przedstawia obraz, dźwięki, tekst, liczbę czy cokolwiek innego. Jest ona tak interesująca, ponieważ dopuszcza dowolny rodzaj regularności w układzie bitów. W szczególności, obejmuje wszystkie metody kompresji opisane powyżej. (Może się wydawać, że taka definicja zależy od szczegółów języka maszynowego komputera, ale pamiętajmy, że każdy komputer może symulować jakikolwiek inny, tak więc różnice pomiędzy typami maszyn związane będą jedynie z krótkim programem, niezbędnym do przeprowadzenia symulacji.)

Kiedy już ciąg znaków tworzących informacje zostanie skompresowany tak bardzo, jak to jest możliwe, nie będzie wykazywał prawidłowości. Jest to oczywiste, ponieważ jakakolwiek regularność stwarzałaby możliwość dalszej kompresji. Ciąg jedynek i zer, przedstawiający optymalnie skompresowany tekst, miałby charakter całkowicie losowy, jak zapis rzutu monetą. W istocie, wielu matematyków przyjmuje tę własność — niemożliwość dalszej kompresji — za definicję przypadkowości. Jest to definicja dość prosta, choć niezbyt użyteczna w praktyce, ponieważ bardzo trudno jest określić, czy dany ciąg bitów jest losowy w tym sensie. Jeśli już odkryjemy jakąś prawidłowość, to łatwo jest stwierdzić, że ciąg może

zostać skompresowany — ale nie możemy udowodnić, że ciąg **nie może** zostać skompresowany, jeśli nie widzimy żadnej prawidłowości. Ciągi liczb pseudolosowych opisane w rozdziale 4. są dobrym przykładem sekwencji, które wydają się losowe, ale kryje się za nimi pewna prawidłowość, czyli algorytm, za pomocą którego zostały wygenerowane. Według powyższej definicji przypadkowości są one w wysokim stopniu nielosowe, ponieważ bardzo długa sekwencja może zostać opisana bardzo krótko, po prostu przez podanie algorytmu, który ją wytworzył — w tym przypadku algorytmu symulacji koła ruletki.

SZYFROWANIE

Ciągi, które wydają się losowe, ale kryją się za nimi jakieś prawidłowości, mogą zostać użyte do szyfrowania danych. Wyobraźmy sobie na przykład, że chcę przesłać poufny list przyjacielowi. Jeśli obaj mamy kopie tego samego generatora liczb losowych, tak że obaj możemy wytworzyć te same ciągi liczb losowych, to możemy posłużyć się nimi do ukrycia zawartości listu przed kimś, kto mógłby go przechwycić. List jest ciągiem bitów reprezentujących litery. Przyjmijmy, że używamy standardowej reprezentacji, w której jeden znak reprezentowany jest przez 8 bitów. Ta standardowa reprezentacja może prawdopodobnie zostać zinterpretowana i zrozumiana przez każdego, kto ma dostęp do naszej korespondencji; jest to coś, co kryptografowie nazywają **tekstem otwartym.** Aby zaszyfrować list, łączymy każdy bit w otwartym tekście z odpowiadającym mu bitem ciągu bitów pseudolosowych. Jeśli pseudoloso-

wym bitem jest 1, to zamieniamy odpowiadający mu bit tekstu otwartego na przeciwny. Jeśli pseudolosowy bitem jest 0, to nie wprowadzamy zmian. Spowoduje to zamianę około połowy bitów w otwartym tekście, ale poza adresatem nikt nie będzie wiedział, której. Otrzymany w ten sposób ciąg sygnałów 1 i 0 będzie dla zewnętrznego obserwatora całkowicie niezrozumiały, chyba że zna on zastosowany ciąg liczb pseudolosowych. Z drugiej strony, mój przyjaciel dokładnie wie, w jaki sposób utworzyć taki sam pseudolosowy ciąg, który posłuży do przywrócenia zamienionych bitów, dzięki czemu list zostanie odtworzony (rozszyfrowany). Ta metoda, lub coś bardzo do niej podobnego, leży u podstaw większości systemów kryptograficznych.

Szyfrowanie przypomina wysyłanie listów w zamkniętej skrzynce, którą można otworzyć jedynie za pomocą specjalnego klucza. W opisanej powyżej metodzie kluczem jest generator liczb losowych. Każdy, kto ma klucz, jest w stanie rozszyfrować list. W podanym przykładzie ten sam klucz używany jest do szyfrowania i rozszyfrowywania, ale możliwe jest również skonstruowanie kodów, w którym klucze te różnią się; wówczas, nawet jeśli ktoś zdobędzie klucz do szyfrowania, nie będzie automatycznie znał klucza dekodującego. Jeśli na przykład chcę, by wysłano do mnie zaszyfrowaną wiadomość, mogę opublikować opis klucza potrzebnego do szyfrowania listów do mnie. Każdy będzie w stanie wysłać do mnie poufny list, niezależnie od tego, czy jest moim znajomym, czy nie. Ponieważ klucz publiczny mówi nadawcy jedynie jak szyfrować list, a nie jak go rozszyfrowywać, to inne osoby nie będą w stanie odczytać zako-

dowanego listu. Jedynie mój prywatny klucz, który trzymam w tajemnicy, pozwala na przekształcanie zakodowanych listów z powrotem w otwarty tekst. Jest to tak zwany **system kryptograficzny z kluczem publicznym** (**PKC** — *Public Key Cryptography*). System ten rozwiązuje ważny problem praktyczny: na przykład wiele firm akceptujących płatność za pomocą kart kredytowych przez Internet publikuje swoje własne klucze publiczne, tak by klienci mogli zaszyfrować numery swoich kart bez obawy, że zostaną one przechwycone i odczytane.

System kryptograficzny z kluczem publicznym może być także zastosowany do uwierzytelniania przesyłanych informacji. Aby to osiągnąć, publikuję klucz do rozszyfrowywania korespondencji, ale utajniam mój klucz do szyfrowania. Kiedy chcę wysłać wiadomość, którą pragnę „podpisać", aby było wiadomo, że pochodzi ode mnie, szyfruję wiadomość za pomocą prywatnego klucza. Wszyscy adresaci mojego listu mogą go rozszyfrować za pomocą klucza publicznego. Będą też pewni, że list pochodzi naprawdę ode mnie, ponieważ tylko ktoś, kto zna mój prywatny klucz, mógłby tak go zaszyfrować.

WYKRYWANIE BŁĘDÓW

Kodowanie i dekodowanie bitów ma wiele innych zastosowań poza kompresją i zabezpieczaniem danych. Na przykład są sytuacje, w których zapisujemy przekaz za pomocą większej liczby bitów, niż to jest niezbędne, w celu zredukowania prawdopodobieństwa wystąpienia błędu. Kody, w których zastosowana jest jakaś forma redundancji w celu wykrycia błę-

dów powstałych podczas transmisji — na przykład, że 0 zostało zarejestrowane jako 1 — nazywane są **kodami detekcyjnymi.** Inne kody, zwane **kodami korekcyjnymi**, zawierają wystarczająco wiele nadmiarowej informacji, aby wykryć i poprawić takie błędy.

Oczywistą formą redundancji jest przesyłanie informacji więcej niż jeden raz, co pozwala na wykrycie błędów. Jeśli przesłane dwie kopie tej samej informacji różnią się od siebie, to w czasie przesyłania musiał nastąpić jakiś błąd. Zastosowanie prostego kodu **korekcyjnego** wymaga przesłania informacji trzykrotnie. Zakładając, że tylko w jednej informacji wystąpiły zakłócenia, adresat mógłby odtworzyć poprawną wersję, wybierając te dwie kopie, które są identyczne.

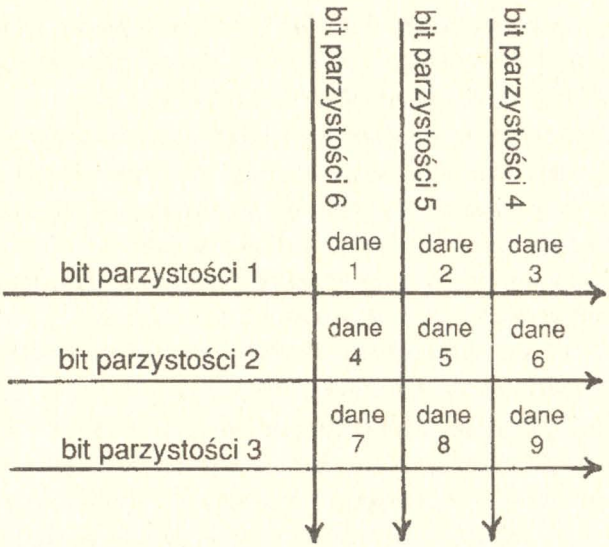
Na szczęście istnieją kody detekcyjne i kody korekcyjne, funkcjonujące przy znacznie mniejszej redundancji. Często stosowanym systemem detekcji błędów jest **kod kontroli parzystości.** System ten pozwala wykrywać jednobitowy błąd w wiadomości o dowolnej długości przez dodanie jednego nadmiarowego bitu. Jako szczególny przykład kodu kontroli parzystości rozważmy 8-bitowy kod, często używany do przesyłania znaków liniami telekomunikacyjnymi z dużymi zakłóceniami. Ósmy bit, zwany **bitem parzystości**, jest 1 wtedy i tylko wtedy, gdy liczba jedynek w siedmiu pozostałych bitach jest parzysta. Oznacza to, że liczba jedynek w 8-bitowym ciągu powinna zawsze być nieparzysta. Jeśli zakłócenie w linii przesyłowej spowoduje zamianę 1 na 0 lub na odwrót, to 8-bitowa wiadomość odebrana przez adresata mieć będzie parzystą liczbę jedynek. Będzie więc

wiadomo, że nastąpił błąd. Podobne systemy kontroli parzystości używane są do wykrywania błędów w układach pamięci komputerów. Jeden bit parzystości pozwala na wykrycie błędu w wiadomości zawierającej dowolną liczbę bitów. Ograniczeniem tego prostego kodu kontroli jest to, że dobrze wykrywa jedynie pojedynczy błąd. Informacja, w której zmienione zostały dwa bity, mieć będzie poprawną parzystość, mimo że dane zawierają błąd.

Wykrywanie błędów wielokrotnych możliwe jest przy zastosowaniu wielu bitów parzystości. Możliwe jest też przekazanie odbiorcy informacji wystarczającej nie tylko do wykrycia błędu, ale też do jego poprawienia; może on wtedy zrekonstruować oryginalną wiadomość, mimo że wystąpił błąd. Przykładem takiego kodu jest dwuwymiarowy kod kontroli parzystości pokazany na rys. 24.

Kod ten zawiera 9 bitów przekazywanej informacji i 6 bitów parzystości. Bity zawierające informację ustawione są w 3 rzędy po 3 bity każdy. Na każdy wiersz i każdą kolumnę przypada jeden bit parzystości. Błąd w wiadomości polegający na zamianie jednego bitu powoduje odkrycie dwóch naruszeń parzystości: jednego w wierszu, drugiego w kolumnie. Odbiorca wiadomości będzie wtedy wiedział, że bit na przecięciu wadliwego wiersza i wadliwej kolumny jest błędny i powinien zostać zamieniony. Jeśli błąd nastąpi w transmisji jednego z bitów parzystości, to albo wiersz, albo kolumna mieć będą niewłaściwą parzystość, ale nie oba naraz. Bity przedstawione są w układzie dwuwymiarowym, aby pomóc w wizualizacji struktury kodu, ale mogą być przesyłane w dowolnym porządku. Takie kody ko-

rekcyjne często używane są do ochrony każdego słowa w pamięci dużych komputerów. Za pomocą podobnych technik można skonstruować wiele innych kodów, które będą wykrywać i/lub poprawiać rozmaite typy błędów.



PRZYKŁAD

	1	0	1
1	1	0	1
1	0	1	1
0	1	0	0

RYSUNEK 24.

Kod korekcyjny z 9 bitami danych i 6 bitami kontrolnymi

Posługując się kodami korekcyjnymi potrafimy poradzić sobie z sygnalizowaniem błędów, zdarzających się podczas przesyłania i przechowywania informacji, ale co z błędami w samych obliczeniach? Okazuje się, że możliwe jest również zbudowanie bloków logicznych, które dają prawidłowe odpowiedzi nawet wtedy, gdy niektóre z ich elementów składowych działają nieprawidłowo. W tym przypadku podstawowym narzędziem znów jest redundancja. Jednym ze sposobów na zbudowanie odpornego na błędy bloku logicznego jest skopiowanie każdego bloku logicznego trzy razy. Do łączenia sygnałów z trzech kopii można posłużyć się blokiem większościowym, jaki został przedstawiony na rys. 12. w rozdziale 2. Jeśli jedna z kopii robi błąd, to zostaje „przegłosowana" przez pozostałe. Ta prosta metoda chroni przed dowolnym pojedynczym błędem (z wyjątkiem błędu w samym bloku większościowym).

Przy ustalonym zbiorze możliwych błędów — takich jak pęknięcia przewodów, zablokowanie przełączników, zamiana 0 na 1 — możliwe jest skonstruowanie urządzenia liczącego o dowolnie dużym stopniu niezawodności za pomocą elementów o dowolnie małym stopniu niezawodności. Zadanie to po prostu wymaga systematycznego stosowania układów logicznych z redundancją. Jeśli na przykład udałoby się wam zbudować nowy rodzaj przełącznika — powiedzmy, przełącznik molekularny — który byłby wyjątkowo szybki, lub niezwykle tani, ale popełniał błędy w 20% przypadków, to w dalszym ciągu moglibyście zbudować za pomocą tego przełącznika komputer dający odpowiedzi z niezawodnością

99,99999%, przez wbudowanie w obwody odpowiedniej nadmiarowości.

Czy to oznacza, że można zbudować dowolnie niezawodny komputer? Ściśle biorąc, nie. Wprawdzie komputer można tak skonstruować, że wyeliminowane zostaną pewne szczególne rodzaje błędów, ale mogą zdarzać się też błędy nieoczekiwane, które powodują powstanie skorelowanych błędów w modułach nadmiarowych. Na przykład, spalenie się jednego modułu może spowodować, że inny moduł się przegrzeje, lub jakiś impuls magnetyczny może sprawić, że wszystkie moduły równocześnie zaczną działać błędnie. Inżynierowie są w stanie projektować bloki logiczne odporne na wszystkie błędy, jakie potrafią sobie wyobrazić, ale historia techniki pokazuje, że nasza wyobraźnia nie zawsze jest wystarczająca. Najbardziej dramatyczne awarie zwykle są zaskakujące.

Od komputerów nie można oczekiwać niezawodności również z innego powodu. Okazuje się, że większość awarii komputerowych nie jest spowodowana nieprawidłowym działaniem układów logicznych. Wynikają one z błędów w projektowaniu — zwykle w projektowaniu oprogramowania. Komputery wraz ze swym oprogramowaniem są prawdopodobnie najbardziej złożonymi systemami, jakie kiedykolwiek skonstruował człowiek. Liczba elementów składowych jest o rzędy wielkości większa niż liczba części w najbardziej skomplikowanym samolocie. Metody stosowane we współczesnej inżynierii nie są przystosowane do projektowania układów o tak dużym stopniu złożoności. Szybkie komputery przeprowadzają równocześnie miliony symultanicznych

operacji logicznych i niemożliwe jest antycypowanie konsekwencji każdej kombinacji zdarzeń. Metody funkcjonalnej abstrakcji, opisane w poprzednich rozdziałach, pomagają zachować kontrolę nad tymi oddziaływaniami, ale te poziomy abstrakcji opierają się na założeniu, że wszystko działa zgodnie z planem. Kiedy zachodzą nieoczekiwane oddziaływania (a zachodzą), to założenia załamują się, a konsekwencje mogą być katastrofalne. W praktyce, zachowanie dużego systemu komputerowego może być czasem nieprzewidywalne nawet bez awarii — i przede wszystkim z tego powodu nie można zaprojektować całkowicie niezawodnego komputera.

Rozdział siódmy

SZYBKOŚĆ: KOMPUTERY O ARCHITEKTURZE RÓWNOLEGŁEJ

Rozmaite typy komputerów uniwersalnych różnią się nie tylko wielkością pamięci, ale również szybkością, z jaką przeprowadzają obliczenia. Szybkość komputera zależy w dużym stopniu od czasu, jaki zajmuje przesyłanie danych do — lub pobieranie ich z — pamięci.

Komputery, o których do tej pory mówiliśmy, to komputery sekwencyjne — komputery takie przetwarzają po jednym słowie maszynowym naraz. Wynika to w dużym stopniu ze względów historycznych. Pod koniec lat czterdziestych i na początku lat pięćdziesiątych, kiedy powstawały podstawy konstrukcji współczesnych komputerów, elementy przełączające (przekaźniki i lampy elektronowe) były drogie, ale względnie szybkie. Elementy pamięci (rtęciowe linie opóźniające, bębny magnetyczne) były zaś stosunkowo tanie i wolne. Elementy te dobrze nadawały się do wytwarzania sekwencyjnych zbiorów danych. Komputery zostały zaprojektowane tak, aby drogie przełączniki procesora były obciążone najbardziej, jak to możliwe, nie stawiając jedno-

cześnie zbyt dużych wymagań co do szybkości pamięci. Te wczesne komputery miały rozmiary pokoju, z drogim procesorem po jednej stronie i powolną pamięcią po drugiej. Między nimi przepływał strumień danych.

W miarę jak technika komputerowa rozwijała się, oprogramowanie stawało się coraz bardziej skomplikowane i drogie i coraz trudniej było wyszkolić programistów, więc, by móc inwestować maksimum w oprogramowanie i szkolenie, nie zmieniano podstawowej struktury komputerów. Brak było zresztą motywacji do zmiany tej dwuczęściowej struktury, ponieważ dzięki postępom technologicznym łatwo można było budować szybsze i tańsze maszyny, po prostu odtwarzając ten sam typ komputera w nowej technologii.

Szybkość komputerów podwaja się co dwa lata. Lampy elektronowe ustąpiły tranzystorom, a te z kolei układom scalonym. Układy pamięci na liniach opóźniających zostały zastąpione przez układy pamięci rdzeniowej, a te zaś — również przez układy scalone. Maszyny wielkości pokoju skurczyły się do płytki krzemowej o rozmiarach paznokcia. Pomimo całej tej zmieniającej się technologii, prosty schemat procesora podłączonego do pamięci pozostał nie zmieniony. Jeśli spojrzymy na płytę z układami współczesnego komputera, to w dalszym ciągu będziemy mogli zobaczyć pozostałości pokoju wypełnionego lampami elektronowymi: procesor i pamięć zajmują na płytce osobne obszary. Tak jest, mimo że części procesora i pamięci komputera obecnie wykonuje się tymi samymi metodami, często na tym samym kawałku płytki. Sposoby działania wciąż opty-

malizuje się pod kątem pierwotnego, dwuczęściowego schematu. Ta część płytki krzemowej, która odpowiada za przetwarzanie, jest nadal obciążana, a część spełniająca funkcje pamięci nadal przekazuje strumień danych po jednym słowie maszynowym naraz.

Przepływ danych między procesorem a pamięcią to wąskie gardło komputerów sekwencyjnych. Główny problemem polega na tym, że pamięć jest tak zaprojektowana, iż w każdym cyklu pracy komputera możliwy jest dostęp tylko do jednej lokalizacji. Dopóki trzymamy się tego rozwiązania, jedynym sposobem na zwiększanie szybkości komputera jest skracanie czasu trwania cyklu. Przez wiele lat czas ten można było zmniejszać, zwiększając szybkość działania przełączników: szybsze przełączniki dawały szybsze komputery. Dziś ta strategia nie jest już efektywna. Szybkość dzisiejszych dużych komputerów jest ograniczona przede wszystkim przez tempo przepływu informacji w obwodach, ono zaś z kolei ograniczone jest przez skończoną prędkość rozchodzenia się światła. Światło w ciągu jednej nanosekundy (10^{-9} sekundy) pokonuje w przybliżeniu 30 cm. Cykl pracy najszybszych współczesnych komputerów wynosi około nanosekundy i nie jest zbiegiem okoliczności, że procesory mają rozmiary mniejsze od 30 cm. Zbliżamy się do granic szybkości komputerów — o ile nie zmienimy ich podstawowego schematu.

WSPÓŁBIEŻNOŚĆ

Aby działać szybciej, dzisiejsze komputery muszą być zdolne do wykonywania więcej niż jednej operacji naraz. Można to osiągnąć, dzieląc pamięć kompu-

tera na wiele małych pamięci i wyposażając je we własne procesory. Takie urządzenia nazywają się **komputerami o architekturze równoległej** lub po prostu **komputerami równoległymi.** Komputery równoległe są praktyczne ze względu na niską cenę i małe rozmiary mikroprocesorów. Możemy zbudować komputer równoległy przez połączenie dziesiątek, setek, a nawet tysięcy mniejszych procesorów. Najszybsze komputery na świecie to komputery **całkowicie równoległe**, w których zastosowano tysiące, a nawet dziesiątki tysięcy procesorów.

Jak już wcześniej pisałem, komputery składają się z hierarchii elementów konstrukcyjnych, przy czym każdy poziom tej hierarchii jest stopniem wiodącym wyżej. W tym schemacie komputery równoległe są naturalnym następnym poziomem, na którym same komputery (sekwencyjne) stają się elementami konstrukcyjnymi. Tego typu konstrukcje nazywa się też niekiedy **sieciami komputerowymi.** Rozróżnienie między komputerami równoległymi a sieciami jest dość arbitralne; istotne jest, jak taki układ jest wykorzystywany. Poza tym komputer równoległy to zwykle urządzenie zlokalizowane w jednym konkretnym miejscu, podczas gdy sieć to komputery rozmieszczone w różnych lokalizacjach, ale są wyjątki od obu tych reguł. Ogólnie biorąc, jeśli grupa połączonych komputerów używana jest w sposób skoordynowany, to taki układ nazywamy komputerem równoległym. Jeśli komputery wykorzystywane są raczej niezależnie, to połączone maszyny nazywa się siecią komputerową.

Łączenie razem dużej liczby komputerów w celu osiągnięcia większej szybkości wydaje się krokiem

oczywistym, ale przez wiele lat wśród informatyków panowała zgoda, że takie posunięcie będzie skuteczne jedynie w przypadku bardzo nielicznych aplikacji. Znaczną cześć mojej kariery poświęciłem na dyskusje z ludźmi wierzącymi, że budowa i zaprogramowanie uniwersalnego komputera całkowicie równoległego będzie trudne, a nawet wręcz niemożliwe. Ten szeroko rozpowszechniony sceptycyzm opierał się na błędnych przekonaniach co do tego, na ile skomplikowane musiałyby być takie systemy i jak miałyby współpracować ze sobą ich elementy.

Naukowcy przeceniali stopień złożoności komputerów równoległych, ponieważ nie doceniali tempa rozwoju techniki wytwarzania układów elektronicznych. Nie wynikało to bynajmniej z niewiedzy — raczej spowodowane było bezprecedensowym tempem rozwoju technologii, wyprzedzającym oczekiwania i intuicje. Pamiętam, że w połowie lat siedemdziesiątych, na konferencji poświęconej komputerom w nowojorskim hotelu Hilton, wygłosiłem wykład, w trakcie którego stwierdziłem, że panujące trendy jednoznacznie wskazują, iż wkrótce w USA będzie więcej mikroprocesorów niż ludzi. W owym czasie uważano to za niedorzeczną ekstrapolację. Choć mikroprocesory były już wówczas produkowane, to w powszechnym odczuciu komputer ciągle był sporym zestawem szaf wielkości lodówki, z migającymi światełkami. W trakcie dyskusji po wykładzie jeden z niezbyt przychylnych słuchaczy zapytał głosem, w którym pełno było sarkazmu: „A co według was ludzie będą robić z tymi wszystkimi komputerami? Przecież nie potrzeba nam komputera do każdej klamki!" Audytorium wybuchło śmiechem, a ja mia-

łem kłopot ze znalezieniem właściwej odpowiedzi — dziś w tym samym hotelu każdy zamek w drzwiach zawiera mikroprocesor.

Inne przyczyny tego sceptycyzmu w stosunku do komputerów równoległych były bardziej subtelne, ale i lepiej uzasadnione. Zastrzeżenia wiązały się z przewidywanym brakiem wydajności w dzieleniu obliczeń na wiele współbieżnych części. Ten problem do dziś ogranicza zastosowania komputerów równoległych, ale nie do tego stopnia, jak dawniej sądzono. Przecenianie stopnia trudności wynikało częściowo z ciągu mylących doświadczeń z pierwszymi maszynami równoległymi. Pierwsze cyfrowe komputery równoległe zostały zbudowane w latach sześćdziesiątych, przez połączenie dwóch lub trzech dużych komputerów sekwencyjnych. W większości przypadków układy przetwarzania współbieżnego były połączone z tym samym, pojedynczym układem pamięci, aby każdy z procesorów miał dostęp do tych samych danych. Te wczesne komputery równoległe były programowane zwykle tak, aby przydzielić każdemu procesorowi inne zadanie: na przykład w programach użytkowych typu baz danych pierwszy procesor odczytywał dane, drugi układał tablice statystyczne, a trzeci drukował wyniki. Procesory były wykorzystywane w sposób bardzo podobny do tego, jak działają pracownicy przy linii montażowej; każdy z nich wykonywał inny etap obliczeń. W takim schemacie nieuchronnie pojawiały się czynniki obniżające efektywność, a ich liczba rosła wraz ze zwiększaniem liczby procesorów. Jednym z problemów była konieczność dzielenia zadania na mniej lub bardziej niezależne etapy. O ile często można było podzielić

problem na dwa lub trzy etapy, to trudno było na pierwszy rzut oka dostrzec, jak można podzielić zadanie na dziesięć lub sto etapów. Jeden z krytyków przetwarzania współbieżnego wyjaśnił to dziennikarzowi w ten sposób: *Dwóch reporterów może szybciej napisać artykuł do gazety, dzieląc się zadaniami; jeden zbiera wiadomości, podczas gdy drugi pisze tekst, ale gdyby stu reporterów pracowało nad jednym artykułem, to prawdopodobnie nigdy by on nie powstał.* Takie argumenty brzmią dość przekonująco.

Innym czynnikiem obniżającym efektywność był brak wspólnego dostępu do pamięci. Typowy układ może pobierać z danego obszaru pamięci tylko jedno słowo maszynowe naraz; to ograniczenie tempa dostępu tworzyło łatwo zauważalne wąskie gardło w systemie i ograniczało jego wydajność. Jeśli zaś do układu, którego wydajność już była ograniczona przez tempo pobierania danych, dodano by więcej procesorów, to w efekcie czas oczekiwania procesorów na dane zwiększyłby się i wydajność układu uległaby zmniejszeniu.

Ponadto procesory musiały uważać, aby nie zmienić danych, z których korzystał inny procesor. Rozważmy na przykład system rezerwacji miejsc w samolotach. Jeśli jakiś procesor pracuje nad zarezerwowaniem miejsc, to sprawdza, czy są wolne miejsca, a jeśli tak jest, to dokonuje rezerwacji. Jeśli dwa procesory dokonują rezerwacji dla dwóch pasażerów równocześnie, to może pojawić się problem: oba zauważają, że jakieś miejsce jest wolne i rezerwują je, zanim którykolwiek zdąży zaznaczyć, że miejsce jest już zajęte. Dla uniknięcia tego typu niepomyślnego zbiegu okoliczności procesor musiał wykonywać sze-

reg skomplikowanych operacji, które uniemożliwiały innym procesorom dostęp do danych w czasie, gdy on je odczytywał. To pogłębiało jeszcze nieefektywność związaną ze współzawodnictwem o pamięć sytemu i w najgorszym przypadku redukowało szybkość systemu wieloprocesorowego do szybkości pojedynczego procesora — a czasem nawet jeszcze bardziej. Jak już powiedziałem, wpływ czynników obniżających efektywność rósł w miarę zwiększania liczby procesorów.

Zasadniczy czynnik obniżający wydajność ma jeszcze bardziej fundamentalny charakter: trudność w równomiernym wykonywaniu zadań przypisanych różnym procesorom. Powróćmy do analogii z linią montażową: łatwo można się przekonać, że szybkość wykonywania obliczeń określana będzie przez czas wykonania najwolniejszej operacji. Nawet jeśli tylko jedna z operacji jest czasochłonna, to tempo obliczeń wyznaczane jest właśnie przez nią. Uzasadnione jest oczekiwanie, że również i ten czynnik będzie silniej redukował wydajność systemu, jeśli zwiększymy liczbę procesorów.

Najlepszym ujęciem tych problemów z wydajnością jest tak zwane prawo Amdahla, odkryte przez projektanta komputerów Gene'a Amdahla w latach sześćdziesiątych. Rozumowanie Amdahla było mniej więcej takie: zawsze będzie jakaś część obliczeń, która ze swej istoty jest sekwencyjna — może być wykonywana tylko przez jeden procesor naraz. Nawet jeśli tylko 10% zadania jest sekwencyjne, to niezależnie od tego, jak bardzo przyspieszymy wykonywanie pozostałych 90%, zakończenie całego obliczenia nigdy nie zostanie przyspieszone bardziej niż o czynnik 10.

Procesory pracujące nad tymi 90% obliczeń, które mogą być wykonane współbieżnie, w efekcie będą czekać na jeden, aż zakończy on wykonywanie swoich 10% zadania. Ten argument sugeruje, że komputer równoległy złożony nawet z tysiąca procesorów byłby wyjątkowo mało wydajny, ponieważ byłby tylko około dziesięć razy szybszy od pojedynczego procesora. Kiedy próbowałem uzyskać fundusze na zbudowanie mojego pierwszego komputera równoległego — całkowicie równoległego komputera z 64 000 procesorów — to pierwszym pytaniem, jakie słyszałem zwykle po zakończeniu mojej prezentacji było: „Czy nie słyszał pan nigdy o prawie Amdahla?"

Oczywiście, **słyszałem** o prawie Amdahla i, mówiąc prawdę, nie widziałem żadnego błędu w rozumowaniu, na którym się opiera. A jednak byłem pewien, nawet jeśli nie mogłem tego udowodnić, że prawo Amdahla nie stosuje się do problemów, które próbowałem rozwiązać. Moja pewność wynikała z faktu, że problemy, nad którymi pracowałem, zostały już rozwiązane za pomocą całkowicie równoległego procesora — ludzkiego mózgu. Byłem studentem w Laboratorium Sztucznej Inteligencji w MIT i chciałem zbudować maszynę, która potrafiłaby myśleć.

Kiedy w roku 1974, jako student pierwszego roku, po raz pierwszy odwiedziłem Laboratorium Sztucznej Inteligencji w MIT, dziedzina ta przeżywała okres gwałtownego rozwoju. Opracowywano pierwsze programy, które miały rozumieć rozkazy zapisane w języku naturalnym i bliskie wydawało się zbudowanie komputera zdolnego do rozumienia mowy ludzkiej. Komputery doskonale radziły sobie w grach takich jak szachy, które jeszcze kilka lat

wcześniej uważano za zbyt skomplikowane dla maszyn. Sztuczne systemy do rozpoznawania obrazów potrafiły identyfikować nieskomplikowane obiekty, jak rysunki kreską lub stosy klocków. Komputery zdawały nawet egzaminy z matematyki i rozwiązywały proste problemy oparte na analogiach, wzięte z testów na inteligencję. Czy uniwersalna maszynowa inteligencja mogła być bardzo daleko?

Kiedy — już jako doktorant — powróciłem do tego samego laboratorium kilka lat później, problemy sprawiały wrażenie trudniejszych. Proste demonstracje okazały się być tylko tym, czym były — prostymi demonstracjami. Choć wypracowano wiele nowych zasad i silnych metod, zastosowanie ich do większych i bardziej skomplikowanych problemów nie dawało rezultatów. Kłopot wynikał częściowo z ograniczonej szybkości komputerów. Badacze sztucznej inteligencji nie widzieli sensu w rozszerzaniu swoich eksperymentów na przypadki związane z większą ilością danych, ponieważ nawet z prostszymi zagadnieniami komputery radziły sobie powoli, a zwiększenie ilości danych powodowało tylko dalsze spowolnienie ich pracy. Frustrujące były na przykład próby rozpoznania przez maszynę stosu przedmiotów, kiedy identyfikacja każdego z nich wymagała całych godzin obliczeń.

Komputery pracowały wolno, ponieważ były sekwencyjne; mogły wykonywać tylko jedną czynność naraz. Maszyna musi oglądać obrazek piksel po pikselu; natomiast mózg odbiera całość w jednej chwili i może równocześnie porównać to, co widzi, z każdym obrazem, jaki zna. Z tego powodu człowiek jest znacznie szybszy od komputera w rozpoznawaniu

przedmiotów, mimo że neurony w ludzkim układzie wzrokowym są znacznie wolniejsze od tranzystorów w komputerze. Ta różnica w konstrukcji zainspirowała mnie — i wielu innych — do poszukiwania metod pozwalających na skonstruowanie takich całkowicie równoległych komputerów, które potrafiłyby wykonywać miliony operacji równolegle i wykorzystywać współbieżność w taki sam sposób, jak mózg. Ponieważ wiedziałem, że mózg potrafi uzyskać dużą szybkość działania z powolnych części składowych, domyślałem się, że prawo Amdahla nie musi działać zawsze.

Dziś wiem już, w którym miejscu w rozumowaniu Amdahla tkwi błąd. Polega on na założeniu, że pewna część obliczeń, nawet tylko 10%, **musi** mieć charakter sekwencyjny. To oszacowanie brzmi wiarygodnie, ale okazuje się fałszywe dla większości dużych obliczeń. Fałszywa intuicja bierze się ze złego zrozumienia jak będą używane procesory równoległe. Sedno sprawy tkwi w sposobie podziału obliczeń między procesory: na pierwszy rzut oka najlepszym sposobem wydaje się przekazanie każdemu procesorowi do wykonania innej części programu. Ten sposób jest dość dobry, ale — jak na to wskazuje wspomniana wyżej dziennikarska analogia — ma te same wady, co przydzielanie zadania zespołowi ludzi: zyski z potencjalnej współbieżności przepadają w wyniku kłopotów z koordynacją. Programowanie komputera przez dzielenie programu na części przypomina sytuację, kiedy duży zespół ludzi maluje płot, przy czym jedna osoba zajmuje się otwieraniem puszek z farbą, inna przygotowuje powierzchnię do malowania, kolejna nakłada farbę, a jeszcze inna czyści

pędzle. Taki podział funkcji wymaga doskonałej koordynacji i od pewnego momentu zwiększanie liczby ludzi nie przyspiesza wykonania zadania.

Bardziej efektywny sposób wykorzystania komputera równoległego polega na wykonywaniu przez wszystkie procesory podobnych zadań, ale na różnych fragmentach danych. Taki podział zadania — z tak zwanym **współbieżnym rozkładem danych** — przypomina przydzielenie każdemu pracownikowi oddzielnego kawałka płotu. Nie wszystkie problemy dają się podzielić tak łatwo, jak malowanie płotu, ale w przypadku dużych obliczeń taka metoda działa zadziwiająco dobrze. Na przykład, zadania związane z przetwarzaniem obrazów mogą zostać podzielone na współbieżne części przez przydzielenie każdemu procesorowi małych fragmentów. Zagadnienia związane z przeszukiwaniem, takie jak gra w szachy, mogą zostać podzielone w ten sposób, że procesory równocześnie analizują różne warianty gry. W tych przykładach stopień przyspieszenia wykonywania zadania jest prawie proporcjonalny do liczby procesorów — czyli im więcej procesorów, tym lepiej. Trochę czasu pochłania rozdzielenie zadań między procesorami i zebranie odpowiedzi razem, ale jeśli zagadnienie jest duże, to obliczenie może zostać wykonane bardzo efektywnie, nawet na komputerze równoległym z dziesiątkami tysięcy procesorów.

W przypadku obliczeń opisanych powyżej jest dość oczywiste, że można je podzielić tak, aby były wykonywane współbieżnie, ale podział ze współbieżnym rozkładem danych da się zastosować także do zadań bardziej skomplikowanych. Zaskakujące jest, jak niewiele jest dużych problemów obliczeniowych,

z którymi nie można by sobie poradzić za pomocą przetwarzania współbieżnego. Nawet problemy uważane przez większość ludzi za z natury sekwencyjne zwykle mogą być efektywnie rozwiązywane na komputerze równoległym. Przykładem jest zagadnienie szukania łańcuchowego. Moje dzieci bawią się w grę zwaną Poszukiwaniem Skarbu, opartą na zasadzie szukania łańcuchowego. Daję im karteczkę ze wskazówką, gdzie ukryta jest następna wskazówka. Ta prowadzi do kolejnej... i tak dalej, aż znajdą na końcu skarb. W komputerowej wersji tej gry program otrzymuje jako dane wejściowe adres miejsca w pamięci, w którym zapisany jest adres innego miejsca w pamięci, również zawierającego adres pewnego miejsca i tak dalej, aż w końcu, pod którymś adresem, znajduje się specjalne słowo oznaczające koniec łańcucha. Zadanie polega na wyjściu z pierwszego miejsca pamięci i znalezieniu ostatniego.

Na pierwszy rzut oka problem szukania łańcuchowego wydaje się ucieleśnieniem obliczenia sekwencyjnego, ponieważ nie widać sposobu, aby komputer mógł odnaleźć poszukiwany adres bez prześledzenia związanych ze sobą adresów w całym łańcuchu. Musi przecież zajrzeć do pierwszego miejsca, aby odnaleźć adres drugiego, potem zajrzeć do drugiego, aby znaleźć trzecie.... Okazuje się jednak, że nawet to zagadnienie może zostać rozwiązane współbieżnie. Komputer równoległy z milionem procesorów może znaleźć ostatni element w łańcuchu o długości miliona adresów w dwudziestu krokach. Trik polega na podzieleniu zadania na połowę w każdym kroku, podobnie jak w przypadku algorytmów sortujących, opisywanych w rozdziale 5. Przyjmijmy, że każde

z miliona miejsc w pamięci ma swój własny procesor, który może przesłać wiadomość do każdego innego procesora. Aby odnaleźć koniec łańcucha, procesor zaczyna od wysłania swojego własnego adresu do procesora, który następuje po nim w łańcuchu — czyli tego, którego adres przechowywany jest w jego pamięci. W tej sytuacji wszystkie procesory znają adres nie tylko procesora następnego w ciągu, ale również poprzedzającego. Każdy z nich wykorzystuje dalej tę informację, do przesłania adresu swojego następcy swojemu poprzednikowi. Teraz każdy zna adres procesora, który znajduje się o dwa kroki przed nim w łańcuchu, tak więc, łańcuch łączący pierwszy procesor z ostatnim ma teraz połowę długości wyjściowej. Ta redukcja jest następnie powtarzana, a za każdym powtórzeniem długość łańcucha zmniejsza się o połowę. Po dwudziestu iteracjach pierwszy procesor w łańcuchu zawierającym milion adresów zna adres ostatniego miejsca w łańcuchu. Podobne metody mogą być stosowane do wielu innych zadań, pozornie idealnie sekwencyjnych.

W chwili, kiedy piszę tę książkę, komputery równoległe ciągle są czymś względnie nowym i nie rozumiemy jeszcze dobrze, jakiego typu zagadnienia mogą być rozkładane tak, aby w efektywny sposób wykorzystywać wiele procesorów. Doświadczenie wskazuje, że współbieżność daje najlepsze wyniki w problemach z dużą ilością danych, ponieważ wtedy jest wiele podobnych zadań, którymi można obdzielić wiele procesorów. Zasada, że większość obliczeń można podzielić na współbieżne podproblemy, wynika między innymi z tego, że wiele obliczeń związanych jest z modelami fizycznej rzeczywistości. Obli-

czenia w ramach modeli fizycznych mogą być przeprowadzane równolegle, ponieważ świat fizyczny działa równolegle. Na przykład, obrazy grafiki często są syntetyzowane za pomocą algorytmów symulujących fizyczny proces odbijania się światła od powierzchni przedmiotów. Obraz otrzymywany jest za pomocą matematycznego opisu kształtów przedmiotów, który pozwala obliczyć, jak każdy promień światła odbijać się będzie od powierzchni w drodze od źródła do oka. Obliczenia dla wszystkich promieni świetlnych mogą być przeprowadzane współbieżnie, ponieważ odbicie światła w rzeczywistości zachodzi współbieżnie.

Typowym przykładem kategorii obliczeń dobrze nadających się dla komputera równoległego jest symulacja atmosfery, wykorzystywana przy prognozach pogody. Trójwymiarowa siatka liczb przedstawiających atmosferę jest analogiczna do trójwymiarowej przestrzeni fizycznej. Każda liczba określa fizyczny parametr pewnej objętości powietrza — na przykład ciśnienie atmosferyczne w sześcianie o boku 1 km. Każdy z tych sześcianów będzie reprezentowany przez kilka liczb, określających średnią temperaturę, ciśnienie, prędkość wiatru i wilgotność. Aby przewidzieć, jak będzie ewoluować powietrze w jednym sześcianie, komputer oblicza przepływ powietrza między sąsiadującymi obszarami; na przykład, jeśli więcej powietrza wpływa do sześcianu niż z niego wypływa, to ciśnienie wzrośnie. Komputer również oblicza zmiany powodowane przez takie czynniki jak światło słoneczne i parowanie. Symulacja atmosfery wykonywana jest w ciągu kroków, z których każdy odpowiada małemu odcinkowi czasu —

powiedzmy pół godziny — tak więc przepływ symulowanego powietrza i wody między sześcianami w sieci jest analogiczny do przepływu rzeczywistego powietrza i wody, tworzących układ pogodowy. W efekcie, wewnątrz komputera powstaje trójwymiarowy ruchomy obraz, zachowujący się zgodnie z prawami fizyki.

Oczywiście wierność tej symulacji zależy od rozdzielczości i wierności trójwymiarowego modelu, co wyjaśnia powszechnie znaną niedokładność prognoz pogody. Jeśli zwiększy się rozdzielczość i dokładniej zmierzy warunki początkowe, to przewidywanie będzie lepsze — choć, nawet przy bardzo wysokiej rozdzielczości, nigdy nie będzie idealne w dłuższych odcinkach czasu, ponieważ warunków początkowych nigdy nie da się ustalić z całkowitą precyzją, układy reprezentujące pogodę bowiem są chaotyczne, tak jak w grze w ruletkę, więc mała zmiana w warunkach początkowych powoduje istotne różnice wyniku. Na komputerze równoległym każdy procesor może być odpowiedzialny za odwzorowywanie zmian pogody w maleńkim obszarze. Kiedy wiatr wieje z jednego obszaru do drugiego, procesory modelujące te obszary muszą się porozumiewać. Natomiast symulacja pogody dla obszarów odległych od siebie może być przeprowadzana praktycznie niezależnie. Obliczenia mają charakter lokalny i współbieżny, ponieważ fizyka, która rządzi pogodą, również ma charakter lokalny i współbieżny.

O ile modelowanie pogody łączy się z prawami fizyki w sposób oczywisty, to wiele innych obliczeń związanych jest z rzeczywistością fizyczną bardziej subtelnie. Na przykład, wyliczanie rachunków za

telefon jest współbieżne, ponieważ telefony (i osoby korzystające z nich) działają w fizycznym świecie niezależnie od siebie. Jedynymi problemami, których nie potrafimy rozwiązywać w efektywny sposób na komputerach równoległych, są zagadnienia, w których rolę analogiczną do czasu pełni rosnąca liczba wymiarów. Przykładem takiego zagadnienia jest problem przewidywania przyszłych pozycji planet (jak na ironię jest to akurat problem, na którego potrzeby rozwinięto wiele z naszych matematycznych narzędzi obliczeniowych). Trajektorie planet wynikają z dobrze określonych praw ruchu i oddziaływań grawitacyjnych między dziewięcioma planetami i Słońcem. Dla uproszczenia pomińmy wpływy małych obiektów, takich jak księżyce i asteroidy. Całą konieczną informację można przedstawić w postaci dziewięciu współrzędnych, więc nie ma tu zbyt wielu danych. Złożoność obliczeniowa tego problemu wynika z faktu, że musimy obliczać — a przynajmniej nie znamy innej metody — kolejne położenia planet w każdym z miliardów maleńkich kroków, reprezentujących krótkie odcinki czasu. Jedynym sposobem na poznanie położenia planet za milion lat jest obliczanie go w każdym z pośrednich momentów, począwszy od dnia dzisiejszego. Nie znamy żadnego triku pozwalającego na rozwiązanie tego problemu w sposób współbieżny, analogicznego do tego, którego użyliśmy w przypadku szukania łańcuchowego. Z drugiej strony — znów na ile mi wiadomo — nikt nie udowodnił, że problem orbit rzeczywiście jest sekwencyjny; pozostaje to kwestią otwartą.

Już dziś współbieżne komputery są stosowane względnie często. Używa się ich głównie przy bardzo

dużych obliczeniach numerycznych (jak symulacja pogody) lub w obliczeniach związanych z dużymi bazami danych (na przykład analiza strategii marketingowych na podstawie danych o transakcjach dokonanych za pomocą kart kredytowych). Ponieważ komputery równoległe są zbudowane z tych samych elementów co komputery osobiste, jest prawdopodobne, że z czasem staną się tańsze i bardziej rozpowszechnione. Dziś jednym z najbardziej interesujących komputerów równoległych jest struktura, która wyłoniła się niemal przez przypadek z połączenia w sieć maszyn sekwencyjnych. Ogólnoświatowej sieci komputerów zwanej Internetem ciągle jeszcze używa się głównie jako systemu komunikacyjnego. Komputery spełniają przeważnie rolę medium — sortują i przesyłają informacje (takie jak poczta elektroniczna), sensowne jedynie dla ludzi. Ale jestem przekonany, że to się zmieni. Już teraz zaczynają pojawiać się standardy pozwalające na wymianę programów i danych. Komputery podłączone do Internetu, pracując razem, mają potencjalną zdolność obliczeniową znacznie przekraczającą możliwości jakiegokolwiek zbudowanego dotychczas pojedynczego komputera.

Wierzę, że Internet rozwinie się kiedyś na tyle, by ogarnąć komputery znajdujące się w sieciach telefonicznych, samochodach lub urządzeniach gospodarstwa domowego. Takie maszyny będą odczytywać dane wejściowe bezpośrednio ze świata fizycznego, zamiast polegać na pośrednictwie ludzi. Spodziewam się, że w miarę wzrostu ilości informacji dostępnej w Internecie i złożoności typów oddziaływań między połączonymi komputerami, Internet zacznie

wykazywać **zachowania emergentne,** wykraczając poza to, co zostało explicite zaprogramowane w systemie. W zasadzie już teraz Internet zaczyna wykazywać oznaki zachowania emergentnego, choć jak dotąd są to rzeczy bardzo proste: plaga wirusów komputerowych czy zaskakujące trasy przesyłanych wiadomości. Podejrzewam, że kiedy komputery w sieci zaczną wymieniać oddziałujące programy zamiast tylko poczty elektronicznej, to Internet zacznie zachowywać się bardziej jak komputer równoległy niż jak sieć komputerowa. Mam wrażenie, że zachowania emergentne Internetu staną się kiedyś o wiele bardziej interesujące.

Rozdział ósmy

KOMPUTERY, KTÓRE UCZĄ SIĘ I DOSTOSOWUJĄ

Programy komputerowe, które do tej pory opisywałem, działają według ustalonych zasad, podanych przez programistę. Nie mają one żadnych możliwości tworzenia nowych reguł działania ani doskonalenia tych, które zostały dla nich ustanowione. Programy szachowe — jeśli programiści ich nie zmodyfikują — będą wciąż popełniały te same błędy, niezależnie od liczby rozegranych partii. W tym sensie komputery są całkowicie przewidywalne; właśnie dlatego mówimy, że komputery mogą „robić tylko to, do czego zostały zaprogramowane" — argument często przytaczany przez obrońców ludzkości w debacie „człowiek kontra maszyna".

Ale nie wszystkie programy są tak mało elastyczne. Możliwe jest tworzenie programów, które w miarę nabywania doświadczeń stają się coraz lepsze. Komputery działające według takich programów mogą uczyć się na swoich błędach i poprawiać je. Osiągają to, wykorzystując **sprzężenie zwrotne**. Każdy układ oparty na sprzężeniu zwrotnym potrzebuje trzech typów informacji:

- Jaki jest pożądany stan (**cel**)?
- Jak jest różnica między stanem obecnym a stanem pożądanym (**odchylenie**)?
- Jakie działania zmniejszą różnicę między stanem obecnym a stanem pożądanym (**reakcja**)?

Sprzężenie zwrotne reguluje reakcję układu zgodnie z tym, jakie jest odchylenie, po to, by osiągnąć cel. Najprostszym i najlepiej znanym rodzajem sprzężenia zwrotnego nie są wcale układy uczące się, ale układy sterujące; dobrym przykładem jest domowy termostat. Ten układ ze sprzężeniem zwrotnym rozpoznaje tylko dwa możliwe odchylenia i wytwarza tylko dwie możliwe reakcje. Celem jest zachowanie określonej temperatury, a możliwe odchylenia odpowiadają sytuacjom, kiedy temperatura jest za wysoka lub za niska. Reakcje są z góry określone: jeśli temperatura jest za niska, to reakcją jest włączenie ogrzewania, a jeśli jest za wysoka — wyłączenie go. Ponieważ termostat może jedynie włączać lub wyłączać ogrzewanie, reakcja nie zależy od wielkości odchylenia. (Wielokrotnie próbowałem wyjaśnić to członkom mojej rodziny, którzy podkręcają termostat zawsze, kiedy w mieszkaniu jest za zimno, w nadziei, że to jakoś przyspieszy nagrzewanie się. Takie postępowanie nic nie daje. Termostat może tylko włączyć ogrzewanie, nie może go zintensyfikować.)

W zasadzie jednak nie ma powodu, aby termostat domowego systemu ogrzewania nie reagował proporcjonalnie do odchylenia. W takim urządzeniu istniałaby możliwość regulowania mocy grzejników, a nie tylko ich włączania i wyłączania. Bez wątpienia byłoby ono bardziej skomplikowane i droższe, ale zapewniałoby też bardziej precyzyjną kontrolę tempe-

ratury. Takich termostatów ze **sterowaniem proporcjonalnym** używa się obecnie w układach sterujących delikatnymi procesami przemysłowymi. Są już urządzenia gospodarstwa domowego — na przykład niektóre japońskie pralki — również wykorzystujące sterowanie proporcjonalne (lub jego przybliżenie), co często jest reklamowane jako **logika rozmyta** (*fuzzy logic*).

Innym przykładem systemu wykorzystującego sterowanie proporcjonalne jest automatyczny pilot w samolotach. Celem w tym przypadku jest utrzymanie samolotu na określonym kursie. Urządzenie określające kierunek lotu, na przykład kompas, mierzy odchylenie, a autopilot reaguje, dokonując odpowiednich zmian w położeniu sterów samolotu, proporcjonalnie do wielkości i kierunku odchylenia. Niewielkie odchylenie od kursu spowoduje jedynie nieznaczną zmianę położenia steru, ale znaczne odstępstwo, spowodowane na przykład zmianą kierunku wiatru, spowoduje dużą zmianę w ustawieniu steru. Gdyby autopilot nie wykorzystywał sterowania proporcjonalnego, ale działał podobnie do domowego termostatu i wychylał ster maksymalnie w lewo lub w prawo, to samolot podlegałby bardzo niewygodnym, a przypuszczalnie i niebezpiecznym oscylacjom.

We wszystkich układach ze sprzężeniem zwrotnym wrażliwość na odchylenia i wielkość reakcji są z góry ustalone przez konstrukcję systemu sterującego. Ale możliwe jest też zaprojektowanie bardziej elastycznego układu ze sprzężeniem zwrotnym, w którym reakcje układu **dostosowują się** w miarę upływu czasu. W tym przypadku parametry wyjściowego układu ze sprzężeniem zwrotnym są re-

gulowane przez drugi układ ze sprzężeniem zwrotnym. Jeśli ten drugi dostosowuje się i doskonali w miarę upływu czasu, to można powiedzieć, że cały układ będzie „uczył się", jakie powinny być parametry sterowania.

Rozważmy na przykład sytuację, kiedy ktoś uczy się pilotować samolot. Początkujący pilot ma zwykle skłonność do przesterowań — zbyt gwałtownie reaguje na każde odchylenie. Takie zachowanie pilota przypomina działanie układu wykorzystywanego przez termostat do sterowania temperaturą: jeśli samolot skręca zbytnio w lewo, trzeba sterować w prawo; jeśli leci w prawo, należy zwrócić się w lewo. Ponieważ pomiędzy zmianą położenia sterów kierunkowych a reakcją samolotu upływa pewien czas, samolot wpada w ruch drgający. Pilot musi nauczyć się, jak poruszać sterem proporcjonalnie do odchylenia. Właściwą czułość na odchylenia można ustalać za pomocą układu ze sprzężeniem zwrotnym; w tym przypadku **celem** jest utrzymanie samolotu na prawidłowym kursie bez wahań, a **odchyleniem** jest wielkość drgań. **Reakcją** zaś jest dostosowanie reakcji pierwotnego sprzężenia zwrotnego, czyli dostosowanie stopnia wychylenia steru przy korygowaniu kursu samolotu. Jeśli pierwsze sprzężenie zwrotne powoduje drgania, to redukuje on jego czułość. Jeśli samolot zaczyna zbaczać z kursu, wzmacnia ją. Kiedy początkujący pilot nauczy się, jaka wrażliwość na odchylenia jest właściwa, wtedy potrafi utrzymać samolot na kursie bez jakichkolwiek oscylacji.

Możliwe jest zbudowanie automatycznego pilota, wykorzystującego drugie sprzężenie zwrotne do modyfikacji własnych parametrów. W tym przypadku

można powiedzieć, że autopilot „uczy" się sterować samolotem, podobnie jak w przypadku pilota-człowieka. O ile wiem, takie dostosowujące się układy automatycznego pilotażu nie są używane w samolotach, ale gdyby były, miałyby pewne zalety. W przypadku uszkodzenia powodującego zmianę w reakcjach samolotu (na przykład częściowa awaria sterów), taki autopilot potrafiłby dostosować się do nowej sytuacji. Mógłby nawet nauczyć się reagować prawidłowo w sytuacji, kiedy silniki obracające ster kierunku zostałyby przypadkowo odwrócone, przez co sygnał, który normalnie powoduje zwrot samolotu w prawo, powodowałby skręt w lewo. Oczywiście, podobnie jak pilot-człowiek, autopilot potrzebowałby czasu, aby dostosować się do tak radykalnej zmiany.

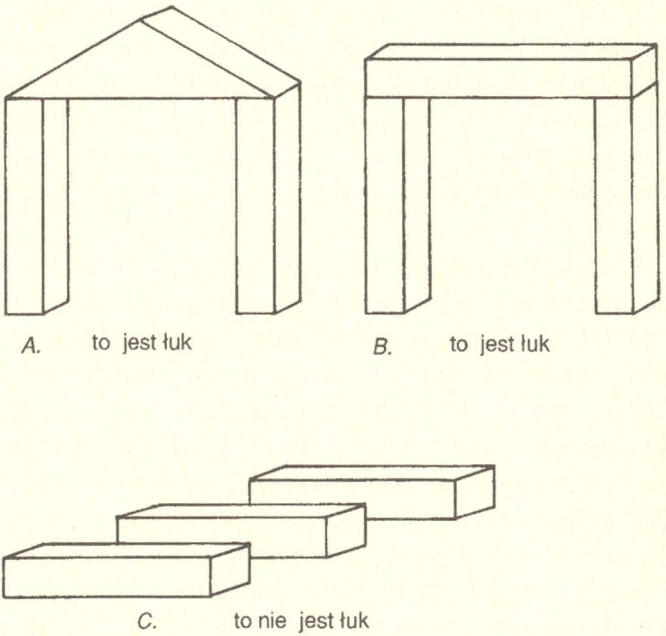
TRENOWANIE KOMPUTERA

Zasada sprzężenia zwrotnego leży u podstaw wszystkich układów uczących się, choć często przyjmuje postać bardziej skomplikowaną niż samoregulujący się autopilot. Często sprzężenie zwrotne w programach komputerowych wprowadzane jest przez **trening**. Trener (zwykle jest nim człowiek) odgrywa rolę nauczyciela, a program staje się uczniem. Klasycznym przykładem trenowanego układu uczącego się jest program napisany przez pioniera badań nad sztuczną inteligencją, Patricka Winstona. Uczy się on definicji takich pojęć jak na przykład „łuk" z ciągu pozytywnych i negatywnych przykładów podawanych przez instruktora. Program Winstona uczy się nowych pojęć, analizując proste szkice przestawiające stosy klocków; potrafi przeanalizo-

wać takie rysunki i wytworzyć symboliczny opis stosu klocków, na przykład: „Dwa stykające się sześciany, podtrzymujące klin". Trener pokazuje programowi przykładowe konfiguracje klocków, tworzące łuki, i takie, które łuków nie tworzą, informując o tym, które z nich są „łukami", a które nie. Początkowo program nie ma żadnej definicji pojęcia „łuku", ale w miarę, jak pokazuje mu się pozytywne i negatywne przykłady, zaczyna formułować definicję roboczą. Za każdym razem, gdy programowi pokazuje się nowy przykład, sprawdza jej poprawność. Jeśli definicja wystarczająco dobrze opisuje przykłady pozytywne, lub wyklucza przykłady negatywne, program jej nie modyfikuje. Jeśli jest błędna, jest modyfikowana tak, aby objąć nowy przypadek.

A oto, jak program uczy się definicji „łuku" na przykładach. Przyjmijmy, że pierwszy układ pokazany programowi jest przykładem pozytywnym: A z rys. 25., dwa ustawione pionowo prostopadłościenne klocki, podtrzymujące klocek trójkątny. Na początku program musi przyjąć jakąś wyjściową definicję łuku. Ta wstępna definicja nie musi być bardzo dokładna, ponieważ z czasem zostanie udoskonalona. Załóżmy, że program przyjmuje na początku następującą definicję: „Łuk to dwa prostopadłościenne klocki i trójkątny klocek". Drugi przykład pokazany programowi może przedstawiać inny układ klocków (C na rys. 25.). To przykład negatywny — przykład czegoś, co łukiem **nie** jest. Ponieważ początkowa robocza definicja programu błędnie identyfikuje ten układ jako łuk, program modyfikuje ją, by wyeliminować ten przykład. Dokonuje tego przez uchwycenie różnic między definicją a przykła-

dem i wykorzystanie ich do dodania ograniczeń do konstruowanej definicji. W tym przypadku różnica polega na wzajemnych relacjach między klockami, tak więc poprawiona definicja zawierać będzie teraz relacje: „Łuk to dwa ustawione **pionowo** prostopadłościenne klocki **podtrzymujące** trójkątny klocek". Przypuśćmy teraz, że trener podaje kolejny przykład pozytywny (B na rys. 25.). W tym przykładzie zamiast trójkątnego klocka na górze znajduje się klocek prostopadłościenny. Ponieważ robocza definicja programu nie jest wystarczająco ogólna, aby rozpoznać ten



RYSUNEK 25.

Układy klocków mogących służyć za pozytywne i negatywne przykłady łuku

przykład, program uogólni swoją definicję „łuku" tak, by dopuszczała inne kształty.

Po obejrzeniu tych i kilku innych przykładów, program wypracuje następującą definicję łuku: „Graniastosłup podtrzymywany przez dwa ustawione pionowo klocki, nie stykające się ze sobą". Każdy element tej definicji pojawił się w wyniku popełnienia jakiegoś błędu, przez co definicja musiała być odpowiednio modyfikowana. Kiedy już program wypracuje właściwą definicję, przestaje robić błędy i nie wprowadza do niej dalszych zmian. Potrafi prawidłowo zidentyfikować jako łuk dowolny spełniający odpowiednie kryteria obiekt, jaki zostanie mu pokazany, nawet jeśli jakiegoś układu klocków nigdy przedtem nie widział. Program nauczył się definicji pojęcia „łuk".

SIECI NEURONOWE

Program Winstona uczy się pojęcia „łuk", ale takie pojęcia jak „stykanie się", „trójkątny klocek" czy „podtrzymywanie" były weń wbudowane od początku. Sposób reprezentacji rzeczywistości wykorzystywany przez ten program jest specjalnie dostosowany do analizy układów klocków. Poszukiwania ogólniejszego i bardziej uniwersalnego sposobu reprezentowania rzeczywistości skłoniły wielu badaczy do zainteresowania się układami komputerowymi o strukturze podobnej do istniejących w mózgu połączonych sieci neuronów. System taki nazywamy sztuczną **siecią neuronową**.

Sieć neuronowa to symulacja sieci neuronów. Taka symulacja może zostać przeprowadzona na do-

wolnym typie komputera, ponieważ jednak sztuczne neurony mogą działać współbieżnie, to naturalnym sposobem jej realizacji jest posłużenie się komputerem równoległym. Każdy sztuczny neuron ma jedno wyjście i wiele wejść, nawet setki lub tysiące. W najbardziej powszechnym typie sieci neuronowych sygnały przesyłane między neuronami są binarne — 1 lub 0. Wyjście jednego neuronu może zostać połączone z wejściami wielu innych. Każde wejście ma związaną z nim liczbę, zwaną **wagą**, która określa, jak silnie określony sygnał wejściowy wpływa na pojedynczy sygnał wyjściowy neuronu. Ta waga może być dowolną liczbą, dodatnią lub ujemną. Sygnał na wyjściu neuronu jest więc określany przez „głosowanie" sygnałów pojawiających się na jego wejściach i zmodyfikowanych przez wagi. Neuron oblicza swój sygnał wyjściowy mnożąc każdy sygnał wejściowy przez jego wagę i dodając wyniki; innymi słowy, dodaje wagi wszystkich wejść, na których pojawia się 1. Jeśli suma ważona osiąga pewną wartość, to sygnałem na wyjściu jest 1; w pozostałych przypadkach na wyjściu pojawia się 0.

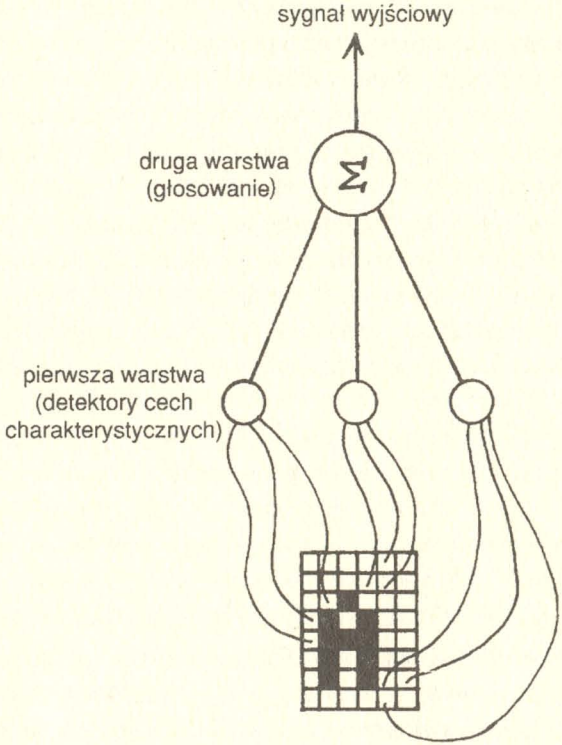
W bardzo grubym przybliżeniu działanie sztucznego neuronu odpowiada działaniu pewnego typu neuronów rzeczywiście występujących w mózgu. Prawdziwe neurony również mają jedno wyjście i wiele wejść, a ich połączenia, zwane **synapsami**, działają z różną siłą (co odpowiada różnym wagom wejściowym). Sygnał wejściowy może zarówno wzmacniać jak i hamować wysyłanie sygnału przez neuron (co odpowiada dodatnim i ujemnym wagom); neuron wysyła sygnał, kiedy połączona stymulacja z sygnałów wejściowych jest równa lub wyższa niż

określony próg. Pod tymi względami sztuczne neurony są podobne do prawdziwych. Neurony w mózgu są pod wieloma względami znacznie bardziej skomplikowane niż sztuczne, ale ta prosta sztuczna wersja wystarcza do zbudowania układów zdolnych do uczenia się.

Ważną właściwością sztucznych neuronów jest możliwość wykorzystania ich do wykonywania operacji logicznych AND, OR i NOT. Neuron będzie realizować funkcję OR, jeśli progiem dla sygnału wyjściowego jest 1 i każda z wag wejściowych jest równa lub większa niż 1. Neuron z progiem równym sumie wag będzie realizował funkcję AND, zaś neurony z jednym wejściem o wadze ujemnej i progiem równym 0 będą realizować funkcję NOT. Ponieważ każda funkcja logiczna może być zbudowana z połączenia funkcji AND, OR i NOT, to sieć neuronowa może realizować dowolną funkcję Boole'a. Sztuczne neurony są zatem uniwersalnymi elementami konstrukcyjnymi.

Niewiele wiadomo o tym, jak działa mózg ludzki, ale wydaje się, że pewne jego partie uczą się przez modyfikowanie siły reakcji synaps łączących neurony. Z pewnością jest to prawda w przypadku prostszych organizmów, na których możemy przeprowadzać eksperymenty — na przykład ślimaków morskich. Ślimaki morskie można nauczyć pewnych reakcji warunkowych i można też pokazać, że uczą się reakcji przez zmianę siły połączeń synaptycznych między neuronami. Jeśli proces uczenia się u ludzi przebiega w ten sam sposób, to wy także (mam nadzieję) odpowiednio dostrajacie połączenia w swoich mózgach w trakcie czytania tej książki.

Sieć sztucznych neuronów może „uczyć się", zmieniając wagi swoich połączeń. Dobrym przykładem jest prosty typ sieci neuronowej, zwany **perceptronem**, który potrafi nauczyć się rozpoznawania obrazów. Sposób, w jaki perceptron się uczy, ilustruje sposób działania większości sieci neuronowych. Perceptron jest siecią z dwiema warstwami neuronów i pojedynczym wyjściem. Każde wejście w pierwszej warstwie podłączone jest do urządzenia sensorycznego — na przykład do detektora światła, mie-



RYSUNEK 26.
Perceptron

rzącego jasność jakiegoś obszaru na obrazie. Każde wejście drugiej warstwy połączone jest z wyjściem z pierwszej warstwy, jak to pokazano na rysunku 26.

Wyobraźmy sobie, że próbujemy nauczyć perceptron rozpoznawania litery A. Można to osiągnąć pokazując mu wiele pozytywnych i negatywnych przykładów, aby dostosował wagi swojej drugiej warstwy w taki sposób, że sygnał wyjściowy będzie równy 1 wtedy i tylko wtedy, gdy pokazuje się mu A. Perceptron osiąga to przez dostrajanie wag za każdym razem, kiedy popełni błąd. Każdy neuron w pierwszej warstwie perceptronu postrzega mały fragment pokazywanego obrazu. Każdy z nich jest zaprogramowany tak, by rozpoznawać charakterystyczne cechy lokalne, jak pewne szczególne kąty lub linie o szczególnej orientacji; dokonuje tego za pomocą ustalonych wag przypisanych sygnałom wejściowym. Poniżej pokazany jest przykład układu negatywnych i pozytywnych wag wejściowych dla pola percepcji neuronu z pierwszej warstwy, zaprogramowanego do rozpoznawania szczytowego kąta litery A:

–	–	–	–	–	–
–	–	+	–	–	–
–	+	+	+	–	–
–	+	+	+	–	–
+	+	+	+	+	+
+	+	+	+	+	+

Pierwsza warstwa perceptronu zawiera tysiące takich neuronów, wykrywających charakterystyczne cechy prezentowanych obrazów; każdy z nich zaprogramowany jest do rozpoznawania szczególnego rodzaju lokalnych cech w określonej części pola

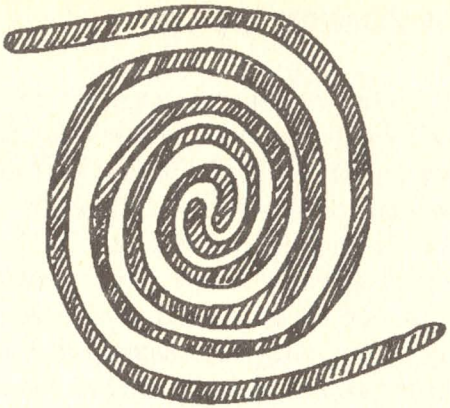
postrzegania. Neurony w pierwszej warstwie wykrywają w obrazie te cechy, które są użyteczne przy rozróżnianiu między literami. Perceptron — tak samo jak ludzkie oko — łatwiej rozpoznaje litery, jeśli zastosujemy czcionkę szeryfową.

Detektory lokalnych cech charakterystycznych w pierwszej warstwie dostarczają danych, a wagi drugiej warstwy określają, w jakich proporcjach te dane zostaną uwzględnione w sygnale wyjściowym. Na przykład, skierowany w górę wierzchołek kąta w górnej części obrazu przemawia za literą A, podczas gdy wierzchołek kąta skierowany w dół w środkowej części przemawia przeciwko tej hipotezie. Perceptron uczy się, dostosowując wagi dla sygnałów wejściowych w drugiej warstwie. Algorytm uczenia się jest bardzo prosty: jeśli trener stwierdzi, że perceptron popełnił błąd, to perceptron zmienia wartości wag wszystkich sygnałów wejściowych, które głosowały za błędnym rozpoznaniem. Czyni to w taki sposób, aby w przyszłości błędy były mniej prawdopodobne. Na przykład, jeśli perceptron nieprawidłowo rozpoznaje obraz jako literę A, to wagi wszystkich wejść, które głosowały na korzyść tego wyniku zostaną zmniejszone. Jeśli zaś perceptron nie zidentyfikuje właściwie litery A, to wejścia, które głosowały na korzyść jej rozpoznania zostaną wzmocnione. Jeśli perceptron ma wystarczającą liczbę detektorów wykrywających cechy charakterystyczne odpowiedniego typu, metoda ta w końcu spowoduje, że układ nauczy się rozpoznawać A.

Procedura uczenia się przez perceptron jest kolejnym przykładem sprzężenia zwrotnego. **Celem** jest właściwe ustawienie wag, **odchylenie** to błędna

identyfikacja przykładów treningowych, a **reakcją** jest dostosowywanie wag. Zauważmy, że perceptron — tak jak program Winstona rozpoznający łuk — uczy się jedynie na błędach. Jest to charakterystyczna cecha wszystkich układów uczących się, opartych na sprzężeniu zwrotnym. Po odpowiednio długim treningu procedura ta będzie zawsze prowadzić do właściwego wyboru wag — oczywiście przy założeniu, że istnieje zbiór wag, przy którym zadanie jest wykonalne. Perceptron może wydawać się idealnym urządzeniem do rozpoznawania obrazów, ale założenie, że istnieje jakiś układ wag, który wykona zadanie, stanowi pewne ograniczenie. Aby rozpoznać literę A w rozmaitych rozmiarach, czcionkach i położeniach perceptron potrzebuje bardzo bogatego zbioru detektorów cech charakterystycznych w pierwszej warstwie.

Perceptrony potrafią nauczyć się rozpoznawać dowolną literę, jeśli otrzymają wystarczającą liczbę obrazów do nauki, ale istnieją pewne rodzaje kształtów bardziej skomplikowanych od liter, które nie mogą zostać rozpoznane metodą jakiegokolwiek sumowania lokalnych cech. Na przykład, proste sumowanie sygnałów z małych fragmentów obrazu nie pozwala perceptronowi na stwierdzenie, czy wszystkie ciemne obszary na rysunku są połączone, ponieważ spójność jest własnością globalną; żadna charakterystyka lokalna sama z siebie nie może świadczyć na korzyść spójności lub przeciwko niej. Rysunek 27., pochodzący z książki Marvina Minsky'ego i Seymoura Paperta *Perceptrons*, pokazuje dlaczego ocena spójności obrazu wyłącznie na podstawie analizy charakterystyk lokalnych niekiedy nie jest możliwa.



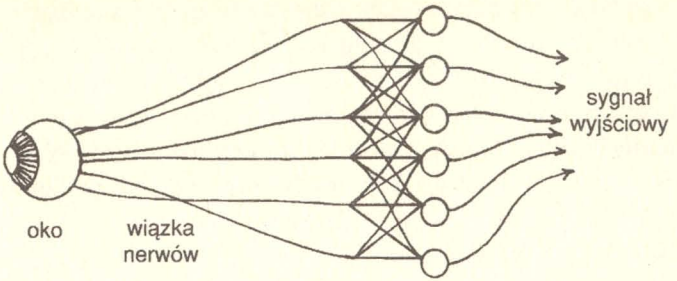
RYSUNEK 27.

Spirala perceptronowa

Z tych między innymi powodów dwuwarstwowe perceptrony nie są najbardziej praktycznymi sieciami neuronowymi do rozpoznawania większości typów kształtów. Bardziej uniwersalne sieci neuronowe, z większą liczbą warstw, potrafią już jednak rozpoznać bardziej skomplikowane wzory. Takie sieci uczą się za pomocą podobnych procedur. Wytrenowane sieci neuronowe tego rodzaju są często wykorzystywane do rozpoznawania obrazów i mowy — zadań, które są trudne do zdefiniowania za pomocą ustalonego zbioru zasad. Na przykład, proste układy rozpoznawania mowy, wbudowywane obecnie w wiele dziecięcych zabawek, opierają się na sieciach neuronowych.

Wadą systemu uczenia się opartego na pozytywnych i negatywnych przykładach jest to, że wymaga on trenera do sklasyfikowania przykładów. Istnieje jednak pewien rodzaj sieci neuronowych nie wymagających trenera — lub, mówiąc inaczej, istnieją sieci, w których sygnały treningowe wytwarzane są przez samą sieć. Taka samodzielnie trenująca się sieć jest układem **samoorganizującym się.** Układy samoorganizujące się badano od lat (ważne rezultaty w tej dziedzinie opublikował Alan Turing), ostatnio zaś nastąpił wyraźny wzrost zainteresowania tą dziedziną. Osiągnięto nawet pewne nowe rezultaty, częściowo dzięki dostępowi do szybszych komputerów. Podobnie jak trenowane sieci neuronowe, układy samoorganizujące się stanowią naturalne pole zastosowań dla komputerów równoległych.

Jako przykład funkcjonującego układu samoorganizującego rozważmy zagadnienie przesyłania obrazu z oka do mózgu (patrz rys. 28.). Siatkówka, na której wyświetlany jest obraz, to powierzchnia pokryta neuronami wrażliwymi na światło. Obraz na siatkówce przekształcany jest przez wiązkę neuronów w podobny obraz „wyświetlany" w mózgu. Jeśli w tej wiązce występuje błąd, to wyświetlany obraz będzie nieco zakłócony; każdy piksel będzie w trochę niewłaściwym miejscu. Opiszę samoorganizującą się sieć neuronową, która potrafi nauczyć się, jak porządkować taki zaburzony obraz, przywracając każdemu pikselowi właściwe miejsce. Układ korygujący zaburzenia składa się z pojedynczej warstwy neuronów, ułożonych w dwuwymiarową sieć.



RYSUNEK 28.

Oko z niewłaściwie podłączoną wiązką nerwów i układ korygujący zaburzenie.

Wyjścia tych neuronów dają skorygowany obraz. Jeśli obraz jest zakłócony tylko nieznacznie, to każdy piksel w zaburzonym obrazie znajduje się niedaleko swego właściwego miejsca. Wejścia każdego neuronu analizują sąsiedztwo pikseli w zaburzonym obrazie, a neuron uczy się, które z tych pikseli powinny być podłączone do wyjścia w celu stworzenia prawidłowego obrazu. Neuron tworzy połączenie, ustawiając wagi poprawnych wejść na 1 i wagi pozostałych wejść na 0.

Algorytm realizujący trening urządzenia do korygowania zaburzeń oparty jest na tym, że obrazy mają strukturę, która nie jest przypadkowa. Jak już mówiliśmy, obrazy zwykle nie są losowymi układami kropek, ale obrazami świata, tak więc sąsiadujące obszary na ogół wyglądają podobnie. Układ korygujący odwraca to stwierdzenie, przyjmując że piksele, które wyglądają podobnie, powinny znajdować się blisko siebie. Działanie neuronów w urządzeniu korygującym polega na mierzeniu korelacji każdego

z wejść z wyjściami sąsiadujących neuronów podczas wyświetlania serii obrazów. Gdy tylko neuron robi „błąd", polegający na przesłaniu sygnału innego od sygnału swoich sąsiadów, neuron zwiększa wagę dla wejścia, które zgadza się z wyjściami sąsiadów i obniża wagę dla pozostałych wejść. Oczywiście sąsiedzi również poznają swoje połączenia w tym samym czasie, można więc powiedzieć, że na początku przypomina to sytuację, kiedy ślepy prowadzi ślepego, ale w końcu niektóre z neuronów urządzenia korygującego zaczynają trafiać na swoje właściwe wejścia i w ten sposób stają się skutecznymi trenerami dla swoich sąsiadów. Znów, jedynymi dostrajanymi neuronami są te, które popełniały błędy. W miarę, jak neurony wzajemnie się trenują, z sygnałów wyjściowych zaczyna wyłaniać się niezaburzony obraz i w końcu sieć sama się organizuje, tworząc idealne odwzorowanie.

Samoregulujący się autopilot, program Patricka Winstona do rozpoznawania łuku, perceptron, urządzenie do korygowania zaburzeń — to tylko nieliczne przykłady układów, które się uczą. Wszystkie te układy działają w oparciu o zewnętrzne lub wewnętrzne sprzężenie zwrotne i wszystkie uczą się przez poprawianie własnych błędów. Konstrukcję każdego z nich inspirowały spełniające podobną funkcję układy biologiczne. Korzystając wyłącznie z tych owoców ewolucji, zachowujemy się trochę jak głupiec z bajki Ezopa *O gęsi, która zniosła złote jajko*, który wolał jajko od gęsi. W następnym rozdziale przyjrzymy się zatem samej gęsi.

Rozdział dziewiąty

WYKRACZAMY POZA INŻYNIERIĘ

Legenda głosi, że trzynastowieczny uczony i mnich Roger Bacon zajmował się również czarną magią i pewnego razu skonstruował mówiącą mechaniczną głowę. Według tej opowieści Bacon chciał obronić Anglię przed najeźdźcami, budując mur wokół królestwa. Stworzył głowę, aby spytać ją, w jaki sposób go zbudować. Bacon wykonał głowę z mosiądzu, dokładnie odtwarzając wszystkie szczegóły ludzkiej anatomii. Nagrzewał ją potem przez wiele dni nad ogniem, wymawiając magiczne zaklęcia. W końcu głowa przebudziła się i zaczęła mówić. Niestety, Bacon był już wtedy tak wyczerpany rzucaniem czarów, że zasnął, a jego młody asystent nie chciał budzić mistrza z powodu jakiegoś tam mamrotania mosiężnej głowy i ta wybuchła nad ogniem, zanim Bacon zdążył ją o cokolwiek zapytać.

Ta legenda o Baconie zawiera elementy wspólne wszystkim historiom o magach, którzy skonstruowali sztuczną inteligencję: Dedalu, Pigmalionie, Albercie Wielkim, rabinie z Pragi. W większości z nich pojawia się wątek gotowania lub jakiejś formy do-

jrzewania, koniecznego, aby coś zaczęło myśleć. W czasach przed nastaniem maszyn liczących niewiele osób wyobrażało sobie, że proces tak skomplikowany jak myślenie mógłby kiedykolwiek być sprowadzony do operacji realizowanych przez maszyny. Powszechnie sądzono, że jeśli kiedykolwiek zostanie stworzona inteligencja, będzie to wynikiem procesu emergentnego — czyli procesu, w którym skomplikowane zachowanie wyłania się jako globalna konsekwencja miliardów słabych oddziaływań lokalnych. Zakładano, że nie jest potrzebny prawidłowy schemat połączeń, ale właściwa recepta, zgodnie z którą składniki **same** zorganizują się w inteligencję. Inteligencja miałaby powstać, mimo że jej twórca — mag — nie rozumiałby dokładnie ani procesu prowadzącego do jej powstania, ani sposobu jej działania.

Może to się wyda dziwne, ale w zasadzie zgadzam się z tą przednaukową koncepcją: wierzę, że uda nam się stworzyć sztuczną inteligencję znacznie wcześniej, niż zrozumiemy inteligencję naturalną. Przypuszczam też, że proces jej konstruowania będzie polegał na stworzeniu warunków, w których inteligencja sama wyłoni się z szeregu złożonych oddziaływań, których nie będziemy rozumieć we wszystkich szczegółach — czyli z procesu bardziej przypominającego pieczenie ciasta lub uprawianie ogrodu niż konstruowanie maszyny. Nie zbudujemy sztucznej inteligencji; raczej przygotujemy właściwe warunki, w których inteligencja będzie mogła się wyłonić. Jest bardzo prawdopodobne, że największym osiągnięciem naszej techniki będzie stworzenie narzędzi, które pozwolą nam na wykroczenie **poza** inżynierię

— pozwolą nam tworzyć więcej, niż jesteśmy w stanie zrozumieć.

Zanim omówimy, w jaki sposób taki emergentny proces konstrukcyjny może przebiegać, rozważmy najlepszy znany nam przykład inteligencji: ludzki mózg. Ponieważ mózg został „zaprojektowany" przez emergentny proces darwinowskiej ewolucji, to porównanie go z omówionymi do tej pory konstrukcjami inżynierskimi może być użyteczne.

MÓZG

Ludzki mózg zawiera około 10^{12} neuronów, a każdy neuron ma przeciętnie 10^5 połączeń. Mózg jest do pewnego stopnia układem samoorganizującym się, ale byłoby niewłaściwe przyjmować, że jest czymś jednorodnym. Zawiera setki różnych typów neuronów, z których wiele występuje jedynie w niektórych obszarach. Badania tkanki mózgowej pokazują, że również sposób łączenia się neuronów jest inny w różnych obszarach mózgu: istnieje około pięćdziesięciu obszarów, w których schematy połączeń zauważalnie się różnią, a przypuszczalnie w znacznie większej liczbie różnice są zbyt subtelne, abyśmy potrafili je dostrzec.

Każdy obszar mózgu jest najwyraźniej wyspecjalizowany pod kątem jakiejś szczególnej funkcji, jak rozpoznawanie barw czy zmian intonacji albo zapamiętywanie nazw. Wiemy o tym stąd, że kiedy jakiś obszar zostaje uszkodzony na skutek wypadku lub udaru, towarzyszy temu również utrata odpowiednich funkcji. Na przykład, uszkodzenia w obszarach 44 i 45, po lewej stronie płata czołowego — nazy-

wanych wspólnie obszarem Broca — często pozbawiają chorego zdolności tworzenia wypowiedzi poprawnych pod względem gramatycznym. Osoby z tego typu uszkodzeniami w dalszym ciągu potrafią wymawiać wyraźnie słowa i rozumieć mowę innych, ale nie umieją konstruować poprawnych gramatycznie zdań. Uszkodzenie w obszarze znanym jako zakręt kątowy (*gyrus angularis*), umiejscowionym nieco dalej z tyłu głowy, powoduje trudności w czytaniu i pisaniu; uszkodzenia w jeszcze innych obszarach powodują niezdolność do przypomnienia sobie nazw dobrze znanych przedmiotów czy rozpoznawania znajomych twarzy.

Oczywiście nie należy twierdzić, że rozmaite obszary mózgu spełniają rolę analogiczną do funkcjonalnych elementów konstrukcyjnych komputera. Po pierwsze, uszkodzenia w większości obszarów nie powodują utraty jakiejś dobrze określonej funkcji: na przykład, usunięcie większości prawego płata czołowego czasami prowadzi do trudnych do zdefiniowania zmian w osobowości, ale w wielu wypadkach w ogóle nie pojawiają się żadne zmiany. Nawet w tych przypadkach, kiedy utrata funkcji jest dobrze określona, wcale nie jest oczywiste, że dana funkcja była w całości realizowana przez uszkodzony obszar; być może po prostu ten obszar dostarczał jakiegoś małego elementu wspomagającego, niezbędnego do jej wykonywania — samochód z rozładowanym akumulatorem nie ruszy z miejsca, ale przecież nie wyciągamy z tego wniosku, że samochód jest napędzany przez akumulator.

Istnieją też obszary w mózgu — w szczególności w tylnej jego części, związane z przetwarzaniem

bodźców wzrokowych — w których możemy znaleźć pewne prawidłowości w układzie połączeń: na przykład połączenia związane z odbieraniem sygnałów z lewego i prawego oka w celu stworzenia poczucia głębi w widzeniu przestrzennym. Ale w przeważającej części mózgu „schemat połączeń" pozostaje zagadką. Nawet sama koncepcja, że większa część mózgu jest w sposób organiczny przystosowana do spełniania pewnych funkcji, może okazać się błędna. Wiemy już, że przetwarzanie języka wydaje się zachodzić głównie po lewej stronie, podczas gdy umiejętności związane z orientacją przestrzenną, na przykład zdolność odczytywania mapy, wydają się być domeną głównie strony prawej. A jednak pod mikroskopem układ tkanek po lewej i prawej stronie wygląda bardzo podobnie. Jeśli jest jakaś systematyczna różnica między schematem połączeń w obu półkulach mózgu, jest ona zbyt subtelna, abyśmy mogli ją zauważyć.

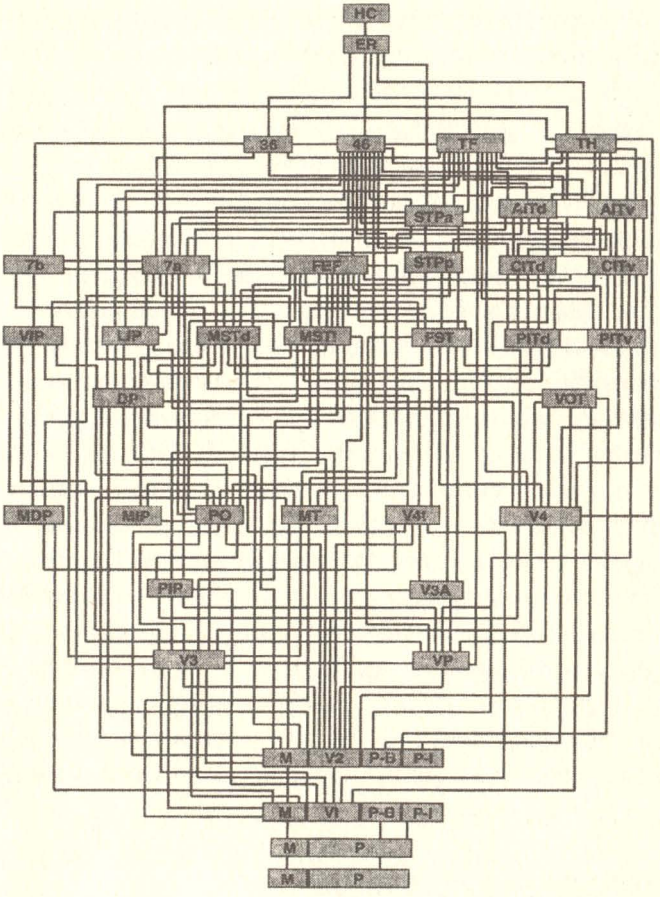
Możliwe jest, że mózg uczy się w swego rodzaju samoorganizującym się procesie, który zmienia siły różnych połączeń synaptycznych w celu przygotowania danego obszaru do wykonywania określonej funkcji. Z pewnością do jakiegoś stopnia tak się dzieje. Wiemy na przykład, że małpy pozbawione palca nadal używają obszaru w mózgu, który wcześniej przetwarzał informacje z tego palca: bezczynne neurony zaczynają przetwarzać impulsy z pozostałych palców. Prawdopodobnie w podobny sposób ludzie przegrupowują funkcje mózgowe w procesie rekonwalescencji po udarze. Ofiara udaru może mieć początkowo trudności z pewnymi zadaniami, takimi jak chociażby rozpoznawanie twarzy, ale potem od-

zyskuje te umiejętności. Ponieważ uszkodzone neurony nie mogą się zregenerować, pacjent prawdopodobnie ponownie uczy się funkcji, angażując neurony z innych partii mózgu.

Jeśli za uczenie się funkcji takich jak rozpoznawanie twarzy czy rozumienie języka odpowiedzialne są różne obszary mózgu, to funkcje te muszą być w jakimś sensie wbudowane od początku. Noworodki są szczególnie zainteresowane twarzami przez pierwszych kilka dni życia i uczą się je rozpoznawać na długo przed tym, nim nauczą się rozróżniać znacznie prostsze kształty, na przykład litery. Podobnie, dzieci wydają się predysponowane do zwracania uwagi na pewne prawidłowości w mowie, pozwalające im nauczyć się słów i gramatyki. Funkcje przetwarzające język czy też odpowiedzialne za rozpoznawanie twarzy umiejscowione zostają w różnych obszarach mózgu przypuszczalnie dlatego, że te właśnie części mózgu zostały w jakiś sposób poinstruowane, aby wykonywać tak różne działania.

Nawet w tych częściach mózgu, którym funkcje wydają się być przypisane w sposób organiczny, układ połączeń wykazuje mało podobieństw do hierarchicznej struktury funkcjonalnych bloków w komputerze: przede wszystkim nie ma prostego układu sygnałów wejściowych, przechodzących w sygnały wyjściowe. Połączenia często są dwukierunkowe, z jednym zbiorem neuronów przekazującym sygnały w jednym kierunku i komplementarnym zbiorem, przekazującym sygnały w kierunku przeciwnym. Rysunek 29. pokazuje schemat połączeń w korze wzrokowej makaka. Każda linia na diagramie przedstawia wiązkę wielu tysięcy neuronów, razem z komplementarną

wiązką biegnącą w przeciwnym kierunku. Na pierwszy rzut oka wydaje się, że wszystko łączy się ze wszystkim — w przeciwieństwie do eleganckich, hierarchicznych schematów obwodów komputerowych.



RYSUNEK 29.

Schemat blokowy kory wzrokowej makaka

Ważne, że mózg jest nie tylko bardzo skomplikowany, ale ma też zupełnie inną strukturę niż wytwory pracy inżyniera. Nie znaczy to wprawdzie, że nigdy nie będziemy mogli skonstruować maszyny realizującej funkcje ludzkiego mózgu, lecz nie możemy oczekiwać, iż zrozumiemy inteligencję przez rozkładanie jej na części i analizowanie tak, jakby była hierarchicznie skonstruowaną maszyną.

Możliwe, że zadowalający opis tego, co robi mózg, będzie równie skomplikowany, jak opis struktury mózgu — oznaczałoby to, że nie ma żadnych rozsądnych kategorii, w których moglibyśmy go zrozumieć. W inżynierii sposobem radzenia sobie ze złożonością jest rozłożenie czegoś na części. Kiedy już zrozumiemy każdą część z osobna, możemy zrozumieć oddziaływania między nimi. Poszczególne elementy poznajemy stosując ten proces w sposób rekursywny, dzieląc każdą z części na mniejsze... i tak dalej. Konstrukcja komputera elektronicznego, razem z całym oprogramowaniem, jest imponującym dowodem na to, jak daleko posunąć można taki proces. Tak długo, jak funkcja każdej części jest uważnie wyspecyfikowana i zrealizowana, i tak długo, jak oddziaływania między częściami są pod kontrolą i są przewidywalne, ten system „dziel i rządź" funkcjonuje bardzo dobrze. Jednak układ będący wynikiem ewolucji, taki jak mózg, wcale nie musi mieć struktury hierarchicznej tego typu.

KŁOPOTY Z MODULARNOŚCIĄ

Poleganie na ścisłej hierarchicznej strukturze jest piętą achillesową projektowanych konstrukcji, po-

nieważ nieuchronnie wiąże się ze specyficznym dla maszyn brakiem elastyczności. Jak to omawialiśmy w rozdziale 6., systemy hierarchiczne są delikatne — w tym sensie, że są podatne na katastrofalne w skutkach awarie. Każda część maszyny, będącej efektem pracy inżyniera, musi odpowiadać wymaganiom projektowym co do oddziaływań z innymi elementami. Te wymagania pełnią funkcję swego rodzaju kontraktu między elementami składowymi. Jeśli jedna ze składowych nie wywiązuje się ze swojego kontraktu, to założenia projektowe układu nie są spełnione i cały układ załamuje się w nieprzewidywalny sposób. Awaria najdrobniejszej części może spowodować w systemie zmiany o katastrofalnych skutkach. Oczywiście układy złożone, jak komputery czy samoloty, konstruowane są tak, by unikać tych tak zwanych „punktowych awarii" dzięki redundancji, opisanej w rozdziale 6. Jednak tego typu zabezpieczenia mogą chronić układ jedynie przed awariami, które potrafimy przewidzieć, co oznacza, że wszystkie możliwe konsekwencje każdej awarii powinny być przewidziane i zrozumiane. A to zadanie staje się coraz trudniejsze w miarę wzrostu stopnia złożoności maszyn.

Problem sięga głębiej niż awarie pojedynczych elementów. W złożonym układzie nawet poprawnie funkcjonujące elementy mogą w wyniku wzajemnych oddziaływań wygenerować zachowania nieoczekiwane. Kiedy duży program źle działa, programiści odpowiedzialni za jego części składowe często potrafią w przekonujący sposób argumentować, że ich podprogramy pracują właściwie. I nierzadko wszyscy mają rację, w tym sensie, że każdy podprogram rzeczywiście prawidłowo wykonuje swoje specyficzne

zadanie. Błąd polega na nieprecyzyjnych instrukcjach dotyczących tego, co poszczególne części powinny robić i jak powinny między sobą oddziaływać. Duże, złożone systemy, takie jak systemy operacyjne komputerów czy sieci telefoniczne, często wykazują zaskakujące i nieoczekiwane zachowania nawet wtedy, gdy każda część działa zgodnie z projektem. Na przykład, parę lat temu przez kilka godzin przestały funkcjonować połączenia międzymiastowe we wschodnich stanach USA. System łączności oparty był na zaawansowanej, odpornej na błędy konstrukcji, działającej zgodnie z zasadą redundancji. Wszystkie elementy funkcjonowały poprawnie, ale nieprzewidziane oddziaływanie między dwiema wersjami oprogramowania, pracującymi w różnych centralach, spowodowało awarię całego systemu.

Czasem wręcz zadziwia mnie, że techniki tworzenia nowych układów funkcjonują tak dobrze. Projektowanie czegoś równie skomplikowanego jak komputer czy system operacyjny może wymagać pracy tysięcy ludzi. Jeśli system jest wystarczająco skomplikowany, to nikt nie ma jego pełnego obrazu. Taka sytuacja prowadzi zwykle do pojawienia się błędów wynikających ze złej komunikacji interfejsów lub nieefektywnych rozwiązań zastosowanych w projekcie. W miarę, jak system staje się coraz bardziej skomplikowany, kłopoty z interfejsami pogłębiają się.

Należy zauważyć, że przedstawione powyżej problemy nie są nieuniknionymi słabościami maszyn czy programów jako takich. Są to niedoskonałości procesu projektowania nowych urządzeń. Wiemy, że nie wszystko, co skomplikowane, musi być jednocześnie podatne na awarie. Mózg jest znacznie bar-

dziej skomplikowany, a jednak znacznie odporniejszy od komputera. Kontrast między stopniem niezawodności mózgu i komputera dobrze ilustruje różnicę między produktami ewolucji a produktami techniki. Pojedynczy błąd w programie komputerowym może spowodować przerwanie pracy systemu operacyjnego, natomiast mózg zwykle radzi sobie ze złymi pomysłami czy błędnymi informacjami, a nawet ze źle działającymi częściami. Pojedyncze neurony w mózgu nieustannie umierają i nigdy nie są zastępowane; jeśli tylko uszkodzenia nie są zbyt poważne, mózg potrafi dostosować się i skompensować powstałe braki. (Jak na ironię, kiedy pisałem ten rozdział mój komputer zawiesił się i musiałem go zresetować. Ludziom rzadko się zdarza, aby padł ich system operacyjny.)

SYMULOWANA EWOLUCJA

Czy mamy jakąś alternatywę dla projektowania gdy chcemy stworzyć sztuczną inteligencję? Jedną z możliwości jest odtworzenie w komputerze procesu ewolucji biologicznej. Symulowana ewolucja to inna metoda konstruowania skomplikowanych układów komputerowych i oprogramowania — metoda, która unika wielu problemów inżynierii. Aby zrozumieć, jak działa symulowana ewolucja, przeanalizujmy konkretny przykład. Przypuśćmy, że chcemy stworzyć program, który będzie ustawiał liczby w porządku malejącym. Standardowym podejściem inżyniera byłoby napisanie takiego programu w oparciu o jeden z algorytmów sortujących, opisanych w rozdzia-

le 5. Rozważmy jednak, jak — zamiast tego — moglibyśmy „wyewoluować" taki program.

Pierwszy krok polega na wytworzeniu „populacji" losowych programów. Możemy ją stworzyć, posługując się generatorem liczb pseudolosowych (patrz rozdział 4.). Aby przyspieszyć ten proces, możemy używać tylko tych rozkazów, które są użyteczne przy sortowaniu, jak na przykład rozkazy porównywania i wymiany. Każdy z tych losowych ciągów rozkazów jest programem: losowa populacja będzie zaś zawierać — powiedzmy — 10 000 takich programów, liczących po kilkaset rozkazów. Kolejny krok polega na przetestowaniu populacji, w celu wybrania programów najbardziej skutecznych. W tym celu musimy wykonać każdy z nich, aby zobaczyć, czy potrafi on prawidłowo posortować testową sekwencję liczb. Oczywiście, ponieważ programy są losowe, jest mało prawdopodobne, by przeszły ten test — ale (przez przypadek) niektóre osiągną uporządkowanie bliższe właściwego niż inne. Na przykład, właśnie przez przypadek, program może przesuwać nieduże liczby na koniec ciągu. Sprawdzając programy na kilku różnych ciągach liczb, możemy każdemu z nich przypisać pewną wielkość, określającą stopień jego umiejętności wykonania zadania.

Następnym krokiem jest stworzenie nowej populacji, wywodzącej się z programów z wyższymi ocenami. Aby to osiągnąć, usuwamy programy z ocenami niższymi od przeciętnej; pozostają tylko programy „najlepiej przystosowane". Nowa populacja powstaje przez tworzenie kopii wyselekcjonowanych programów, z małymi losowymi zmianami, czyli analogicznie do procesu rozmnażania bezpłciowego z mu-

tacjami. Inna możliwość polega na „rozmnażaniu" programów, przez łączenie w pary programów wyselekcjonowanych z poprzedniego pokolenia, czyli w sposób analogiczny do rozmnażania płciowego. Programy-dzieci tworzymy przez łączenie ciągów rozkazów pobranych z programów-rodziców. „Rodzice" przetrwali selekcję prawdopodobnie dlatego, że zawierali użyteczne ciągi rozkazów, istnieje więc pewne prawdopodobieństwo, że „dziecko" odziedziczy najbardziej użyteczne cechy po każdym ze swoich „rodziców". Nowe pokolenie programów zostaje poddane tym samym procedurom testowania i selekcji, i znów przeżywają i rozmnażają się „osobniki" najlepiej przystosowane. Komputer równoległy będzie wytwarzać nowe pokolenie co kilka sekund, tak więc procesy selekcji i różnicowania mogą być bez trudu powtarzane wiele tysięcy razy. W każdym kolejnym pokoleniu przeciętny stopień zdatności programów będzie rósł — mówiąc inaczej, programy będą coraz lepiej przeprowadzały sortowanie, a po kilku tysiącach pokoleń będą sortować bezbłędnie.

Wiem, że ten proces właśnie tak działa, ponieważ sam używałem symulowanej ewolucji do tworzenia programów, mających rozwiązać konkretne problemy wymagające sortowania. W swoich eksperymentach premiowałem właśnie programy, które sortowały próbne ciągi szybko, w ten sposób szybsze programy miały więcej szans na przetrwanie. Ten ewolucyjny proces doprowadził do powstania bardzo szybkich programów sortujących. Dla zagadnień, które mnie interesowały, programy otrzymane drogą ewolucji były nawet nieznacznie szybsze niż którykolwiek z algorytmów opisanych w rozdziale 5. —

i nawet szybciej porządkowały ciągi liczby niż jakikolwiek program, który sam potrafiłbym napisać.

Ciekawe, że sam nie rozumiem, w jaki sposób działają programy sortujące otrzymane drogą ewolucji w moim doświadczeniu. Uważnie analizowałem występujące w nich ciągi poleceń, ale nadal ich nie rozumiem: nie potrafię znaleźć żadnego prostszego wytłumaczenia, jak te programy działają, oprócz samych ciągów rozkazów. Być może takich programów w ogóle nie da się zrozumieć — może nie istnieje sposób, by ich działanie rozłożyć na hierarchie możliwych do zrozumienia elementów. Jeśli to prawda — jeśli ewolucja może wytworzyć coś tak prostego jak program sortujący, który ze swej istoty jest niemożliwy do zrozumienia — to nie wróży to dobrze próbom zrozumienia działania ludzkiego mózgu. To, że programy sortujące otrzymane w procesie ewolucji sortują rzeczywiście bezbłędnie, udowodniłem za pomocą matematycznych testów, ale i tak bardziej wierzę w proces, który doprowadził do ich powstania, niż w moje testy. Po prostu wiem, że każdy z programów sortujących otrzymanych drogą ewolucji jest potomkiem długiej linii „przodków", których przetrwanie zależało od ich zdolności do sortowania.

Fakt, że oprogramowanie otrzymane drogą ewolucji czasami może być niezrozumiałe, powoduje, że niektórzy wahają się przed używaniem takich programów w praktyce. Myślę jednak, że ich obawy wynikają z błędnych założeń. Jednym z nich jest sąd, że struktura systemów powstałych według projektów inżynierskich daje się łatwo wyjaśnić — tymczasem jest to prawdą jedynie w przypadku systemów względnie prostych. Na przykład, nie ma ludzi, któ-

rzy naprawdę dobrze rozumieliby złożone systemy operacyjne. Drugim fałszywym założeniem jest, że systemy są mniej wiarygodne, jeśli nie potrafimy wytłumaczyć, jak działają. Mając do wyboru lot samolotem pilotowanym przez zaprojektowany program komputerowy a samolotem pilotowanym przez człowieka, wybrałbym to drugie; i zrobiłbym tak, mimo że nie rozumiem, jak działa człowiek-pilot. Po prostu wierzę w proces, którego tworem jest człowiek. Tak jak w przypadku programów sortujących, wiem, że pilot wywodzi się z długiej linii jednostek, które przetrwały selekcję. Dlatego też, jeśli bezpieczeństwo samolotu zależałoby od poprawnego sortowania liczb, to wolałbym skorzystać z programu otrzymanego drogą ewolucji niż programu napisanego przez zespół programistów.

JAK WYEWOLUOWAĆ MASZYNĘ MYŚLĄCĄ?

Symulowana ewolucja sama w sobie nie jest rozwiązaniem problemu stworzenia maszyny myślącej, ale wskazuje nam właściwy kierunek postępowania. Kluczowy pomysł polega na rezygnacji ze skomplikowanych konstrukcji o strukturze hierarchicznej i wykorzystaniu kombinatorycznych możliwości komputera. W istocie, symulowana ewolucja jest rodzajem heurystycznej techniki poszukiwań w przestrzeni możliwych konstrukcji. Heurystyki używane do przeszukiwania przestrzeni możliwości to **Wypróbuj konstrukcje podobne do najlepszych konstrukcji znalezionych do tej pory** i **Połącz elementy dwóch udanych konstrukcji.** Obie dają dobre efekty.

Symulowana ewolucja jest dobrym sposobem na tworzenie nowych struktur, ale jest mało efektywna przy doskonaleniu konstrukcji istniejących. Jej wady i zalety wynikają z faktu, że ewolucja z istoty swej nie dostrzega pytań, które można zawrzeć w samej konstrukcji. W odróżnieniu od układów ze sprzężeniem zwrotnym, opisanych w poprzednich rozdziałach, w których wprowadzane były określone zmiany, aby poprawić konkretne błędy, ewolucja wybiera warianty na ślepo, nie biorąc pod uwagę, w jaki sposób zmiany wpłyną na wynik.

Ludzki mózg wykorzystuje oba mechanizmy: jest w takim samym stopniu produktem procesów kształcenia, co ewolucji. Ewolucja tworzy podstawy, a rozwój osobnika — w interakcji z otoczeniem — dopełnia obrazu. W istocie, produktem ewolucji jest nie tyle konstrukcja mózgu, ale konstrukcja procesu, który tworzy mózg — nie plany, ale przepis. Mamy więc wiele poziomów, na których procesy emergentne funkcjonują równocześnie. Proces ewolucyjny tworzy przepis na wyhodowanie mózgu, a procesy rozwojowe oddziałują z otoczeniem w celu wytworzenia w nim odpowiednich połączeń. Proces rozwojowy obejmuje zarówno stymulowane wewnętrznie procesy morfogenezy, jak i stymulowane zewnętrznie procesy uczenia się. Rozwojowe siły morfogenezy powodują, że komórki nerwowe rozwijają się we właściwe układy, a proces uczenia się dostraja połączenia między komórkami. Ostatecznym etapem uczenia się mózgu jest proces uczestniczenia w kulturze, w którym przekazywana jest mu wiedza zebrana przez inne jednostki na przestrzeni wielu pokoleń.

Opisałem każdy z tych emergentnych mechanizmów (ewolucja, morfogeneza, uczenie się) tak, jakby były odrębnymi procesami, ale w rzeczywistości są one synergicznie splecione. Nie ma wyraźnej linii, która oddzielałaby rozwojowe siły morfogenezy od kształcących procesów kultury. Kiedy matka przemawia czule do swojego nowonarodzonego dziecka, jest to zarówno proces kształcący, jak i proces stymulujący rozwój dziecięcego mózgu. Proces morfogenezy jest sam w sobie procesem adaptacyjnym, w którym każda komórka rozwija się w nieustannym oddziaływaniu z pozostałymi komórkami w organizmie, w złożonym procesie ze sprzężeniem zwrotnym, który dąży do poprawiania błędów i utrzymania właściwego kierunku rozwoju organizmu.

Zachodzą też synergiczne oddziaływania między procesami ewolucyjnymi, powodującymi powstanie gatunków, a procesami rozwojowymi powodującymi powstanie osobników. Najlepszy przykład oddziaływania między rozwojem a ewolucją znany jest jako efekt Baldwina, jako że po raz pierwszy został opisany przez biologa ewolucyjnego Jamesa Baldwina w roku 1896, ponownie zaś odkrył go informatyk Geoffrey Hinton prawie sto lat później. Podstawową koncepcją w efekcie Baldwina jest obserwacja, że jeśli połączyć ewolucję z rozwojem, to ewolucja może przebiegać szybciej; proces adaptacyjny rozwoju jednostki może naprawić błędy w niedoskonałej konstrukcji ewolucyjnej. Aby zrozumieć efekt Baldwina należy docenić, jak trudne jest otrzymanie na drodze ewolucji cech wymagających zajścia wielu mutacji naraz. Rozważmy ewolucję instynktu budowania gniazda u ptaków. Wydaje się rozsądne założenie, że

budowa gniazda wymaga wielu samodzielnych kroków — znalezienie gałązki, podniesienie jej, przeniesienie do gniazda i tak dalej. Przyjmujmy również, dla potrzeb tego przykładu, że każdy z tych kroków wymaga innej mutacji, a korzyść dla ptaka (w postaci ukończonego gniazda) wymaga kompletnego zbioru mutacji. Innymi słowy, nawet jeśli brakuje tylko pojedynczego kroku, to gniazdo nie zostanie w ogóle zbudowane i ptak nie będzie lepiej przystosowany niż jego towarzysze — nie osiągnie żadnej przewagi ewolucyjnej. Oczywiście problem z otrzymywaniem takich cech drogą ewolucji polega na tym, że ewolucja preferuje każdą z potrzebnych mutacji tylko wtedy, gdy obecne są wszystkie pozostałe: zaś równoczesne zajście wszystkich niezbędnych mutacji u jednego osobnika jest zdarzeniem bardzo mało prawdopodobnym. Ponieważ żaden z poszczególnych kroków nie przynosi korzyści sam w sobie, trudno wyobrazić sobie, jak zachowanie takie jak budowanie gniazd w ogóle mogło się pojawić na drodze ewolucji.

Efekt Baldwina polega na synergicznym oddziaływaniu między ewolucją a uczeniem się. To właśnie oddziaływanie pomaga rozwiązać problem, premiując ptaka za każdą mutację, która odpowiada za pojedynczy krok w tym zadaniu. Ptak rodzący się z wiedzą o tym, jak wykonywać niektóre kroki, będzie mieć przewagę nad innym, który tego nie wie, ponieważ będzie musiał nauczyć się mniejszej ilości kroków. Jest więc bardziej prawdopodobne, że posiądzie umiejętności umożliwiające skuteczną budowę gniazd. Każda pojedyncza umiejętność, z którą ptak się rodzi, zwiększa możliwości uczenia się, jest zatem wartościowa sama w sobie. Z tej perspektywy widać, że

każda pojedyncza mutacja będzie premiowana niezależnie, więc zdolność budowania gniazd wyniknie z kroków, które dodawane są do instynktownego repertuaru zachowań ptaka stopniowo i w krótszym czasie, niż zajęłoby to probabilistycznej fluktuacji, wytwarzającej wszystkie mutacje za jednym razem, w jednym osobniku. Dzięki temu, że ptak potrafi się uczyć, ewolucja przebiega szybciej. Efekt Baldwina stosuje się nie tylko do uczenia się, ale też do wielu mechanizmów adaptacyjnych w rozwoju jednostki.

Mój optymizm co do perspektyw stworzenia maszyny myślącej drogą ewolucji wynika częściowo z tego, że nie musimy zaczynać od zera. Możemy wyposażyć początkową populację maszyn w struktury takie, jakie obserwujemy w mózgu. Możemy również zacząć od wszelkich wzorów rozwoju i uczenia się, jakie można dostrzec w systemach naturalnych, nawet jeśli nie rozumiemy ich do końca. To powinno pomóc, także wówczas gdy nasze przypuszczenia nie są zupełnie słuszne, ponieważ rozpoczęcie poszukiwań gdziekolwiek w pobliżu rozwiązania jest przypuszczalnie znacznie lepsze od zaczynania ich w sposób przypadkowy. Jeśli włączymy do tego procesu **jakikolwiek** model rozwoju, to ewolucja maszyny myślącej będzie mogła skorzystać z efektu Baldwina.

Inny proces, radykalnie skracający czas potrzeby do wykształcenia złożonych zachowań, to uczenie. Ludzkie dziecko rozwija swoją inteligencję przynajmniej częściowo dzięki temu, że żyje wśród innych ludzi, od których może się uczyć. Część tej wiedzy zostaje nabyta na zasadzie zwykłej imitacji, część zaś przez świadome uczenie się. Ludzki język jest spektakularnym mechanizmem, umożliwiającym prze-

kaz koncepcji z jednego umysłu do drugiego, pozwalającym nam na kumulowanie użytecznej wiedzy i zachowań na przestrzeni wielu pokoleń w tempie znacznie szybszym niż ewolucja biologiczna. „Przepis" na ludzką inteligencję opiera się w takim samym stopniu na ludzkiej kulturze, co na ludzkim genotypie.

Jednak, jeśli nawet zaczniemy z całą dostępną obecnie wiedzą, nie spodziewam się, aby udało się nam otrzymać drogą ewolucji rozwiniętą sztuczną inteligencję w jednym kroku. Oto szkic tego, jak mogłyby przebiegać kolejne etapy. Rozpocząć trzeba by prawdopodobnie od wyewoluowania konstrukcji z inteligencją, powiedzmy, owada, tworząc proste środowisko, w którym inteligencja na takim poziomie jest premiowana. Wystartować należałoby zatem od populacji predysponowanej (przez mechanizmy rozwojowe) do wykształcenia tego rodzaju struktur neuronowych, jakie spotykamy u owadów. Przez sekwencję kolejno coraz bogatszych symulowanych środowisk moglibyśmy ostatecznie doprowadzić do ewolucji naszej inteligencji w inteligencję żaby, potem myszy itd. Nawet osiągnięcie tego etapu pochłonie prawdopodobnie dziesiątki lat pracy i wielokrotnie zabrniemy w ślepe uliczki, ale ostatecznie właśnie ten kierunek badań może prowadzić do otrzymania na drodze ewolucji sztucznej inteligencji o stopniu złożoności i elastyczności porównywalnym z mózgiem naczelnych.

Gdyby kiedykolwiek udało się nam otrzymać metodą ewolucji maszynę, która może rozumieć język, bylibyśmy w stanie pójść szybko do przodu, korzystając z ludzkiej kultury. Wyobrażam sobie, że musielibyśmy uczyć inteligentną maszynę w dużym

stopniu tak, jak uczymy ludzkie dzieci, z tą samą mieszaniną umiejętności, faktów, morałów i bajek. Ponieważ włączalibyśmy ludzką kulturę do przepisu na inteligencję maszyny, to konstrukcja, która by w ten sposób powstała, nie byłaby w zasadzie sztuczną inteligencją, ale raczej ludzką inteligencją zawartą w sztucznym umyśle. Dlatego właśnie oczekuję, że szybko czyniłaby postępy. Oczywiście zdaję sobie sprawę, że budowanie takiej maszyny stworzy mnóstwo problemów moralnych. Na przykład, kiedy już zostanie stworzona, czy niemoralne będzie jej wyłączenie? Przypuszczam, że byłoby niewłaściwe, ale nie chcę sprawić wrażenia, że mam pewność co do moralnego statusu inteligentnego urządzenia. Na szczęście, będziemy mieć dużo czasu na znalezienie odpowiedzi na takie pytania.

Zwykle jednak ludzi interesują nie tyle praktyczne problemy moralne hipotetycznej przyszłości, co raczej filozoficzne pytania, jakie nasuwa sama możliwość istnienia sztucznej inteligencji. Większość z nas nie lubi być porównywana z maszynami. Jest to zrozumiałe: porównywanie do głupich maszyn, jak tostery, samochody, czy nawet dzisiejsze komputery, powinno być dla nas obrazą. Tezę, że umysł jest kuzynem współczesnego komputera, odbieramy jako równie poniżającą, co stwierdzenie, że istota ludzka jest spokrewniona ze ślimakiem. A jednak oba twierdzenia są prawdziwe i oba mogą być pomocne. Tak, jak możemy nauczyć się czegoś o nas samych, badając struktury neuronowe ślimaka, możemy też nauczyć się czegoś o sobie, przyglądając się prostej karykaturze procesów myślowych w dzisiejszych kom-

puterach. Możemy być zwierzętami, ale w pewnym sensie nasz mózg jest rodzajem maszyny.

Wielu moich wierzących przyjaciół jest zaszokowanych, że postrzegam ludzki mózg jako maszynę a myślenie jako proces obliczeniowy. Z drugiej strony, wielu moich przyjaciół naukowców zarzuca mi, że jestem mistykiem, ponieważ wierzę, że możemy nigdy nie osiągnąć całkowitego zrozumienia istoty fenomenu myślenia. A jednak, jestem przekonany, że ani religia, ani nauka, nie wiedzą jeszcze wszystkiego. Podejrzewam, że świadomość jest konsekwencją działania zwykłych praw fizycznych i przejawem złożonych obliczeń, ale nie wydaje mi się ona przez to mniej tajemnicza i wspaniała — jest raczej wręcz przeciwnie. Między sygnałami naszych neuronów i naszymi myślami widnieje przepaść tak duża, że być może nigdy nie zostanie nad nią przerzucony pomost ludzkiego zrozumienia. Tak więc, kiedy mówię, że mózg jest maszyną, to nie chcę ubliżyć umysłowi, ale chcę uznać potencjalne możliwości maszyny. Nie twierdzę, że ludzki umysł jest czymś skromniejszym, niż to sobie wyobrażamy — sądzę raczej, że maszyna może być czymś znacznie, znacznie większym.

LITERATURA UZUPEŁNIAJĄCA

Hillis W. Daniel, *The Connection Machine*, MIT Press Series in Artificial Intelligence, MIT 1989.

Knuth Donald Ervin, *The Art of Computer Programming*, Addison-Wesley 1997.

Minsky Marvin Lee, *Computation: Finite and Infinite Machines*, Prentice Hall 1967.

Patterson David A. i John L. Hennessy, *Computer Architecture: A Quantitative Approach*, wyd. drugie, Morgan Kaufman Publishers 1996.

Wiener Norbert, *Human Use of Human Beings: Cybernetics and Society*, Avon 1986.

Winston Patrick Henry, *Artificial Intelligence,* wyd. trzecie, Addison-Wesley 1998.

PODZIĘKOWANIA

Pomysł napisania tej książki podsunął mi John Brockman, który uważał, że potrzebne jest jakieś krótkie opracowanie, podsumowujące koncepcje związane z komputerami. Początkowo wydawało mi się to prostym zadaniem, ale szybko odkryłem, że napisanie krótkiej książki o tak złożonym zagadnieniu może być trudniejsze, niż napisanie czegoś obszernego.

Wdzięczny jestem Johnowi — a także Williamowi Fruchtowi z Basic Books — za ich pomoc w mym przedsięwzięciu. Napisałem tę książkę w trakcie pobytu w Media Laboratory na MIT i chciałbym podziękować wszystkim profesorom i studentom, którzy pomagali mi i wspierali mnie w mej pracy, szczególnie zaś twórcy laboratorium i jego dyrektorowi, Nicholasowi Negroponte. W przygotowaniu pierwszych wersji książki bardzo pomogła mi Debbie Widener, a także Bettylou McClanahan i Peggi Oakley. Większości z tego, co opisane jest w mojej książce, nauczyłem się od mojego przyjaciela i mistrza Marvina Minsky'ego i innych naukowców z MIT, w tym od

Geralda Sussmana, Claude'a Shannona, Seymoura Paperta, Tomaso Poggio, Patricka Winstona i Toma Knighta.

Chciałby również podziękować tym, którzy przeczytali wczesne wersje tej książki i przekazali mi pomocne komentarze. Są to: Jerry Lyons, Seymour Papert, George Dyson, Chris Sykes, Brian Eno, Po Bronson oraz Argye i Pati Hillis. Pomocne uwagi na temat niektórych rozdziałów przekazali mi też Tommy Poggio, Neal Gershenfeld, Simon Garfinkel, Mitchell Resnick i Marvin Minsky. Miałem wielkie szczęście, mogąc skorzystać z pomocy Sary Lippincott, która zredagowała cały tekst, znacznie go przy tym udoskonalając. Chciałbym w końcu podziękować mojej rodzinie: rodzicom — Argye i Billowi — którzy popierali moje zainteresowania projektowaniem złożonych maszyn; dzieciom — Noahowi, Asie i Indii; szczególnie zaś mojej żonie Pati, która cierpliwie mnie zachęcała i wspierała w trakcie pracy nad książką.

W. DANIEL HILLIS jest specjalistą od komputerów, współzałożycielem i głównym teoretykiem Thinking Machines, Inc. oraz wiceprezesem zarządu Disneya. Jest właścicielem prawie czterdziestu amerykańskich patentów, autorem licznych publikacji naukowych i książek (w tym *The Connection Machine* z roku 1985) oraz wydawcą kilku czasopism naukowych (m.in. „Artificial Life", „Complexity", „Complex Systems" i „Future Generation Computer Systems"). Jest też jednym z założycieli Fundacji Długiej Teraźniejszości.

MARVIN MINSKY: Danny Hillis jest jednym z najbardziej twórczych ludzi, jakich spotkałem, i zarazem jednym z najgłębszych myślicieli. Wniósł do nauk komputerowych wiele ważnych idei — zwłaszcza, choć nie tylko, w dziedzinie obliczeń równoległych. Dla wielu algorytmów, o których sądzono, że mogą działać tylko w komputerach sekwencyjnych, znalazł nowe zastosowania w o wiele szybszych obliczeniach równoległych. Gdy wpada na nowy pomysł, szybko odkrywa sposób przetestowania go i zbudowania maszyn wykorzystujących nową ideę. Po dokonaniu wspaniałych rzeczy w naukach komputerowych Hillis zainteresował się ewolucją i, jak sądzę, jest obecnie na dobrej drodze by stać się jednym z głównych teoretyków tej dziedziny. Jest także dobry w opowiadaniu anegdotek. Ma fascynujące uzdolnienia techniczne i nie chodzi mi tu tylko o wiedzę, jak uformować i połączyć potrzebne materiały: Danny jest jedną z niewielu osób, które można nazwać artystami. Gdy myśli, jak coś skonstruować, zdarza się, że chodzi po pokoju i nagle dostrzega, że coś do siebie doskonale pasuje i ma akurat te właściwości, których szukał. Sam jestem w tym dosyć dobry, lecz Danny jest o wiele lepszy.

DANIEL C. DENNETT: Pamiętam pierwsze spotkanie z Dannym, gdy był studentem w Laboratorium AI na MIT, i już wówczas było jasne, że ma pełno obrazoburczych pomysłów. Jeden z nich, szczególnie mocno go w tym czasie intrygujący, a z którego później uczynił swe sztandarowe osiągnięcie, to maszyna połączeń. Danny wymyślił ogromną maszynę o architekturze równoległej, która mogłaby badać dowolną część przestrzeni możliwych obliczeń i ta idea otworzyła nowy i rozległy obszar badawczy.

To, czego dokonał brytyjski matematyk Alan Turing (czyli to, co dzisiaj nazywamy maszyną Turinga), to było zwięzłe określenie całej przestrzeni wszystkich możliwych obliczeń. Maszyna zbudowana przez Johna von Neumanna była mechaniczną realizacją idei Turinga. Nasz domowy komputer to właśnie maszyna von Neumanna — standardowy komputer sekwencyjny. Maszyna von Neumanna — która dla wszystkich praktycznych celów jest uniwersalną maszyną Turinga — może w zasadzie obliczyć dowolną funkcję obliczalną; jednak, jeżeli nie mamy miliarda lat na to, by czekać na wyniki, nie możemy w rzeczywistości zbadać naprawdę ciekawych części tej przestrzeni. Rzeczywista przestrzeń badana przez jedną architekturę jest bardzo ograniczona. To zupełnie tak, jakbyśmy wysyłali znikomo cienki promień w ogromną i wielowymiarową przestrzeń. Aby zbadać inne jej części, musimy wymyślić inne rodzaje architektury komputera. Dlatego właśnie wszyscy wybierają ogromne architektury równoległe. Danny stworzył jeżeli nie pierwszy, to jeden z pierwszych rzeczywiście działających ogromnych komputerów równoległych. Było to odkrycie złotej żyły. Dostaliśmy nowy pojazd badawczy, który zaglądał do takich części przestrzeni projektowania, do których poprzednio nikt nie potrafił dotrzeć. Danny'emu dobrze szła sprzedaż tego pomysłu ludziom z różnych obszarów badań i pokazywanie, dzięki pewnym szczególnym zastosowaniom, jak potężny i ekscytujący jest to pojazd.

CHRISTOPHER G. LANGTON: Danny Hillis jest jednym z najinteligentniejszych ludzi jakich znam. Byłem pod wpływem koncepcji Danny'ego od czasu „Memoriału Laboratorium AI" w MIT, w którym po raz pierwszy wyłożył swój pomysł maszyny połączeń. Danny posiada godną uwagi zdolność zagłębienia się w nowej dla siebie dziedzinie i szybkiego osiągania w niej mistrzostwa. Potrafi nieomal natychmiast dotrzeć do najważniejszych problemów każdej dziedziny i dokonać nowych istotnych odkryć. Mam szczerą nadzieję, że potrafi się też wyplątać z administracyjnych zadań swojej firmy Thinking Machines i znaleźć czas i środki na własną pracą naukową. Nie mam wątpliwości, że znowu osiągnie ważne rezultaty.

FRANCISCO VARELA: Powiedziałbym o Dannym, że jest jednym z najlepszych specjalistów od precyzyjnego wytwarzania systemów złożonych. Nie tylko z wielką inwencją wkroczył na obszar robienia obliczeń, lecz zrobił z tego dochodowe przedsięwzięcie. Za pomocą maszyny połączeń dokonał wspaniałych rzeczy — na przykład, zrealizował rzeczywistą ewolucję oprogramowania w symulowanym krajobrazie ewolucyjnym. Robi to naprawdę duże wrażenie. Nie jestem pewien, czy ma to wiele wspólnego z ewolucją biologiczną, lecz tworzy sztuczną ewolucję, która na pewno jest bardzo ciekawa. Jest to bardzo pomysłowa koncepcja.

Istnieje kierunek, o którym, jak wiem, Danny myśli bardzo poważnie: koncepcja sztucznych światów i kreowanie równoległych wszechświatów. Jesteśmy dopiero na początku tej drogi i na razie to, o czym myślimy jako o „sztucznym życiu", powinno być nazywane raczej „sztucznymi światami". Interesującą częścią programu symulacji idei biologicznych jest rozważanie bytów biologicznych i ich światów jako zamkniętego systemu, w którym mamy do czynienia z brakiem podziału na to, co zewnętrzne, i to, co wewnętrzne, i gdzie pozwalamy systemowi biologicznemu rozgrywać pełną grę życia w świecie.

MURRAY GELL-MANN: Bardzo lubię Hillisa i bardzo go cenię. Mam wrażenie, że jest nie tylko odważnym (o tym wszyscy wiemy) lecz także głębokim i skutecznym myślicielem. Chciałbym więcej wiedzieć i rozumieć więcej z jego prac.

JOHN BROCKMAN: Każda nowa technologia jest równoważna nowej percepcji. Gdy tworzymy nowe narzędzia, sami przekształcamy się na ich obraz i podobieństwo. Mechanika Newtona dała początek metaforze serca jako pompy. Pokolenie przed nami, wraz z początkami cybernetyki, nauki o informacji i sztucznej inteligencji, zaczęło myśleć o mózgu jako o komputerze. Teraz doszliśmy do nowego skrzyżowania empirii i epistemologii. Ostatnie przełomowe osiągnięcia w dziedzinie budowy wielkich komputerów równoległych i związanych z nimi algorytmów, mają wielki wpływ na nasz obraz siebie i naszego miejsca we Wszechświecie. Przebiliśmy się wreszcie przez wąskie gardło komputera sekwencyjnego von Neumanna.

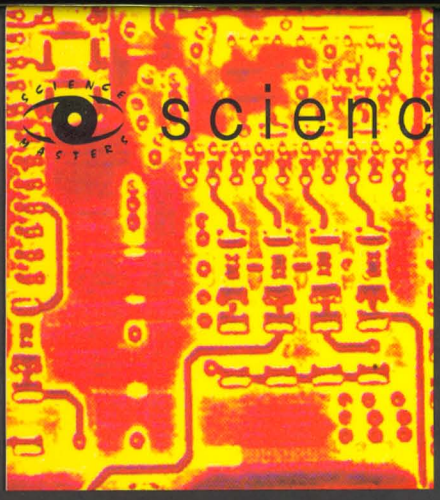
W. Daniel Hillis łączy w swej działalności wiele idei — społeczeństwo umysłu Marvina Minsky'ego, sztuczne życie Christophera G. Langtona, Richarda Dawkinsa spojrzenie okiem genu, plektykę uprawianą w Santa Fe. Hillis opracował algorytmy, dzięki którym możliwe stało się zbudowanie wielkich komputerów równoległych. Rozpoczął od fizyki, potem przeszedł do nauki o komputerach — rewolucjonizując tę dziedzinę — a obecnie zaczął stosować swoje algorytmy do badania ewolucji.

[Cytowane opinie pochodzą z książki Johna Brockmana *Trzecia kultura*, opublikowanej przez Wydawnictwo CiS w roku 1996.]

W. Daniel Hillis

WZORY NA KRZEMOWEJ PŁYTCE

CiS



science masters

W. Daniel Hillis

WZORY
NA KRZEMOWEJ PŁYTCE

SCIENCE MASTERS 2000

Początek Wszechświata John D. Barrow
Ostatnie trzy minuty Paul Davies
Pochodzenie człowieka Richard Leakey
Rzeka genów Richard Dawkins
Liczby natury Ian Stewart
Kraina pierwiastków Peter Atkins
Laboratorium Ziemia Stephen H. Schneider
Natura umysłów Daniel C. Dennett
Jak myśli mózg William H. Calvin
Światełko mydliczki George C. Williams
Dlaczego lubimy seks? Jared Diamond
Mózg Susan A. Greenfield
Samolubna komórka Robert A. Weinberg
Symbiotyczna planeta Lynn Margulis
Wzory na krzemowej płytce W. Daniel Hillis
Tylko sześć liczb Martin Rees

Więcej informacji o naszych książkach można znaleźć w Internecie, na stronie **http://www.cis.pl**

83 85458 71 9
ISBN 83-85458-71-9