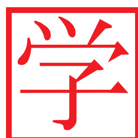
POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Piotr Jurgaś

Informatyczne podstawy grafiki komputerowej



WYDAWNICTWO
PJWSTK

Notka biograficzna
Dr inż. Piotr Jurgaś - pracownik Polsko - Japońskiej Wyższej Szkoły Technik Komputerowych znany jest ze swojego niekonwencjonalnego podejścia do prowadzonych przez siebie zajęć, które bez względu na to, czy dotyczą matematyki, czy informatyki, odwołują się do całości wiedzy, przeżyć, a nawet uczuć biorących w nich udział osób. Z tego powodu uznawany jest za jednego z najlepszych wykładowców Wydziału Zamiejscowego PJWSTK w Bytomiu. Jest też autorem licznych nowatorskich algorytmów i rozwiązań informatycznych na czele z poolicz.pl, czyli uhonorowaną nagrodą Webstar, pierwszą na świecie stroną internetową pomagającą w rozwiązywaniu zadań z matematyki.

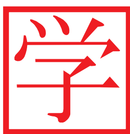
Streszczenie
Postęp nauk ścisłych i techniki w dzisiejszych czasach jest tak szybki, że trudno już nadążyć nawet za osiągnięciami konkretnej gałęzi wiedzy. Z tego powodu należy uznać, że czasy geniuszy pokroju Leonarda da Vinci już dawno minęły. Najwybitniejsze umysły XXI w. to niemal wyłącznie specjaliści w wybranych dziedzinach. Inaczej niż należałoby się jednak spodziewać, najbardziej cenieni spośród nich są nie ci, którzy uważają, że przedmiot ich badań jest najważniejszy na świecie, tylko ci, którzy potrafią na chwilę go porzucić i spojrzeć na problem z perspektywy, którą mogą zaoferować inne formy ekspresji ludzkiego umysłu.

Podobnie jest ze sztuką i naukami humanistycznymi, które coraz częściej sięgają po precyzyjne narzędzia wypracowane przez matematyków i informatyków. Zadaniem niniejszej monografii jest przybliżenie wszystkim ludziom, którym bliżej do sztuki niż nauki, zagadnień, które legły u podstaw informatyki, lub z których oni często korzystają nie zdając sobie nawet sprawy z tego, w jaki sposób to wszystko działa, lub jak barwna jest tego historia.

PJWSTK
ポーランド日本情報工科大学
POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Piotr Jurgaś

Informatyczne podstawy grafiki komputerowej



WYDAWNICTWO
PJWSTK

© by Piotr Jurgaś
Warszawa 2010

© by Wydawnictwo PJWSTK
Warszawa 2010

Wszystkie nazwy produktów są zastrzeżonymi nazwami handlowymi lub znakami towarowymi odpowiednich firm.

Książki w całości lub w części nie wolno powielać ani przekazywać w żaden sposób, nawet za pomocą nośników mechanicznych i elektronicznych (np. zapis magnetyczny) bez uzyskania pisemnej zgody Wydawnictwa.

Edytor
Leonard Bolc

Kierownik projektu
Konrad Wojciechowski

Redaktor techniczny
Ada Jedlińska

Korekta
Anna Bittner

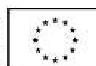
Komputerowy skład tekstu
Grażyna Domańska-Żurek

Projekt okładki
Andrzej Pilich

Wydawnictwo
Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych
ul. Koszykowa 86, 02-008 Warszawa
tel. 022 58 44 526; fax 022 58 44 503
e-mail:oficyna@pjwstk.edu.pl

Oprawa miękka
ISBN 978-83-89244-85-7

Wersja elektroniczna
ISBN 978-83-63103-49-1



Projekt „Nowoczesna kadra dla e-gospodarki - program rozwoju Wydziału Zamiejscowego Informatyki w Bytomiu Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych, współfinansowany przez Unię Europejską ze środków Europejskiego Funduszu Społecznego w ramach Podziałania 4.1.1 "Wzmocnienie potencjału dydaktycznego uczelni" Programu Operacyjnego Kapitał Ludzki

This book should be cited as:
Jurgaś, P., 2010. Informatyczne podstawy grafiki komputerowej. Warszawa: Wydawnictwo PJWSTK.

Spis treści

Przedmowa i

1 Historia obliczeń, maszyn liczących i komputerów 1

2 Modele komputerów 9
2.1 Przykłady 10

3 Bramki logiczne i ich układy 17
3.1 Przykłady 18
3.2 Zadania i pytania 22

4 Wprowadzenie do architektury komputerów 25

5 Systemy operacyjne 33
5.1 Przykład 37

6 Podstawowe pojęcia języków programowania 39
6.1 Zmienne 39
6.1.1 Przykłady 40
6.2 Tablice 41
6.3 Instrukcje bezwarunkowe 41
6.4 Instrukcje warunkowe 42
6.5 Pętle 43
6.6 Grupowanie instrukcji 44
6.7 Zadania i pytania 45

7 Sposoby projektowania programów komputerowych 47
7.1 Schematy blokowe 48
7.2 Pseudokod 52
7.2.1 Przykłady 54
7.3 Zadania i pytania 57

8 Złożoność obliczeniowa programów komputerowych 59
8.1 Przykłady 60

9 Klasyfikacja języków programowania 67
9.1 Przykłady 68

10 Zagadnienie kompresji danych 79
10.1 Przykłady 79

11 Grafika komputerowa 85
11.1 Przykład 87

12 Komputerowy dźwięk 91

13 Internet 97
13.1 Przykład 100

14 Gromadzenie i przetwarzanie danych 105
14.1 Przykład 111

Literatura 115

Przedmowa

Kiedy poproszono mnie o napisanie tego podręcznika, którego odbiorcami będą studenci kierunku „Grafika komputerowa", zdałem sobie sprawę, że zadanie to nie będzie należało do najłatwiejszych. Informatyka jest jedną z najszybciej rozwijających się nauk i nie ma chyba na świecie człowieka, który znałby się na wszystkich jej zagadnieniach. Ponadto niektóre z nich, jak choćby badania nad sztuczną inteligencją są na tyle skomplikowane, że do ludzi, którzy mogliby napisać na ten temat coś sensownego, należy garstka wybitnych specjalistów.

Każdy z opisanych w tym opracowaniu tematów zasługuje na osobną monografię, którą należałoby potem referować co najmniej w semestralnym ciągu wykładów. To jednak nie jest możliwe z uwagi na specyfikę studiów z pogranicza informatyki i sztuki. W związku z tym zdecydowałem się na utrzymanie tego skryptu w charakterze popularnonaukowym, tak by chociaż pobieżnie zaakcentować większość tematów, z którymi mają styczność studenci informatyki.

Wśród autorów książek popularnonaukowych panuje przekonanie, że każde użycie dowolnego wzoru matematycznego w tekście redukuje o połowę liczbę jego czytelników. Z uwagi na swoją specyfikę opracowanie to jest kierowane do dosyć wąskiej grupy ludzi, więc świadome działanie w kierunku uszczuplania tej grupy odbiorców przypomina strzelanie sobie w kolano. Dlatego aparat matematyczny zawarty w tej książce został ograniczony do niezbędnego minimum, a tu i ówdzie wpleciono garść ciekawostek dotyczących omawianych tematów.

Pozostaje już tylko życzyć miłej lektury, zachęcić do samodzielnego zgłębiania zasygnalizowanych tematów i dzielenia się z autorem uwagami na temat tego tekstu.

1

Historia obliczeń, maszyn liczących i komputerów

Gdyby przyszło nam odpowiedzieć sobie na pytanie, co właściwie potrafi robić komputer, to poza odpowiedzią, że prawie wszystko, należałoby spojrzeć nieco głębiej. Wyświetlane grafiki i filmy, odgrywane dźwięki, a także efekty działania wszystkich programów użytkowych i gier komputerowych to właściwie nic innego jak ogromna liczba obliczeń interpretowanych czasem w bardzo zaskakujący sposób. Być może z tego właśnie powodu w większości popularnych języków świata słowo oznaczające urządzenie, które oznacza urządzenie potrafiące sobie poradzić z takimi zadaniami, zostało wywiedzione od angielskiego czasownika „to compute", co znaczy obliczać. Wyjątkiem jest język francuski, w którym na komputer mówi się „ordinateur", co dla Polaka jest mniej więcej tak śmieszne, jak dla Rosjanina fakt, że na ich „intiegrał" my mówimy „całka".

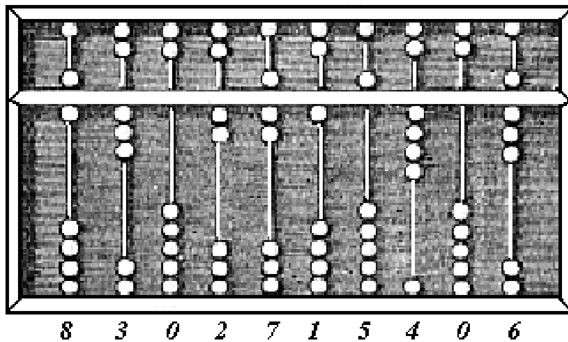
Historia obliczeń jest prawdopodobnie tak długa jak historia ludzkości, wliczając nawet gatunki poprzedzające homo sapiens. Początkowo proces obliczeń prawdopodobnie odbywał się w nie do końca uświadomiony sposób z wykorzystaniem czegoś, co matematycy nazywają relacją równoliczności. Aby policzyć upolowane zwierzaki lub biegające dookoła brzdące, wystarczyło każdemu z nich przypisać jeden palec i zaginać go podczas badania liczby elementów. Jak jednak nie trudno się domyślić, sytuacja mocno komplikowała się w przypadku, gdy łowy były wyjątkowo udane, albo gdy dzieci posiadało się sporo, o co w obliczu braku telewizji i internetu specjalnie trudno nie było.

Prawdopodobnie właśnie w ten sposób powstały zręby idei, którą znamy dziś pod nazwą liczby naturalnej i dopiero w XIX wieku doczekała się ona ścisłej definicji wprowadzonej przez włoskiego matematyka, Giuseppe Peano. Liczby naturalne przez długi czas wystarczały ludzkości do obliczeń związanych z codziennym życiem. Co ciekawe i warte odnotowania, dla ludzi z dawnych czasów pojęcie „niczego", czyli zera, nie było specjalnie interesujące. I to pewnie dlatego liczbę oraz cyfrę zero odkryto stosunkowo niedawno, co pozwoliło na powstanie systemów pozycyjnych. Do tej pory liczby notowano na wiele różnych, najczęściej dziwacznych z naszego punktu widzenia sposobów. W dzisiejszych czasach w pewnych okolicznościach stosowany jest już właści-

wie chyba tylko system rzymski. Jak bardzo jest on jednak niepraktyczny, wie każdy, kto próbował pomnożyć przez siebie MXL i CCIV bez zamiany tych liczb na używany przez nas system dziesiątkowy. Co więc takiego specjalnego jest w naszym sposobie liczenia, że jest tak dobry? Odpowiedź jest prosta, to jego pozycyjność. Dla przykładu liczba 1234 to nic innego jak: 1 tysiąc 2 setki 3 dziesiątki oraz 4 jednostki. Tu sytuacja jest czysta i jasna. Jak jednak notować liczby, w których nie ma żadnych setek, dziesiątek lub jedności? W przypadku, gdy jest nam nieznany symbol zera, można próbować sobie radzić jak Majowie, którzy symbol ten zastępowali pustym miejscem. Nietrudno jednak zauważyć, że w takiej sytuacji bardzo łatwo jest pomylić liczbę 1 1 z liczbą 1 1. W przypadku abstrakcyjnych rachunków prowadzonych dla zabicia czasu może nie ma to większego znaczenia, jednak posiadanie na koncie bankowym 1001 lub 101 złotych to spora różnica. Z tego właśnie powodu odkrycie symbolu zera, choć wydaje się niepozorne, wywołało w matematyce i naukach z niej korzystających prawdziwą rewolucję.

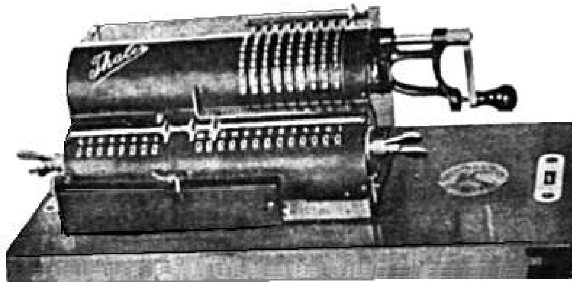
Z uwagi na to, że w większości przypadków ludzie dysponują dziesięcioma palcami, dosyć powszechnym pozycyjnym systemem obliczeń stał się system dziesiątkowy. I tu po raz kolejny Francuzi nie byliby sobą, gdyby nie próbowali zrobić czegoś po swojemu. W czasach Rewolucji Francuskiej promowany był nie tylko inny kalendarz, ale również inny system pozycyjny oparty na potęgach liczby 12 (np. 12, 144, 1728 itp.). Na szczęście jednak wraz z końcem czasów, w których prości Francuzi mogli się postarać o dodatkowe palce, które wcześniej należały do arystokracji, system ten używany jest raczej sporadycznie.

Wraz z rozwojem bankowości na świecie potrzeba przeprowadzania coraz większej liczby obliczeń stawała się inspiracją do powstawania wynalazków mających ten proces usprawnić. W bardzo wielu cywilizacjach odkryto liczydła i abakusy, które jednak głównie wspomagały pamięć rachmistrzów i cały czas wymagały od nich posiadania nie lada umiejętności. Poniżej widoczny jest abakus (rys. 1.1) z zaznaczoną na nim pewną sporą liczbą.



Rysunek 1.1.

Przez długi czas taki stan rzeczy utrzymywał się, aż w końcu ktoś wpadł na pomysł zaprzęgnięcia do obliczeń osiągnięć mechaniki precyzyjnej. W ten sposób pojawiły się na świecie urządzenia zwane arytmometrami, na których można już było w sposób automatyczny dokonywać dodawania i odejmowania. Operacje mnożenia i dzielenia były natomiast wykonywane przy użyciu wielokrotnego dodawania i odejmowania. Wszystko to miało miejsce jeszcze przed okiełznaniem elektryczności, przez co większość z nich posiadała korbki przeznaczone do ich napędzania. Poniżej znajduje się ilustracja przedstawiająca arytmometr Odhnera (rys. 1.2.), który zwany był powszechnie kręciołkiem.

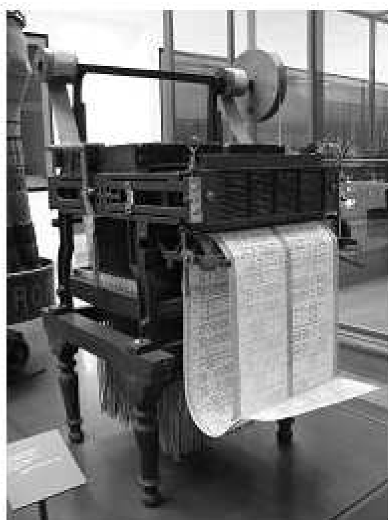


Rysunek 1.2.

Choć w zakresie obliczeń tamte urządzenia nie umiały robić wiele więcej niż dzisiejsze komputery, to trzeba uczciwie przyznać, że odrobinę rozmijają się one z naszym obecnym rozumieniem komputerów. To, czego w nich brakowało, to możliwość ich programowania, czyli zadawania im ciągów instrukcji, które mają wykonywać według pewnego planu.

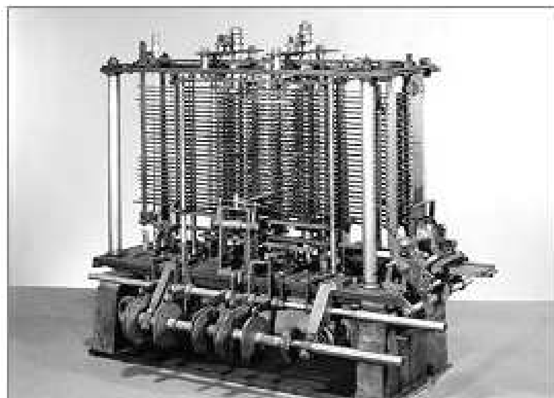
Niejeden raz w historii nauki i techniki zdarzało się, że pewne rozwiązanie, które miało pomagać w przeprowadzaniu jednej czynności, stawało się podstawą, na której opierano coś całkiem innego. Choć może trudno w to uwierzyć, pierwszym programowanym urządzeniem na świecie nie był wcale komputer. Palmę pierwszeństwa pod tym względem przypisuje się wynalezionym przez Josepha Marie Jaquarda krosnom tkackim (rys. 1.3.), w których wzory tkanin były programowane przy pomocy serii kart perforowanych przesuwających się w mechanicznym czytniku.

Nieco później Charles Babbage stworzył projekt „maszyny różnicowej”, która była ulepszoną wersją mechanicznego kalkulatora. Próbom przejścia od projektu do rzeczywistości Babbage poświęcił sporą część życia i ostatecznie projekt ten porzucił. Maszyna różnicowa powstała w końcu na podstawie jego oryginalnych planów i od 1991 roku można ją oglądać w Muzeum Nauki w Londynie. Co ciekawe, maszyna ta jest w pełni sprawna i potrafi wykonywać skomplikowane obliczenia z bardzo dużą jak na urządzenie mechaniczne precyzją. Jakkolwiek przedstawiony poniżej rysunek przez niewprawne oko może być



Rysunek 1.3.

odebrany jako chińskie narzędzie tortur, to jednak przedstawia on właśnie tę maszynę (rys. 1.4.).



Rysunek 1.4.

Jednak najbardziej rewolucyjną koncepcją wprowadzoną przez Babbage'a była tak zwana „maszyna analityczna", która co prawda nigdy nie została zmontowana, ale której plany przez znacznie późniejszych naukowców z Johnem von Neumannem na czele zostały uznane za pierwowzór dzisiejszych kom-

puterów. Po raz pierwszy w historii obliczeń pojawiało się tam założenie o konieczności wprowadzenia możliwości grupowania obliczeń i przechowywania je w mechanicznej pamięci.

Jakkolwiek był to kamień milowy na drodze powstawania informatyki, to jednak można tu dostrzec pewne analogie z maszyną parową. Kiedy powstawała, przyśpieszyła rozwój technologiczny. Gdyby jednak się uprzeć, by używać jej powszechnie dzisiaj, to z pewnością by ten rozwój spowolniła. Mechaniczne kalkulatory, nawet tak zaawansowane jak maszyna Babbage'a poza tym, że najczęściej są piękne jako przedmioty, to mają jedną dyskwalifikującą je cechę, a jest nią szybkość pracy. Przekazywanie impulsów mechanicznych jest bardzo wolnym procesem i nie da się go przyspieszyć, nie ryzykując przy tym popsucia całej maszyny.

Inaczej sprawa ma się z elektrycznością. Prąd elektryczny to, jak wiadomo, uporządkowany ruch elektronów. Same elektrony poruszają się stosunkowo wolno, ale już to, co je napędza, czyli napięcie elektryczne, rozprzestrzenia się z prędkością światła, która na obecnym etapie wiedzy uznawana jest za największą prędkość w kosmosie. Naturalna jest więc próba wykorzystania tego zjawiska w procesie wykonywania obliczeń. Jak jednak pogodzić ze sobą taką prędkość i system dziesiętny? W jaki sposób przesyłać pomiędzy różnymi punktami komputera cyfry systemu dziesiętnego? Na upartego można wyobrazić sobie miernik napięcia prądu, który pewnym przedziałom jego wartości będzie przypisywał konkretne cyfry oraz modulator sterujący tymi wartościami, a wszystko to połączone drucikiem. Intuicja podpowiada jednak, że nie jest to rozwiązanie optymalne. I tu po raz kolejny wracamy do koncepcji systemu liczenia innego niż system dziesiętny. Każda liczba naturalna może zostać zapisana tylko przy pomocy zer i jedynek jako suma odpowiednich potęg liczby 2 (1, 2, 4, 8, 16,...). System wykorzystujący ten zapis nazywamy binarnym lub dwójkowym. Dla przykładu liczba 23 zapisana w tym systemie to 10111 bo składa się na nią jedna szesnastka, zero ósemek, jedna czwórka, jedna dwójka i jedna jedynka. Od tego spostrzeżenia już tylko krok dzieli nas od pomysłu na sposób przesyłania informacji w dzisiejszych komputerach. Jeśli umówimy się, że brak napięcia będzie oznaczał zero, a niezerowe napięcie będzie oznaczało jedynkę, to zyskujemy sposób błyskawicznego przesyłania informacji między różnymi częściami komputera. Po raz kolejny widać tu też wpływ pomysłu wykorzystanego w krosnach Jaquerde'a.

Kolejnym istotnym etapem w powstawaniu komputerów było pojawienie się lampy elektronowej, która stała się jednym z podstawowych elementów wchodzących w skład pierwszych komputerów, jakimi były brytyjski Colossus i amerykański Eniac (*Electronic Numerical Integrator And Computer*). Poniżej (rys. 1.5.) znajduje się ilustracja lampy elektronowej.

Gołym okiem widać, że nie można temu urządzeniu odmówić urody. Paradoksalnie jednak fakt, że widać go gołym okiem, stał się przyczyną końca komputerów, które opierały swoje działanie na ich wykorzystaniu. Wspomniane już pierwsze komputery składały się z wielu tysięcy lamp elektronowych, co przy ich rozmiarach wiązało się z faktem, że zajmowały ogromne objętości.

Rysunek 1.5.

Oprócz tego lampy elektronowe miały przykrą tendencję do stosunkowo krótkiego działania (liczonego najwyżej w tysiącach godzin) i wrażliwości na różnego rodzaju uszkodzenia, które dziś należą do rzadkości. Popularny obecnie wśród programistów termin debugowanie (po polsku odrobaczanie) powstał właśnie w tamtych czasach i został ukuty przez jednego z techników nadzorujących pracę Eniaca, która dosyć często przerywana była przez spięcia wywołane przez łażące po przewodach żyjątka.

Z tego powodu każda innowacja pozwalająca na wyeliminowanie tych niedogodności przy zachowaniu dotychczasowej funkcjonalności była skazana na sukces. Innowacją tą stały się układy scalone, czyli zminiaturyzowane układy elektroniczne zawierające w swoim wnętrzu (w dzisiejszych czasach) miliony elementów takich jak tranzystory, diody, rezystory i kondensatory. Pierwowzorem tego typu układów była wyprodukowana w 1926 r. lampa próżniowa Loewe 3NF, zawierająca w swoim wnętrzu trzy triody, dwa kondensatory oraz cztery rezystory. Pierwszą osobą, która opracowała teoretyczne podstawy układów scalonych, był angielski naukowiec Geoffrey Dummer, ale podobnie jak Babbage'owi jemu również nie udało się przejść od planów do prototypu. W roku 1958 Jack Kilby i Robert Noyce niezależnie od siebie zbudowali działające modele układów scalonych. J.Kilby wyprzedził konkurencję o pół roku i okazało się to rozwiązaniem na tyle przełomowym, że niemal pół wieku później w 2000 roku został za to uhonorowany Nagrodą Nobla z fizyki.

Ostatnie kilka stron to sprint przez kilkadziesiąt tysięcy lat ludzkiej historii od zaistnienia jakichkolwiek obliczeń, po doprowadzenie do tego, by te obliczenia przebiegały zupełnie poza naszą świadomością. Trudno byłoby sobie dziś wyobrazić świat bez komputerów, ale warto pamiętać o jednym. Wszystko, co

one potrafią robić, to obliczenia prowadzone z zawrotnymi prędkościami, a to ma się nijak do ludzkiego intelektu, który jest potrzebny, aby te obliczenia zrozumieć, docenić i zinterpretować.

2

Modele komputerów

W nauce i technice mało jest udanych rzeczy, które powstały w spontaniczny sposób. Podobnie jest z komputerami. Zanim doszło do pojawienia się ich prototypów i ich dzisiejszych następców, znane były ich matematyczne modele. Jednym z pierwszych była tzw. maszyna Turinga, opisana przez angielskiego matematyka Alana Turinga. Maszyna ta składa się z nieskończenie długiej taśmy podzielonej na komórki, w których może się znaleźć zapis jednego z N stanów, głowicy odczytująco-zapisującej, która może przybierać M stanów i układu sterowania głowicą. Choć może się to wydawać niewiarygodne, dzisiejsze komputery mają wiele wspólnego z tym tworem i uważa się je za praktyczne przybliżenie idei maszyny Turinga, która w sposób dokładny nie zostanie zrealizowana prawdopodobnie nigdy. Łatwo zauważyć, że w komputerze rolę nieskończonej taśmy pełni pamięć, rolę głowicy odczytująco-zapisującej pełnią układy wejścia-wyjścia, a rolę układu sterowania procesor. Dlaczego więc komputer uznawany jest za przybliżenie maszyny Turinga? Odpowiedź na to pytanie jest niezwykle prosta: ponieważ każda pamięć, obojętnie jak duża, zawsze jest skończona, więc dosyć łatwo jest stworzyć program, który wypełni ją w całości i jeszcze mu mało będzie. Dowodem na to mogą być np. dowolne nowoczesne programy firmy Microsoft zainstalowane na kilkuletnim sprzęcie.

Zależnie od stanu głowicy oraz stanu komórki taśmy, maszyna może odczytać z klatki wartość, zapisać wartość w klatce lub przesunąć się o jedno pole w lewo lub prawo. Działanie takie zwane jest instrukcją i formalnie opisywane jest w następujący sposób: <A,B,C,D,E>, gdzie:

A - oznacza symbol odczytany z bieżącej komórki taśmy (pamięci),
B - oznacza bieżący stan układu sterowania (procesora),
C - symbol, który zostanie zapisany w bieżącej komórce taśmy,
D - stan, w który przejdzie układ sterowania po wykonaniu tej instrukcji,
E - symbol kierunku, w którym ma się przesunąć głowica po zakończeniu, przetwarzania tej instrukcji, L jeżeli ma się poruszać w lewo i P jeżeli ma się poruszać w prawo.

2.1 Przykłady

Przykład 1:

Rozważmy następującą instrukcję: <1,2,3,2,L>. Jakie jest jej znaczenie? Jedynka oznacza symbol odczytany z bieżącej komórki taśmy. Znaczy to, że instrukcja ta wykona się, jeżeli głowica odczytująco - zapisująca znajdzie się nad jedynką. Pierwsza dwójka oznacza bieżący stan układu sterowania i może być utożsamiana z numerem pewnego rozkazu procesora. Trójka to symbol, który ma zostać zapisany w bieżącej komórce taśmy po przetworzeniu tej instrukcji. Druga dwójka oznacza stan, w którym ma się znaleźć układ sterowania maszyną po przetworzeniu tej instrukcji. Ponieważ jest on taki sam jak stan początkowy, więc znaczy to tyle, że układ sterowania nie zmieni swojego stanu. Litera L na końcu powyższej piątki oznacza, że po wszystkim głowica maszyny przesunie się o jedno miejsce w lewo.

Bardzo ważną kwestią dotyczącą maszyny Turinga jest fakt, że program opisujący jej zachowanie podany jest w postaci listy instrukcji, których kolejność nie ma większego znaczenia. Pierwsze dwa symbole składające się na instrukcję określają warunki, które muszą zostać spełnione, by taka instrukcja została wykonana. Pozostałe trzy opisują to, co potem ma się stać z maszyną. Jest to wyraźna różnica w stosunku do programów opisujących działanie dzisiejszych komputerów. Programy te, zwłaszcza pisane w językach strukturalnych, o których będzie mowa w jednym z kolejnych rozdziałów, rzadko mają formę listy, a jeszcze rzadziej kolejność wchodzących w ich skład instrukcji nie gra roli.

Przykład 2:

Rozważmy następujący program dla maszyny Turinga:

<0,A,1,A,L>
<1,A,0,A,L>

Pierwsza instrukcja zostanie przetworzona, gdy na taśmie znajdzie się 0, a układ sterowania będzie pozostawał w stanie A. Jak widać, po przetworzeniu tej instrukcji stan układu sterowania się nie zmieni, bieżąca komórka zmieni swoją wartość na 1, a głowica maszyny przesunie się o jedno miejsce w lewo. Analogicznie druga instrukcja zamieni w bieżącej komórce taśmy 1 na 0, nie zmieniając stanu układu sterowania i przesuwając po wszystkich głowicę maszyny o jedno miejsce w lewo.

Załóżmy, że układ sterowania maszyny znajduje się w stanie A. Co więc stanie się w przypadku, gdy przedstawimy maszynie do przetworzenia następującą taśmę, na której początkową pozycję głowicy oznaczono tłem w innym kolorze?

2	0	1	0	1

Odpowiedź jest prosta: w pierwszym kroku znajdująca się pod głowicą komórka zmieni wartość z 1 na 0, a głowica przesunie się o jedno pole w lewo.

2	0	1	**0**	1

Po drugim kroku taśma z zaznaczoną pozycją głowicy będzie już wyglądała w sposób następujący:

2	0	**1**	0	1

Po trzecim kroku uzyskamy następującą postać taśmy i pozycję głowicy:

2	**0**	1	0	1

Po czwartym kroku taśma wraz z głowicą będą wyglądać, jak następuje:

2	0	1	0	1

W tym momencie maszyna odczytuje z taśmy symbol, z którym nie może sobie poradzić, bo nie ma w swoim spisie instrukcji, która dopuszczałaby symbol 2 jako stan taśmy. W takim układzie program maszyny musi zakończyć swoje działanie, które, jak widać, sprowadziło się do zamienienia wszystkich jedynek na zera i zer na jedynki. Operację taką nazywamy bitową negacją.

Przykład 3:

Przyjmijmy, że układ sterowania maszyny znajduje się w stanie A, taśma z zaznaczoną pozycją głowicy sterującej wygląda, jak następuje:

0	**0**

a program maszyny wyraża się listą instrukcji

<0,A,0,B,L>
<0,B,0,A,R>.

Pierwsza instrukcja sprawia, że gdy maszyna, której układ sterowania znajduje się w stanie A, znajdzie na taśmie 0, to zmieni ona stan maszyny na B i przesunie głowicę o jedno miejsce w lewo. Analogicznie druga instrukcja sprawia, że gdy maszyna jest w stanie B, a na taśmie jest 0, to stan maszyny zmienia się na A, a głowica wędruje o jedno miejsce w prawo.

Co więc się stanie z maszyną o takim programie, stanie początkowym i taśmie? Odpowiedź jest prosta: taśma pozostanie niezmienna, stan będzie się bez końca zmieniał z A na B i z powrotem, a głowica będzie przeskakiwać pomiędzy sąsiednimi komórkami taśmy.

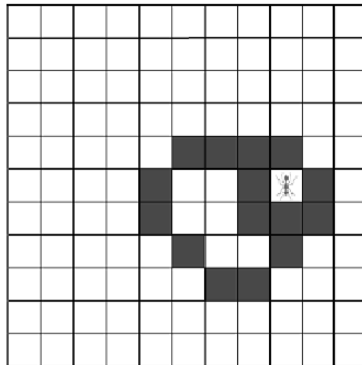
Przedstawiona wyżej maszyna to bodaj najprostszy przykład maszyny Turinga. W informatyce rozważano także maszyny z wieloma taśmami, wieloma głowicami oraz wieloma układami sterowania, co można byłoby sobie wyobrazić jako komputery z wieloma różnymi pamięciami, wieloma układami wejścia-wyjścia i maszyny wieloprocesorowe. Znane są też rozważania nad maszynami posiadającymi taśmy, których nie da się zawrzeć w jednym wymiarze.

Najprostszym przykładem takiej maszyny jest tzw. mrówka Langtona. Taśmą dla takiej maszyny jest pokratkowana płaszczyzna. Głowica może poruszać się do góry, na dół, w lewo oraz w prawo. Działanie maszyny da się opisać w następujący sposób:

Jeżeli głowica znajdzie się na polu białym, to zamienia to pole na czarne, a sama obraca się o kąt prosty w lewo i przechodzi do następnego pola.

Jeżeli głowica znajdzie się na polu czarnym, to zamienia to pole na białe, a sama obraca się o kąt prosty w prawo i przechodzi do następnej komórki.

Poniżej można obejrzeć przykład działania mrówki Langtona (rys. 2.1.).



Rysunek 2.1.

Choć zestaw reguł opisujących działanie tej maszyny jest nader ubogi, to jednak graficzny efekt jej działania prezentuje się okazale, zwłaszcza gdy taśma maszyny początkowo nie była wypełniona jednolitym kolorem, albo gdy porusza się po niej więcej niż jedna „mrówka”.

Warto w tym miejscu zaznaczyć, że choć działanie każdego komputera da się sprowadzić do pewnej maszyny Turinga, to jednak jest to najczęściej zadanie bardzo trudne, a uzyskane w ten sposób programy są długie i trudne w lekturze. Za to, z uwagi na swoją konstrukcję, hipotetyczne oczywiście maszyny Turinga mają znacznie większą moc obliczeniową niż rzeczywiste komputery. Mimo to jednak istnieją problemy, które nie mogą być rozwiązane nawet przy ich pomocy. Przykładem takiego zagadnienia jest problem stopu, czyli pytanie o to, czy zadany program komputerowy zakończy swoje działanie dla każdych dopuszczalnych danych wejściowych. Z tego powodu od czasów Turinga mate-

matycy i informatycy rozważają istnienie hipotetycznych maszyn o znacznie większej mocy obliczeniowej. Maszyny takie znane są pod nazwą hiperkomputerów, przy czym na obecnym etapie wiedzy pomimo intensywnych badań prowadzonych w tym kierunku nie wiadomo, czy fizyka rządząca naszym wszechświatem pozwala na zbudowanie rzeczywistych odpowiedników hiperkomputerów tak jak pozwoliła na zbudowanie komputerów będących przybliżeniem maszyn Turinga.

Uważny czytelnik zauważy z pewnością fakt, że maszyny Turinga dysponują pewną cechą, której nie ma żaden istniejący komputer. Cechą tą jest fakt, że choć lista instrukcji tej maszyny może mieć niewyobrażalne rozmiary, to i tak właściwa dla konkretnego stanu układu sterowania i taśmy instrukcja zostanie wybrana natychmiast. Przypomina to trochę sytuację, w której układamy na sobie kolejne warstwy filtrów wyspecjalizowanych do wyłapywania konkretnego rodzaju chemikaliów oraz tłoczymy przez nie roztwór pod ciśnieniem, choć nawet i w tym przypadku interesujące nas związki zatrzymają się na stosownym filtrze dopiero po upływie określonego czasu.

Jak do tej pory nie udało się zbudować niczego, co zasadą swojego działania zbliżałoby się chociaż do ideału wyznaczonego przez myślową konstrukcję maszyny Turinga. Na szczęście nie jest to jedyny model komputera i obok takich konstrukcji jak rachunek Lambda znaleźć można również i coś, co bardzo bliskie jest koncepcji protoplasty wszystkich języków programowania, czyli tzw. asemblera. Ów model to tzw. maszyna RAM.

Istnieje mnóstwo różnych sposobów opisu tej koncepcji. Prawdopodobnie najprostszy z nich zakłada, że maszyna RAM podobnie jak maszyna Turinga ma nieskończoną pamięć, ale jej komórki są ponumerowane kolejnymi liczbami naturalnymi. Aby było wiadomo, gdzie program ma się zacząć, maszyna RAM wyposażona jest w tzw. wskaźnik rozkazów, czyli numer aktualnie przetwarzanej instrukcji.

Zestaw instrukcji rozpoznawanych przez maszynę RAM nie jest imponujący i zawiera jedynie 4 pozycje. Pierwsza instrukcja oznaczana jest symbolem $Z(n)$, gdzie n to pewna liczba naturalna. Zadaniem tej instrukcji jest wyzerowanie komórki pamięci leżącej pod numerem n i zwiększenie wskaźnika rozkazów o jeden. Dla osób zaznajomionych choć odrobinę z programowaniem komputerów, instrukcję tę można zapisać w postaci $z[n] = 0$.

Druga instrukcja oznaczana symbolem $A(n)$ powoduje zwiększenie o jeden wartości przechowywanej w komórce o numerze n i zwiększenie o jeden wskaźnika rozkazów. Symboliczny opis tej instrukcji to $z[n] = z[n] + 1$.

Trzecia instrukcja to $C(m, n)$ i służy ona do kopiowania wartości przechowywanej w m-tej komórce pamięci do n-tej komórki. Ubocznym efektem działania tej instrukcji jest tak samo, jak w powyższych przypadkach, zwiększanie wskaźnika rozkazów o jeden. Symbolicznie można to zapisać jako $z[n] = z[m]$.

Czwarta i ostatnia instrukcja to $I(m, n, q)$ i jej zadaniem jest sprawdzenie, czy wartości przechowywane w komórkach o numerach m i n są równe. Jeżeli tak się stanie, to wskaźnik rozkazów jest ustawiany na q. W przeciwnym zaś przypadku wskaźnik rozkazów zwiększa się o jeden.

Przykład 4:

Rozważmy następujący program maszyny RAM:

0	C(0,3)
1	I(1,2,5)
2	A(2)
3	A(3)
4	I(0,0,1)

Załóżmy także, że wskaźnik rozkazów ustawiony jest na zero, a dane wejściowe dla tego programu przechowywane są w komórkach o numerach 0 i 1. Tymczasowe obliczenia przechowywane będą w komórce o numerze 2, a końcowy wynik działania tego programu pojawi się w komórce o numerze 3. Przyjmijmy też, że pozostałe komórki na początku działania programu są wyzerowane.

Przy takich założeniach program rozpocznie swoje działanie od wykonania instrukcji C(0,3), która kopiuje zawartość zerowej komórki (wartość pierwszej danej wejściowej) do trzeciej komórki pamięci. Zaraz potem przechodzimy do wykonania instrukcji I(1,2,5), która sprawdza, czy wartości pierwszej i drugiej komórki są równe. Jeżeli tak się stanie, to program zwiększy wskaźnik rozkazów do 5, a to będzie równoważne z zakończeniem programu. Jeżeli jednak powyższy warunek nie będzie spełniony, to przejdziemy do instrukcji A(2) i A(3), co zwiększy o jeden wartości przechowywane w drugiej i trzeciej komórce pamięci. Następnie przejdziemy do instrukcji I(0,0,1). Porównanie wartości zerowej komórki z nią samą zawsze zakończy się orzeczeniem ich równości, a to sprawi, że zawrócimy do instrukcji I(1,2,5).

Prześledźmy teraz działanie tego programu na konkretnych danych.

Wskaźnik rozkazu	Rozkaz	P[0]	P[1]	P[2]	P[3]
0	C(0,3)	1	2	0	1
1	I(1,2,5)	1	2	0	1
2	A(2)	1	2	1	1
3	A(3)	1	2	1	2
4	I(0,0,1)	1	2	1	2
1	I(1,2,5)	1	2	1	2
2	A(2)	1	2	2	2
3	A(3)	1	2	2	3
4	I(0,0,1)	1	2	2	3
1	1(1,2,5)	1	2	2	3
5	-	1	2	2	3

Jak widać, po zakończeniu działania programu w komórce trzeciej, w której miał pojawić się wynik, pojawiło się $3 = 1 + 2$. Nietrudno jest pokazać, że dla dowolnych dodatnich danych wejściowych program ten doda do siebie ich wartości i umieści wynik tej operacji w komórce o numerze 3.

Przedstawiono tutaj tylko dwa matematyczne modele komputerów. Co ciekawe, żaden z nich nie wyprzedził powstania prawdziwego komputera. Historycznie pierwszy z nich powstał w wyniku tego, że Alan Turing nie uzyskał dostępu do pierwszego komputera na świecie, jakim był brytyjski Colossus. Czy zatem rozważania na nimi były jedynie zabijaniem czasu? Nie, bo dzięki nim informatyka wzbogaciła się o wiele czysto teoretycznie wypracowanych koncepcji i rozwiązań problemów.

3

Bramki logiczne i ich układy

Dzisiejsze komputery, aby działać, potrzebują elektryczności. W poprzednich rozdziałach wspominano już o możliwości skonstruowania maszyn przypominających komputery zasadą działania, ale opartych na rozwiązaniach z mechaniki precyzyjnej. Możliwe jest też skonstruowanie komputera w oparciu o zasady hydrauliki i pneumatyki. Na obecnym poziomie wiedzy byłaby to jednak sztuka dla sztuki.

Napisano również, że komputery przetwarzają dane w postaci długich ciągów zer i jedynek reprezentowanych przez brak napięcia elektrycznego i napięcie. Powstaje więc w tym momencie pytanie o to, w jaki sposób przetwarza się tak zapisane informacje. Odpowiedzią na to pytanie są zaskakująco proste struktury zwane bramkami logicznymi.

Jedną z najstarszych gałęzi matematyki jest logika, czyli nauka o wnioskowaniu na podstawie pewnego rodzaju zdań. Uściślając: logika nie zajmuje się wszystkimi zdaniami. W pierwszym rzucie można pominąć wszystkie zdania pytające i te, które kończą się wykrzyknikiem. Zostają nam więc tylko zdania orzekające, ale nawet wśród nich są takie, którymi nie zajmuje się logika. Na przykład spośród dwóch zdań

„Słonko świeci, ptaszek kwili.
Może byśmy coś wypili"

oba są orzekające, ale tylko pierwsze może być traktowane jak zdanie logiczne. Dzieje się tak dlatego, że rozglądając się dookoła i nastawiając uszu, możemy jasno powiedzieć, czy jest ono prawdziwe, czy fałszywe. Drugie natomiast określa nie do końca sprecyzowany zamiar.

W pierwszym zdaniu powyższego przykładu pojawiło się w nieformalny sposób coś jeszcze. Tak naprawdę składa się ono z dwóch zdań, które połączone są spójnikiem „i", co oznacza, że zarówno świecenie, jak i kwilenie zachodzi w tym samym momencie. Znaczy to więc, że całe zdanie będzie prawdziwe, jeżeli aktywność ptasia i słoneczna spotkają się w tej samej chwili.

Oprócz spójnika „i” w logice występuje jeszcze spójnik „lub” oraz funkcja zaprzeczenia znana również negacją. Spójniki te można zdefiniować w następujący sposób:

– *zdanie składające się z dwóch zdań połączonych spójnikiem „i” jest prawdziwe, jeżeli prawdziwe są oba jego zdania składowe, a w pozostałych przypadkach jest fałszywe,*

– *zdanie składające się z dwóch zdań połączonych spójnikiem „lub” jest prawdziwe, jeżeli co najmniej jedno z jego zdań składowych jest prawdziwe, a w pozostałych przypadkach jest fałszywe,*

– *zaprzeczenie zdania zmienia zdanie prawdziwe w fałszywe, a zdanie fałszywe w prawdziwe.*

3.1 Przykłady

Przykład 1:

Zdanie „Polacy grają w piłkę i są w tym najlepsi na świecie” w chwili powstawania tego tekstu nie jest prawdziwe, choć prawdziwe jest jego pierwsze zdanie składowe. Za to zdanie „Polscy piłkarze są słabi i polskie mecze ligowe przez długi czas były ustawiane” jest prawdziwe, bo obie jego składowe są prawdziwe.

Zdanie „Rumianek jest zdrowy lub cyjanek jest zdrowy” jest prawdziwe z uwagi na prawdziwość jednego ze zdań składowych, która utrzymuje się nawet w obliczu przynależenia czytelnika do subkultury Emo. Za to zdanie „Praca w policji jest bezpieczna lub bycie złodziejem jest legalne” jest fałszywe, bo oba zdania składowe są nieprawdziwe.

Zdanie „Nieprawda, że Wisła płynie przez Kraków” jest fałszywe, natomiast zdanie „Nieprawdą jest, że Kraków jest stolicą Polski” jest prawdziwe.

Powyższy przykład poza pokazaniem sposobu łączenia zdań unaocznił jeszcze jeden problem. Zapisywanie zdań logicznych w ten sposób jest rozwlekłe, co w przypadku bardziej skomplikowanych złożeń robi się mocno niepraktyczne. Z tego powodu bardzo często zdania składowe oznacza się często symbolami literowymi, a słowa „prawda” i „fałsz” zastępuje się odpowiednio przez 1 i 0. Przy takich założeniach można się już pokusić o stworzenie tabelek określających prawdziwość zdań, w których występują opisane powyżej spójniki. Oto one:

„i”	0	1
0	0	0
1	0	1

„lub”	0	1
0	0	1
1	1	1

„nie”	0	1
	1	0

Odrobinę większego wysiłku umysłowego wymaga dowód faktu, że z powyższych trzech spójników potrzebne są tak naprawdę tylko dwa, przy czym

mogą to być pary („i”, „nie”) lub („lub”, „nie”). Obie te pary tworzą tzw. zbiory zupełne, co znaczy, że przy ich pomocy można skonstruować każdą inną funkcję, która pobiera dwie wartości zerojedynkowe i przekształca je na zero lub jeden. To niepozorne stwierdzenie przybliża nas o kolejny krok do skonstruowania podstawowych cegiełek budujących komputery.

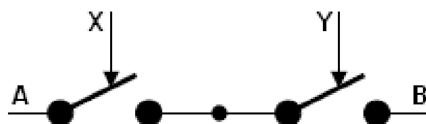
Przykład 2:

Załóżmy, że spójnik „lub” chcemy wyrazić przy pomocy spójników („i”, „nie”). Jak to zrobić? Okazuje się, że „a lub b” to to samo, co „nie (nie a i nie b)”. A oto jeden ze sposobów, w jaki można pokazać, że to prawda.

a	b	a lub b	nie a	nie b	nie a i nie b	nie(nie a i nie b)
0	0	**0**	1	1	1	**0**
0	1	**1**	1	0	0	**1**
1	0	**1**	0	1	0	**1**
1	1	**1**	0	0	0	**1**

Skoro wiemy już, że każdą funkcję o dwóch argumentach zerojedynkowych da się wyrazić poprzez odpowiednie zestawienie funkcji (spójników) „i”, „lub” oraz „nie”, to można skojarzyć z faktem, że w komputerze jedynki i zera reprezentowane są przez prąd i brak prądu, i zapytać, jak to ze sobą połączyć? Jak zaprojektować układy elektryczne, które będą realizowały te funkcje?

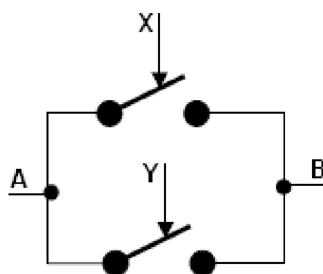
Okazuje się, że funkcje te mogą zostać zrealizowane przy użyciu przełączników zwanych też kluczami. Prześledźmy kolejno przepływ prądu w kilku układach, zaczynając od następującego:



Rysunek 3.1.

Załóżmy, że chcemy przesłać prąd z punktu A do punktu B. Kiedy to będzie możliwe? Tylko i wyłącznie w sytuacji, w której popchniemy (tj. zamkniemy) klucz X i klucz Y. Dzieje się tak dlatego, że prąd może w powyższym układzie płynąć tylko jedną drogą i jeżeli coś ją przerwie, nie będzie to możliwe. Jest więc to prosty model działania spójnika „i”, który wymaga, by spełnione były naraz dwa warunki. A mówiąc jeszcze prościej, popchnięcie klucza jest interpretowane jako jedynka, a zostawienie go w spokoju jako zero.

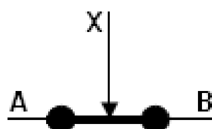
Drugi schemat obrazuje sposób, w jaki można modelować działanie spójnika „lub”.



Rysunek 3.2.

Aby prąd przepłynął z punktu A do punktu B, potrzebne jest zamknięcie tylko jednego z kluczy X lub Y.

Najmniej intuicyjny jest schemat spójnika „nie”, który wygląda w następujący sposób:



Rysunek 3.3.

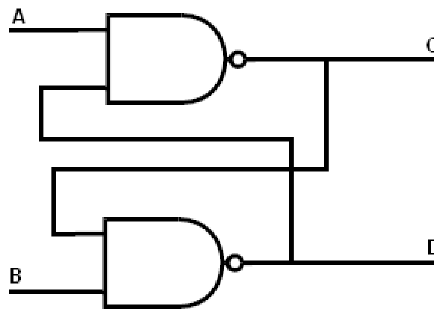
Jak widać, prąd płynie z punktu A do punktu B, jeżeli nie dotykamy klucza X. Znaczy to, że zgodnie z powyższą umową wprowadzamy do układu sygnał zero, a układ odpowiada jedynką, bo jest w stanie przewodzić prąd pomiędzy zadanymi punktami. Co się jednak stanie, jeżeli popchniemy klucz X? Jest to równoznaczne z tym, że droga, którą płynął prąd, zostanie przerwana, a jeszcze inaczej: my wprowadzamy do układu (umowną) jedynkę, a układ reaguje umownym zerem.

Opisane wyżej układy to jeden z możliwych sposobów stworzenia obwodów elektrycznych realizujących podstawowe operacje logiczne. Układy te zwane są bramkami logicznymi, a ponieważ mogą być realizowane w najrozmaitsze sposoby, więc na ich oznaczenie wprowadzono specjalne symbole graficzne sugerujące, co bramka robi, a nie w jaki sposób została wykonana. Oto one:

Po oswojeniu się z istnieniem i działaniem bramek logicznych warto zaznajomić się z ich zwyczajowym nazewnictwem związanym z angielskimi wersjami słów „i", „lub" i „nie". I tak na bramkę modelującą działanie spójnika „i" mówimy „bramka AND", na tę, która modeluje spójnik „lub", mówimy „bramka OR", a tę, która zastępuje negację, nazywamy „bramką NOT".

Bramki, podobnie jak ich abstrakcyjne odpowiedniki można łączyć w większe struktury. Na przykład ustawiając bramkę NOT tuż po bramce AND, otrzymujemy tzw. bramkę NAND, podobnie z bramek OR i NOT można zbudować bramkę NOR. Nie jest to jednak koniec możliwości połączeń bramek logicznych, a dopiero ich początek.

Z dwóch bramek NAND lub dwóch bramek NOR można zbudować coś w rodzaju prostej pamięci zwanej przerzutnikiem. Schemat takiego układu złożonego z dwóch bramek NAND przedstawiony jest na poniższym rysunku.



Rysunek 3.4.

Wejście A jest często opisywane angielskim słowem Set, co ma sugerować, że tym wejściem ustawiamy sygnał, który ma być zapamiętany. Wejście B oznaczane jest słowem Reset. Aby zrozumieć zasadę działania tego urządzenia najlepiej, jest prześledzić, jak ono się zachowuje pod wpływem różnych sygnałów.

Załóżmy na razie, że to faktycznie pamięć jednobitowa, przechowująca wartość zero. Znaczy to, że w danej chwili na wyjściu C znajduje się zero. Załóżmy dodatkowo, że na wejściach A i B wchodzą jedynki. Co się dzieje? Do dolnej bramki wchodzą dwa sygnały. Pierwszy z nich to jedynka, która ustawiana jest na wejściu B, a drugi to zero, które jest aktualnym stanem układu przechowywanym na wyjściu C. W związku z tym zdanie „1 i 0" ma wartość logiczną 0, a jego zaprzeczenie ma wartość 1 i to jest wartość, która w tym momencie pojawia się w punkcie D i na drugim wejściu górnej bramki. Do górnej bramki wchodzą wobec tego dwie jedynki, które są przetwarzane na 0 i wysyłane do punktu C.

W skrócie wygląda to więc tak: w układzie zapamiętano zero i pod wpływem umówionego impulsu wartość ta została wysłana na wyjście.

Załóżmy teraz, że w układzie zapamiętana była jedynka, a na wejścia A i B wchodzą dwie jedynki. Co dzieje się teraz? Zacznijmy znowu od analizy tego, co dzieje się w dolnej bramce. Z założenia, wchodzą do niej dwie jedynki, które przez bramkę NAND przekształcane są w zero, które wysyłane jest do punktu D i na drugie wejście górnej bramki. Z założenia górna bramka musi więc przetworzyć 0 i 1, co zamienia się w jedynkę. Po raz kolejny okazało się więc, że układ oddał wartość, która została w nim wcześniej zmagazynowana.

Pełny zestaw sygnałów wejściowych i związanych z nimi odpowiedzi przedstawia poniższa tabelka.

A	B	S(n)
1	1	S(n-1)
0	1	1
1	0	0
0	0	Stan zabroniony

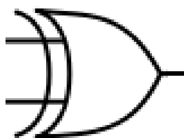
Gdzie S(n) oznacza n-ty stan zapamiętany w urządzeniu.

Idąc za ciosem, możemy się pokusić o zgromadzenie wielu przerzutników w jednym miejscu, co sprawi, że stworzymy wielobitową pamięć. Połączenie tych samych przerzutników w sposób szeregowy może doprowadzić do powstania czegoś w rodzaju licznika.

Dzięki takiemu podejściu można budować coraz bardziej złożone struktury, które modelują coraz bardziej złożone zadania stawiane przed komputerami. Warto zauważyć, że przedstawione tutaj układy były jednymi z najprostszych. Dzisiejsze komputery są znacznie bardziej skomplikowane i oprócz pamiętania wartości i wykonywania na nich prostych operacji logicznych muszą sobie radzić z pełnym zakresem działań matematycznych, przy pomocy których opisuje się zachowanie programów tak różnych jak Microsoft Excel i najnowsze gry akcji. A wszystko to możliwe jest do zrealizowania przy pomocy jedynie trzech elementów, od których wprowadzania rozpoczął się ten rozdział.

3.2 Zadania i pytania

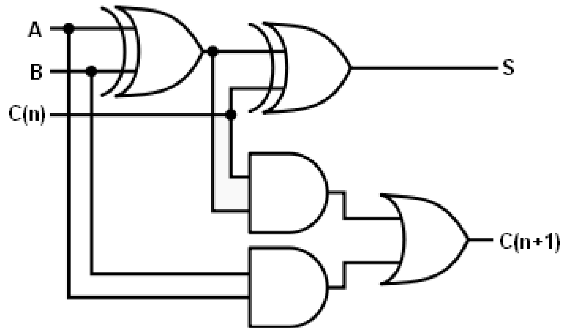
1) Wiedząc, że



jest symbolem bramki XOR, która ma następującą tabelę wartości:

„xor”	0	1
0	0	1
1	1	0

pokazać, że układ o schemacie



Rysunek 3.5.

gdzie:
A - pierwszy składnik sumy
B - drugi składnik sumy
C(n) - przeniesienie z poprzedniej pozycji
C(n+1) - przeniesienie na następną pozycję
S - suma

odpowiada za dodawania do siebie dwóch liczb jednobitowych z uwzględnieniem przeniesienia z poprzedniej pozycji.

2) Oznaczając układ z poprzedniego zadania w wybrany przez siebie sposób, zaproponować sposób zbudowania układu potrafiącego dodawać do siebie liczby czterobitowe.

4

Wprowadzenie do architektury komputerów

Przed nadejściem rewolucji przemysłowej wszystko robiło się ręcznie i każdy wyprodukowany przedmiot był inny. Mówiło się nawet, że posiadał duszę. Na swój sposób było to oczywiście piękne i niezwykle romantyczne, ale przy okazji bardzo niepraktyczne. Wyobraźmy sobie przez chwilę sytuację, w której ogólnie przyjętym sposobem wytwarzania dóbr materialnych jest dalej manufaktura, ale jakimś cudem udało się skonstruować komputer. Co by to dla nas znaczyło? Praktycznie tyle, że każdy komputer byłby inny i chcąc go na przykład rozbudować, musielibyśmy pofatygować się osobiście do rzemieślnika, który go zrobił, bo żaden inny mógłby sobie z tym zadaniem nie dać rady. To oczywiście przerysowana sytuacja, z której jednak wyziera dyktowany regułami ery informatycznej zamysł ujednolicania komputerów, tak by mogły ze sobą współpracować i wymieniać się nie tylko danymi, ale również częściami.

Pierwszy stopień tego ujednolicenia polega na założeniu, że wszystkie komputery będą pracować na podobnych zasadach. Sprowadza się to właściwie do tego, że uznajemy konieczność wprowadzenie podziału na procesor, pamięć i układy wejścia-wyjścia. Gdyby jednak położyć te trzy grupy elementów obok siebie na stole, to raczej na pewno nie doprowadziłoby to do powstania komputera. Potrzebne jest jeszcze coś, co połączy je w logiczną całość. Tym czymś jest tzw. magistrala komputerowa.

W tym miejscu pojawia się pojęcie architektury komputera. Trzeba przyznać, że jest to pojęcie, które nie jest do końca precyzyjne i bywa też stosowane zamiennie dla określenia wielu rzeczy. Zazwyczaj jednak określa ono sposób połączenia pomiędzy procesorem, pamięcią i układami wejścia-wyjścia, czyli innymi słowy mówi o tym, w jaki sposób zbudowana jest magistrala.

Magistrala to nic innego jak zbiór linii oraz układów dbających o przesyłanie sygnałów w urządzeniach mikroprocesorowych. Magistrale budowane są z szyn, czyli przewodów przesyłających informacje konkretnego typu. Pierwszym typem są szyny sterujące, w których przesyłana jest informacja o tym, czy spodziewamy się zapisania, czy odczytania danych, których opis wysyłany jest innymi szynami. Drugi typ to szyny adresowe, którymi przesyłana

jest informacja o adresie (numerze komórki w pamięci), do którego ma zostać zapisany lub z którego ma zostać odczytany zestaw danych. Trzeci i ostatni typ to szyna danych, którą płyną dane.

Podstawowym kryterium rozróżniania magistral jest sposób, w jaki przesyłane są nimi informacje. Wyróżniamy magistrale jednokierunkowe i dwukierunkowe. Oprócz tego bardzo ważnym kryterium jest szerokość magistrali, czyli ilość bitów, które jest ona w stanie przesłać za jednym zamachem. Oczywistym faktem jest to, że im szersza magistrala, tym lepiej dla szybkości naszego komputera. Niemniej ważną wielkością określającą magistrale jest częstotliwość ich taktowania, która jest tym lepsza, im większa, bo wtedy czas, który musi upłynąć między kolejnymi wysyłkami danych, jest krótszy. Dlatego właśnie niemądrym postępowaniem jest zapożyczanie się na najnowszy i najdroższy procesor w sytuacji, w której mamy starą płytę główną z wolną magistralą. Dzieje się tak dlatego, że to, co wypracuje procesor i tak będzie musiało poczekać na wysłanie do innych części komputera, tak jak musi poczekać woda, która leje się do wanny szerokim strumieniem, a ma się z niej wylać malutką dziurką. I to tyle na temat magistrali, którą można by porównać do kręgosłupa komputera.

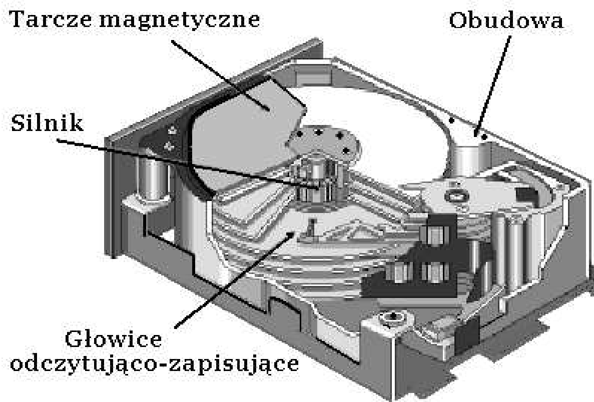
Drugą, nie mniej ważną kategorią części komputerowych są urządzenia wejścia-wyjścia. Należą do nich:

- pamięci masowe, a wśród nich dyskietki FDD (powoli stające się przeżytkiem), dyski twarde HDD, dyski optyczne (CD-ROM, CD-RW, DVD, HD DVD, BlueRay),
- klawiatury,
- karty graficzne i monitory,
- karty dźwiękowe i głośniki,
- urządzenia wskazujące, czyli np. myszy i tablety graficzne,
- karty sieciowe, modemy itp.

Znane jest słynne powiedzenie Billa Gatesa o tym, że każdemu powinno wystarczyć 640 kilobajtów pamięci. Podobne przekonanie panowało kiedyś na temat oszałamiającej ilości danych, które da się upchnąć na dyskietce FDD, a było to niecałe półtora megabajta. Na szczęście czasy popularności tego nośnika mijają prawdopodobnie chyba na zawsze, a zwiastun ich końca pojawił się na początku lat 90, kiedy na polskich giełdach komputerowych pojawiła się jedna z pierwszych gier komputerowych wydanych na CD-ROM. Była to westernowa gra „Mad Dog McCree”, a czasy były takie, że mało kto miał czytnik CD-ROM, a jeszcze mniej ludzi miało jakąkolwiek legalną grę, wliczając w to windowsowego „Sapera”. Zgodnie z hasłem „Polak potrafi” gra została skopiowana na dyskietki i już wtedy co bystrzejsi zaczynali podejrzewać, że coś jest nie tak, jak powinno, skoro na jedną grę potrzeba około 50 dyskietek.

Znacznie lepszym rozwiązaniem w tej materii wydają się dyski twarde, których obecność w komputerze wydaje się dziś oczywistością, a ich pojemności mierzone są setkami gigabajtów. Warto jednak przypomnieć, że pierwszy

komputer wyposażony w dysk twardy o wstrząsającej pojemności 4.4 megabajta pojawił się dopiero w 1956 roku, a jeszcze w latach 90 ubiegłego wieku niemal wszystkie komputery firmy Commodore nie miały dysków twardych. To tyle tytułem ciekawostek, zastanówmy się teraz czym właściwie jest dysk twardy i w jaki sposób działa. Schematyczna budowa mechanicznych dysków twardych przedstawiona jest na poniższym rysunku.



Rysunek 4.1.

Dysk stały składa się z zamkniętego w obudowie, wirującego talerza (dysku) lub zespołu talerzy pokrytych nośnikiem magnetycznym oraz z głowic elektromagnetycznych umożliwiających zapis oraz odczyt danych. Na każdą powierzchnię talerza dysku przypada po jednej głowicy odczytu i zapisu. Głowice są umieszczone na ramionach i w stanie spoczynku stykają się z talerzem blisko jego osi, a w czasie pracy unoszą się tak, by nie rysować jego powierzchni.

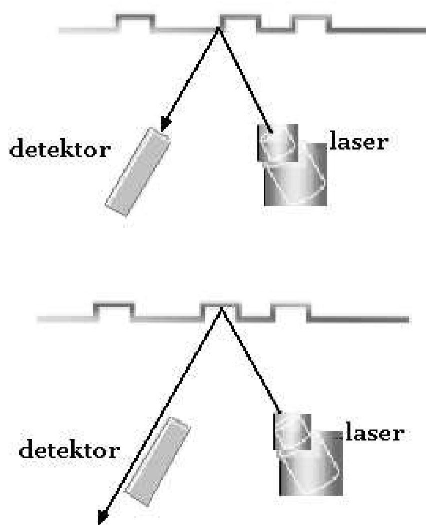
Informacja jest zapisywana na dyskach poprzez zmianę kierunku namagnesowania fragmentów materiału magnetycznego, który je pokrywa. Odczyt informacji jest możliwy dzięki wykorzystaniu indukcji napięcia elektrycznego stwarzanej przez wirujące, namagnesowane talerze.

Aby to wszystko mogło działać, potrzebne są oczywiście zintegrowane układy elektryczne, które sterują ruchem talerzy, ramion i przygotowaniem danych do zapisu oraz obróbką danych, które zostały odczytane z dysku.

O ile dyski twarde są bardzo wygodne, to jednak mają przykrą tendencję do szybkiego zapełniania się. Pół biedy, kiedy te dane są mało ważne, co jednak zrobić, gdy ich znaczenie jest z jakichś względów trudne do przecenienia, tak jak choćby zestawienie informacji o kontach bankowych? Jedną z możliwości wybrnięcia z tej sytuacji jest archiwizacja danych i skasowanie z dysku twardego tych, które aktualnie są rzadko używane. W przypadku kont bankowych mogłoby to być na przykład skasowanie historii transakcji starszych niż, powiedzmy, miesiąc.

Logiczne wydaje się więc, że do magazynowania takich danych będziemy potrzebowali nośników o ogromnej pojemności, które nie są na stałe zamontowane w komputerze. Co prawda, w przypadku banków sposób archiwizacji danych jest najczęściej objęty tajemnicą, ale można się założyć, że dane te nie są gromadzone na płytach CD-ROM czy DVD. Zupełnie inaczej sprawa się ma z prywatnymi użytkownikami komputerów, dla których kupno drogich urządzeń archiwizacyjnych mija się po prostu z celem.

Dane gromadzone na napędach optycznych zapisywane są również w binarnej postaci, a rolę zer i jedynek w tym zapisie spełniają mikroskopijne wgłębienia i wypukłości, które na różne sposoby odbijają skierowane na nie światło laserowe. Zasadę odczytu dysków optycznych można prześledzić na poniższym rysunku (rys. 4.2.).



Rysunek 4.2.

Jak widać powyżej, światło skierowane na wybrzuszenie trafia do detektora, co może być zinterpretowane jako jedynka, a światło skierowane na wgłębienie omija ten detektor, co jest interpretowane jako zero.

Bez względu na to, czy mamy tu do czynienia z dyskami CD-ROM, DVD, HD DVD, czy też BlueRay, idea ich odczytu sprowadza się do jakieś wariacji na powyższy temat. Jedyną znaczącą różnicą jest tutaj długość fali świetlnej służącej do ich odczytywania. Im jest ona mniejsza, tym gęściej dane mogą zostać upakowane na płycie. To właśnie dlatego płyty CD-ROM, DVD i BlueRay mają te same rozmiary, a ich pojemności różnią się w znaczący sposób.

Odrobinę inaczej sprawa się ma w przypadku dysków optycznych przeznaczonych do samodzielnego zapisu. Tutaj zamiast wgłębień oraz wybrzuszeń wykorzystuje się zjawiska rozproszenia i odbicia światła. Nagrywanie płyt (potocznie zwane wypalaniem) sprowadza się do bardzo szybkiej zmiany energii

lasera skierowanego na pustą płytę. W przypadku, gdy energia ta podgrzeje płytę punktowo do około 700 stopni Celsjusza, dojdzie do stopienia materiału, który od tej pory będzie rozpraszał światło, przez co nie trafi ono do detektora i zostanie zinterpretowane jako zero. Jeżeli natomiast ten sam laser podgrzeje płytę do zaledwie 400 stopni Celsjusza, to dojdzie do punktowej krystalizacji, która będzie podczas oświetlania odbijała światło, które trafi do detektora i zostanie zinterpretowane jako jedynka.

Aby dane mogły zostać zapisane na nośnikach, muszą w jakiś sposób zostać wprowadzone do komputerów. Jeśli są to dane tekstowe, to najlepszym rozwiązaniem są klawiatury, które na szczęście wyparły czytniki kart oraz taśm perforowanych. W przypadku tworzenia grafiki i sterowania komputerem najlepszymi rozwiązaniami wydają się myszy i tablety graficzne, które odczytują fizyczne położenie kontrolera i wysyłają je w numerycznej postaci do komputera. Wszystkim malkontentom narzekającym na taki stan rzeczy warto uświadomić fakt, że grafikę można tworzyć w inny, mniej wygodny sposób i wielu grafików komputerowych zaczynających swoją przygodę od komputerów Commodore C64 właśnie to robiło, używając w tym celu joysticka.

Zasada działania innych urządzeń wejścia-wyjścia zostanie opisana w rozdziałach dotyczących grafiki komputerowej, dźwięku i Internetu.

Tak samo jak człowiek będący w bibliotece potrzebuje pamięci, aby móc czytać, tak samo komputer potrzebuje pamięci, by mógł przetwarzać napływające do niego dane wejściowe, przekształcając je do innej postaci. Istnieją różne rodzaje pamięci. Pierwszy poważny podział dotyczy możliwości ich modyfikowania i wskazuje na istnienie pamięci typu ROM i RAM.

Pamięć ROM to pamięć, z której łatwiej jest coś odczytać aniżeli zapisać, a bywa i tak, że możliwością jej zapisu dysponuje jedynie jej producent. Stąd właśnie wziął się jej skrót (ang. *Read Only Memory* - pamięć tylko do odczytu). Z uwagi na taką przypadłość w pamięciach tego typu przechowywane są dane, które muszą pozostać nienaruszone nawet jeżeli odetniemy komputerowi zasilanie, a które z jakichś powodów nie mogą być przechowywane na dysku twardym. Pamięci tego typu dzielą się na kilka podtypów, które wyszczególniono poniżej:

- MROM (ang. *Mask programmable ROM*), które są programowane w trakcie produkcji,
- PROM (ang. *Programmable ROM*), czyli pamięć programowalna z wielokrotnym odczytem i jednokrotnym zapisem. Pierwsze pamięci tego typu były programowane poprzez przepalenie cienkich drucików wbudowanych w ich strukturę,
- EPROM (ang. *Erasable Programmable ROM*) to kasowalne pamięci tylko do odczytu. Są to pamięci, do których zaprogramowania potrzebne jest też specjalne urządzenie zwane programatorem PROM (PROM Programmer albo PROM Burner). Pamięci tego typu montowane są zazwyczaj w obudowie ceramicznej ze szklanym „okienkiem” umożliwiającym skasowanie ich zawartości poprzez naświetlanie ultrafioletem,

- EEPROM (ang. *Electrically Erasable Programmable ROM*) to pamięci kasowalne i programowalne elektrycznie,
- Flash EEPROM to pamięci, dla których kasowanie danych, a co za tym idzie także ich zapisywanie odbywa się tylko dla pewnej liczby komórek pamięci podczas operacji programowania.

Zupełnie innym rodzajem pamięci są pamięci typu RAM (od ang. *Random Access Memory*, czyli pamięć o dostępie swobodnym). W pamięciach tego typu przechowuje się programy oraz ich dane wykorzystując w ten sposób podstawową zaletę tychże pamięci, czyli elastyczność ich stosowania. Podstawową wadą jest w tym przypadku fakt, że dla większości rodzajów pamięci zawarte w nich dane są bezpowrotnie tracone kilka sekund po odcięciu zasilania.

Najogólniejszym sposobem podziału pamięci typu RAM jest podział na statyczne (skrót SRAM) i dynamiczne (skrót DRAM). Inaczej niż zwykle pamięci dynamiczne wcale nie są lepsze od statycznych, a wręcz przeciwnie. Pamięci statyczne są znacznie szybsze od dynamicznych i nie potrzebują tzw. odświeżania, bez którego szybko tracą swoją zawartość. Przyczyna, dla której pomimo wszystko wykorzystuje się pamięci dynamiczne, jest bardzo prozaiczna i jest nią cena.

Nieco inne kryterium podziału pamięci tego typu polega na rozróżnieniu ich umiejscowienia. Jak już wcześniej wspomniano, pamięć i procesor połączone są magistralą, przez co bywa, że dane zapisane w jednym lub drugim miejscu muszą przebyć długą i powolną drogę z jednego miejsca na drugie. Coraz większe prędkości pracy procesorów wymagają zwiększenia prędkości pamięci, co nie zawsze idzie ze sobą w parze. Wtedy procesor musi czekać coraz dłużej na kolejną porcję danych odczytywanych lub zapisywanych w pamięci RAM. Tymczasem badania statystyczne dowiodły, że większość odwołań do pamięci mieści się najczęściej w obszarze o rozmiarze ok. 16 kilobajtów. Konsekwencją tego spostrzeżenia było pojawienie się małych, lecz bardzo szybkich pamięci podręcznych określanych angielskim słowem cache. Pamięci te są integrowane z procesorami, przez co dostęp do nich jest kilkukrotnie szybszy i zoptymalizowany niż do pamięci, o których mowa była wcześniej.

Skoro mowa o pamięciach, to należy wspomnieć również o tzw. pamięciach wirtualnych, które z pamięcią i wirtualnością mają tyle wspólnego, co świnka morska ze schabowym i Bałtykiem. Pamięci wirtualne są swego rodzaju oszustwem, którego działanie polega na zapisywaniu danych na dysku twardym w pliku lub partycji wymiany. Ze względu na to, że dzisiejsze dyski twarde wielokrotnie przewyższają objętościowo rozmiary pamięci RAM, której nowoczesne programy komputerowe wymagają naprawdę dużo, stosuje się wybieg polegający na zapisywaniu tychże danych na dysk komputerowy. Uczciwie trzeba powiedzieć, że takie podejście sprawdza się pod względem poprawności danych, ale na pewno nie pod względem szybkości dostępu do nich. Z tego też powodu bardzo wolne działanie naszego komputera może być dla nas wskazówką do tego, aby sprawdzić, czy pamięć wirtualna naszego komputera jest wykorzystywana w optymalny sposób.

Ostatnią, choć kto wie, czy nie najważniejszą częścią komputera jest jego procesor, określany często skrótem CPU (ang. *Central Processing Unit*, czyli centralna jednostka przetwarzająca). Jest to urządzenie cyfrowe, które w sposób sekwencyjny pobiera dane z pamięci i interpretuje je jako rozkazy do wykonania. Liczba tych rozkazów jest zwykle bardzo ograniczona i z góry określona przez producenta jako lista rozkazów procesora.

Procesory (zwane mikroprocesorami) wykonywane są zwykle jako układy scalone zamknięte w hermetycznej obudowie. Ich sercem jest monokryształ krzemu, na który techniką fotolitografii nanosi się szereg warstw półprzewodnikowych tworzących, w zależności od zastosowania, sieć od kilku tysięcy do kilkuset milionów tranzystorów.

Jedną z podstawowych atrybutów procesora jest liczba bitów wchodzących w skład zestawu zwanego słowem, na którym wykonywane są podstawowe operacje obliczeniowe. Jeśli słowo ma np. 64 bity, mówimy, że procesor jest 64-bitowy.

Inną ważną cechą określającą procesor jest szybkość, z jaką wykonuje on program. Przy danej architekturze procesora szybkość ta w znacznym stopniu zależy od czasu trwania pojedynczego taktu.

W strukturze procesora można wyróżnić następujące elementy:

- zespół rejestrów do przechowywania danych wejściowych i wyników przeprowadzonych na nich operacji,
- jednostkę arytmetyczną (arytmometr), której zadaniem jest wykonywanie operacji obliczeniowych na wskazanych danych,
- układ sterujący przebiegiem wykonywania programu.

Początkowo procesory miały tylko jeden rdzeń, zaś zwiększanie szybkości ich działania odbywało się poprzez zwiększanie tzw. częstotliwości taktowania, czyli podawania kolejnych danych na jego wejście. Jednakże, wiązało się to ze znaczącym wzrostem energii pobieranej przez procesor, a co za tym idzie, ze wzrostem jego ciepłoty, co dla jego delikatnej struktury mogło być zabójstwem. Rozwiązaniem tego problemu stało się zwiększanie liczby rdzeni procesora. Pierwszym modelem procesora, w którym zastosowano takie podejście, był Intel Pentium D, ale prawdziwą popularność zyskał dopiero Intel Core 2, który na dobre ugruntował ten kierunek zmian.

W dzisiejszych procesorach zdarzają się już struktury czterordzeniowe (Intel Quad i AMD Phenom), a firma Intel prowadzi testy procesora ośmiordzeniowego.

Na koniec warto jeszcze wspomnieć o planach na przyszłość. Przez ostatnich parę lat mówiło się o tzw. kresie możliwości technologicznych, czyli sytuacji, w której procesory nie będą już mogły pracować z większymi szybkościami. Kres taki wieszczono na koniec roku 2008 i jak do tej pory do niego nie doszło. Ewolucja procesorów, jakie znamy dzisiaj, podąża w kierunku zwiększania liczby wchodzących w ich skład tranzystorów przy jednoczesnym utrzymywaniu tych samych rozmiarów, tak by zmniejszyć drogę przemieszczających

się w nich sygnałów. Wiąże się to ze zmniejszeniem rozmiarów tranzystorów i odległości między nimi. Dziś odległości te są rzędu 45 nm (45 miliardowych części metra) i jak nietrudno się domyślić, kiedyś to zmniejszanie będzie musiało się skończyć, chyba że ktoś wymyśli, jak robić tranzystory mniejsze od atomów.

Istnieją jednak inne podejścia do tego tematu, które z uwagi na to, że bazują na całkiem innych założeniach, rokują duże nadzieje na przyszłość. Jednym z takich podejść jest tzw. komputer kwantowy. Działanie takiej maszyny oparte jest na tzw. qbitach, czyli bitach kwantowych. O ile tradycyjny bit występuje w dwóch wykluczających się stanach (zero albo jeden), o tyle qbit ma tych stanów tyle, ile stanów ma reprezentująca go struktura zbudowana z cząstek elementarnych, a co więcej wszystkie te stany występują jednocześnie. Wydaje się to kompletnym wariactwem, ale dzięki tej cesze możliwe jest przeformułowanie wielu stosowanych powszechnie w informatyce algorytmów do postaci nadającej im szybkość, która prawdopodobnie nie zostanie nigdy osiągnięta przez tradycyjne procesory. Mało tego, podwojenie szybkości działania procesora kwantowego osiąga się poprzez oddanie do jego dyspozycji tylko jednego dodatkowego qbitu.

Wszystko to sprawia, że mimo czarnych wizji futurystów informatycy jeszcze przez długi czas będą mieli nad czym myśleć, choć na razie nic nie wskazuje na to, by raz ustalony podział na procesor, pamięć, urządzenia wejścia-wyjścia i łączącą je magistralę miał się kiedykolwiek zmienić.

5

Systemy operacyjne

W dzisiejszych czasach coraz trudniej jest kupić gotowy zestaw komputerowy bez zainstalowanego systemu operacyjnego najczęściej pochodzącego od firmy Microsoft. Bardziej zaprawieni w bojach informatycy-majsterkowicze decydują się czasem na samodzielne złożenie komputera z zakupionych wcześniej części. Dla nich zaskoczeniem nie jest, że świeżo złożony komputer świetnie sprawdza się co najwyżej w roli podstawki pod kurz, gdyż całe oprogramowanie, które się w nim znajduje, to najczęściej jedynie BIOS (ang. Basic Input Output System - podstawowy system wejścia wyjścia). Dopiero użycie stosownego oprogramowania w postaci systemu operacyjnego sprawia, że możemy przy pomocy komputera rozpocząć pracę, zabawę czy relaks. I choć pojęcie to jest szeroko znane, to jednak często zdarza się, że fraza „system operacyjny" pada w rozmowie bez zrozumienia tego, czym właściwie jest i jak się zachowuje oprogramowanie kryjące się pod tą nazwą. Przejdźmy zatem do wyjaśnienia tego tematu.

Na początku komputery były urządzeniami o astronomicznych wręcz cenach. Mało kogo byłoby wtedy stać na posiadanie prywatnego komputera. Znane jest stwierdzenie jednego z członków zarządu firmy IBM, który kilka dekad temu stwierdził, że aby zaspokoić potrzeby obliczeniowe całego świata należałoby wyprodukować ok. 60 (słownie sześćdziesięciu) komputerów. Stwierdzenie to może się dziś wydać śmieszne zwłaszcza w ustach człowieka związanego z taką firmą, obrazuje jednak skalę kosztowności komputerów w tamtych czasach i wynikającą z tego konieczność optymalizacji ich wydajności bez ich rozbudowywania, co było albo technicznie niemożliwe, albo niewyobrażalnie drogie.

Szybko stało się jasne, że wprowadzenie danych do komputera i wyprowadzenie ich na wyjście (wyświetlenie na ekranie, wydrukowanie czy skierowanie do pliku wyjściowego) jest najczęściej dużo dłuższe niż działanie obliczeniowej części programu. Z tego powodu pojawiła się koncepcja wydzielenia zestawu programów pozwalających na komunikację użytkownika ze sprzętem i umieszczenie ich na stałe w pamięci komputera, co owocowało pewną oszczędnością czasu oraz uproszczeniem programów tworzonych przez użytkowników. Warto

tutaj zauważyć, że czasy, w których takie postępowanie nie było standardem nie są zbyt odległe. Jeszcze naście lat temu, gdy czasy swojej największej popularności święciły komputery z linii Commodore Amiga, programiści gier komputerowych za każdym razem stawali przed problemem samodzielnego wymuszenia na komputerze odczytu oraz zapisu danych na dyskietkę i trzeba przyznać, że czasem realizowali to w bardzo wymyślne sposoby. Koronnym przykładem może tu być gra pt. „Cannon fodder", która pozwalała na sformatowanie dyskietki przechowującej zapis stanów tej gry i robiła to w tak cudaczny sposób, że każdy inny program uznawał tę dyskietkę za popsutą.

Wydzielenie procedur wejścia/wyjścia nie rozwiązało jednak problemu ostatecznie. Warto zauważyć, że na przykład w czasie między naciśnięciem kolejnych klawiszy na klawiaturze, komputer nie wykonuje żadnej pracy poza czekaniem, czyli de facto leży odłogiem. Z tego powodu pojawiła się koncepcja, by tego typu przerwy wykorzystać na wykonywanie programów, które rozpoczęły swoje działanie wcześniej na innym zestawie danych wejściowych. Dziś funkcjonalność taka znana jest pod nazwą wielozadaniowości i przyjmuje się ją za oczywistość. Warto jednak przypomnieć, że wcale nie tak dawno jednym z najpopularniejszych systemów komputerowych był MS-DOS, który pozwalał na jedynie szczątkową wielozadaniowość związaną z obsługą tak zwanych programów rezydentnych.

Innym ciekawym rozwiązaniem było przerzucenie kontroli nad komputerem na język programowania zaszyty na stałe w pamięci komputera. Takie postępowanie było bardzo powszechne w czasach komputerów ośmiobitowych takich jak komputery Commodore C64. Najczęściej wykorzystywano do tego celu język BASIC, który do cudów informatyki nie należy, ale za to ma małe wymagania. Użytkownik komputera wyposażonego w program potrafiący interpretować programy napisane w BASICU mógł sterować pracą tego komputera, tworząc proste programy, co na owe czasy wydawało się rewolucją, a dziś przez większość użytkowników litościwie zostałoby skwitowane jako niedorzeczność.

Z czasem zbiór spostrzeżeń i doświadczeń różnych programistów zaczął ewoluować w kierunku ujednolicenia, co doprowadziło do powstania Systemów operacyjnych (ang. skrót OS *Operating System*), czyli oprogramowania zarządzającego sprzętem komputerowym, które tworzy środowisko do uruchamiania i kontroli zadań użytkownika.

W dzisiejszych czasach od systemu operacyjnego wymaga się, żeby radził sobie z następującymi sprawami:

– planowaniem oraz przydziałem czasu procesora poszczególnym zadaniom (jest to szczególnie ważne ze względu na fakt, że niemal wszystkie obecnie używane systemy operacyjne są wielozadaniowe, co znaczy, że potrafią sobie poradzić z obsługą wielu jednocześnie uruchomionych programów),
– kontrolą i przydziałem pamięci operacyjnej dla uruchomionych zadań,
– dostarczaniem mechanizmów do synchronizacji zadań i komunikacji pomiędzy zadaniami,

- obsługą sprzętu oraz zapewnieniem równolegle wykonywanym zadaniom jednolitego wolnego od konfliktów dostępu do sprzętu,
- ustalaniem i obsługą połączeń sieciowych,
- zarządzaniem plikami.

Oprócz tego wiele systemów operacyjnych posiada środowiska graficzne ułatwiające komunikację maszyny z użytkownikiem. Nie jest to jednak obowiązkowy składnik systemów operacyjnych, a jedynie miły akcent, który zwiększa komfort pracy z komputerem. Warto zauważyć, że coraz bardziej popularne systemy z rodziny Linux umożliwiają w trakcie instalacji wybór sposobu, w jaki chcemy komunikować się z komputerem. Możliwy jest wybór środowiska tekstowego lub różnych środowisk graficznych takich jak Gnome czy KDE.

Jakikolwiek byłby system, to jego działanie polega na wykorzystaniu dwustronnego przepływu danych przez różne warstwy komputera. Zaczynając od użytkownika mamy, jego dane, które wędrują do programu, a ten przekazuje je do systemu, który dogaduje się ze sprzętem. Aby przesłać opracowane dane do użytkownika, pokonywana jest ta sama droga, tyle że w przeciwnym kierunku.

By to wszystko było możliwe, system musi udostępniać interfejs (czyli swego rodzaju zestaw gniazdek, do których programy będą mogły podłączać się na zasadzie wtyczki). Wymaganie torealizuje się zwykle poprzez udostępnienie tzw. API (ang. *Application Programming Interface*), czyli zestawu funkcji realizujących podstawowe zadania, z którymi radzą sobie komputery, lub poprzez tzw. wywołania systemowe. Wszystko to przypomina klocki, z których programista aplikacyjny składa swoje programy, nie martwiąc się za każdym razem o konieczność pisania podprogramu, który pozwoli mu na przykład na odczytanie zawartości pliku, o ile jest to w danym systemie możliwe. Takie podejście sprawia, że programista najczęściej nie musi dysponować szczegółową wiedzą na temat sprzętu, który może się składać z tysięcy różnych części komputerowych. Dzieje się tak dlatego, że wiedza ta jest niejako zmagazynowana w systemie komputerowym, który przejmuje na siebie odpowiedzialność za komunikację ze sprzętem, eliminując w ten sposób wiele potencjalnych problemów związanych z nieznajomością specyfikacji poszczególnych urządzeń.

Jeszcze precyzyjniej należałoby powiedzieć, że system składa się ze swego rodzaju warstw otaczających sprzęt. Najbliżej sprzętu są sterowniki, czyli programy zbierające informację z przypisanych do nich urządzeń i dostarczające im danych z systemu. Później mamy jądro systemu, a nad nim powłokę. Oprócz tego wyróżnia się jeszcze tzw. system plików, czyli sposób zapisu struktur danych na nośniku informacji, jakim obecnie najczęściej jest dysk twardy.

Jądro systemu wykonuje i kontroluje programy uruchamiane przez użytkownika i sam system. Składają się na niego moduł planisty oraz przełącznika zadań i czasem dodatkowe moduły, które zostaną opisane poniżej. Moduł planisty ustala, które zadanie będzie aktualnie wykonywane i ile czasu powinno się na to poświęcić. Moduł przełącznika zadań, jak sama nazwa wskazuje, odpowiada za przełączanie się między zadaniami. Oprócz tego w zależności od systemu zdarzają się moduły, które zapewniają synchronizację i komunikację

między zadaniami, oraz takie, które dbają o przydział i ochronę pamięci wykorzystywanej przez uruchomione zadanie.

Skoro już tyle napisano odnośnie tego, że systemy mogą się od siebie różnić, to warto zastanowić się nad podstawowymi kryteriami ich klasyfikowania. Najogólniejszym kryterium jest tutaj podział na systemy czasu rzeczywistego (RTOS od ang. *Real Time Operating System*) i systemy czasowo niedeterministyczne. Najprościej byłoby powiedzieć, że systemy czasu rzeczywistego obsługują urządzenia, w których czas wypracowania odpowiedzi na bodziec pojawiający się w ich otoczeniu musi być niewielki ze względu na ogromną cenę tych urządzeń bądź ich strategiczne znaczenie. Jak nietrudno się zorientować, taki wymóg nie zawsze może być spełniony, więc dodatkowo systemy te dzieli się na twarde i miękkie. Twarde systemy operacyjne czasu rzeczywistego charakteryzują się tym, że czas odpowiedzi na każdy bodziec jest w ich przypadku znany i ograniczony z góry, a miękkie to systemy, dla których przynajmniej jeden bodziec może doprowadzić do nieokreślonej długości oczekiwania na odpowiedź. Oczywiście ideałem byłoby doprowadzenie do sytuacji, w której każdy system komputerowy jest twardym systemem operacyjnym czasu rzeczywistego, ale na razie jest to nie bardzo możliwe i tego typu systemy instalowane są na przykład w samochodowych systemach ABS, centralach telefonicznych czy lądownikach sond, które wysyłano na inne planety.

Ze względu na to, w jaki sposób dochodzi do przełączenia się między kolejnymi zadaniami, systemy operacyjne dzieli się na systemy z wywłaszczeniem zadań i bez wywłaszczenia. W systemach z wywłaszczeniem zadań możliwa jest sytuacja, w której może dojść do wstrzymania aktualnie wykonywanego zadania, jeżeli stosowny algorytm uzna, że trwa ono już zbyt długo. W systemach bez wywłaszczenia zadania informują tenże algorytm o tym, kiedy planują przekazać innym zadaniom możliwość wykonywania. Jak łatwo się domyślić, wywłaszczenie jest czymś w rodzaju wentyla bezpieczeństwa, który chroni przed zablokowaniem komputera przez program, który np. ze względu na zawarte w sobie błędy zaczął zachowywać się w nieprzewidywalny sposób.

Ze względu na to, na ile system operacyjny zezwala na swoją modyfikację, systemy dzieli się na otwarte i wbudowane. Otwarte systemy operacyjne można uruchomić na dowolnej maszynie należącej do pewnej kategorii. Może to być na przykład dowolny komputer klasy PC. Ponieważ w takich przypadkach nie da się przewidzieć choćby wszystkich kombinacji części wchodzących w składa takich urządzeń, więc dopuszcza się pewną możliwość modyfikowania systemu przez użytkownika. Przykładem takiej modyfikacji może być instalowanie sterowników pozwalających na wykorzystanie pełnej mocy karty graficznej w komputerze działającym w oparciu o system Windows. Wbudowane systemy dadzą się uruchomić jedynie na konkretnym urządzeniu takim jak na przykład konkretny model telefonu komórkowego czy urządzenie wspomagające nawigację satelitarną. Ze względu na takie ograniczenie w systemach wbudowanych, które najczęściej instalowane są fabrycznie przez producentów skojarzonych z nimi urządzeń, ogranicza się możliwość dokonywania zmian, które mogłyby doprowadzić do destabilizacji pracy takich urządzeń.

Pod względem środowiska użytego do implementacji systemu można wprowadzić podział na programowe i sprzętowe. Sprzętowe systemy operacyjne to takie, które są przypisane do pewnego rodzaju procesorów, które dzięki temu mogą optymalizować pewien zakres czynności wykonywanych przez system. W przypadku systemów programowych nie może być mowy o takim przypisaniu, muszę one być oparte na algorytmach działających w zbliżony sposób na wszystkich rodzajach procesorów.

Ze wszystkich tych podziałów, wyłania się prawdziwa w większości przypadków reguła mówiąca, że otwartymi systemami operacyjnymi są najczęściej systemy w pełni programowe, czasowo niedeterministyczne (czyli inne niż systemy RTOS) i stosujące wywłaszczenie przy przełączaniu zadań. W przypadku wbudowanych systemów operacyjnych możemy mówić, że są one najczęściej czasowo deterministyczne i że zwykle nie stosują wywłaszczenia zadań, a bywa nawet, że są w całości zaszywane w sprzęcie.

Bez względu na to, jakiego typu będzie system operacyjny, będzie musiał sobie poradzić z obsługą tzw. zasobów sprzętowych, na które składa się procesor i pamięć, a najczęściej również pewne urządzenia zewnętrzne i system plików. W przypadku procesora problem sprowadza się do przydzielania jego czasu poszczególnym zadaniom uruchomionym w obrębie systemu. Zarządzanie pamięcią sprowadza się do przydzielania zadaniom pewnego jej obszaru, który mogą modyfikować w bezpieczny sposób. Jeżeli chodzi o urządzenia zewnętrzne, to ich kontrola polega między innymi na udostępnianiu i sterowaniu urządzeniami pamięci masowej takimi jak twarde dyski czy karty pamięci, na przydzielaniu przestrzeni na dyskach, udostępnianiu i sterowaniu pracy drukarek, skanerów itp. Zarządzanie ostatnią częścią zasobów systemowych, czyli systemem plików sprowadza się do organizacji i udostępniania informacji (grupowanie w katalogi, umożliwianie odczytu i zapisu) oraz na ochronie i autoryzacji dostępu do określonych informacji, co polega najczęściej na nadawaniu im stosownych praw odczytu, zapisu i wykonania.

Dokładniej rzecz ujmując, do głównych zadań systemu operacyjnego należy zaliczyć cztery czynności. Pierwsza z nich to tworzenie deskryptora zasobu. Deskryptor zasobu to struktura, która przechowuje informacje o dostępności i zajętości danego zasobu. Analogicznie rozważamy również usuwanie deskryptora zasobu. W przypadku, gdy deskryptor zasobu wskazuje, że zasób ten jest nieużywany, może dojść do realizacji żądania przydziału, a potem do jego zwolnienia i odzyskania zasobu. W zależności od tego, z jakim zasobem mamy do czynienia, zadania te będą się sprowadzały do różnych czynności.

5.1 Przykład

Uruchomione i działające aktualnie w komputerze programy nazywamy procesami. Nawet jeżeli proces zajmuje się robieniem niczego, to i tak wymaga przydzielenia mu pewnej części pamięci i czasu procesora. Częściej wymaga jeszcze dostępu do urządzeń wejścia/wyjścia i systemu plików, a wszystko to

powinno się odbywać w pełnej świadomości faktu, że obok działają inne procesy, które mogą wymagać podobnych zasobów. Dlatego też system poza tworzeniem, wstrzymywaniem i wznawianiem pracy procesu powinien dysponować również mechanizmami wspierającymi synchroniczną pracę procesów oraz komunikację między nimi.

Innym zasobem jest pamięć, której stan jest bardzo ulotny. Z tego powodu konieczne jest utrzymywanie informacji o tym, która jej część jest aktualnie wykorzystywana i przez kogo, tak by nie dopuścić na przykład do podglądania dokumentów obrabianych przez wielu użytkowników na jednym serwerze. Dzięki takim informacjom łatwiejsze staje się przydzielanie i zwalnianie pamięci oraz decydowanie, który z procesów powinien zostać załadowany do pamięci, jeśli jest ona wolna. Zarządzanie i ochronę pamięci realizuje się obecnie przez tzw. kontrolery pamięci, które wbudowane są w procesor lub stosowny chipset. Ogólna zasada ich działania polega na udostępnianiu wyłącznego dostępu do rozłącznych obszarów pamięci konkretnym procesom. W przypadku wystąpienia awarii lub niewłaściwego działania minimalizuje to prawdopodobieństwo wystąpienia zaburzeń w pamięci przynależącej do innych procesów, bo sam procesor zakańcza takie niesforne zadania

Gdyby to wszystko zebrać w całość, to tytułem podsumowania należałoby powiedzieć, że gdyby porównać komputer do człowieka, to hardware byłby jego ciałem, system operacyjny duszą, programy myślami, a dane bodźcami, na które reaguje. Nie zapominajmy jednak, że ponad tym wszystkim znajduje się jego stwórca, i po części to my nim jesteśmy.

6

Podstawowe pojęcia języków programowania

Wystarczającym narzędziem komunikacji międzyludzkiej jest język, jakim się posługujemy. Ma on jednak pewną cechę, która jest jednocześnie jego wadą i zaletą w zależności od tego, do czego miałby być zastosowany. Cechą tą jest jego bogactwo przejawiające się między innymi w wieloznaczności pewnych stwierdzeń i istnieniu idiomów. Dla człowieka nie stanowi to problemu, a wręcz zaletę pozwalającą na ekspresję wszystkich jego emocji i przemyśleń na swój indywidualny sposób, bo każdy z nas jest inny. To właśnie różni nas od komputerów, które choć złożone z różnych części, w swojej naturze wszystkie są takie same.

Z tego powodu języki komputerowe są znacznie uboższe, ale za to znacznie bardziej precyzyjne od ludzkich.

Jakkolwiek źródła internetowe w momencie powstawania tego opracowania wskazywały na istnienie na świecie około 6500 różnych języków programowania, wśród których są języki naprawdę użyteczne, takie jak C++ czy Java i tworzone z nudów i używane dla zabawy, takie jak Brainfuck, Whitespace czy Shakespeare, to jednak jest kilka pojęć, które łączą niemal wszystkie z nich. Należą do nich:

– zmienne,
– tablice,
– instrukcje bezwarunkowe,
– instrukcje warunkowe,
– pętle,
– grupowanie instrukcji,

a ich pobieżny opis znajduje się poniżej.

6.1 Zmienne

Jeżeli całe działanie programu ma się skupić na wyświetleniu pewnego komunikatu bez jakiejkolwiek interakcji z użytkownikiem programu, to zmienne są

niepotrzebne, podobnie zresztą jak programy, które robią właśnie coś takiego. Czym zatem jest zmienna? Najogólniej ujmując, jest to pewne miejsce w pamięci komputera, w którym przechowywana jest informacja pewnego typu. Może to być kod klawisza naciśniętego przed chwilą przez użytkownika, wynik pewnych obliczeń, komunikat wysłany za pośrednictwem Internetu przez zdalny komputer czy też dowolna inna informacja dająca się zakodować w postaci ciągu bitów.

Ponieważ zmienna to tak naprawdę pewien obszar w pamięci, który zaczyna się pod pewnym adresem i kończy pod jakimś innym, to jej użytkowanie bez pewnego ułatwienia byłoby cokolwiek niewygodne. Korzystanie ze zmiennych określanych wyłącznie przez ich adresy wiązałoby się z koniecznością pamiętania tychże adresów, które przecież są dosyć długimi liczbami. Mało tego, ten sam program uruchomiony dwa razy może przechowywać informacje w zmiennych znajdujących się w dwóch różnych miejscach w pamięci, co dodatkowo komplikuje ich wykorzystanie.

Wszystkie te problemy da się obejść w prosty i chytry sposób. Wystarczy nadać zmiennym pewne nazwy oraz wymusić na programie ich tłumaczenie na adresy. Jakkolwiek dąży się do tego, żeby nazwą zmiennej mógł być niemal dowolny ciąg znaków, to jednak dobrą praktyką programistyczną bez względu na język jest używanie nazw, które rozpoczynają się literą alfabetu angielskiego i zawierają wyłącznie litery tego alfabetu, cyfry od 0 do 9 i ewentualnie znaki podkreślenia.

6.1.1 Przykłady

Przykład 1:

`zmienna` - poprawna nazwa zmiennej w większości języków programowania (wyjątkiem może być PHP, w którym zmienna ta powinna nazywać się $zmienna),

`zażółć23` - nazwa zmiennej poprawna w nowszych wersjach języka Java, ale niepoprawna w ANSI C z uwagi na użycie typowo polskich liter.

Przyjęcie takiej konwencji w przypadku użytkowania języków o podobnej składni (np. ANSI C i Java) sprawia, że dysponując kodem źródłowym pewnego programu napisanym w jednym z tych języków, można go przenieść mniejszym nakładem pracy na drugi z nich.

Ze zmienną wiąże się pojęcie deklaracji. Deklaracja, to sugestia dotycząca tego, by stworzyć miejsce w pamięci pozwalające na przechowywanie danych wskazanego typu.

Przykład 2:

`int x`; w języku C oznacza deklarację zmiennej o nazwie x, która będzie mogła przechowywać liczby całkowite (int od angielskiego *integer* - całkowity),

`float pi;` w języku Java oznacza deklarację zmiennej o nazwie pi, która będzie mogła przechowywać tzw. liczby zmiennoprzecinkowe (float od angielskiego floating point).

Istnieją języki wymagające zadeklarowania zmiennych przed ich użyciem takie jak wspomniane C++ czy Java oraz takie, w których to nie jest wymagane takie jak JavaScript, PHP czy Perl. W przypadku tych drugich zmienna jest automatycznie deklarowana, czyli tworzona w momencie jej pierwszego użycia.

6.2 Tablice

W przypadku, gdy musimy przechowywać tylko pojedyncze informacje danego typu (np. programując zegarek, dość stworzyć tylko zmienne opisujące aktualną godzinę, minutę oraz sekundę) w zupełności wystarczy dysponować wiedzą na temat zmiennych i sposobów ich wykorzystania. Czasem jednak może się zdarzyć tak, że będziemy musieli przechowywać w pamięci komputera większą ilość informacji tego samego typu. Wyobraźmy sobie na przykład rozkład jazdy pewnego autobusu zapisany na stałe w kodzie źródłowym naszego programu. Nawet gdyby się okazało, że autobus ten kursuje w każdy dzień według tego samego planu 20 razy na dzień, to oznaczałoby to zadeklarowanie następujących zmiennych:

```
godzina1,godzina2,..., godzina20
minuta1,minuta2,...,minuta20
```

Nietrudno zauważyć, że w przypadku wyjątkowo dużych zbiorów danych takie podejście do sprawy robi się już nawet nie tyle niewygodne, co niepraktyczne. Dlatego też, wprowadzono możliwość jednoczesnego deklarowania wielu zmiennych tego samego typu. Struktury, pozwalające na to, zwą się tablicami.

Reguły dotyczące nazewnictwa oraz deklarowania zmiennych przenoszą się na tablice.

6.3 Instrukcje bezwarunkowe

Czasem zdarza się tak, że program, który mamy stworzyć, musi za każdym razem i bez względu na jakiekolwiek warunki wykonać tę samą serię kroków. Gdyby to odnieść do ludzkiej działalności, to można byłoby napisać, że chcąc włączyć lampę, musimy wcisnąć przełącznik, który steruje dopływem prądu do tejże lampy. I póki ktoś nie wymyśli innego sposobu włączania lamp, to bez względu na warunki będziemy to robili zawsze tak samo.

Instrukcje bezwarunkowe można oczywiście grupować, przy czym przypomina to tworzenie listy zadań, które mamy wykonywać po kolei od góry do dołu.

Przykład

Rozważmy zadanie odnalezienia pewnej informacji w Internecie. Zakładając, że wszystko działa jak należy, zadanie to można podzielić na następujące kroki:

– znajdź komputer z dostępem do Internetu,
– włącz komputer,
– włącz przeglądarkę internetową,
– wejdź na stronę jednej z wyszukiwarek,
– wpisz poszukiwane słowo lub frazę,
– spośród wyszukanych informacji wybierz Twoim zdaniem najlepszą.

Ponieważ z założenia komputer był sprawny i miał dostęp do Internetu, więc każdy z tych kroków wykonuje się bezwarunkowo. Usunięcie któregoś z nich może doprowadzić do zawalenia się całego planu. Gdyby na przykład powyższa lista wyglądała tak:

– znajdź komputer z dostępem do Internetu,
– włącz przeglądarkę internetową,
– wejdź na stronę jednej z wyszukiwarek,
– wpisz poszukiwane słowo lub frazę,
– spośród wyszukanych informacji wybierz Twoim zdaniem najlepszą.

To plan spaliłby na panewce z prostego powodu: trudno jest włączyć przeglądarkę internetową na wyłączonym komputerze. Podobnie jest w przypadku zastosowania kompletnej, ale poprzestawianej listy. Gdybyśmy na przykład zamienili miejscami wpisywanie poszukiwanej frazy i wejście na stronę jednej z wyszukiwarek, to w zależności od tego, jaka strona pokazałaby się po włączeniu przeglądarki, cały plan mógłby wziąć w łeb, prowadząc do nieprzewidywalnych wyników.

6.4 Instrukcje warunkowe

Podobnie jak w przypadku programów niekorzystających ze zmiennych, tak i programy wykorzystujące jedynie instrukcje bezwarunkowe rzadko bywają naprawdę użyteczne. Wyobraźmy sobie na przykład automatycznego ochroniarza ustawionego na bramce popularnego klubu i postawmy przed nim zadanie wpuszczania jedynie osób pełnoletnich. Takie założenie sprawia, że pojawia się pewien warunek, którego spełnienie lub nie wpływa na dalszą pracę programu ochroniarza.

O ile niektórzy nie mieliby nic przeciwko temu, żeby program ochroniarza wykorzystywał jedynie instrukcje bezwarunkowe, co sprowadzałoby się do wpuszczania na dyskotekę każdego człowieka, o tyle drugi przykład powinien wzbudzić większą wiarę w sensowność istnienia instrukcji warunkowych. Wyobraźmy sobie mianowicie bankomat, który po włożeniu do niego karty wypłaca z naszego konta pieniądze bez względu na to, czy podano poprawny czy

błędny PIN. To wydaje się już znacznie bardziej bolesną perspektywą zwłaszcza w obliczu zgubienia lub kradzieży naszej karty.

Instrukcje warunkowe z uwagi na konieczność zapewnienia interakcji z użytkownikiem lub innymi programami czy urządzeniami są bardzo ważne, a ich rolę trudno przecenić. Ich konstrukcja w językach programowania sprowadza się najczęściej do jednego z dwóch schematów:

1. Sprawdź warunek i jeżeli był prawdziwy, to wykonaj zadanie A, po czym bezwarunkowo przejdź do dalszej części programu.
2. Sprawdź warunek i jeżeli był prawdziwy, to wykonaj zadanie A, a w przeciwnym przypadku wykonaj zadanie B, po czym bezwarunkowo przejdź do dalszej części programu.

Gdyby oprzeć działanie bankomatu na pierwszej konstrukcji, to warunkiem byłaby zgodność numeru karty z numerem PIN, a działaniem A, które miałoby nastąpić po sprawdzeniu warunku, byłaby wypłata gotówki. W przypadku automatycznego ochroniarza warunkiem byłoby bycie pełnoletnim, działaniem A byłoby wpuszczenie do dyskoteki, a działaniem B wskazanie delikwentowi konieczności raptownego dorośnięcia.

6.5 Pętle

Mimo tego, że opisane powyżej pojęcia dają już sporą swobodę tworzenia sensownych programów, to jednak często zdarza się, że pewne czynności muszą być wykonywane w tej samej lub zbliżonej formie wielką ilość razy. O ile znając ilość powtórzeń pewnej danej czynności, można je po prostu powtórzyć na liście zadaną ilość razy, to takie podejście ma jednak dwie poważne wady. Po pierwsze sprawia, że dla dużej ilości powtórzeń lista rozrośnie się do niebotycznych rozmiarów, a po drugie takie podejście sprawia, że program przestanie działać w przypadku, gdy zadanie trzeba będzie powtórzyć inną ilość razy.

Te dwie niedogodności da się załatwić w bardzo prosty sposób właśnie przy pomocy użycia pętli. Choć postać notacji pętli zależy od użytego języka programowania oraz intencji autora programu, to jednak jej ogólny schemat jest zawsze taki sam i sprowadza się do wykonania następującej listy instrukcji:

– oznacz to miejsce jako `"początek_pętli"`,
– wykonaj zadanie, które ma się powtarzać,
– sprawdź, czy zadanie ma się znów powtórzyć - a jeżeli tak - to wróć na `"początek_pętli"`,
– przejdź do dalszej części programu.

Przykład:

Marek Kondrat w filmie „Dzień Świra" wykonując różne powtarzalne czynności, odliczał do 44 lub 77. Chcąc napisać dla niego listę zadań pozwalających

na mieszanie herbaty, to bez użycia pętli wyglądałaby w następujący sposób:

– włóż łyżeczkę do herbaty,
– zamieszaj, wykonując pełny obrót łyżeczką,
... 42 razy to samo,
– zamieszaj, wykonując pełny obrót łyżeczką,
– zacznij pić herbatę.

Natomiast, z wykorzystaniem pętli powyższa lista zmieni się w sposób następujący:

– `"licznik_mieszanina = 0"`,
– `"początek_mieszanina"`,
– zamieszaj, wykonując pełny obrót łyżeczką,
– zwiększ wartość zmiennej `"licznik_mieszanina"`,
– jeżeli wartość zmiennej `"licznik_mieszanina` < 44" to
– wróć na `"początek_mieszanina"`,
– zacznij pić herbatę.

6.6 Grupowanie instrukcji

Ostatnim z pojęć spotykanych w odniesieniu do prawdopodobnie wszystkich języków programowania bez względu na ich charakter jest pojęcie grupowania instrukcji. Doświadczeni programiści wyznają w swojej pracy dwie zasady, z których pierwsza to KISS (Keep It Simple Stupid - Utrzymuj To w Prostocie Głupolu), a druga DRY (Don't Repeat Yourself - Nie Powtarzaj Się). Zasady te mają wiele sensu z prostego powodu. Nieustanne powtarzanie tych samych fragmentów listy powoduje jej wydłużenie, a w przypadku, gdy fragment ten zawierał błąd, który chcemy poprawić, prowadzi do konieczności uważnego przejrzenia całej listy. Na szczęście obie niedogodności można obejść w sposób równie prosty jak ten, który pozwalał na wygodne użytkowanie zmiennych. Wystarczy po prostu wydzielić pewną listę instrukcji, która będzie się pojawiała w całym programie i zamiast wklejać ją w każdym miejscu, które tego wymaga, wklejać tam instrukcję zmuszającą komputer do odwołania się do tego wydzielonego fragmentu. Tego typu podejście, z uwagi na różny charakter języków programowania znane jest pod różnymi nazwami, z których najczęściej występujące w literaturze to **procedura, funkcja i metoda**.

Przykład:

Wyobraźmy sobie rozkład zajęć na uczelni jako pewną listę zadań, które mają zostać przez nas wykonywane cyklicznie przez jakiś czas. To będą nasze procedury, funkcje lub metody. Z drugiej strony wyobraźmy sobie nasz kalendarz, w którym jest następujący zapis

-uczelnia,
-kino,
-impreza.

Wyraz uczelnia sugeruje nam odwołanie się do bardziej szczegółowego zestawu instrukcji, który przechowywany jest w innym miejscu. To jest nasze wywołanie procedury, funkcji lub metody.

6.7 Zadania i pytania

1. Czy w przypadku konieczności składowania w programie liczb określających kursy różnych walut lepiej jest do tego zastosować zmienne czy tablice? Odpowiedź uzasadnij.
2. Przygotować słowny opis programu (listę kroków) potrzebnych do rozwiązania równania kwadratowego. *Wskazówka:* wymagane jest użycie instrukcji warunkowej.
3. Przygotować słowny opis programu (listę kroków) potrzebnych do przygotowania ulubionej kanapki.
4. Przygotować słowny opis programu pozwalającego na przygotowanie kanapek dla zadanej liczby osób.

7

Sposoby projektowania programów komputerowych

Spora część ludzi, którzy dopiero rozpoczynają swoją przygodę z programowaniem, zabiera się do tego bez żadnego planu, wychodząc z założenia, że jakoś to będzie. O ile takie podejście w zupełności sprawdza się dla prostych i krótkich programów, o tyle dłuższe programy w obliczu braku planowania potrafią zaskoczyć.

Dosyć dobrym przykładem jest tutaj program sprawdzający, czy dany element należy do posortowanej tablicy elementów tego samego typu.

Przykład:

Rozważmy następującą tablicę liczb:

Numer komórki	0	1	2	3	4	5	6	7	8
Wartość komórki	1	2	3	4	5	6	7	8	9

Załóżmy, że poszukiwaną liczbą jest 4,5. Rzut oka na powyższą tabelę upewnia nas, że ten element tam nie występuje. Komputer musi to jednak sprawdzić po swojemu, co wiąże się z pewnym planowym postępowaniem.

Pierwsze z możliwych podejść polega na przejściu przez wszystkie komórki tabeli i sprawdzeniu, czy znajdujące się w nich elementy są równe wartości 4,5, co ostatecznie doprowadza komputer do stwierdzenia, że tej wartości w tablicy nie ma.

W przypadku jednak, kiedy ta tablica jest ogromna (i dalej posortowana) takie podejście zaczyna być niewydajne, a całą sprawę da się załatwić znacznie lepiej, choć może się to wydawać mało intuicyjne. Sprawdźmy najpierw, jaki element znajduje się na końcu tej tablicy. W naszym przypadku jest to $9 > 4,5$. 9 znajduje się w tablicy pod numerem 8. Podzielmy więc 8 (czyli numer komórki zajmowanej przez 9) przez 2 i zobaczmy, jaka liczba znajduje się w tablicy pod numerem 4. Jest to $5 > 4,5$. To proste stwierdzenie w zestawieniu z faktem, że tablica jest posortowana, sprawiło, że połowę pracy mamy już za sobą, bo poszukiwana wartość z pewnością nie wystąpi wśród elementów

większych od piątki. W ten sposób okroiliśmy niejako pierwotną tablicę tak, że jej pierwsza komórka dalej znajduje się pod numerem zero, ale ostatnia ma numer cztery, a nie osiem, jak do tej pory. Podzielmy zatem numer ostatniej komórki przez dwa i zobaczmy, jaka liczba kryje się w tablicy pod tym numerem. Jest to $3 < 4,5$. Dla programu powinien być to znak, że zabrnął za daleko i zamiast zmniejszać numer ostatniej komórki tablicy, powinien zwiększyć numer jej początku. Przyjmijmy więc, że początek tablicy znajduje się teraz pod numerem dwa, a jej koniec pod numerem cztery. Sprawdźmy, jaka liczba znajduje się w tablicy pod numerem 3, czyli dokładnie pomiędzy aktualnym początkiem a końcem rozpatrywanego obszaru tablicy. Liczbą tą w naszym przykładzie jest $4 \neq 4,5$, co ostatecznie powinno utwierdzić komputer w przekonaniu tym, że poszukiwana liczba nie występuje w danej tablicy.

Opisany wyżej sposób postępowania to nic innego jak **algorytm**. W pierwszym przypadku był to algorytm przeszukiwania sekwencyjnego, a w drugim algorytm przeszukiwania binarnego. Jak nietrudno zauważyć, działanie tego programu prawie na pewno zmieniłoby się w przypadku, gdyby rozpatrywana tablica była dłuższa lub krótsza, lub gdyby zawierała inne elementy. Mało tego, przedstawiony powyżej opis jest bardzo rozwlekły i na dobrą sprawę tłumaczy tylko to, co komputer powinien zrobić w tym konkretnym przypadku. Jak więc ma poradzić sobie z opisem tego zadania w ogólnym przypadku? Na to pytanie nie ma jednoznacznej odpowiedzi. Spośród mnóstwa różnych rozwiązań i ich modyfikacji tworzonych przez poszczególnych programistów na ich własny użytek, najpopularniejsze i stosowane do dziś są dwa podejścia. Pierwsze z nich to notowanie struktury programu w sposób graficzny pod postacią tzw. schematów blokowych, drugie zaś polega na notowaniu programu przy użyciu czegoś w rodzaju namiastki języka programowania określanego pseudokodem.

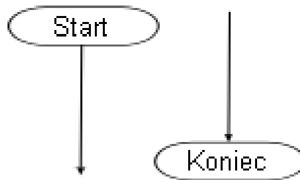
7.1 Schematy blokowe

Schemat blokowy jest sposobem graficznej prezentacji zadań, jakie stawiamy przed programem, który zamierzamy stworzyć. Rodzaj wykonywanych czynności składających się na program jest oznaczany odpowiednią ramką (blokiem) otaczającą opis tej czynności, a kolejność ich wykonania jest sugerowana przez strzałki, które łączą odpowiednie bloki.

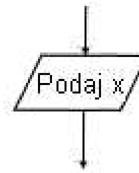
W najbardziej rozpowszechnionej notacji dotyczącej schematów blokowych wyróżnia się 9 typów bloków wskazujących na rodzaj obejmowanych przez nie instrukcji lub pojęć.

Pierwszy z nich to blok początku lub końca programu oznaczany owalem (rys. 7.1.).

Drugi to blok wejścia lub wyjścia programu, czyli miejsce sugerujące pojawienie się interakcji z użytkownikiem. Blok tego typu oznaczany jest przez równoległobok, w którym wpisuje się treść instrukcji wejścia/wyjścia, której się spodziewamy w tym miejscu. Przykład takiego bloku widać na rys. 7.2.



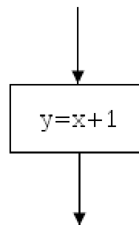
Rysunek 7.1.



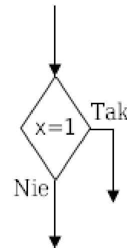
Rysunek 7.2.

Trzeci typ to blok instrukcji obliczeniowych oznaczany prostokątem. Jego pojawienie się sugeruje najczęściej konieczność wykonania pewnych operacji matematycznych, których wynik zostanie przypisany do zmiennej. Przykład wykorzystania takiego bloku widać na rys. 7.3.

Czwarty typ to blok decyzyjny, czyli graficzna interpretacja instrukcji warunkowej. Oznacza się ją rombem, w którym podaje się warunek, który trzeba sprawdzić przed przejściem do kolejnych bloków. Jest to jedyny blok, z którego wychodzą dwie strzałki, przy których zwyczajowo zapisuje się słowa takńie lub prawda/fałsz sugerujące, czy warunek zapisany w środku takiego bloku był prawdziwy, czy też nie. Przykład takiego bloku przedstawia rys. 7.4.

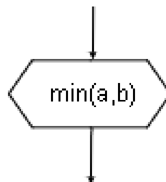


Rysunek 7.3.



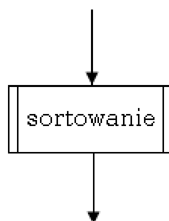
Rysunek 7.4.

Piąty, to blok wywołania podprogramu (procedury, funkcji albo metody). Oznacza się go sześciokątem, w którym wpisuje się nazwę wywoływanego podprogramu wraz z jego argumentami. Przykład na rys. 7.5. pokazuje sposób wywołania procedury wyznaczającej minimum wartości dwóch zmiennych.



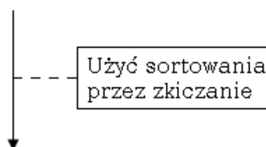
Rysunek 7.5.

Szósty to blok powszechnie stosowanego typu postępowania, co do którego istnieje pewna dowolność wyboru jego realizacji. Poniżej przedstawiono blok sugerujący konieczność wykonania sortowania elementów (rys. 7.6.).



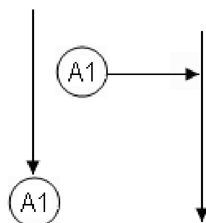
Rysunek 7.6.

Siódmy to blok komentarza. Dzięki niemu możemy w formalny i precyzyjny sposób dodawać do schematu notatki sugerujące na przykład sposób realizacji konkretnych zadań czy cel ich wykonania, w przypadku gdy jest on na pierwszy rzut oka niejasny.



Rysunek 7.7.

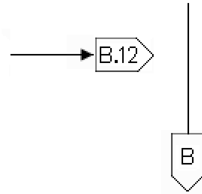
Ósmy to blok łącznika między częściami programu znajdującymi się w różnych miejscach schematu blokowego na tej samej stronie. Miejsca, które należy połączyć, oznaczane są wtedy jednakowymi symbolami. Oto przykład takiego połączenia (rys. 7.8):



Rysunek 7.8.

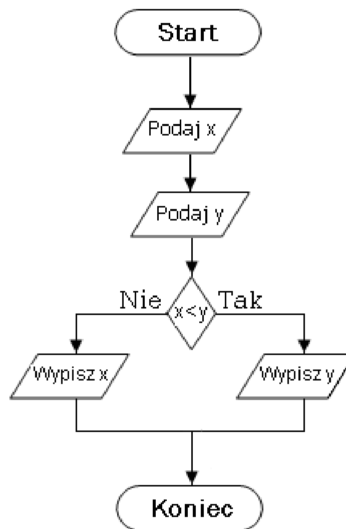
Dziewiąty, ostatni to blok łącznika między częściami programu znajdującymi się na różnych stronach w dokumentacji. Miejsca, które należy połączyć,

podobnie jak w poprzednim przypadku, oznacza się tymi samymi symbolami, a dodatkowo dla zwiększenia czytelności na początku łącznika notuje się zwykle numer strony, na której znajduje się zakończenie łącznika.



Rysunek 7.9.

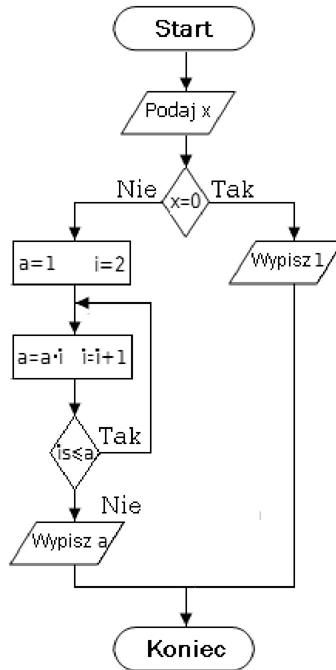
Rozważmy teraz przykład prostego programu, który poprosi o wprowadzenie dwóch liczb i wybierze większą z nich. Jak będzie wyglądał schemat blokowy programu realizującego to zadanie? Poniżej znajduje się rysunek 7.10., który go przedstawia.



Rysunek 7.10.

Na schemacie rysunku 7.10. znajdują się instrukcje bezwarunkowe „Podaj x", „Podaj y", „Wypisz x" oraz „Wypisz y". Program zakłada też istnienie dwóch zmiennych x i y. Oprócz tego występuje w nim pojedyncza instrukcja warunkowa sprawdzająca, czy wartość zmiennej x jest mniejsza od wartości zmiennej y. W tym miejscu, w zależności od sytuacji program rozgałęzia się, co sprawia, że jego dalsza praca zaczyna przebiegać w rożny sposób.

Rozważmy jeszcze jeden prosty schemat blokowy programu wyznaczającego silnię liczby naturalnej.



Rysunek 7.11.

Przykład ten prezentuje sposób zapisywania pętli. Jak widać (rys. 7.11.), program zawiera instrukcję warunkową, która sprawdza, czy wartość zmiennej i jest mniejsza lub równa wartości zmiennej a. Jeżeli tak, to program zawraca i ponownie wykonuje instrukcje mnożenia liczby a przez liczbę i oraz zwiększania wartości zmiennej i o jeden. W momencie, w którym warunek stanie się nieprawdziwy, program przejdzie do wypisania wartości zmiennej a, po czym zakończy swoje działanie.

7.2 Pseudokod

Jakkolwiek nietrudno jest sobie wyobrazić program, który potrafiłby interpretować schematy blokowe, to jednak w przypadku bardziej rozbudowanych programów mogłoby się to w końcu okazać niewygodnym rozwiązaniem. Między innymi z tego powodu, drugim równie popularnym sposobem notowania struktury programu jest tzw. pseudokod, czyli na wpół sformalizowany słowny opis działań, które ma wykonać projektowany program. Ponieważ opis ten, nie jest do końca formalny i jest przeznaczony do interpretacji przez ludzki umysł,

a nie komputerowy procesor, to właściwie każdy może stworzyć swój rodzaj pseudokodu. W przypadku jednak, gdy chcemy w łatwy sposób przejść od pseudokodu do realnego programu, dobrze jest przynajmniej w pewnej części polegać na konwencjach językowych zapożyczonych z języka, którym ostatecznie będziemy się posługiwać.

Na początku lat siedemdziesiątych XX wieku stworzono język C. Prostota i finezja jego składni i gramatyki okazała się na tyle dobrym pomysłem, że wielu twórców nowocześniejszych języków programowania takich jak C++, Java czy PHP przejęło te koncepcje i zastosowało je w swoich rozwiązaniach. Z tego powodu prawdziwe różnice między językami widoczne są dopiero po dogłębnej analizie ich dokumentacji, podczas gdy często się zdarza, że fragmenty kodów źródłowych napisanych w nich programów trudno jest od siebie odróżnić. Skoro takie podobieństwa występują w rzeczywistych językach, to czemu z tego nie skorzystać, sprawiając, że nasz pseudokod się do nich upodobni i będzie łatwy do przetworzenia w prawdziwy, działający w komputerze program?

Zajmijmy się zatem wprowadzeniem pseudokodu, który przypomina nieco język C.

Bez względu na to, co program właściwie robi, to i tak coś liczy. Dlatego rozpocznijmy od wprowadzenia umowy dotyczącej symboli określających kolejne symbole działań matematycznych. Dodawanie będzie oznaczane znakiem '+', odejmowanie będzie oznaczane przez '-', mnożenie przez '*', a dzielenie przez '/'. Podobnie też równość dwóch wyrażeń i przypisanie wartości wyrażenia do zmiennej będą oznaczane znakiem '='. Nierówność będzie oznaczana znakiem '!=', nierówności ostre znakami '<' i '>', zaś nierówności nieostre (mniejsze lub równe i większe lub równe) odpowiednio znakami '<=' oraz '>='.

Dodatkowo będziemy potrzebowali instrukcji pozwalających na współpracę z użytkownikiem. Ustalmy zatem, że instrukcja pobierająca wartość zmiennej x będzie oznaczana przez `input(x)` (od angielskiego *input* - wprowadź), a instrukcja wyświetlająca na ekranie wartość tej zmiennej będzie oznaczana przez `print(x)` (od angielskiego *print* - drukuj).

Aby uniknąć niedomówień, przyjmiemy, że kolejne instrukcje bezwarunkowe będą oddzielone od siebie średnikami. Dla przykładu aby pomnożyć zmienną a przez zmienną b i zapisać wynik w zmiennej c, której wartość potem wyświetlimy na ekranie, moglibyśmy użyć następującego pseudokodu:

```
c=a*b;
print(c);
```

Na oznaczenie instrukcji warunkowej przyjmiemy następujące konstrukcje:

```
if(warunek)
{
 ciąg instrukcji wykonywanych, gdy warunek jest prawdziwy
}

if(warunek)
```

```
{
 ciąg instrukcji wykonywanych, gdy warunek jest prawdziwy
}
else
{
 ciąg instrukcji wykonywanych, gdy warunek NIE jest prawdziwy
}
```

Słowa *if* oraz *else* oznaczają w języku angielskim odpowiednio jeżeli i w przeciwnym wypadku. Oto przykład pseudokodu wyznaczającego minimum dwóch liczb a i b:

```
if(a<b){wynik = a}
else{wynik = b}
```

W przypadku schematów blokowych każda pętla notowana była praktycznie w ten sam sposób - użyciem strzałki wiodącej do miejsca, od którego mamy rozpocząć powtarzanie pewnej części programu. W przypadku języka C wyróżniono 4 rodzaje pętli, których odpowiedniki zostaną poniżej zaadaptowane na potrzeby naszego pseudokodu.

Pierwsza z pętli to pętla `goto` (od angielskiego *go to* - idź do). Jej użycie w wielu językach programowania uważa się za przejaw wyjątkowo złego smaku i zaleca się stosowanie wyłącznie w sytuacjach, w których zastosowanie pętli jest konieczne, a użycie innego jej typu byłoby bardzo trudne. Są też i takie języki, np. Java, w których użycie tego typu pętli jest w ogóle niemożliwe, co wymusza też na programiście większą kulturę pisania. Aby stworzyć pętlę `goto`, należy rozpocząć od zdefiniowania etykiety, czyli wyróżnionego pewną unikatową nazwą miejsca w programie, do którego program ma powrócić, gdy natknie się na skojarzoną z nią instrukcję `goto`. Definiowanie etykiety zwykle załatwia się przy pomocy podania jej nazwy zakończonej dwukropkiem.

7.2.1 Przykłady

Przykład 1:

```
Etykieta:
TuSkocz:
```

Po etykiecie może się pojawić dowolny ciąg instrukcji warunkowych i bezwarunkowych oraz pętli, których działanie chcemy powtarzać, a na końcu tego ciągu powinna się znaleźć instrukcja `goto` wraz z nazwą etykiety, do której chcemy wrócić.

Przykład 2:

```
etykietka:
x=x+1;
goto etykietka;
```

Działanie powyższego pseudokodu będzie się sprowadzało do pobierania starej wartości zmiennej x, zwiększania jej o jeden $(x+1)$ i zapisywania nowej wartości w zmiennej $x(x = x+1)$. Ponieważ pętla nie ma zdefiniowanego żadnego warunku, który by ją kończył, więc jej działanie będzie polegało na ustawicznym zwiększaniu wartości zmiennej x.

Drugim rodzajem pętli występującym w języku C i podobnych jest pętla sprawdzająca warunek trwania na swoim początku. Notuje się ją w następujący sposób:

```
while(warunek)
 {
  ciąg instrukcji, które mają wykonywać się cyklicznie
 }
```

Słowo *while* w języku angielskim oznacza dopóki. Ponieważ warunek trwania pętli sprawdzany jest na początku pętli, to może się zdarzyć tak, że instrukcje zawarte w pętli nie wykonają się ani razu, bo pętla będzie wykonywana tak długo, jak długo warunek opisany w nawiasie będzie prawdziwy.

Przykład 3:

```
a=2;
while(a<10)
  {
   print(a);
   a=a+2;
  }
```

Warunek trwania pętli polega na sprawdzeniu, czy wartość zmiennej a jest mniejsza od 10. Ponieważ na początku programu $a = 2$, więc instrukcje zawarte w pętli wykonają się, a efektem działania programu będzie wypisanie na ekranie liczb 2, 4, 6 i 8.

Trzecim rodzajem pętli jest pętla, która warunek swojego trwania sprawdza na końcu, po wykonaniu wszystkich instrukcji przewidzianych do wielokrotnego wykonywania. Notuje się ją w następujący sposób:

```
do
  {
   ciąg instrukcji przewidzianych do powtarzania
  }
while(warunek)
```

Przykład 4:

```
a=2;
do
```

```
  {
   print(a);
   a=a+2;
  }
while(a<10)
```

Nietrudno zauważyć, że powyższy program jest bardzo podobny do poprzedniego. Różni je tylko miejsce sprawdzenia warunku trwania pętli. W konsekwencji efektem działania powyższego programu będzie wypisanie liczb 2, 4, 6, 8 i 10.

Ostatnim rodzajem pętli jest pętla, której sama konstrukcja wymusza nadawanie warunku początkowego i warunku trwania pętli oraz sposobu, w jaki ma się zmieniać zmienna numerująca powtórzenia pętli, czyli tzw. indeks. Pętlę tę notuje się w następujący sposób:

```
for(warunek początkowy ; warunek trwania ; krok pętli)
{
 Instrukcje, które mają zostać powtórzone
}
```

Przykład 5:

```
for(i=0;i<10;i=i+1)
{
 print(i);
}
```

Indeksem tej pętli jest zmienna i. Warunkiem początkowym dla tej pętli jest wyrażenie $i = 0$. Warunkiem trwania tej pętli jest wyrażenie $i < 10$, które sprawia, że będzie się ona wykonywała tak długo, jak długo wartość zmiennej i będzie mniejsza od 10. Krok pętli opisany jest wyrażeniem $i = i + 1$, co znaczy, że po każdym przebiegu pętli jej indeks zwiększy się o jeden. Efektem działania programu opisanego powyższym kodem będzie wyświetlenie liczb 0, 1, 2, 3, 4, 5, 6, 7, 8 i 9.

Należy tu wspomnieć, że zmiennych zwykle nie deklaruje się ich w pseudokodzie, tylko po prostu, umieszcza ich nazwy w miejscu, w którym powinny być użyte. Podobnie jest z tablicami. Jedyną konwencją, z którą warto się zaznajomić, jest sposób odczytywania oraz zapisywania poszczególnych komórek tablicy. Zwykle oznacza się to przy użyciu nawiasów prostokątnych, wewnątrz których wpisuje się wyrażenie opisujące numer komórki w tabeli, którą chcemy odczytać lub zapisać.

Przykład 6:

Chcąc odczytać piątą komórkę tablicy o nazwie `tab`, należy użyć następującego pseudokodu:

```
tab[5];
```

Chcąc przypisać komórce o numerze opisanym przez wyrażenie $i + 1$ znajdującej się w tablicy `tab` wartość 17, należy użyć następującego pseudokodu:

```
tab[i+1]=17;
```

7.3 Zadania i pytania

1. Przygotować schemat blokowy i pseudokod programu pobierającego wartości dziesięciu liczb i umieszczającego je w stosownej tablicy.
2. Przygotować schemat blokowy i pseudokod programu pobierającego trzy wartości liczbowe a, b i c i wyznaczającego rozwiązanie równanie kwadratowego o takich współczynnikach.
3. Przygotować pseudokod programu wyznaczającego silnię liczby naturalnej, dbając o to, by wejściowe wartości ujemne były wykluczane na początku programu jako niepoprawne.

8

Złożoność obliczeniowa programów komputerowych

Działanie każdego programu komputerowego musi zajmować jakiś czas i trudno z tym dyskutować na obecnym poziomie wiedzy. Za to zastanawianie się nad tym, dlaczego to zajmuje aż tyle czasu, to już całkiem inna historia i jako takie jest całkowicie uzasadnione. Na początku komputery miały bardzo wolne procesory oraz śmieszne jak na dzisiejsze standardy ilości pamięci operacyjnej. W konsekwencji pisanie programów, które miały robić cokolwiek pożytecznego, urastało do miana wirtuozerii i wiązało się z koniecznością znajomości matematycznych trików i sztuczek pozwalających na obejście tych ograniczeń. Z czasem procesory robiły się coraz szybsze, pamięci coraz większe, zaś programiści coraz bardziej niechlujni. W dzisiejszych czasach, zwłaszcza wśród producentów gier komputerowych, panuje przekonanie, że jeżeli program komuś nie działa, to jest to wyłącznie jego wina, bo już dawno powinien kupić sobie komputer o potężniejszej mocy obliczeniowej. Celem tego rozdziału jest przedstawienie podstaw wiedzy na temat złożoności obliczeniowej programów komputerowych i obnażenie fałszywości przekonania, że szybszy komputer załatwi to, z czym nie poradził sobie leniwy programista.

W poprzednich rozdziałach przedstawiono podstawowe pojęcia stosowane w programowaniu komputerów. Była tam mowa o tym, że program składa się z instrukcji. Nietrudno jest zrozumieć, że każda z tym instrukcji trwa przez określony czas, więc im więcej będzie ich w programie, tym gorzej dla niecierpliwych użytkowników. To intuicyjne stwierdzenie da się jednak ująć w ramy rachunków, które w miarę precyzyjnie wskazują na to, który z dwóch różnych programów robiących to samo będzie szybszy. Konkretne wykonanie się każdego programu da się rozłożyć na ciąg niepodzielnych instrukcji języka maszynowego. Podobnie jak cząstki elementarne, które są malutkie, ale różnią się rozmiarem, tak samo jest z tymi instrukcjami. Chociaż czas wykonania każdej z nich jest zaniedbywalnie mały, to jednak konieczność wykonania miliardów takich operacji sprawia, że wykonanie całego programu zajmuje już długi czas. Dlatego programując na takim poziomie, warto jest stosować takie niepodzielne instrukcje, które robią to samo, a zajmują mniej czasu. Dosyć

dobrym przykładem może tu być zastępowanie mnożenia lub dzielenia liczb przez 2 przez odpowiednie przesunięcie bitowe.

Z powyższego wynika, że aby naprawdę precyzyjnie opisywać czas wykonania się programu, należałoby znać dokładny czas wykonania się każdej niepodzielnej instrukcji, a potem je sumować. W praktycznym przypadku jest to jednak zadanie na tyle trudne, że często upraszcza się je, wprowadzając umowę o tym, że każda niepodzielna instrukcja trwa dokładnie tyle samo czasu co inna niepodzielna instrukcja.

8.1 Przykłady

Przykład 1:

Instrukcja

```
x=1+2*3/4;
```

będzie zajmowała 4 jednostki czasu, bo wiąże się z wykonaniem następujących instrukcji niepodzielnych (konkretnie w tej kolejności):

– mnożenie 2*3,
– dzielenie wyniku poprzedniej operacji przez 4,
– dodawanie wyniku poprzedniej operacji do jedynki,
– przypisywanie wyliczonej wartości do zmiennej x.

W przypadku programów zawierających wyłącznie instrukcje bezwarunkowe oszacowanie czasu ich wykonania polega więc właściwie na dokładnym policzeniu wszystkich działań, które w nim występują. Co jednak zrobić w przypadku wystąpienia instrukcji warunkowych? Przetworzenie instrukcji warunkowej wymaga sprawdzenia prawdziwości zawartego w niej warunku i przejściu do odpowiedniej części programu związanej z jego prawdziwością lub fałszem. Jeżeli chcemy sprawdzić, jak długo będzie trwało działanie naszego programu dla konkretnych danych, to możemy dla niego sprawdzić prawdziwość warunku, podliczając liczbę niepodzielnych instrukcji, które trzeba wykonać, by to zrobić, po czym podliczyć liczbę niepodzielnych instrukcji związanych z prawdziwością lub fałszem warunku.

Przykład 2:

Rozważmy następujący pseudokod:

```
a=2;
if(a<3)
 {
   a=a+1;
 }
else
```

```
 {
   a=a*2;
 }
```

Ponieważ na początku wartość zmiennej a jest ustalana na 2, więc warunek $a < 3$ jest prawdziwy. W związku z tym część programu występująca po słowie else po prostu się nie wykona, a działanie całego programu będzie właściwie tożsame z wykonaniem następujących instrukcji:

```
a=2;
a<3
a=a+1;
```

które trwają 4 jednostki czasu (jedno przypisanie w pierwszej linii, jedno porównanie w drugiej linii, jedno dodawanie i jedno przypisanie w trzeciej linii).

Jednak co zrobić, kiedy chcemy oszacować długość działania programu w przypadku, gdy prawdziwość warunku nie jest znana od początku? Jedno z możliwych podejść zakłada wykorzystanie w tym celu prawdopodobieństwa prawdziwości warunku.

Przykład 3:

Rozważmy następujący pseudokod pozwalający na wyznaczenie wartości funkcji $|x| + 1$:

```
if(x<0)
{
  a=-x+1;
}
else
{
  a=x+1;
}
```

Wartość zmiennej x jest w tym przypadku nieznana. Każda liczba rzeczywista może zostać zaznaczona na osi liczbowej, której połowę zajmują liczby ujemne, a drugą połowę liczby dodatnie i zero. Zatem można przyjąć, że prawdopodobieństwo wystąpienia liczby ujemnej to 0.5, a prawdopodobieństwo wystąpienia liczby dodatniej lub zera to 0.5. Wynika z tego, że warunek $x < 0$ będzie musiał być sprawdzony za każdym razem, natomiast instrukcje $a = -x + 1$ i $a = x + 1$ będą przetwarzane jedynie z zadanymi prawdopodobieństwami, co ostatecznie sprawia, że czas działania powyższego pseudokodu możemy oszacować przy pomocy wyrażenia $1 + 0.5{\cdot}3 + 0.5{\cdot}2 = 1 + 0.5{\cdot}5 = 3.5$.

W przypadku pętli sprawa polega na podliczaniu długości trwania instrukcji znajdujących się w jej wnętrzu tak długo, jak długo spełnione są warunki jej trwania. W prostszych przypadkach polega to na rozwinięciu zapisu pętli do postaci kolejnych instrukcji wchodzących w jej skład tyle razy, ile to jest potrzebne.

Przykład 4:

Rozważmy następujący pseudokod:

```
a=0;
while(a<3)
{
  a=a+1;
}
```

Działanie powyższego pseudokodu jest równoważne z działaniem następującego ciągu instrukcji:

```
a=0;
a<3;      (prawda)
a=a+1;    (a=1)
a<3;      (prawda)
a=a+1;    (a=2)
a<3;      (prawda)
a=a+1;    (a=3)
a<3;      (nieprawda, koniec pętli)
```

Łącznie daje to więc 11 instrukcji zajmujących 11 jednostek czasu.

W przypadku pętli typu `goto` i `do-while` rachunki wyglądają bardzo podobnie. Odrobinę inaczej sprawa wygląda w przypadku pętli for, którą wygodnie jest doprowadzić do równoważnej jej pętli while.

Przykład 5:

Pseudokod

```
a=0;
for(i=0;i<10;i=i+1)
{
  a=a+i;
}
```

może zostać zapisany w postaci

```
a=0;
i=0;
while(i<10)
{
  i=a+i;
  i=i+1;
}
```

Tak spreparowany pseudokod może już zostać potraktowany analogicznie jak w poprzednim przykładzie.

Dotychczasowe przykłady powinny doprowadzić do dwóch spostrzeżeń. Po pierwsze, im więcej w programie jest pętli, tym większa jest szansa na to, że jego wykonanie będzie zajmowało dużo czasu. Po drugie, z im większymi danymi program będzie sobie musiał poradzić, tym więcej czasu może mu to zająć. Tak jedno jak i drugie stwierdzenie jest oczywiście jedynie pewną intuicją. W pierwszym przypadku łatwo jest wskazać przykłady programów zawierających mnóstwo pętli, które tak naprawdę nigdy się nie wykonają. W drugim zaś, wystarczy zdać sobie sprawę z tego, że testowanie, czy liczba o wielu tysiącach cyfr jest pierwsza, może się zakończyć wraz ze spostrzeżeniem, że jej ostatnia cyfra jest parzysta, podczas gdy sprawdzenie, że znacznie mniejsza liczba jest pierwsza, będzie wymagało sprawdzenia, czy dzieli się ona przez każdą liczbę naturalną większą jeden i mniejszą niż pierwiastek z tej liczby.

W związku z tym, że powyższa intuicja podaje niezbyt precyzyjne kryteria oceny prędkości programów, wprowadzono pojęcie funkcji kosztu, czyli wzoru łączącego rozmiar danych, które mamy przetworzyć z czasem działania realizującego to programu. W rozpatrywanych do tej pory przykładach czas działania programu właściwie nie zależał od wielkości danych, które były przez nie przetwarzane. Zajmijmy się teraz przypadkiem, w którym taka zależność występuje.

Przykład 6:

```
a=0;
i=0;
while(i<=n)
{
  a=a+i;
  i=i+1;
}
```

W powyższym przykładzie wartość zmiennej n jest nieokreślona. Nie można więc rozwinąć tej pętli do równoważnej postaci niezapętlonej listy instrukcji, bo nie wiadomo, ile będzie dokładnie powtórzeń instrukcji zawartych w pętli. Można za to zauważyć, że za każdym razem powtórzy się ten sam zestaw instrukcji.

```
i<=n (sprawdzanie warunku trwania pętli)
a=a+i
i=i+1
```

Powyższy ciąg instrukcji zajmie 5 jednostek czasu i powtórzy się dla $i = 0, 1, \ldots, n$, czyli $n + 1$ razy. Wobec tego właściwe działanie pętli zajmie $5(n + 1)$ jednostek czasu. Oprócz tego na początku mieliśmy instrukcje $a = 0$ i $i = 0$, które zajmują łącznie 2 jednostki czasu, a po wszystkim trzeba będzie jeszcze raz sprawdzić warunek, który dla $i = n + 1$ będzie już nieprawdziwy. Wiąże się to z dodatkową instrukcją $i <= n$ kosztującą jedną jednostkę czasu. Podsumowując, otrzymujemy wzór podający oszacowanie długości trwania prog-

ramu opisanego danym pseudokodem w postaci:

$$K(n) = 5(n+1) + 3$$

który stanowi funkcję kosztu dla danego pseudokodu.

To oczywiste, że postać funkcji kosztu będzie zależała od programu, który ta funkcja opisuje. Skoro zaś dysponujemy nieskończoną ilością możliwości zestawiania ze sobą wszelkiego rodzaju instrukcji, to należy się spodziewać, że również i funkcje kosztu będą mogły przyjmować nieskończoną ilość postaci. Aby w jakiś sposób skatalogować tę obfitość, stworzono pojęcia notacji asymptotycznych i klas złożoności obliczeniowej.

Zajmijmy się najpierw notacjami asymptotycznymi. Bywa tak, że interesuje nas dolne ograniczenie dla czasu przetwarzania jakichś danych. Naturalnie znacznie bardziej interesujące jest to, jak długo będą przetwarzane duże ilości danych, a nie wyjątkowo mała próbka. Dlatego też wprowadzono następującą definicję:

Definicja:
Mówimy, że funkcja $f(x) = \Omega(g(x))$, jeżeli istnieją takie stałe $c > 0$ i z, że dla każdego $x > z$ zachodzi nierówność

$$c \cdot g(x) < f(x).$$

Jeżeli natomiast interesuje nas jedynie górne ograniczenie dla czasu przetwarzania dużych danych, to znacznie lepsze będzie użycie poniższej definicji.

Definicja:
Mówimy, że funkcja $f(x) = O(g(x))$, jeżeli istnieją takie stałe $c > 0$ i z, że dla każdego $x > z$ zachodzi nierówność

$$c \cdot g(x) > f(x).$$

Ostatnia i najbardziej precyzyjna notacja opisuje sytuacje, w których możemy uzyskać ograniczenie tego samego typu zarówno z góry, jak i z dołu.

Definicja:
Mówimy, że funkcja $f(x) = \Theta(g(x))$, jeżeli istnieją takie stałe $0 < c < d$ i z, że dla każdego $x > z$ zachodzi nierówność

$$c \cdot g(x) < f(x) < d \cdot g(x).$$

Przykład 7:

Funkcja $f(x) = logx = O(x)$, bo istnieją stałe $c = 1$ i $z = 0$, takie, że dla każdego $x > z$ zachodzi nierówność $1 \cdot x > logx$.

Funkcja $f(x) = |sinx| + 1 = \Omega(1)$, bo istnieją stałe $c = 0,5$ i $z = 0$, takie, że dla każdego $x > z$ zachodzi nierówność $0,5 \cdot 1 < |sinx| + 1$. Funkcja

$f(x) = 4x^2 = \Theta(x^2)$, bo istnieją takie stałe $c = 3$ i $d = 5$ oraz $z = 0$, że dla każdego $x > z$ zachodzi nierówność $3x^2 < 4x^2 < 5x^2$.

Z pojęciem notacji Θ łączy się pojęcie klas złożoności obliczeniowej. Jeżeli wyznaczymy funkcję kosztu dla naszego programu i okaże się, że możemy ją wyrazić przy pomocy notacji Θ, to możemy dla niej wyznaczyć również klasę złożoności obliczeniowej. Najczęściej występujące klasy złożoności to:

- stałej, gdy funkcja kosztu $K(n) = \Theta(1)$,
- logarytmicznej, gdy funkcja kosztu $K(n) = \Theta(\ln n)$,
- liniowej, gdy funkcja kosztu $K(n) = \Theta(n)$,
- kwadratowej, gdy funkcja kosztu $K(n) = \Theta(n^2)$,
- wielomianowej, gdy funkcja kosztu $K(n) = \Theta(W(n))$ i $W(n)$ jest wielomianem zmiennej n,
- wykładniczej, gdy funkcja kosztu $K(n) = \Theta(f(n))$ i $f(n)$ jest funkcją wykładniczą zmiennej n.

Często, jeśli funkcja kosztu programu należy do pewnej klasy złożoności obliczeniowej, mówi się w skrócie, że program ma złożoność tej klasy. Dla przykładu jeżeli funkcja kosztu należy do klasy złożoności stałej, mówi się, że program ma stałą złożoność obliczeniową.

Przykład 8:

Wracając do funkcji z poprzedniego przykładu, należałoby powiedzieć, że $\log x$ należy do klasy złożoności logarytmicznej, a $4x^2$ do klasy złożoności kwadratowej. Najbardziej zadziwiający jest przypadek funkcji $|sinx|+1$, która należy do klasy złożoności stałej, choć sama stała nie jest. Wynika to z tego, że wartość tej funkcji jest ograniczona z dołu przez 1, a z góry przez 2, czyli zgodnie z definicją $|sinx| + 1 = \Theta(1)$.

Za każdym razem, gdy w książce pojawia się jakiś wzór matematyczny, powinien on mieć jakieś uzasadnienie. I to uzasadnienie istnieje. Wyobraźmy sobie teraz, że mamy kilka programów realizujących to samo zadanie na różne sposoby oraz załóżmy, że ich funkcje kosztu należą do różnych klas złożoności obliczeniowej. Który z nich wybrać, by nasz program działał jak najszybciej? Jeżeli funkcje kosztu tych programów należą do klas wymienionych w powyższym zestawieniu, to najlepiej wybrać program o złożoności stałej, jeżeli go nie ma, to następny w kolejności byłby program o złożoności logarytmicznej, kolejny to ten o złożoności liniowej. Gorszy od programu o złożoności liniowej jest ten, który ma złożoność kwadratową, jeszcze gorszy jest program o złożoności wielomianowej w przypadku, gdy stopień tego wielomianu jest wyższy niż 2, a już najgorszy i wybierany w ostatniej kolejności powinien być program o złożoności wykładniczej.

Wróćmy teraz, do upowszechnionego podejrzenia, że jeżeli program działa zbyt wolno, to na szybszym komputerze w wyraźmy sposób zyskamy na wydajności. Wyobraźmy sobie, że mamy pewien program o złożoności logarytmicznej, przy czym funkcja kosztu dla tego programu to $K(n) = \lg n$. Załóżmy

dodatkowo, że dane rozmiaru n są przez niego przetwarzane w czasie t. Zatem $\lg n = t$, a z tego z kolei wynika, że na ośmiokrotnie szybszym komputerze te same dane zostaną przetworzone w czasie $0,125t$, czyli

$$\lg n = 0,125t$$

$$8 \lg n = t$$

$$\lg n^8 = t.$$

Jeśli zatem nasz program w czasie t na starszym komputerze przetwarzał dane o rozmiarze 10, to na nowszym i szybszym poradzi już sobie z danymi o rozmiarze 100000000.

Jeżeli teraz założymy, że rozpatrywany program ma złożoność liniową, a jego funkcja kosztu $K(n) = n$, to można napisać, że $n = t$ i na ośmiokrotnie szybszym komputerze uzyskamy równość $n = 0,125t$, z czego wynika, że $8n = t$, a to prowadzi do wniosku, że na szybszym komputerze w tym samym czasie zostanie przetworzona ośmiokrotnie większa ilość danych, czyli na przykład 80 zamiast 10.

A jak to będzie w przypadku programu o złożoności wykładniczej? Jeżeli funkcja kosztu będzie $K(n) = 2^n$, możemy więc napisać, że $2^n = t$, a na nowszym komputerze zamieni się to w równość $2^n = 0.125t$, z której wynika, że $8 \cdot 2^n = t$ i $2^{n+3} = t$. Z ostatniej równości wynika, że na ośmiokrotnie szybszym komputerze w tym samym czasie uda nam się przetworzyć dane tylko o trzy jednostki większe niż w przypadku starszego komputera!

To oczywiście wydumany przykład, który pokazuje tylko różnicę między programem optymalnym i napisanym według zasady „jakoś to będzie”. Nie zawsze, a raczej stosunkowo rzadko jest tak, że to samo zadanie da się zrealizować bez nadmiernego udziwniania przy pomocy programów o tak różnych złożonościach obliczeniowych. Niemniej liczby nie kłamią, a w tym przypadku mówią jedno: nie warto polegać na pisanych byle jak programach, inwestując w coraz nowsze komputery.

9

Klasyfikacja języków programowania

Jedną z cech różniących ludzi od komputerów jest to, że ludzie dysponują inteligencją, a komputery nie. To co określane jest mianem sztucznej inteligencji jak do tej pory nie jest nawet marną namiastką ludzkiego intelektu i sprowadza się do nauczenia maszyn wykonywania rzeczy, które dla większości ludzi są oczywiste. Oprócz tego ludzie potrafią interpretować informacje w sposób zależny od swoich emocji i interakcji, w jakie wchodzą aktualnie z otoczeniem. Często pozornie bardzo podobne informacje mogą być przetworzone przez tego samego człowieka na różne sposoby. Przykładem może być interpretacja prostego tekstu: „jesteś głupi". W przypadku, gdy taki tekst został wysłany przez naszego szefa, z którym znamy się jedynie zawodowo, nie wróży on niczego dobrego. Za to, kiedy ten sam tekst wygląda tak: „jesteś głupi :*", a wysyła go dziewczyna, której właśnie usiłujemy zaimponować, to może to znaczyć, że wbrew pozorom zbliżamy się do zamierzonego celu.

Przy obecnym poziomie wiedzy skonstruowanie programu, który umiałby w pełni poprawnie zinterpretować już nawet nie wymawiane w jego kierunku, ale podawane mu tekstowo instrukcje zapisane w języku naturalnym, jest bardzo trudne. Oprócz tego nawet gdyby to się udało, mogłoby się okazać, że takie rozwiązanie wcale nas nie zadowala, gdyż ludzki język bywa na tyle niejednoznaczny, że przetworzenie go według prawideł logiki może prowadzić do przedziwnych wyników. Znany jest dowcip o żonie programisty, która zwraca się do niego tymi słowami: „idź do sklepu i kup jajka, jak będą parówki, to weź cztery", po czym programista idzie do sklepu i pyta: „są parówki?" i gdy sprzedawca mówi, że tak, programista dodaje: „to poproszę cztery jajka". To jest oczywiście mocno przerysowana sytuacja, która jednak pokazuje, jak mogłyby reagować komputery i dlaczego warto tworzyć dla nich sztuczne języki mimo tego, że i naturalnych na Ziemi jest obfitość.

Intuicja podpowiada, że język komputerowy powinien być jednocześnie elastyczny i nie dopuszczać do powstawania niejednoznaczności. Nie zawsze da się to osiągnąć nawet w tak logicznych konstrukcjach jak język C. Jedyną radą na obsłużenie takich sytuacji jest najczęściej zakończenie przetwarzania programu w sposób odpowiedni dla danego języka.

9.1 Przykłady

Przykład 1:

W języku C istnieją dwa operatory oznaczane znakiem „++”. Pierwszy z nich zastosowany na zmiennej x wyglądałby tak: „++x” i znaczyłby mniej więcej tyle: przed wykonaniem dalszych obliczeń zwiększ wartość zmiennej x o jeden. Drugi wygląda tak: „x++” i oznacza konieczność zwiększenia zmiennej x o jeden, ale po zakończeniu obliczeń. Oprócz tego w języku tym występuje jeszcze symbol zwykłego dodawania, który w zapisie wygląda na przykład tak: „$x+y$”. Wiedząc już to wszystko, zastanówmy się nad tym, w jaki sposób należałoby zrozumieć instrukcję „x+++y”? Czy oznacza ona, że przed wykonaniem obliczeń zwiększamy wartość zmiennej y o jeden i dodajemy do zmiennej x? Czy też chodzi o to, aby dodać do siebie zmienne x i y, a po wszystkim zwiększyć wartość zmiennej x o jeden?

Powyższy przykład miał za zadanie wskazać konieczność wprowadzenia pojęcia składni języka programowania. Składnia to nic innego jak zestaw reguł opisujących rodzaje dostępnych w danym języku symboli oraz sposoby łączenia tych symboli w większe i bardziej złożone struktury. Jedna z tych reguł dla języka C wyklucza możliwość przetworzenia programu zawierającego instrukcję „x+++y”. Aby mógł on zostać przetworzony, należałoby użyć jednego z dwóch zapisów „(x++)+y” lub „x+(++y)”, w których nawiasy sugerują dokładną kolejność wykonywania działań, a co za tym idzie również sens naszych zamierzeń.

Warto tutaj zauważyć jeszcze jedną rzecz. To, że program jest poprawny pod względem składniowym, nie znaczy jeszcze, że ma jakikolwiek sens. Występuje tu pełna analogia z językami naturalnymi. Dla przykładu zdanie: „Węchacz iżbnął zapawiony” jest poprawne gramatycznie, gdyż występują w nim słowa, których konstrukcja przypomina przymiotniki, rzeczowniki i czasowniki ułożone w stosownej kolejności. Jednak, konia z rzędem temu, kto rozumie o co w nim chodzi. Podobnie jest też z językami programowania. Pozornie niewinna instrukcja typu: „$x + y$”, może się okazać niewykonalna z powodu tego, że x jest zmienną opisującą pewien łańcuch tekstowy, zaś y jest na przykład tablicą liczb całkowitych. I choć osobno te symbole są poprawne, a łącząca je formuła ma sens, to jednak nijak nie da się ich przetworzyć w sensowny sposób.

W tym miejscu warto jest więc przejść do kolejnego ważnego pojęcia związanego z językami programowania, jakim jest semantyka. Semantyka to zestaw reguł określających znaczenie każdego symbolu języka i jego funkcję w programie. Najczęściej zestaw ten definiuje się słownie z uwagi na stopień komplikacji interakcji między regułami mającymi opisywać rzeczywiście użyteczny język programowania. Dzięki semantyce już na wstępnym etapie przetwarzania kodu można wychwycić sporo błędów takich jak na przykład próby wywołań nieistniejących funkcji, czy niemożliwe do realizacji sposoby przerabiania jednych danych na inne (tzw. rzutowanie). Najczęściej jednak jest tak, że mimo istnie-

nia składni i semantyki część wyjątkowo rzadkich błędów ujawnia się czasami dopiero w trakcie działania programu.

Skoro już o tym mowa, to warto zauważyć, że języki programowania dzielą się na dwie ważne kategorie związane ze sposobem wykonywania programów. Pierwsze z nich to języki kompilowane, a drugie to języki interpretowalne.

Języki kompilowane to takie, w których stworzony przez programistę kod źródłowy programu jest przetwarzany przez program zwany kompilatorem. Zadaniem kompilatora jest zamiana tego kodu na ciąg instrukcji języka maszynowego. Zaletą takich programów jest to, że z uwagi na swoje odwołania bezpośrednio do rozkazów procesora działają one na ogół znacznie szybciej niż języki interpretowalne. Ich wadą natomiast jest to, że są nieprzenośne. Jeżeli na przykład stworzymy program w języku C i skompilujemy go na komputerze klasy PC, to mamy marne szanse na to, aby działał na Sony Playstation. Co więcej, jeżeli ten sam program skompilujemy w systemie Microsoft Windows, to prawie na pewno nie uda nam się z niego korzystać bez dodatkowych sztuczek na tym samym komputerze, na którym zainstalujemy system Linux.

Z drugiej strony mamy języki interpretowalne, w których kod źródłowy napisany przez programistę jest na bieżąco przetwarzany przez program zwany interpreterem. Zaletą tych języków jest ich przenośność. Jeśli chcemy np. uruchomić nasz program na komputerze z innym systemem czy dowolnym innym urządzeniu, to nie musimy go w żaden sposób modyfikować, wystarczy, że będziemy dysponować interpreterem odpowiednim dla tego urządzenia i platformy. Wadą jest tutaj oczywiście ich wydajność, która jest znacznie mniejsza niż w przypadku programów pisanych w językach kompilowanych. Dzieje się tak z prostego powodu: aby zinterpretować jakąś instrukcję, trzeba ją przetworzyć na zapis zrozumiały dla interpretera, który dopiero musi wywołać jeden ze swoich podprogramów, który wiąże się z wykonaniem pewnego ciągu rozkazów języka maszynowego odpowiedniego dla danego urządzenia. Występują więc tutaj dodatkowe stopnie pośredniości, które w konsekwencji muszą zajmować więcej czasu. Przykładami języków prezentujących takie podejście są PHP, Perl czy Python.

Ostatnimi czasy coraz popularniejsze stają się języki, w których występuje połączenie tych dwóch idei. Program jest kompilowany, ale nie do postaci ciągu rozkazów języka maszynowego, lecz do tzw. kodu bajtowego, który jest taki sam w przypadku wszystkich systemów komputerowych i urządzeń oraz stanowi coś w rodzaju wysoce zoptymalizowanego języka interpretowalnego, który najczęściej jest kompletnie nieczytelny dla człowieka, za to jest wygodny w przetwarzaniu przez komputer. Dzięki takiemu podejściu zachowujemy przenośność, choć cały czas trzeba pamiętać o konieczności napisania stosownych programów interpretacyjnych i zyskujemy na wydajności w stosunku do wcześniejszych języków interpretowalnych, zachowując względną wygodę programowania. Przykładami języków, w których zastosowano takie podejście, są Java i C#.

Myliłby się ten, kto pomyślałby, że języki programowania różnią się między sobą jedynie sposobem przetwarzania ich kodów źródłowych. Na stronie

http:/www.99-bottles-of-beer.net można znaleźć kody źródłowe tego samego programu napisanego w ponad tysiącu różnych języków programowania. To wystarczająco duża próbka, aby zauważyć, że nawet języki, których składnia oraz semantyka są stosunkowo podobne, pozwalają na wyrażanie algorytmów w zupełnie różne sposoby. Te sposoby to tzw. paradygmaty programowania. Obecnie istnieje 11 paradygmatów programowania. Niektóre języki w swoich ramach łączą kilka z nich. Zajmijmy się teraz krótkim omówieniem każdego paradygmatu.

Programowania proceduralne to pierwszy z nich. Programowanie proceduralne polega na podziale kodu tworzonego programu na logiczne części zwane procedurami, z których każda wykonuje ściśle określone zadanie. Podejście to zakłada, że dobrą praktyką programistyczną jest przekazywanie danych przewidzianych do przetworzenia przez procedury poprzez tzw. parametry wywołania, a nie zmienne globalne.

Przykład 2:

```
void kwadrat(int x)
{
  printf("kwadrat = %d",x*x);
}
```

Jest to przykład procedury napisanej w języku C, której zadaniem jest wypisywanie kwadratu zmiennej x, stanowiącej tutaj, parametr wywołania. Ten sam fragment programu mógłby zostać zapisany przy użyciu zmiennej globalnej i wyglądałby wtedy tak:

```
int x;
void kwadrat()
{
  printf("kwadrat = %d",x*x);
}
```

Na czym polega różnica? Na sposobie wywoływania tej procedury. W pierwszym przypadku wywołanie procedury kwadrat wiąże się podaniem zmiennej, której kwadrat chcemy zobaczyć (np. kwadrat(2)). W drugim przypadku nie musimy podawać wartości zmiennej, którą chcemy zobaczyć, bo zostanie ona odczytana ze zmiennej x znajdującej się poza tą procedurą. W związku z tym, wywołując tę procedurę (kwadrat()), zobaczymy kwadrat ostatniej liczby, jakiej wartość była przechowywana w zmiennej x.

Drugi paradygmat programowania to programowanie strukturalne. Podobnie jak w programowaniu strukturalnym zakłada się tu podział kodu na logiczne części. Tutaj jednak dodatkowo dodaje się wymóg istnienie pewnej ich hierarchii. Chodzi o to, by w tworzonym kodzie wyodrębniać bloki, z których każdy ma tylko jedno wejście i co najmniej jedno wyjście. Dobrym przykładem języka o paradygmacie strukturalnym jest język C, w którym bloki programu

ogranicza się przy pomocy nawiasów klamrowych. O ile w językach czysto proceduralnych dosyć nagminnie stosowaną instrukcją jest instrukcja skoku do etykiety (goto), o tyle w językach strukturalnych nawet jeżeli umożliwia się jej używanie, to uznaje się go za wyjątkowo zły pomysł. Dobrymi instrukcjami strukturalnymi są pętle, instrukcje warunkowe i instrukcje wyboru. Instrukcjami zaburzającymi strukturalność są instrukcje: *break* (przerwanie działania pętli w trakcie jej wykonywania), *continue* (przejście na początek pętli z opuszczeniem instrukcji następujących po continue aż do końca pętli) oraz *switch* (instrukcja przypominająca w działaniu bardzo rozgałęzioną instrukcję warunkową).

Przykład 3:

To dobry kod strukturalny

```
if(x<0)
{
  printf("Ujemne");
}
else
{
  printf("Nieujemne")
}
```

A to kod, któremu do strukturalności dużo brakuje:

```
if(x<0) goto mniejsze;
else goto większe;
mniejsze:
printf("Ujemne");
goto koniec;
większe:
printf("Większe");
koniec:
```

Kolejnym paradygmatem jest programowanie deklaratywne. Języki oparte na tym paradygmacie umożliwiają opisywanie efektów, które chcemy osiągnąć bez opisywania sposobu, w jaki ten efekt ma powstać. Strategią tworzenia zajmuje się wtedy na przykład interpreter takiego języka. Dobrym przykładem takiego języka o paradygmacie deklaratywnym jest SQL, czyli Structured Query Language, będący językiem, w którym formułuje się zapytania do baz danych.

Przykład 4:

Załóżmy, że chcielibyśmy wydobyć ze stworzonej wcześniej bazy danych wszystkie dane znajdujące się w tabeli o nazwie zarobki. Gdybyśmy używali języka

niedeklaratywnego, musielibyśmy dokładnie opisać, w jaki sposób ma to być wykonane. Co gorsza, w przypadku przesiadki na inną bazę danych mogłoby się okazać, że ten program robi coś innego niż zakładamy.

Na szczęście perspektywę katorżniczej pracy programistycznej, która kończy się fiaskiem, odsuwa od nas możliwość zastosowania języka SQL, w którym to zadanie opisuje poniższy program.

```
SELECT * FROM zarobki;
```

SELECT - sugeruje bazie danych konieczność wybrania pewnego zakresu danych. Gwiazdka jest umownym znakiem, który sugeruje, że chodzi o całość danych, a FROM zarobki mówi serwerowi o tym, że chodzi o dane zgromadzone w tabeli o nazwie zarobki. I tyle! Zgłaszamy zapotrzebowanie i nie interesuje nas sposób jego realizacji.

Odmianą programowania deklaratywnego jest programowanie funkcyjne. W językach tego typu funkcje (czyli wyodrębnione bloki kodu przetwarzające dane wejściowe na wyniki) traktuje się jak wartości podstawowe. Dodatkowo spory nacisk kładzie się też na tzw. wartościowanie, czyli obliczanie wartości wyrażeń. Wyróżnia się trzy typy języków funkcyjnych.

Pierwszy typ to języki typowo funkcyjne, czyli takie, w których nie występują zmienne ani tzw. efekty uboczne. Efektem ubocznym nazywamy działanie, które wykonuje się niejako przy okazji wykonywania innych działań. Możemy np. stworzyć funkcję, która będzie wyliczała pewną wartość i po drodze wypisywała wynik częściowych obliczeń. Efektem ubocznym będzie w tym przypadku między innymi owo wypisywanie danych na ekranie. W językach funkcyjnych stosuje się też wartościowanie leniwe. Jest to sposób przetwarzania tylko tych fragmentów obliczeń, które są niezbędne do wykonania kolejnej instrukcji.

Przykład 5:

Załóżmy, że mamy wyznaczyć wartość następującego zdania:

$$a(x) \text{ lub } b(y),$$

gdzie $a(x)$ i $b(y)$ to pewne funkcje dające wartości postaci prawda lub fałsz.

Jeżeli przydarzy się, że wartość $a(x)$, to prawda, nie ma sensu wyznaczać wartości $b(y)$, bo ona i tak nie wniesie niczego nowego do procesu wyznaczania wartości wyrażenia $a(x)$ lub $b(y)$. Takie podejście nazywamy wartościowaniem leniwym, w przeciwieństwie do wartościowania zachłannego, które wymagałoby wyznaczenia wartości wszystkich wyrażeń wchodzących w skład powyższego przykładu przed przystąpieniem do ostatecznych obliczeń.

Oprócz tego w językach typowo funkcyjnych wejście i wyjście programu nie może być załatwione z użyciem standardowych metod polegających na wykorzystaniu klawiatury czy też odczytu danych z pliku. Najprostszym sposobem jest oczywiście umieszczenie interesujących nas danych bezpośrednio w źródle

programu. Przez takie podejście traci się jednak prawdziwą interakcję z programem, a zyskać ją można jedynie przy pomocy znacznie bardziej wyrafinowanych metod. Stąd być może bierze się mała popularność języków typowo funkcyjnych takich jak Haskell czy Clean, które mimo swej elegancji wymagają dodatkowych zabiegów prowadzących do zrobienia czegoś, co w innych językach dostaje się niejako za darmo.

Drugi typ języków funkcyjnych to języki mieszane, w których dopuszcza się używanie zmiennych i efektów ubocznych jak również tradycyjnego wejścia i wyjścia, a wartościowanie jest zachłanne. W tego typu językach najczęściej umożliwia się też mieszanie podejścia funkcyjnego z obiektowym i imperatywnym. Przykładami takich języków mogą być LISP i Nemerle.

Jeszcze inna odmiana programowania deklaratywnego to programowanie logiczne. Pozwala ono na tworzenie programu poprzez formułowanie zależności logicznych między różnymi obiektami i pytanie o prawdziwość pewnego twierdzenia, które na tej podstawie można udowodnić lub obalić. Przykładem języka prezentującego to podejście jest Prolog.

Przykład 6:

Rozważmy prosty pseudokod zapisany przy pomocy podejścia logicznego:

A dzieli B
B dzieli C
Sprawdź, czy A dzieli C

W pierwszych dwóch liniach program uzyskał informacje na temat relacji wiążącej liczby A i B, a w drugiej informacje na temat B i C. Ostatnia linia to instrukcja zapoczątkowująca przetwarzanie zgromadzonych danych. Warto zauważyć, że nie interesuje nas sposób ich przetworzenia (to właśnie programowanie logiczne przejęło po programowaniu deklaratywnym), a jedynie prawdziwość lub nieprawdziwość sformułowanego zapytania.

Kolejny paradygmat to programowanie uogólnione. Polega ono na umożliwieniu pisaniu programów bez wcześniejszej wiedzy na temat tego, na jakich typach danych ten program będzie pracował.

Przykład 7:

Gdybyśmy potrzebowali stworzyć taki program, który będzie sortował tablice zmiennych różnego typu, to w językach, które nie mają niczego wspólnego z programowaniem uogólnionym, wiązałoby się to z koniecznością stworzenia niemal identycznych funkcji, z których każda sortowałaby tablice elementów tylko jednego typu. W przypadku programowania uogólnionego tworzy się jeden kod, skupiający się na podobieństwach między tymi procesami, ewentualne różnice zostawiając późniejszej rozwadze.

Program sortujący liczby całkowite w języku C wyglądałby np. tak:

```
void sortuj(int[] a,int[] b,int n)
```

```
{
  int  i,j,c;
  for(i=0;i<n;i++)

  {
    for(j=i+1;j<n;j++)
    {
      if(a[i]<b[j])
      {
        c=a[i];
        a[i]=b[j];
        b[j]=c;
      }
    }
  }
}
```

Bardzo podobny program służy do sortowania tablic znaków. Jedyna różnica w tym przypadku polega na zmienieniu pierwszej linijki w następujący sposób:

```
sortuj(char[] a,char[] b,int n)
```

Oczywiście może się zdarzyć konieczność sortowania danych, których nie da się porównywać przy pomocy operatora $<$. Można jednak wtedy lekko zmodyfikować ten kod (pisząc go w innym języku dopuszczającym programowanie uogólnione), uzyskując na przykład coś takiego:

```
void sortuj(T[] a,T[] b,int n)
{
  int  i,j,c;
  for(i=0;i<n;i++)
  {
    for(j=i+1;j<n;j++)
    {
      if(less (a[i],b[j]))
      {
        c=a[i];
        a[i]=b[j];
        b[j]=c;
      }
    }
  }
}
```

gdzie funkcja `less(a,b)` zwraca prawdziwą wartość, gdy `a` jest mniejsze od `b`. W tym momencie musielibyśmy już tylko napisać funkcję porównującą ele-

menty wszystkich typów danych, których sortowanie nas interesuje. Kod odpowiedzialny za sortowanie w każdym z tych przypadków pozostałby już taki sam.

Obecnie bardzo popularne są programy z graficznym interfejsem użytkownika. W ich przypadku bardzo przydatnym okazał się predykat programowania zdarzeniowego polegający na tym, że program przez cały czas zasypywany jest informacjami o zdarzeniach, które mogą go interesować. Z tego powodu niemal niemożliwe jest przewidzenie wszystkich możliwości przepływu danych w programach napisanych w ten sposób. Łatwo sobie wyobrazić kogoś, kto bezmyślnie wali w klawiaturę i jednocześnie przesuwa po ekranie Worda wskaźnik myszy, której oba przyciski są wciśnięte. Program komputerowy (w tym przypadku Word) musi na to wszystko w jakiś sposób zareagować, a jest to o tyle trudne, że niektóre z tych zdarzeń będą miały miejsce w tym samym momencie. Wniosek z tego jest prosty: na obsługę każdego zdarzenia należy przeznaczać tak mało czasu jak tylko się da. Oprócz tego często korzysta się z asynchronicznego wejścia i wyjścia, dzieli obsługę zdarzeń na podzdarzenia i obsługuje je z wykorzystaniem wielu wątków tego samego procesu. Programowanie zdarzeniowe bardzo dobrze widoczne jest w programach napisanych w języku Java z wykorzystaniem pakietów Swing lub AWT.

W przypadku programów, w których jedna czynność wiąże się z mnóstwem różnego rodzaju działań pobocznych, których zgromadzenie w jednym miejscu mogłoby doprowadzić do znacznego zaciemnienia konstrukcji programu, używa się często paradygmatu programowania aspektowego. Polega on na separowaniu zagadnień i rozdzielaniu programu na części, które mogą wymieniać ze sobą informacje jedynie w ściśle określonych miejscach, co pozwala na zwiększenie kontroli nad zadaniem, które chcemy przetworzyć. Do tej pory najpopularniejszym językiem prezentującym podejście aspektowe jest AspectJ stworzony przez Gregora Kiczalesa w firmie Xerox PARC.

Przejdźmy teraz do prawdopodobnie obecnie najpopularniejszego paradygmatu programowania, jakim jest programowanie obiektowe. Podejście to wykorzystywane jest w językach Java i C++, w których tworzona jest większość liczących się rozwiązań informatycznych na świecie. W językach obiektowych tworzy się obiekty, czyli elementy łączące w sobie informacje na temat swojego aktualnego stanu (najczęściej nazywane polami) z procedurami pozwalającymi ten stan zmieniać (najczęściej nazywanymi metodami). Programy obiektowe definiuje się przy pomocy takich elementów, których wzajemna interakcja prowadzi do rozwiązania stawianych przed nimi problemów.

Programy obiektowe charakteryzują się kilkoma bardzo użytecznymi cechami. Pierwszą z nich jest tzw. dziedziczenie. Każdy człowiek, który kiedykolwiek rozkręcił dwa tanie zegarki kwarcowe, zauważył z pewnością, że ich wnętrza są niemal identyczne. Dzieje się tak z powodu tego, że nie opłaci się za każdym razem na nowo projektować mechanizmu zegarka. Znacznie prościej jest zmodyfikować go w wymaganym stopniu (na przykład dodając datownik) i dostosować do tych poprawek jego obudowę. Podobnie chcąc mieć zegarek z datownikiem i pozytywką prawdopodobnie, wykorzystamy istniejący pro-

jekt zegarka z datownikiem. Dzięki temu oszczędzamy masę czasu, bo kolejne projekty dziedziczą cechy po swoich poprzednikach, zachowując się przez to w bardziej przewidywalny sposób.

Innym ciekawym mechanizmem języków obiektowych jest hermetyzacja. Dobrze jest mieć pewność, że dane przetwarzane przez nasz program nie będą się zmieniały z powodu błędnego działania części programu, która nie powinna mieć do nich dostępu.

W językach nie obiektowych trzeba było o to zadbać samodzielnie, podczas gdy języki obiektowe pozwalają na określanie tego, jaka część programu i w jaki sposób może się dobrać do danych i metod zgromadzonych w innej części programu.

Ciekawym rozwiązaniem jest również tzw. polimorfizm, czyli możliwość nazywania w ten sam sposób metod robiących podobne rzeczy na różnych zestawach danych. Może nie jest to tak przełomowe jak dziedziczenie, ale w znaczący sposób wpływa na czytelność kodu, co przekłada się na prostotę jego konwersacji.

Podobnie jest z abstrakcją, która pozwala na komunikowanie się ze sobą obiektów, które nie bardzo wiedzą, z czego się składają. Przypomina to trochę sytuację, w której mamy w ścianie gniazdko elektryczne i wtyczkę radia, potrafimy sprawić, by radio zaczęło działać mimo tego, że nie bardzo wiemy, skąd pochodzi prąd ani jak działo radio.

Opisane wyżej triki sprawiły, że programowanie znacznie bardziej zaczęło przypominać planowanie współdziałania obiektów pochodzących z rzeczywistego świata, a nie próbę ujarzmienia jakichś niezrozumiałych formuł matematycznych. Prawdopodobnie z tego powodu podejście to zaczęło być powszechnie stosowane i nic nie wskazuje, na to by miało się to szybko zmienić.

Szczególnym przypadkiem programowania obiektowego jest programowanie agentowe, którego nazwa pochodzi od pewnego typu programów komputerowych zwanych agentami, które charakteryzują się następującymi cechami.

- autonomicznością, co sprowadza się do zdolności samodzielnego podejmowania decyzji,
- komunikatywnością, objawiającą się możliwościami wymiany informacji z innymi agentami,
- percepcją, czyli zdolnością do reagowania otaczającego ich środowiska.

Faktem jest, że agenty mogą, lecz nie muszą wymieniać się informacjami. Jednakże każdy agent musi poprawnie zareagować na błędne dane przesłane mu przez innego agenta, bądź też na brak danych w ogóle. Jest to znacząca różnica w stosunku do programów obiektowych, w których zakładało się, że każdy obiekt przetwarza i wysyła dane w poprawnej, zaplanowanej przez autora programu formie.

Oprócz tego w programach agentowych naturalną cechą jest fakt, że wielu agentów przetwarza jedno zadanie, wykorzystując wzajemną interakcję do wyznaczenia optymalnego rozwiązania postawionego przed nimi problemu. W programowaniu czysto obiektowym uznane to byłoby za marnowanie mocy

obliczeniowej procesora, ze wskazaniem na to, by rozpatrywane zadanie zostało przetworzone przez jeden obiekt.

Niewątpliwą zaletą programów wieloagentowych jest to, że świetnie sprawdzają się w sieciach komputerowych. Ich działanie sprowadza się wtedy często do wykorzystania mocy obliczeniowej wielu komputerów połączonej z wymianą informacji prowadzoną za pośrednictwem sieci Internet. Dzięki wspólnemu dążeniu do rozwiązania problemu prowadzonego przez wiele odrębnych programów, zwiększa się też odporność na awarie sieci, sabotaż, czy inne nieprzewidywalne zagrożenia.

Na zakończenie warto dodać, że chyba nie da się wskazać przykładu języka, który oparty byłby na dokładnie jednym z tych predykatów. Najczęściej łączą się one ze sobą, tworząc nową jakość pozwalającą na sprawniejsze wyrażenie pewnego typu algorytmów w sposób, który będzie jednocześnie tak samo czytelny zarówno dla człowieka, jak i dla komputera.

10

Zagadnienie kompresji danych

Jak daleko człowiek sięgnie pamięcią, to komputery miały zawsze za małe pamięci, a Internet zawsze był za wolny, by pomieścić i przesłać wszystkie dane, które nas interesują. Zdając się w tym przypadku na rozwój technologiczny, trzeba cierpliwie poczekać aż on się pojawi, po czym wydać niemałe pieniądze na modernizację naszego sprzętu. Problem ten można oczywiście obejść z innej strony, wykorzystując w tym celu osiągnięcia jednej z gałęzi matematyki zwanej teorią informacji.

Mowa tu oczywiście o kompresji danych, która polega na próbie zapisania pierwotnych danych w taki sposób, aby zajmowały mniej miejsca i po dekompresji reprezentowały te same lub zbliżone dane.

10.1 Przykłady

Przykład 1:

Rozważmy następujący ciąg liter:

aaaaabbbbbbbccccdddddddd

Jest ich 25 i na pierwszy rzut oka widać, że dałoby się pominąć niektóre z nich, doprowadzając do zapisu, z którego dokładnie da się odtworzyć oryginał. Można to zrobić na przykład w ten sposób:

5a7b5c8d

co przekłada się na zmniejszenie rozmiaru pierwotnych danych z 25 do 8 znaków. Dekompresja polega w tym przypadku na odczytywaniu liczby i występującego po niej znaku oraz na powtarzaniu tej liczby tyle razy, ile wskazuje na to ta liczba. Pomysł ten wydaje się rewelacyjny dopóki nie zdamy sobie sprawy z możliwości wystąpienia danych postaci

11111111333333333333344444444444455555555555

w których kodowanie 81133134115 nie pozwala na przeprowadzenie jednoznacznej dekompresji. Próba modyfikacji zapisu do postaci: 8 1 13 3 13 4 11 5, też nie załatwia sprawy, gdyż łatwo sobie wtedy wyobrazić dane, w przypadku których kompresja prowadzi do zwiększenia ilości informacji potrzebnych do zapisania pierwotnych danych. Przykładowo dla ciągu 1234 skompresowane dane wyglądałyby wtedy w ten sposób: 1 1 1 2 1 3 1 4, co jest trzykrotnie dłuższe od oryginału.

Nie ulega wątpliwości, że jest mnóstwo informacji, których kompresja musi gwarantować możliwość dekompresji do postaci dokładnie tożsamej z pierwotną. Źle by się stało, gdyby np. po awarii systemu bankowego oraz odzyskaniu danych komputerowych ze skompresowanych kopii bezpieczeństwa ktoś odkrył, że na jego koncie zamiast 12345678 dolarów jest 12 dolarów i 34 centy. Algorytmy kompresji, które wykluczają możliwość pojawienia się takich sytuacji, nazywamy algorytmami kompresji bezstratnej.

Z drugiej strony, jesteśmy czasem w stanie poświęcić część z jakiegoś powodu nieistotnych informacji zawartych w oryginalnym pliku w celu radykalnego zmniejszenia jego objętości. Algorytmy, które wykorzystują tego typu podejście, nazywamy algorytmami kompresji stratnej.

W pierwszym przykładzie tego rozdziału przedstawiono bardzo naiwny algorytm kompresji bezstratnej. Prześledźmy teraz przykład bardziej skomplikowanego algorytmu, który został odkryty w 1952 roku przez Davida Huffmana.

Kolejne kroki tego algorytmu można opisać w następujący sposób:

1. Określ prawdopodobieństwo wystąpienia każdego znaku w oryginalnych danych.
2. Utwórz listę par uporządkowanych zawierających symbol i prawdopodobieństwo jego wystąpienia. Pary te w kolejnych krokach będą stawały się wierzchołkami pewnego grafu. Na razie jednak, będą tworzyły listę oddzielnych grafów złożonych z pojedynczych wierzchołków będących tzw. korzeniami.
3. Dopóki na liście znajduje się więcej niż jeden graf, wybieraj z listy grafy, których korzenie mają najmniejszą sumę prawdopodobieństw wystąpienia.
4. Połącz znalezione w punkcie czwartym grafy w jeden, którego korzeń będzie zawierał sumę prawdopodobieństw zapisanych w ich korzeniach.

Na podstawie utworzonego w ten sposób grafu tworzone są ciągi bitów kodujących kolejne symbole (są to tzw. słowa kodujące). Tworzy się, je przechodząc od korzenia do konkretnego symbolu po krawędziach grafu. Za każdym razem, gdy wybierzemy krawędź lewą, będziemy dodawać do ciągu 0, a gdy skręcimy w prawo, dodamy do ciągu jedynkę.

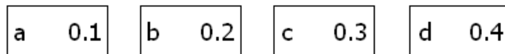
Przykład 2:

Rozważmy następujące dane: dabcbdcdcd. Jak widać, kodowanie ich przy pomocy wprowadzonego na początku algorytmu nie ma najmniejszego sensu,

gdyż żadna z występującym w tym ciągu liter nie występuje kilka razy z rzędu. Spróbujmy zatem skompresować te dane przy użyciu algorytmu Huffmana.

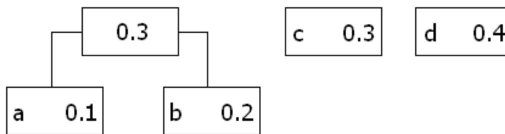
Całe dane reprezentowane są przez 10 znaków. Litera „a" występuje w tym ciągu 1 raz, więc prawdopodobieństwo jej wystąpienia to 0.1. Litera „b" występuje 2 razy, więc prawdopodobieństwo jej wystąpienia to 0.2. Analogicznie prawdopodobieństwa wystąpienia liter „c" i „d" to odpowiednio 0.3 i 0.4. W ten sposób zakończyliśmy wykonywanie pierwszego punktu opisanego algorytmu.

W punkcie drugim tworzymy uporządkowane pary postaci (symbol, prawdopodobieństwo jego wystąpienia), które w tym przypadku będą wyglądały, jak następuje: $(a, 0.1), (b, 0.2), (c, 0.3)$ i $(d, 0.4)$ lub w postaci graficznej.



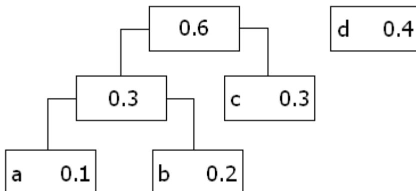
Rysunek 10.1.

W punkcie trzecim przychodzi czas na odnalezienie tych grafów (na razie złożonych z pojedynczych wierzchołków), które mają najmniejsze prawdopodobieństwo występowania. Jak widać, są to znaki „a" oraz „b", których łączne prawdopodobieństwo występowania to $0.1 + 0.2 = 0.3$. Przejdźmy zatem do punktu czwartego i połączmy je w jeden graf, uzyskując następującą listę grafów:

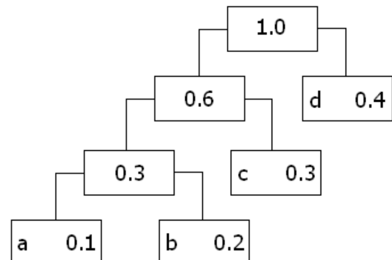


Rysunek 10.2.

Powtarzając kroki trzeci i czwarty, otrzymamy odpowiednio kolejne listy grafów (por. rys. 10.3 i 10.4).



Rysunek 10.3.



Rysunek 10.4.

Jak teraz wybrać ciągi bitów kodujące kolejne znaki? Spójrzmy na powyższy rysunek. Aby z korzenia powyższego grafu dostać się do litery wierzchołka zawierającego symbol „a”, musimy trzykrotnie skręcić w lewo, czyli ciąg kodujący tę literę będzie miał postać 000. Aby zakodować symbol b, trzeba dwukrotnie skręcić w lewo i raz w prawo. Zatem ciąg bitów kodujący symbol „b” to 001. Analogicznie dla c i d uzyskamy odpowiednio ciągi 01 i 1. Jak widać, dzięki takiemu zabiegowi symbol występujący w danych najczęściej jest kodowany przez najmniejszą ilość bitów, co przekłada się na oszczędność miejsca zajmowanego przez dane skompresowane w stosunku do oryginalnych.

Dysponując już ciągami bitów kodujących poszczególne symbole, możemy przystąpić do ostatniego kroku kompresji, jakim jest zamiana tych symboli poprzez odpowiadające im ciągi. Po tej operacji dane dabcbdcdcd (które ze względu na to, że każda litera jest kodowana przez jeden bajt, są tak naprawdę ośmiokrotnie dłuższe i zajmują 80 bitów), skurczą się do postaci

$$1000001010011011011,$$

która zajmuje 19 bitów.

To imponujący efekt, ale nietrudno zauważyć, że sam ciąg

$$1000001010011011011$$

jest bezużyteczny z powodu tego, że nie wiadomo, jaki symbol stoi za jaką sekwencją bitów. Dlatego też dane te mogłyby zostać równie dobrze zdekompresowane do postaci `dabcbdcdcd`, jak i do `nabcbncncn`. Wynika z tego, że aby możliwa była prawidłowa dekompresja, należy w jakiś sposób dodać do skompresowanych danych tzw. słownik, czyli listę ciągów bitów kodujących kolejne symbole i odpowiadających im symboli. Jak nietrudno się domyślić, wpłynie to na stopień kompresji, choć przy dużych plikach zawierających tylko kilka różnych symboli wpływ ten będzie nieznaczny.

Opisany powyżej algorytm to algorytm o zmiennej długości słowa kodującego, czyli ciągu znaków, którym będziemy zastępować oryginalne symbole. Możliwe jest też jednak inne podejście polegające na przypisywaniu kolejnym znakom słów kodujących o ustalonej długości.

Przykład 3:

Rozważmy ponownie dane postaci `dabcbdcdcd`. Jak widać, składają się one tylko z czterech liter, które można zakodować przy pomocy dwóch bitów. Umówmy się, że „a” będzie reprezentowane przez ciąg 00, „b” przez „01”, „c” przez 10, a „d” przez 11. Dzięki takiej umowie oryginalne dane mogą zostać zapisane w postaci 11000110011110111011, co łącznie daje 20 bitów i jest trochę gorszym wynikiem niż ten, który osiągnęliśmy przy pomocy kodowania Huffmana. Podobnie jak w poprzednim przypadku, tak i tu musimy jednak pamiętać o dodaniu słownika, który pozwoli na dekompresję danych do ich pierwotnej postaci.

Opisane powyżej algorytmy kompresji to bodaj najprostsze sposoby bezstratnego kompresowania danych. W ciągu ostatnich lat powstały dziesiątki

innych algorytmów pozwalających na szybszą lub wydajniejszą kompresję bezstratną. Wszystkie bazują jednak na znacznie bardziej wyrafinowanym aparacie matematycznym i ich opisywanie w tym miejscu byłoby zbyteczne dla przedstawienia zarysu zagadnienia kompresji. Dodajmy, że popularny na całym świecie WinZip wykorzystuje obecnie algorytm bzip2, w którym kompresowane dane są dzielone na bloki po około 100 kB. Każdy z takich bloków jest przetwarzany z użyciem tzw. transformaty Burrowsa-Wheelera, a następnie przekształcany przez algorytm Move To Front, by ostatecznie skompresować go przy pomocy algorytmu Huffmana. A to wcale nie jest najbardziej skomplikowany z aktualnie istniejących i wykorzystywanych algorytmów kompresji bezstratnej.

Przejdźmy teraz do algorytmów kompresji stratnej. Podobnie jak iluzjoniści, algorytmy te bazują najczęściej na niedoskonałości ludzkich zmysłów. Z uwagi na to nie istnieją więc algorytmy kompresji stratnej odpowiednie dla każdego zestawu danych, bo każdy z naszych zmysłów w inny sposób reaguje na bodźce i inaczej da się go oszukać.

Przykład 4:

Jednym z najbardziej popularnych rozszerzeń plików w ostatnich czasach jest mp3. Rozszerzeniem tym oznaczane są skompresowane w sposób stratny pliki muzyczne. Dla plików tego typu ważne jest również, że w sporej ilości przypadków strata na ich jakości w stosunku do muzyki nieskompresowanej jest możliwa do wychwycenia jedynie przez prawdziwych audiofili na bardzo drogim sprzęcie. Dzieje się tak dlatego, że format ten wykorzystuje między innymi tzw. model psychoakustyczny, czyli matematyczny opis tego, co właściwie jest słyszalne przez przeciętne ludzkie ucho.

Najprostsze z założeń tego modelu polega na tym, że przeciętny człowiek słyszy dźwięki o częstotliwościach z przedziału pomiędzy 20 Hz a 20 KHz. Jeżeli zatem w oryginalnym pliku muzycznym znajdują się zapisy dźwięków wykraczających poza ten zakres, to można założyć, że mało kto usłyszy ich brak, a co za tym idzie, można je z powodzeniem pominąć jeszcze przed fazą kompresji właściwej.

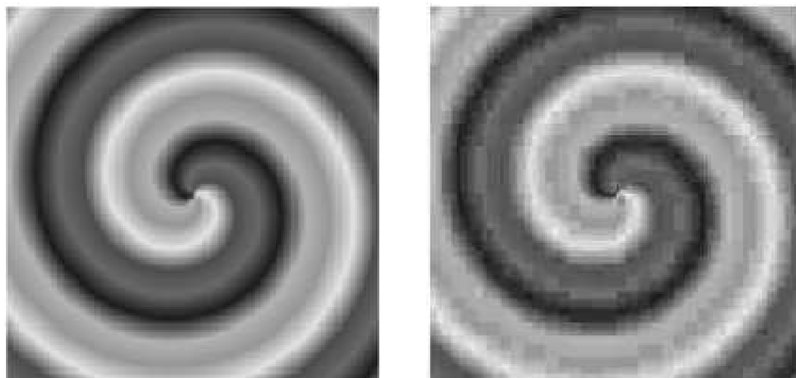
Oprócz tego wykorzystuje się serię trików polegających na maskowaniu jednych częstotliwości innymi. Dla przykładu ciche dźwięki o częstotliwościach zbliżonych do występującego w tym samym momencie dźwięku głośnego są przez ludzkie ucho słyszalne słabo lub wcale, więc również można je pominąć, tak samo jak dźwięki ciche występujące tuż po dźwięku głośnym.

Tego typu obcinanie niepotrzebnych danych to oczywiście tylko fragment całego procesu, którego ukoronowaniem jest kompresja z użyciem tzw. zmodyfikowanej dyskretnej transformaty kosinusowej.

Przykład 5:

W sposób stratny można kompresować również obrazy. Jednym z formatów zapisu, które na to pozwalają, jest format JPEG. Podczas kompresji plików

do tego formatu wykorzystuje się między innymi fakt, że ludzkie oko znacznie lepiej rozróżnia małe różnice jasności niż małe różnice barwy. W związku z tym w pewnych przypadkach wprowadza się uśrednienie barwy sąsiadujących punktów, co w przypadku kompresji o rozsądnym stopniu jest niezauważalnym oszustwem. W przypadku jednak gdy będziemy chcieli skompresować oryginalny plik ponad dopuszczalną miarę, może to wpłynąć na powstanie zauważalnych przekłamań, które można zauważyć na poniższym przykładzie.



Rysunek 10.5.

Z lewej umieszczono oryginalny obrazek, zaś z prawej obrazek skompresowany do około 5% pierwotnej objętości, co spowodowało pojawienie się w nim wielu miejsc o przekłamanych kolorach i zaburzenie gładkości linii występujących w oryginale.

Podsumowując, powiemy, że przedstawione tutaj zagadnienia zostały dobrane w ten sposób by zasygnalizować najważniejsze hasła dotyczące opisywanego tematu i z natury swojej są dosyć pobieżne. Swego czasu bardzo popularny był dowcip o Chucku Norrisie, który jako jedyny człowiek na świecie mógł zmieścić na dyskietce cały Internet. Z oczywistych względów jest to niemożliwe, choć niewątpliwie algorytm pozwalający na urzeczywistnienie takiego wyczynu byłby niemal bezcenny. Warto zauważyć, że firma Google wydaje rocznie prawie ćwierć miliarda dolarów na sam prąd potrzebny do działania jej serwerów, z których spora część musi obsługiwać ruch generowany przez serwis YouTube. Powstaje więc pytanie: ile jej właściciele byliby w stanie zapłacić za program, który w sprytny sposób kompresowałby filmy tak, by do ich rozpowszechniania wystarczyło choćby o połowę mniej serwerów? Przeczucie podpowiada, że sporo. Niech więc stanie się ono inspiracją do dalszych rozmyślań nad niezwykle ważnym w dzisiejszych czasach zagadnieniem kompresji danych.

11

Grafika komputerowa

Dziś trudno w to uwierzyć, ale pierwsze komputery nie miały monitorów. Jeżeli działający na nich program komputerowy miał wyrzucać z siebie jakieś informacje, to po prostu były one kierowane na drukarkę. Nie były to też drukarki w dzisiejszym rozumieniu tego słowa, lecz raczej zdalnie sterowane maszyny do pisania. Po tamtych czasach pozostało już niewiele. Pamięć po nich zachowała się najwyraźniej w językach programowania, w których instrukcja pozwalająca na wyświetlanie tekstowej informacji na ekranie monitora najczęściej zawiera w sobie słowo „`print`” od angielskiego „`drukuj`”.

Kilka dekad później komputery zawitały pod strzechy i trzeba przyznać, że przypominały już wtedy swoim wyglądem dzisiejsze ich odpowiedniki. Rolę monitorów pełniły wówczas najczęściej telewizory, a posiadanie prawdziwego monitora komputerowego było równoznaczne z byciem na technologicznym topie.

Pomysł na to, że komputer może wyświetlać coś więcej niż literki, w swojej istocie jest bardzo prosty, choć przez lata ewoluował, by dojrzeć do dzisiejszej formy. Monitor to przecież nic innego jak mnóstwo malutkich lampek ułożonych obok siebie w taki sposób, aby zapełnić sporą (najczęściej) prostokątną powierzchnię. Początkowo te lampki nie były zbyt skomplikowane, mogły też jedynie świecić lub być zgaszone. Jeżeli więc świeceniu przypisać jedynkę, zaś wygaszeniu zero, to można pójść krok dalej i pokusić się o to, by pewien obszar w pamięci komputera odpowiadał za aktualną zawartość wyświetlacza monitora. Biorąc pod uwagę pierwszy bajt tego zakresu, otrzymujemy kombinację ośmiu zer i jedynek, co odpowiada pierwszym ośmiu punktom na ekranie. Każdy następny bajt odpowiada kolejnym ośmiu punktom, i tak aż do momentu, gdy na ekranie już więcej punktów się nie zmieści.

Trochę później pojawiły się pomysły na to, aby jednemu punktowi odpowiadał jeden bajt. Ponieważ jeden bajt to osiem bitów, więc przy jego pomocy można zakodować 256 liczb, które mogą odpowiadać 256 odcieniom szarości. To już był znaczący postęp w stosunku do grafiki, która była czarno-biała, niemniej było to dalekie od tego, do czego przywykliśmy w ciągu kilku ostatnich lat.

Kolejnym przełomowym pomysłem było wprowadzenie palet, czyli zestawów kolorów, z których każdy miał przypisany unikatowy numer. W momencie odmalowywania zawartości ekranu przeglądało się pamięć z nim związaną i interpretowało jej kolejne bajty jako numery kolorów, które mają przyjąć kolejne punkty ekranu. W ten sposób narodziła się kolorowa grafika komputerowa, ale 256 kolorów na jednym ekranie to naprawdę nie jest ilość, która sprawiałaby, że wyrażone przy ich pomocy obrazki mogłyby udawać rzeczywistość.

Po drodze pojawiały się jeszcze różne inne, mniej lub bardziej udane koncepcje, takie jak chociażby zastosowany w komputerach Commodore Amiga tryb HAM (Hold And Modify), w którym poprzez zastosowanie pewnej sztuczki możliwe stało się wyświetlenie wstrząsającej jak na owe czasy liczby 4096 kolorów na jednym ekranie.

Przez długi czas komputery mogące w rozsądnym czasie tworzyć grafikę na przyzwoitym poziomie były zbyt drogie na kieszeń przeciętnego człowieka oraz zbyt wolne na to, by mogły obsłużyć większość nowoczesnych gier komputerowych. Zekranizowany w 1993 roku film „Jurrasic Park" zawierał w sobie mniej niż 5 minut scen z udziałem wygenerowanych komputerowo dinozaurów, zaś przygotowanie tego materiału według zapewnień producentów trwało wiele miesięcy.

W dzisiejszych czasach stosuje się najczęściej pewną modyfikację pomysłu, na którym bazuje model RGB. Badania wykazały, że ludzkie oko odbiera poszczególne kolory jako zestawienia trzech barw o różnej jasności. Te barwy to czerwona (ang. `red`), zielona (ang. `green`) i niebieska (ang. `blue`), a nazwa modelu pochodzi od pierwszych liter angielskich nazw kolorów składowych. Przyjmując, że na opis jasności każdego z tych kolorów wykorzystamy jeden bajt (jest to tak zwany sRGB, czyli standarized RGB), otrzymamy 3 bajtową kombinację, w której można zapisać $256 \cdot 256 \cdot 256$ kolorów, a to jest już liczba imponująca. Poniżej znajduje się kilka przykładów kolorów i odpowiadających im liczb opisujących jasności kolorów składowych.

Czarny	R=0, G=0, B=0
Biały	R=255, G=255, B=255
Czerwony	R=255, G=0, B=0
Zielony	R=0, G=255, B=0
Niebieski	R=0, G=0, B=255
Fioletowy	R=255, G=0, B=255
Żółty	R=255, G=255, B=0

Choć w powyższym przykładzie użyto wyłącznie skrajnych wartości, to do opisu barw można wykorzystywać oczywiście dowolne kombinacje liczb z zakresu 0-255. A jak to wygląda w praktyce? O ile na początku jeden punkt odpowiadał jednej lampce, która mogła znajdować się w jednym z dwóch stanów (włączona lub wyłączona), o tyle, już w nowoczesnych monitorach na każdy punkt składają się trzy lampki, z których każda może świecić z różną jasnością.

Ponieważ całość jest na tyle mała, że ludzkie oko ich nie rozróżnia, więc w konsekwencji dostrzegamy barwy poszczególnych punktów jako mieszaniny barw składowych.

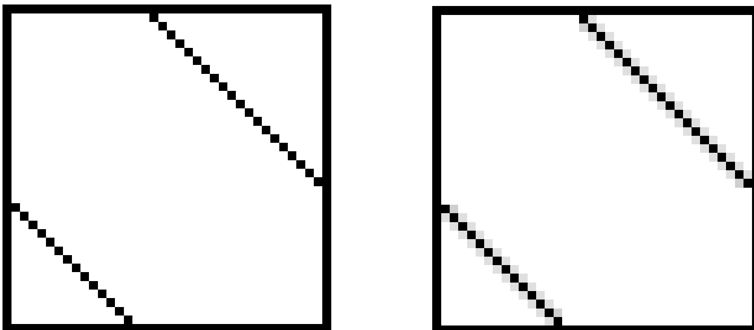
Jak już wspomniano, powyższy model opisu kolorów przyjął się w przypadku monitorów komputerowych. W zastosowaniach poligraficznych najczęściej używa się modelu o nazwie CMYK, w którym barwy składa się z użyciem kolorów cyjanowego (ang. `Cyan`), magenty (ang. `Magenta`), żółtego (ang. `Yellow`) i czarnego (ang. `black`). Oprócz tego istnieje wiele innych mniej popularnych modeli kolorów takich jak choćby HSL i HSV, w których możliwa do osiągnięcia paleta barw przyjmuje również postać wielowymiarowej przestrzeni, której wymiary opisane są innymi parametrami.

Ciekawą kwestią związaną z grafiką komputerową jest sposób przechowywania informacji opisujących obrazy. Zasadniczo wyróżnia się tutaj dwa podejścia. Pierwsze z nich to grafika rastrowa, a drugie to grafika wektorowa.

Obraz zapisany przy pomocy techniki rastrowej, to nic innego jak rodzaj mozaiki złożonej z punktów o różnych barwach opisanych przez pewien model koloru. Niewątpliwą zaletą takiego podejścia jest fakt, że na obecnym poziomie wiedzy informatycznej obrazy zapisane w ten sposób znacznie lepiej odwzorowują rzeczywistość niż obrazy wykorzystujące podejście czysto wektorowe. Wadą jest to, że skalowanie takich obrazów bez utraty jakości możliwe jest wyłącznie w pewnym zakresie, powyżej którego na obrazie pojawiają się nieładne przekłamania kolorów lub kształtów.

11.1 Przykład

Poniżej zamieszczono obrazek oryginalny i jego pięciokrotne powiększenie. Jeden i drugi obrazek został wykonany przy użyciu programu Microsoft Paint, który nie należy do czołówki oprogramowania wspierającego tworzenie grafiki komputerowej, ale podobnych efektów należałoby się spodziewać przy wykorzystaniu innych narzędzi.



Rysunek 11.1.

Jak widać, to co w oryginale mogło uchodzić za linie skośne, to nic innego jak ułożone obok siebie piksele, które na powiększeniu zamiast linii prostej tworzą coś na kształt zębatki. Oprócz tego oryginalny rysunek miał wyłącznie barwy białą i czarną, a na powiększeniu pojawiają się różnego rodzaju szarości.

Drugim podejściem jest tak zwana grafika wektorowa. W jej przypadku obraz składany jest z kształtów opisywanych przy pomocy matematycznych formuł. Zaletą takiego podejścia jest to, że obraz wektorowy może być powiększany oraz zmniejszany bez żadnych ograniczeń i zawsze będzie zachowywał tę samą wysoką jakość. Wadą natomiast jest to, że na obecnym etapie wiedzy informatycznej obrazy czysto wektorowe w dużym stopniu odbiegają swoim wyglądem od zdjęć rzeczywistego świata.

Z uwagi na to grafika wektorowa stosowana jest do przechowywania obrazów, w których pewna część danych jest nieistotna lub wręcz powinna zostać pominięta w celu zwiększenia czytelności przekazu. Przykładami takich zastosowań mogą być mapy, znaki drogowe, czy rysunki techniczne.

Zdarza się i tak, że obrazy wektorowe i rastrowe łączone są w jedno. Jest to możliwe na przykład przy użyciu technologii Flash stworzonej przez firmę Macromedia i rozwijanej obecnie przez firmę Adobe.

Kolejnym sposobem klasyfikowania grafiki jest jej podział na grafikę dwuwymiarową (oznaczaną skrótem 2D od angielskiego słowa „*dimension*” oznaczającego wymiar) i grafikę trójwymiarową (oznaczaną skrótem 3D). Z uwagi na fakt, że nie ma jeszcze urządzeń, które byłyby w stanie obrazować grafikę w trzech wymiarach, to zarówno grafika 2D, jak i 3D obrazowana jest w postaci płaskiej. Istnieją co prawda pewne techniki pozwalające na oszukanie ludzkiego zmysłu wzroku i pozorne przedstawienie głębi na obrazie wyświetlanym na płaskim ekranie. Są to jednak rozwiązania drogie, niedoskonałe i najczęściej spotykane w niektórych kinach. Przykładem mogą być tu kina wykorzystujące technologię IMAX.

Grafika 2D wykorzystywana jest najczęściej do zadań, w których przed powstaniem komputerów wykorzystywano drukowanie i rysowanie. Można tu więc wskazać na kreślarstwo, kartografię czy proste filmy animowane. Z oczywistych też względów grafika 2D tak naprawdę nie zawiera żadnych informacji o tym, co dzieje się w głębi obrazka. Innymi słowy, jeśli na obrazku dwuwymiarowym obiekt A przykryje obiekt B, to w zasadzie jest to równoważne z faktem, że obiektu B w ogóle na obrazku nie ma. Pewnym sposobem obejścia tego ograniczenia jest wykorzystanie warstw, które swoim działaniem nawiązują do nakładających się na siebie folii, z których każda niesie informacje o kolejnych planach uwidocznionych na obrazie. Jakkolwiek jest to rozwiązanie genialne w swojej prostocie, to jednak nie można go utożsamiać z grafiką trójwymiarową. Warstwy, to nic innego, jak dwuwymiarowe obrazy, więc obracając je względem osi obrazka, nie powinniśmy się spodziewać, że dostrzeżemy przedstawiane na nich obiekty z innej perspektywy. Aby to było możliwe, należy skorzystać z możliwości, jakie niesie ze sobą grafika 3D.

Grafika trójwymiarowa wykorzystywana jest we wszystkich tych sytuacjach, w których grafika płaska jest niewystarczająca i w związku z tym należy

się spodziewać, że związane z nią pojęcia będą bardziej skomplikowane niż w przypadku grafiki 2D.

Geometria obiektów trójwymiarowych może być reprezentowana na różne sposoby. Pierwszy z nich opiera się na wykorzystaniu tak zwanych vokseli, które są trójwymiarowymi odpowiednikami pikseli, czyli sześciennymi kostkami o przypisanej im barwie. Tego typu reprezentacja jest szczególnie rozpowszechniona w zastosowaniach medycznych, w których ciało pacjenta jest skanowane i odwzorowywane w pamięci komputera.

Drugi sposób opisywania grafiki trójwymiarowej oparty jest na wykorzystaniu siatek wielokątów. Siatki takie najczęściej zbudowane są z trójkątów i czworokątów, które mają wspólne wierzchołki i krawędzie. Jest to więc jeden z trójwymiarowych sposobów na wykorzystanie grafiki wektorowej. Z tego powodu możliwe jest dowolne powiększanie i zmniejszanie szczegółów siatki bez utraty jakości, a co za tym idzie, zdolny grafik jest w stanie przy pomocy stosownego oprogramowania stworzyć siatkę modelującą niemal dowolny kształt przy pomocy użycia tysięcy wielokątów.

Ostatni sposób opisu geometrii opiera się na wykorzystaniu formuł matematycznych opisujących nieskomplikowane figury geometryczne takie jak kule czy stożki.

Obecnie duże znaczenie w grafice 3D ma realizm tworzonych obrazów. Uzyskuje się go na wiele bardzo różnych sposobów związanych z modelowaniem oświetlenia i nakładaniem tekstur na siatki modelujące obiekty. Warto tutaj wspomnieć, że mimo upływu 16 lat od powstania wspomnianego już w tym rozdziale filmu „Jurrasic Park” nie zmniejszyła się złożoność algorytmów komputerowych pozwalających na fotorealistyczne modelowanie scen podobnych do tych, które w tym filmie się pojawiły. Przez ten czas wzrosła oczywiście moc obliczeniowa komputerów, ale jak wykazano w rozdziale o złożoności obliczeniowej, nie musi i nie przekłada się to na liniowy wzrost szybkości tego typu oprogramowania. Z tego powodu techniki takie jak raytracing czy radiosity wykorzystywane są głównie do tworzenia filmowych efektów specjalnych. Dzieje się tak dlatego, że tutaj na stworzenie jednej klatki można trochę poczekać, bo od filmu nie wymaga się interakcji, tylko wysokiej jakości.

W przypadku gier komputerowych takie podejście jest niepraktyczne. Nawet najlepsza gra komputerowa, w której trzeba odczekać kilka minut pomiędzy kolejnymi klatkami animacji, nie znalazłaby pewnie zbyt wielu nabywców. Z tego powodu ogranicza się w nich szczegółowość, stawiając na wydajność procesu generowania kolejnych klatek animacji. W dzisiejszych czasach w celu uzyskania realizmu w trójwymiarowych grach komputerowych stosuje się technikę wyznaczania oświetlenia dla każdego piksela osobno. Jest to operacja wymagająca ogromnej ilości obliczeń. Dzięki temu jednak, że firmy produkujące karty graficzne prześcigają się w konstruowaniu coraz doskonalszych układów specjalizujących się w tego typu obliczeniach, całą pracę przerzuca się zwykle na nie, odciążając w ten sposób procesor. Układy tego typu znane są też pod nazwą Pixel Shader.

Na zakończenie warto przypomnieć, że niemal 80% informacji dociera do nas przez oczy. Wzrok jest więc niejako naszym najszerszym kanałem pobierania informacji. Każdy kanał ma jednak swoje ograniczenia i ilość informacji, jakie można przy jego pomocy przesłać, zależy w ścisły sposób od optymalizacji jego wykorzystania. Jest wiele prawdy w przysłowiu, że jeden obraz wart jest tysiąca słów. Od drukowania słów to wszystko się zaczęło. Teraz otaczają nas płaskie obrazy. Nic jednak nie wskazuje na to, by miało to być ostatnie słowo w dziedzinie grafiki komputerowej. Ciekawe, jak wtedy zmieni się wspomniane przysłowie.

12

Komputerowy dźwięk

Blisko 80% informacji, które docierają do naszej świadomości zawdzięczamy wzrokowi. Ta porażająca większość obrazuje w jakimś stopniu dramat ludzi, których wzrok się pogarsza lub z jakichś powodów zanika. Niektórzy ludzie mają jeszcze inny problem wynikający z tego, że dotknęła ich synestezja. Właściwie nie ma jednej opinii, czy jest to choroba, czy stan umysłu. Nie ma też jednego zdania na temat tego, czy dotknięci nią ludzie w jakiś sposób cierpią. Synestetyk może doświadczać bodźców odbieranych jednym zmysłem przez drugi zmysł. Z tego powodu możliwe jest dla nich na przykład dostrzeganie barwnych plam w czasie dotykania przedmiotów charakteryzujących się taką, a nie inną fakturą.

Skoro może się zdarzyć, że człowiek nie będzie mógł liczyć na swój wzrok w kontakcie z komputerem, to pojawia się pytanie, w jaki sposób można mu to wynagrodzić. Zacięci fani gier komputerowych znają z pewnością wibrujące joysticki, kierownice, a nawet kamizelki, które stymulują na ciele gracza wrażenia zbliżone do tych, które wynikają z działań podejmowanych w grach akcji. Widać z tego, że możliwe jest komunikowanie się komputera z użytkownikiem za pomocą dotyku. Niestety, generowane w ten sposób sygnały są mało precyzyjne i zbyt wolne, by dało się je wykorzystać do czegoś więcej niż ciekawy dodatek do rozrywki.

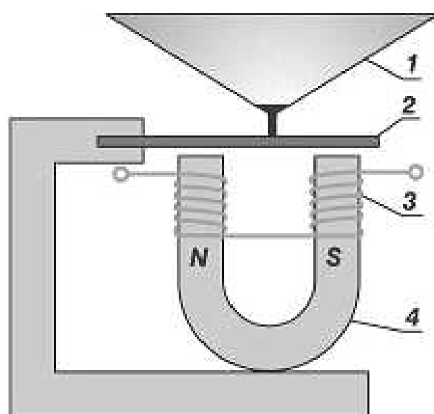
Na razie nie ma jeszcze chyba żadnych urządzeń komputerowych stymulujących smak i węch, chociaż biorąc pod uwagę odrażającą zawartość niejednej gry, należy się chyba z tego bardziej cieszyć, niż nad tym ubolewać. Wygląda jednak na to, że zmysły te mają zbyt dużą inercję, by można je było wykorzystywać do komunikacji z komputerem. Znana jest na przykład zasada, że nie powinno się naraz wąchać więcej niż trzech rodzajów perfum, bo potem i tak się nie wyczuje różnicy. Podobnie jest w przypadku niektórych potraw. Przykładem może być sushi, które podaje się zwykle w towarzystwie marynowanego imbiru, który zabija smak poprzedniego kęsa zanim spróbujemy czegoś nowego.

Ostatnim zmysłem, jakim dysponuje większość populacji, jest słuch. Źródła bodźców, które potrafi on interpretować, mają to do siebie, że dla naszego

dobra powinny się znajdować z daleka od naszych uszu. Oprócz tego nawet średnio wprawne ucho jest w stanie wychwycić i rozróżnić wiele różnych dźwięków w stosunkowo krótkim czasie.

Odkąd w XIX wieku Edison uzyskał patent na swój fonograf, sposób tworzenia dźwięku bardzo się zmienił, ale w swej istocie pozostał taki sam. I wtedy, i dziś dźwięk powstaje dzięki bardzo szybko wibrującemu elementowi, który tworzy fale dźwiękowe naśladujące ich prawdziwe źródło. Wtedy dźwięk był bardzo cichy i marnej jakości, dziś może być dowolnie głośny, a od oryginału potrafią go odróżnić często jedynie wyjątkowi audiofile.

Budowę pierwszego głośnika stworzonego według obowiązującej do dziś koncepcji przedstawia poniższy rysunek 12.1.



Rysunek 12.1.

Jedynką oznaczono na nim papierową stożkową membranę, która była wprawiana w drgania przy pomocy sprężystej blaszki stalowej oznaczonej numerem dwa. Blaszka ta bywała z kolei wprawiana w ruch przez elektromagnes zbudowany z magnesu stałego (3) i nawiniętej na niego cewki (4), zaś całość zasilał prąd elektryczny niosący informacje z mikrofonu lub sporządzonego wcześniej nagrania.

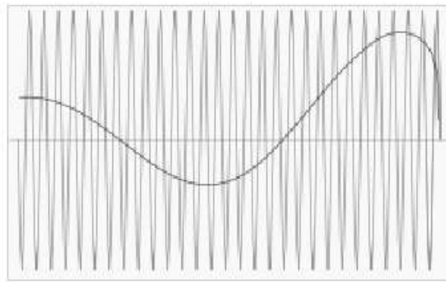
Przez lata od upowszechnienia się tej konstrukcji głośniki zmieniły się tak bardzo jak silniki odrzutowe. Rozkręcając nowoczesny głośnik nawet marnej jakości, zobaczymy w nim sporo elektroniki, a nierzadko również bardzo delikatnych i precyzyjnie wykonanych części z wymyślnych materiałów. Inny też będzie kształt membrany, a sposób doboru jej kształtu często jest wynikiem długoletnich badań i bywa objęty tajemnicą przemysłową. I tylko idea działania wciąż pozostaje ta sama.

W poprzednich rozdziałach wspomniano już o tym, że membrana głośnika drga pod wpływem elektromagnesu zasilanego prądem ze źródła dźwięku. By-

ła to jednak zdawkowa informacja, która teraz zostanie rozwinięta ze szczególnym uwzględnieniem możliwości dzisiejszych komputerów. Niemal każdy z dzisiejszych komputerów domowych wyposażony jest w urządzenie zwane kartą muzyczną lub dźwiękową. Urządzenia tego typu służą do rejestracji, przetwarzania i odtwarzania dźwięku. Obecnie większość kart dźwiękowych wystaczających do zastosowań nieprofesjonalnych jest integrowana z płytą główną.

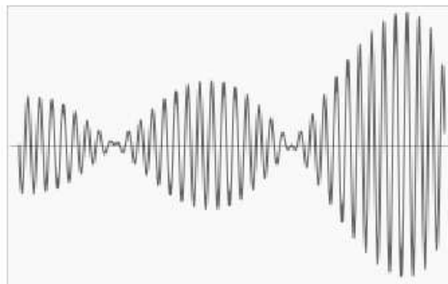
Starsze karty dźwiękowe wyposażone były w generatory dźwięku i szumu, które składane były jako sygnał wyjściowy. Dźwięk był tworzony przy użyciu modulacji amplitudy (generatory AM) lub przy pomocy modulacji częstotliwości.

Modulacja amplitudy - polega na zakodowaniu sygnału informacyjnego o małej częstotliwości przy pomocy chwilowych zmian amplitudy sygnału nośnego. Na poniższym rysunku 12.2. zaznaczona środkowa linia przedstawia sygnał, który chcemy przekazać, a linia sinusoidalna to obraz sygnału nośnego. Jak widać, sygnał nośny ma stałą amplitudę, czyli odległość „szczytu" lub „dna" wykresu od jego przecięcia z osią poziomą.



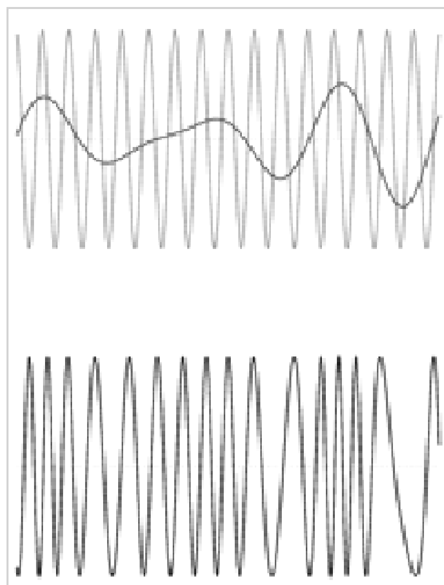
Rysunek 12.2.

Na drugim rysunku 12.3. widać już sposób, w jaki interesujący nas sygnał jest przybliżany za pomocą sygnału nośnego.



Rysunek 12.3.

Drugi typ modulacji sygnału polega na zmienianiu częstotliwości sygnału nośnego w taki sposób, by oddać informację zawartą w sygnale nadawanym. Tu także sygnał nośny oznaczono sinusoidą, sygnał nadawany linią biegnącą przez środek, a sygnał wyjściowy zaznaczono w dolnej części wykresu. Zależności graficzne między całą tą trójką prezentuje rysunek 12.3.



Rysunek 12.4.

W czasach, w których nikogo już nie dziwi telefonia internetowa, montowanie w kartach dźwiękowych przetworników AĆ (analogowo-cyfrowych) i CĄ (cyfrowo-analogowych) nikogo już nie dziwi. Pierwsze z nich umożliwiają łatwe wprowadzanie dźwięku do pamięci komputera, a drugie zamieniają informacje zapisane w pamięci na stosowny sygnał wysyłany do głośników.

Coraz mniejszą fanaberią jest też mikser dźwięku, czyli urządzenie służące do łączenia sygnałów dźwiękowych z różnych źródeł takich jak: przetworniki C/A, generatory dźwięku, wejścia zewnętrzne i inne. Nie można też zapominać o wciąż bardzo popularnym interfejsie MIDI i interfejsie, przez który karta dźwiękowa komunikuje się i wymienia dane z komputerem.

Wykorzystanie całego tego sprzętu wymaga oczywiście zastosowania różnego rodzaju technik, których krótkie charakteryzację, zostaną umieszczone poniżej.

Pierwszą z nich jest próbkowanie, powszechniej znane pod swoją angielską nazwą, jaką jest sampling. Proces ten polega na przybliżaniu funkcji opisującej przebieg dźwięku przy pomocy funkcji schodkowej. Dzięki takiemu podejściu

nie musimy znać pełnej, najczęściej bardzo skomplikowanej funkcji opisującej wahania dźwięku. Wystarczy znać rozmiary schodka, które da się opisać w postaci numerycznej. Starsze karty wykorzystywały w tym celu liczby 8-bitowe, co pozwalało na przechowywanie jedynie 256 wartości i dawało dźwięk wątpliwej jakości. Nowsze wykorzystują liczby 16-bitowe, co skutkuje możliwością zapisania 65536 różnych wartości dla każdego kanału stereo, dzięki czemu generowany dźwięk ma już naturalne brzmienie o jakości hi-fi.

Oprócz ilości bitów opisujących pojedynczą próbkę niezwykle ważnym parametrem jest częstotliwość próbkowania. Im jest ona wyższa, tym mniejszy jest czas, który upływa pomiędzy pobraniami próbek i tym lepiej jest dla jakości dźwięku. Jak dużo tych próbek trzeba wziąć, by uzyskać zadowalający efekt, niech zilustruje fakt, że dla przesłania rozmowy telefonicznej potrzeba ich ok. 8000 w sekundzie (częstotl. 8 kHz), a dla jakości porównywalnej z płytą CD potrzeba ich w tym samym czasie już 44100 (częstotliwość 44.1 kHz).

Jak już wcześniej wspomniano, karty muzyczne nie tylko potrafią odtwarzać dźwięki, ale również je tworzą. Obecnie realizowane jest to przy pomocy dwóch podejść. Pierwsze z nich znane jest jako synteza FM, a drugie jako synteza wavetable.

Wszystkie układy FM działają na tej samej zasadzie. Starają się opisać za pomocą prostych funkcji matematycznych krzywe drgań, które jednak tylko w przybliżeniu imitują działanie oryginalnych instrumentów muzycznych. To oczywiście rozwiązanie dużo lepsze niż kompletny brak dźwięku, ale warto zauważyć, że to, co jest naturalne, rzadko przejawia matematyczną doskonałość. Zachodzi tu podobne zjawisko jak w przypadku grafiki rastrowej i wektorowej. Grafika wektorowa da się do woli skalować bez utraty jakości, lecz to chropawość i ziarnistość grafiki rastrowej nadaje jej bardziej naturalny wygląd.

Zamiast więc używania matematycznych funkcji opisujących nienaturalne brzmienia, można podejść do sprawy inaczej, wykorzystując syntezę wavetable znaną również pod nazwą Advanced Wave Memory, czyli zaawansowana pamięć fal. Podejście to polega na nagraniu w profesjonalnym studiu dźwiękowym popularnych instrumentów i zgromadzeniu ich w pamięci karty muzycznej, a potem w razie potrzeby odtwarzania ich przez wymagany czas i w stosownej wysokości. Uzyskuje się w ten sposób znacznie lepszą jakość dźwięku niż przy użyciu syntezy FM, ale i tak dalekie jest to od brzmienia naturalnego, a poza tym w pamięci karty dźwiękowej nie da się zgromadzić brzmień wszystkich instrumentów na świecie.

Jednak bez względu na to, czy używamy syntezy FM, czy też wavetable, dobrze by było, byśmy mogli używać tego w wygodny, najlepiej jednolity dla wszystkich przypadków sposób. Urządzeniem, które pozwala na taki komfort, jest tzw. interfejs MIDI, czyli cyfrowe złącze instrumentów muzycznych. Złącze to pozwala na wymianę informacji i synchronizację sprzętu muzycznego za pomocą standardowych komunikatów.

Warto tutaj zauważyć, że złącze to nie przesyła dźwięku, lecz informację, które go opisują. Może to być na przykład komunikat nakazujący dwusekundowe odtwarzanie dźwięku o pewnej zadanej wysokości wydobywanego z pew-

nej konkretnej gitary. Jak widać, pomysłem przypomina to trochę klasyczny zapis nutowy, ale ma znacznie bogatsze możliwości. Tworzona w ten sposób muzyka może być zapisywana w standardowych plikach z rozszerzeniem MID, których niezaprzeczalną zaletą jest fakt, że w porównaniu z plikami muzycznymi powstałymi w procesie próbkowania są one bardzo małe. Dla przykładu jednominutowy utwór muzyczny zapisany w formacie MIDI mieści się w pliku muzycznym o objętości około 20 kilobajtów, podczas gdy ten sam utwór zapisany w jednym z formatów cyfrowych może mieć nawet kilka megabajtów.

To co w branży komputerowo-informatycznej zachwycało nas kilka lat temu, szybko się dewaluuje. Tutaj lata mijają jak epoki, zaś dekady są czasem wręcz niewyobrażalnym. Nie każdy musi zdawać sobie sprawę z technologicznego zaplecza takiego stanu rzeczy, bo nie każdy jest programistą czy inżynierem. Spora część ludzi, którzy nie należą ani do jednej, ani do drugiej kategorii, to z pewnością użytkownicy gier komputerowych i powinni się oni oddać chwili refleksji. Zastanówmy się przez moment, co tak naprawdę przyciągało nas do gier sprzed lat? Czy była to wspaniała grafika i dźwięk? Nie! Te gry nie mogły się tym obronić. Każda z nich musiała być oparta na ciekawym pomyśle, który niejednokrotnie jest lepszy od idei przyświecających niejednej świeższej produkcji. Koronnymi przykładami tego typu gier są odświeżone wersje niektórych części „Tomb Ridera" czy „The secret of Monkey Island - Special Edition", w której można zobaczyć różnice pomiędzy dawnym a dzisiejszym anturażem tej wspaniałej gry. Ktoś powie - bezczelność i wyłudzanie pieniędzy za odgrzewane kotlety, ale można to również uznać za świetny pomysł, bo czymże innym są remaki filmów?

W tym co napisano, pobrzmiewa z pewnością lekka nutka nostalgii za czasami, które już nie powrócą, ale ważniejsza jest tutaj inną myśl, którą uparcie promujemy: to nie sprzęt jest najważniejszy, a to co ożywia wychodzące z niego ciągi zer i jedynek. A tym czymś niezmiennie pozostają pomysłowe algorytmy i nasza niczym nieskrępowana wyobraźnia.

13

Internet

Żyjemy otoczeni wieloma przedmiotami, które powstały dlatego, że jedni ludzie chcieli zrobić wrażenie na drugich, niespecjalnie ich lubili, albo chociaż im nie ufali. Tak się jakoś składa, że największym motorem postępu cywilizacji jest wyścig zbrojeń i zapewniające nieustający efekt propagandowy próby wysłania człowieka w dalsze rejony kosmosu.

Jeszcze w trakcie trwania II wojny światowej USA i ówczesny ZSRR, czyli Związek Socjalistycznych Republik Radzieckich przestały darzyć się sympatią. Po zakończeniu wojny ta niechęć przybrała na sile, co w konsekwencji doprowadziło do rozpoczęcia okresu zwanego źimną wojną". W owych czasach wszyscy już wiedzieli, jak wielkim zagrożeniem jest broń nuklearna, ponieważ bombardowania Hiroshimy i Nagasaki były jeszcze stosunkowo świeżymi wydarzeniami. Mimo to oba ówczesne supermocarstwa produkowały bez opamiętania kolejne bomby tylko po to by mieć ich więcej niż konkurencja. I do nikogo nie docierał fakt, że w pewnym momencie światowy arsenał jądrowy wystarczyłby do spopielenia całej Ziemi. Co gorsza, rywalizacja między tymi dwoma państwami z czasem zaczęła przybierać inne formy i przeniosła się w przestrzeń kosmiczną. W 1957 roku ZSSR wystrzelił w kosmos Sputnika - pierwszego sztucznego satelitę Ziemi. W odpowiedzi na to rząd USA powołał organizację ARPA (Advanced Research Projects Agency), której zadaniem było obserwowanie i wspieranie inicjatyw powstających na uczelniach w USA, które miały szczególne znaczenie dla obronności Stanów Zjednoczonych.

Jak wiadomo, w czasie wojny niszczy się wrogowi w pierwszej kolejności to, co ułatwia jej prowadzenie. Są to zakłady zbrojeniowe, skupiska wojsk, centra dowodzenia i system łączności. W tamtych czasach łączność można było prowadzić głównie przy pomocy telefonii naziemnej i radiostacji. Oba te rozwiązania w obliczu zagrożenia atomowego miały poważne wady. W przypadku telefonii wadą było to, że niszcząc wyłącznie centralę telefoniczną, unieruchamiało się łączność na ogromnym obszarze. Jeżeli zaś chodzi o łączność radiową, to należało się liczyć z tym, że w obliczu skażenia radioaktywnego utrzymującego się na obszarze wybuchu byłaby ona mocno utrudniona. Z tego powodu

coraz częściej zaczęto mówić o konieczności znalezienia jakiejś alternatywy dla tych rozwiązań.

W roku 1964 Paul Baran z RAND Corporation publikuje raport „On Distributed Communications Networks”, w którym opisuje pomysł zdecentralizowanej sieci komputerowej, która może działać nawet w przypadku awarii wielu jej węzłów. W 1967 roku ARPA zorganizowała konferencję naukową na temat możliwości budowy rozległych sieci komputerowych o rozproszonym zarządzaniu. Najciekawsze rozwiązanie wskazał wtedy Alex McKenize z Uniwersytetu Stanforda. Zaproponował on, aby informacje rozchodziły się w sieci w formie pakietów z przypisanymi adresami, które krążyłyby po sieci, szukając adresata tak jak robią to tradycyjne listy. Pomysł ten okazał się na tyle chwytliwy oraz sensowny, że ARPA zdecydowała się wesprzeć jego realizację i w roku 1968 zespół pod kierownictwem Alexa McKenize stworzył podstawy objętego wówczas tajemnicą protokołu TCP i dokonał prezentacji automatycznego wyszukiwania połączenia w sieci łączącej kilkanaście serwerów rozproszonych na trzech uniwersytetach. W 1971 roku ARPA zdecydowała się na odtajnienie protokołu TCP i wydała zezwolenie na poszerzenie sieci o kolejne sieci akademickie. Przez kilka kolejnych lat ARPAnet pozostawał pod ścisłym nadzorem wojskowym, aż w końcu w 1980 roku wydarzyło się coś, co stało się kolejnym przełomem. Tym faktem była seria efektownych crackerskich włamań na serwery ARPAnetu, po której zdecydowano się na całkowite oddzielenie części wojskowej od części akademickiej, która od tamtej pory nazywana jest wymyślonym przez Vintona Cerfa słowem Internet.

To tyle tytułem tła historycznego. Zastanówmy się teraz, jak to się dzieje, że Internet w ogóle działa. W jaki sposób dogadują się ze sobą miliony komputerów i telefonów komórkowych? U podstaw działania całej tej niewyobrażalnie złożonej sieci leżą tzw. protokoły internetowe, z których jeden (TCP) został wymieniony już wcześniej.

Po pierwsze: każde urządzenie podpięte do Internetu musi być w jakiś sposób identyfikowane (im prościej, tym lepiej). W tym celu przyjęło się używać między innymi adresów IP, które są niczym innym jak sprytnym sposobem numerowania węzłów Internetu.

Adres IP nie jest czymś w rodzaju tablicy rejestracyjnej urządzenia. Z tego powodu, nie musi być na stałe przypisany do komputera czy telefonu komórkowego. Co więcej, taka sytuacja to rzadkość i najczęściej dotyczy serwerów udostępniających popularne serwisy internetowe.

W czasie, gdy powstawał ten tekst, najpopularniejszym sposobem nadawania adresów IP był ten związany z protokołem IPv4. Adresy stworzone w oparciu o ten protokół są 32-bitowymi liczbami całkowitymi. Taka liczba bitów pozwala się ustawić na ponad 4 miliardy sposobów, co początkowo było liczbą wystarczającą. Coraz częściej słyszy się jednak opinie, że to za mało i należy się przygotować na nadejście nowszego protokołu o nazwie IPv6. Dla przykładu: w chwili obecnej strona www.pjwstk.edu.pl ma przypisany adres IP równy 2488372496. Liczba ta jest spora i trudna do zapamiętania. Z tego powodu zwykle zapisuje się ją w postaci szesnastkowej i grupuje po dwie cyfry,

oddzielając je dwukropkami. W tym przypadku adres ten to 94:51:8D:10. Jak widać, jest trochę prościej, ale dla większości osób system szesnastkowy to i tak zbyt wysoki stopień abstrakcji. Dlatego adresy IP znacznie częściej podaje się w postaci zapisu dziesiętnego. I tak 94:51:8D:10 zamienia się wtedy w 148.81.141.16. Powyższy numer ma dziesięć cyfr, czyli niewiele więcej, niż numery telefonów komórkowych. Na upartego więc, ludzie mogliby używać ich na przykład do pobierania zawartości ze zdalnych komputerów. Nawiasem mówiąc, jest to możliwe. Aby obejrzeć stronę PJWSTK, w okienku adresu można wpisać właśnie powyższy adres IP. To jednak jest niezbyt wygodne, choć znacznie prostsze niż pamiętanie liczby 2488372496. W tym miejscu przychodzi już jednak czas na coś zupełnie innego.

Chcąc wpisać w okienku przeglądarki adres: `www.pjwstk.edu.pl` i zobaczyć stronę, której się spodziewamy, potrzeba czegoś, co będzie umiało zamienić taki słowny adres na adres IP lub inną wartość pozwalającą zidentyfikować urządzenie, z którym chcemy się połączyć. Owo coś to tzw. DNS (ang. *Domain Name System* - System Nazw Domenowych). Na DNS składa się światowa sieć serwerów i protokół pozwalający na komunikowanie się między sobą serwerów i klientów, czyli przeglądarek, gier sieciowych itp.

Każdy z serwerów DNS zawiera spis łatwych do zapamiętania adresów i odpowiadających im identyfikatorów urządzeń internetowych, przy czym identyfikatory te to najczęściej właśnie adresy IP. Dzięki takiemu podejściu użytkownik nie musi już pamiętać ciągu cyfr i kropek, bo zanim dostanie się do jakichś zdalnych zasobów, podany przez niego adres i tak zostanie przerobiony na coś znacznie bardziej zrozumiałego dla komputerów.

Jakkolwiek dzisiejsze przeglądarki internetowe pozwalają nam na wpisanie w okienku adresu czegokolwiek, to jednak jest to ukłon w stronę ich użytkowników. Dla przykładu, wpisując w okienku adresu Firefoxa śtrona Polsko-Japońskiej Wyższej Szkoły Technik Komputerowychźostaniemy, o dziwo, przekierowani na zamierzoną stronę, ale to bardziej wyjątek niż reguła. W sytuacji, w której przeglądarka nie znajdzie na serwerze DNS wymaganego adresu, będzie się starała wyświetlić wyniki wyszukiwania wpisanej frazy przeprowadzone z użyciem jednej z wyszukiwarek internetowych. Język ludzki bywa jednak na tyle wieloznaczny, że można nie znaleźć tego, co się zamierzyło. Tytułem anegdoty można przytoczyć historię pewnej osoby, która pisała pracę naukową z historii dawnych instrumentów muzycznych w Polsce i po wpisaniu w oknie wyszukiwarki frazy „organy w Oliwie", w pewnym sensie dostała poprawną odpowiedź.

Jak więc należy sobie poradzić z tym problemem? Odpowiedzią po raz kolejny jest standaryzacja. W tym przypadku jest nią ujednolicony format adresowania zasobów znany częściej ze swojej skrótowej nazwy URL (Uniform Resource Locator). Poprawny URL składa się z części określającej rodzaj zasobu/usługi, dwukropka i części zależnej od rodzaju zasobu. Najpopularniejsze rodzaje zasobów to: http, https, ftp, mailto i file, ale jest ich znacznie więcej. Jeżeli chcemy się dostać do jakiegoś pliku na zdalnym serwerze, to jego URL będzie zbudowany według zasady

`rodzaj_zasobu/adres_serwera:port/sciezka_dostepu`.

13.1 Przykład

Chcąc złożyć w dziekanacie podanie drogą elektroniczną, możemy użyć następującego URL:

`https:/podania.pjwstk.edu.pl:443/login.aspx`

gdzie:

`https:` - to rodzaj zasobu,
`podania.pjwstk.edu.pl` - to adres serwera,
`443` - to numer portu (w tym wypadku jest niepotrzebny, bo 443 to standardowy numer portu dla zasobów rodzaju https),
`login.aspx` - to ścieżka dostępu do konkretnego pliku.

Czasem bywa i tak, że jakieś zasoby zabezpieczone są hasłem. Wtedy podaje się je według formatu

`rodzaj_zasobu/nazwa_uzytkownika:haslo@adres_serwera/sciezka_dostepu`.

Jeżeli natomiast musimy opisać URL czegoś innego niż plik (np. adresu email), to tworzymy go według formatu

`nazwa_uzytkownika@adres_serwera`.

Wiemy już teraz, w jaki sposób uporządkowana jest zawartość Internetu, ale najważniejsze pytanie, jak to wszystko działa, pozostaje jeszcze bez odpowiedzi. Czas więc to teraz zmienić i napisać kilka słów więcej na temat wspomnianego już kilkakrotnie protokołu TCP.

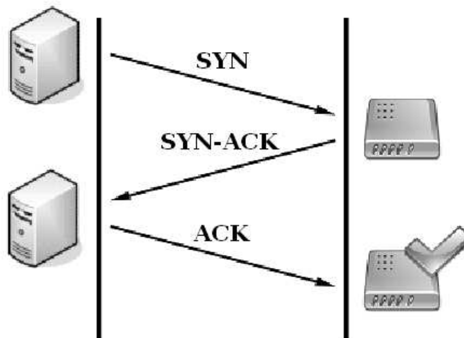
TCP (ang. Transmission Control Protocol - protokół kontroli transmisji) to protokół określający sposób komunikacji między dwoma komputerami. Został stworzony przez Vintona Cerfa i Roberta Kahna. Jest to protokół działający w trybie klient-serwer. Serwer oczekuje na nawiązanie połączenia na określonym porcie. Klient inicjuje połączenie z serwerem, zaś potem oba te węzły wymieniają się ze sobą informacjami w postaci tzw. pakietów. Każdy pakiet w tym protokole oprócz danych, które są przesyłane, zawiera jeszcze nagłówek, który zwiększa ilość przesyłanych danych. Bywa jednak i tak, że w przypadku odrzucenia pakietu trzeba go nadać ponownie. Są to niewątpliwe wady tego protokołu, niemniej dzięki tym dodatkowym informacjom uzyskujemy gwarancję, że o ile nie wystąpi awaria lub przerwanie połączenia, wszystkie wysłane pakiety trafią do adresata.

We wspomnianym nagłówku przesyłane są między innymi informacje na temat tego, co ma się stać z pakietem. Informacje te zwane są flagami i występują w ośmiu odmianach, z czego interesować nas będą teraz cztery:

ACK - gdy jej wartość to 1, informuje o istotności zapisanych w nagłówku informacji o numerze potwierdzenia,
RST - gdy jej wartość to 1, następuje reset połączenia,
SYN - gdy jej wartość to 1, następuje synchronizacja kolejnego numeru sekwencyjnego,
FIN - gdy jej wartość to 1, następuje zakończenie przekazu danych.

Z wykorzystaniem tych jednobitowych informacji nawiązuje się połączenie między klientem a serwerem. Procedura, która temu towarzyszy, jest charakterystyczna dla protokołu TCP i z angielska zwana jest „*three-way handshake*", czyli trójstronny uścisk dłoni. Załóżmy, że mamy komputer, który chce się połączyć z konkretnym serwerem w Internecie. Komputer wysyła w tym celu do serwera pakiet TCP z ustawioną flagą SYN. Serwer decyduje, czy chce obsłużyć takie połączenie i jeżeli tak, to odsyła do naszego komputera pakiet z ustawionymi flagami SYN i ACK. W tym momencie nasz komputer powinien wysłać pierwszą porcję danych, ustawiając już tylko flagę ACK. Po takich korowodach może się zdarzyć, że serwer nie będzie mógł lub chciał odebrać takiego połączenia i wtedy odpowie pakietem z ustawioną flagą RST. Jeżeli natomiast wszystko zakończy się powodzeniem, to rozpoczęta zostanie transmisja danych, która zakończy się wysłaniem pakietu z ustawioną flagą FIN, którą trzeba będzie potwierdzić wysłaniem pakietu z ustawioną flagą ACK.

Przykład podobnego nawiązania połączenia prezentuje rysunek 13.1.



Rysunek 13.1.

Warto w tym momencie przypomnieć, że podobnie jak w komputerze niepodłączonym do sieci, tak samo w Internecie wszelkie dane rozprzestrzeniają się w postaci liczb zapisanych w systemie dwójkowym. To, w jaki sposób zostaną one zinterpretowane, zależy w dużej mierze od tego, czego się po takich danych spodziewamy.

W części poświęconej formatowi URL wspomniano już o istnieniu różnych zasobów internetowych. Czas nieco rozwinąć ten wątek i rzucić nieco więcej światła na te typy, które zachowały swoją popularność do dnia dzisiejszego.

W dzisiejszych czasach dzielenie się plikami nie jest niczym nowym i najczęściej niczym legalnym. Zupełnie odwrotna sytuacja miała miejsce w roku 1970, kiedy powstał protokół ftp (ang. File Transfer Protocol - protokół transferu plików). Jak sama nazwa wskazuje, protokół ten został zaprojektowany, by wymieniać się danymi pod różnymi postaciami. Prawdopodobnie dlatego został pomyślany jako protokół ośmiobitowy. Znaczy to tyle, że każdy bajt danych, które chcemy wysłać lub pobrać, jest przesyłany w swojej oryginalnej postaci.

Protokół ten do komunikacji wykorzystuje dwa połączenia TCP. Pierwsze z nich jest połączeniem kontrolnym i wysyła się za jego pomocą na przykład polecenia dla serwera. Drugie połączenie wykorzystywane jest w całości do przesyłania danych.

Inną bardzo popularną aktywnością sieciową jest wysyłanie poczty elektronicznej nazywanej potocznie mailem lub emailem. Dziś robi to każdy, nieraz nawet dziesiątki razy dziennie i na nikim nie robi to wrażenia. Dlatego z pewną zazdrością można podejść do człowieka, który zrobił to jako pierwszy i przeszedł w ten sposób do historii, a był nim niejaki Ray Tomlinson. Korespondencja internetowa jest dziś prowadzona z wykorzystaniem kilku protokołów, z których najpopularniejsze to SMTP, POP oraz IMAP.

SMTP (ang. *Simple Mail Transfer Protocol* - prosty protokół transferu poczty) jest względnie prostym, tekstowym protokółem, w którym określa się co najmniej jednego odbiorcę wiadomości (w większości przypadków weryfikowane jest jego istnienie), a następnie przekazuje treść wiadomości. Do wad tego protokołu należy zaliczyć przede wszystkim to, że radził on sobie słabo z przesyłaniem plików zawierających dane binarne, czyli właściwie wszystko poza tekstem. Aby obejść ten problem, wymyślono standard MIME (ang. *Multipurpose Internet Mail Extensions* - rozszerzenia poczty internetowej wielorakiego zastosowania). Standard ten koduje zawartość plików przy pomocy nagłówka określającego ich typ i treści, w której oryginalną zawartość pliku koduje się przy pomocy siedmiobitowego zestawu znaków. W dzisiejszych czasach większość serwerów SMTP obsługuje rozszerzenie 8BITMIME pozwalające przesyłać pliki binarne równie łatwo jak tekst. Drugą nie mniej ważną wadą tego protokołu był fakt, że nie pozwalał on pobieranie wiadomości ze zdalnego serwera. Najważniejszym zaś problemem związanym z tym protokołem jest brak mechanizmu weryfikacji nadawcy, co nawet dla człowieka o bardzo małej wiedzy informatycznej otwierało furtkę do przeróżnych nadużyć od podszywania się pod innych nadawców, po rozsyłanie wirusów i złośliwego oprogramowania. Żeby temu zaradzić, stworzono rozszerzenie SMTP-AUTH, które jest jednak tylko częściowym rozwiązaniem problemu. Dzieje się tak dlatego, że ogranicza ono wykorzystanie serwera wymagającego autoryzacji do zwielokrotniania poczty. Nadal nie istnieje metoda, dzięki której odbiorca mógłby w jednoznaczny sposób autoryzować nadawcę, przez co nadawca może "udawaćśerwer i wysłać dowolny komunikat do dowolnego odbiorcy.

Inne podejście prezentuje protokół POP (obecnie w wersji trzeciej, co oznacza się jako POP3). Protokół ten powstał z myślą o ludziach, którzy nie mają

stałego dostępu do Internetu. Jeżeli ktoś łączy się z siecią tylko na chwilę, to poczta nie może dotrzeć do niego z wykorzystaniem protokołu SMTP. W takiej sytuacji w sieci istnieje specjalny serwer, który przez SMTP odbiera przychodzącą pocztę i ustawia ją w kolejce. I kiedy tylko użytkownik połączy się z siecią, czekające na niego listy zostaną mu dostarczone z wykorzystaniem opisywanego protokołu. Wydaje się to idealnym rozwiązaniem, ale niestety ma sporo ograniczeń i niedogodności. Po pierwsze: połączenie trwa tylko, jeżeli użytkownik pobiera pocztę i nie może pozostać uśpione. Po drugie: do jednej skrzynki pocztowej może się podłączyć tylko jeden program (klient) równocześnie. Po trzecie: wszystkie odbierane listy trafiają do jednej skrzynki i nie ma sposobu, by utworzyć ich więcej. Po czwarte: serwer POP3 nie potrafi sam przeszukiwać czekających w kolejce listów, co mogłoby ograniczyć ilość niechcianej poczty jeszcze przed jej pobraniem. Ostatnie ograniczenie polega zaś na tym, że każdy list musi być pobierany razem z załącznikami i żadnej jego części nie można w łatwy sposób pominąć. Z tego powodu zmuszani jesteśmy często do pobrania całej zawartości maila, którego treść jest pusta, a załączniki obfitują w mało śmieszne filmiki lub głupawe łańcuszki szczęścia.

Rozwiązaniem tych problemów okazał się coraz popularniejszy protokół IMAP (ang. *Internet Message Access Protocol* - protokół dostępu do wiadomości internetowych). W przeciwieństwie do POP3, który umożliwia jedynie pobieranie i kasowanie poczty, IMAP pozwala na zarządzanie wieloma folderami pocztowymi oraz operowanie na listach znajdujących się na zdalnym serwerze. Niezaprzeczalną zaletą tego protokołu jest fakt, że pozwala on na ściągnięcie nagłówków wiadomości i wybranie, które z nich chcemy ściągnąć na komputer lokalny.

W 1945 roku Vannevar Bush opublikował w czasopiśmie Atlantic Monthly artykuł pod tytułem „As We May Think”, w którym przedstawił idee leżące u podstaw hipertekstu. Wtedy było to tylko mgliste marzenie, które musiało odczekać ładnych parę lat na pojawienie się technicznych możliwości jego realizacji. Dziś z hipertekstu korzystają niemal wszyscy użytkownicy Internetu, a wspierają ich w tym dwa protokoły: HTTP (ang. *Hypertext Transfer Protocol* - protokół przesyłania dokumentów hipertekstowych) i jego szyfrowana wersja, czyli HTTPS. Jest to protokół sieci WWW (ang. *World Wide Web*), a jego użyteczność wynika z udostępnienia znormalizowanego sposobu komunikowania się komputerów ze sobą nawzajem. Określa on też formę żądań dotyczących danych oraz formę, w jakiej ma zostać udzielona odpowiedź serwera. Za pomocą protokołu HTTP przesyła się żądania udostępnienia dokumentów WWW i informacje o kliknięciu odnośnika oraz informacje z formularzy. Zadaniem stron WWW jest publikowanie informacji, co da się osiągnąć właśnie z użyciem protokołów HTTP i HTTPS.

Przy całej swojej wspaniałości protokół ma też jedną cechę, która jest jednocześnie i zaletą i wadą. Jest on mianowicie zaliczany do protokołów bezstanowych (ang. *stateless*), ponieważ nie zachowuje żadnych informacji o poprzednich transakcjach z klientem, co znaczy, że po zakończeniu połączenia wszystkie dane, które go dotyczyły, po prostu przepadają. Pozwala to znacz-

nie zmniejszyć obciążenie serwera, jednak jest kłopotliwe w sytuacji, gdy np. trzeba zapamiętać konkretny stan dla użytkownika, który wcześniej łączył się już z serwerem. Najczęstszym rozwiązaniem tego problemu jest wprowadzenie mechanizmu ciasteczek (ang. *cookies*), czyli małych porcji informacji tworzonych przez przeglądarkę po stronie użytkownika. Możliwe jest też rozwiązanie tego problemu po stronie serwera, co wiąże się z użyciem tzw. sesji, ukrytych parametrów oraz parametrów podawanych w adresie strony (jak np. `/index.php?lang=pl`).

Pomijając ten drobny problem, z którym i tak muszą sobie radzić programiści, a nie przeciętni użytkownicy komputerów, należy uznać, że oba te protokoły okazały się na tyle elastyczne i wygodne w użyciu, że coraz częściej oparte na nich rozwiązania zaczynają wypierać używanie programów opartych na innych protokołach. Warto tu wspomnieć o poczcie Google Mail czy serwisach wymiany plików takich jak rapidshare.com, które mają funkcjonalność zbliżoną do tej, którą udostępnia protokół FTP.

Tym oto sposobem coś, co zostało pomyślane jako tajna część gigantycznej militarnej machiny i było dostępne tylko garstce specjalistów potrafiących to obsłużyć, spowszedniało i trafiło pod strzechy. W dzisiejszych czasach Internet nie służy już do toczenia wojen z wyjątkiem ich wirtualnych odpowiedników w grach sieciowych. Internet stał się miejscem pokojowej wymiany informacji, a jedyna rzecz, nad którą należy ubolewać, to fakt, że spora jej część hołduje najniższym ludzkim instynktom.

14

Gromadzenie i przetwarzanie danych

Jeszcze kilkanaście lat temu Internet nie był tak popularny jak dziś. Każdy miał swoje pliki w swoim komputerze i dzielił się nimi najczęściej z niewielką grupą ludzi. Z powodu ich nieliczności mógł to robić ręcznie, a dane te mogły być ułożone według najbardziej szalonych reguł lub po prostu tworzyć jeden wielki bałagan.

Nieliczni wybrańcy mający dostęp do sieci zachwycali się w owych czasach stronami internetowymi wykorzystującymi niemal wyłącznie statyczne, czyli jednakowe dla wszystkich dane. Z powodu tego, że z Internetem łączono się przy użyciu modemów, które podpięte były do linii telefonicznych, przemierzanie sieci było bardzo drogie, bo płaciło się właściwie nie za ilość pobranych danych, ale za czas połączenia. Dlatego dosyć popularne były programy pozwalające skopiować całą statyczną zawartość serwisu internetowego, tak by można ją było już za darmo przeglądać w domu.

Z czasem w sieci pojawiło się tyle tego typu stron, że po pierwsze ludziom się to opatrzyło, a po drugie pojawiła się potrzeba jakiegoś usystematyzowania tych zasobów. Z tej potrzeby powstał pomysł na dynamiczne strony internetowe. Pomysł ten polegał na tym, by po stronie serwera zamiast gotowego dokumentu zawierającego kompletny opis strony internetowej umieścić program, który interpretowałby dane nadesłane przez użytkownika i w locie tworzył stronę przygotowaną niejako konkretnie dla niego.

Jedną z pierwszych technik, które pozwalały na tego typu interpretacje, było CGI (ang. *Common Gateway Interface*). W dzisiejszych czasach standard ten wyszedł już nieco z mody, ale wciąż ma się całkiem nieźle. Jest stabilny od 1995 roku i oferuje wsparcie w postaci specjalnych bibliotek dla wielu popularnych języków programowania. Niestety, standard ten powstawał w czasach, gdy systemy wielozadaniowe nie były w powszechnym użyciu, a wielowątkowość była jeszcze w powijakach. Z tego powodu obsługa CGI kojarzy się najczęściej z niezbyt gospodarnym podejściem do zasobów serwera. Wynika to z faktu, że obsługa skryptu CGI wiąże się najczęściej z tworzeniem nowego procesu przypisanego do obsługi konkretnego zapytania, podczas gdy inne

techniki delegują do tego zadania pojedyncze wątki. Aby zrozumieć tę rozrzutność, można wyobrazić sobie proces jako dywan, podczas gdy wątek to pojedyncza nitka tego dywanu.

Oprócz tego CGI mają też pewną cechę, która jest jednocześnie i zaletą i wadą zależnie od tego, jak umiejętnie się ją stosuje. Cechą tą jest fakt, że programy CGI nie różnią się praktycznie niczym od pozostałych programów uruchamianych na serwerze. Jeżeli więc uruchamiane są z konta o uprawnieniach administratora, to mogą z tym serwerem zrobić praktycznie wszystko. Jest to bardzo pożądana i użyteczna własność w przypadku, gdy działanie takiego programu zostało od początku do końca bardzo dobrze przemyślane i wytestowane. W pozostałych przypadkach wiąże się z wieloma niebezpieczeństwami, przed którymi można się w pewnym stopniu obronić, stosując dobre praktyki programistyczne.

Najważniejszą z tych praktyk jest dokładne sprawdzanie danych, które mogą zostać dopuszczone do przetwarzania. Chodzi tutaj o dokładną weryfikację pod kątem ich rozmiaru, zgodności typu i zawartości. Wynika to z faktu, że program, który będzie przeznaczony do przetwarzania krótkich danych tekstowych, prawie na pewno nie poradzi sobie z obsługą ogromnego obrazka, powodując powstanie lawiny błędów, które wprawnemu hakerowi wiele powiedzą na temat możliwych do wykorzystania luk w systemie.

Ponieważ programy CGI nie różnią się praktycznie niczym od pozostałych programów uruchamianych na serwerze, są tak samo podatne na występowanie błędów i awarie. Ponieważ jednak, w większości przypadków programy/ skrypty CGI otrzymują oraz przetwarzają dane „z zewnątrz”, bardzo ważne, jest aby sprawdzić dokładnie ich poprawność. Nie ma żadnej gwarancji, że użytkownik strony wypełni wszystkie pola formularza zgodnie z przeznaczeniem, wyśle do serwera poprawne dane lub program/skrypt CGI nie stanie się przedmiotem ataku krakera. Z tych powodów zaleca się stosowanie pewnych reguł podczas pisania programówśkryptów CGI.

Warto też pamiętać o tym, by oprzeć się pokusie nadawania skryptom CGI szerokich możliwości do ingerencji w system. Dlatego nadaje się im możliwie najniższe uprawnienia lub jeśli tylko jest to możliwe uruchamia się je w środowiskach odizolowanych. Dobrym pomysłem jest też umieszczanie wszystkich skryptów w specjalnie wydzielonym dla nich katalogu.

Poniżej zamieszczony jest przykład bardzo prostego skryptu CGI napisanego w języku Perl i przeznaczonego do uruchomienia (najlepiej) pod kontrolą systemu Unix lub systemów do niego podobnych w działaniu.

```
1 #!/usr/bin/perl
2 use CGI;
3 my $zapytanie = new CGI;
4 my $imie = $zapytanie->param('imie');
5 print $zapytanie->header();
6 print "Twoje imie to:", $zapytanie->escapeHTML($imie);
```

Pierwsza linijka powyższego kodu odpowiedzialna jest za uruchomienie interpretera języka Perl. W drugiej sugerujemy programowi, żeby użył specjalnego rozszerzenia języka Perl (biblioteki) wspierającego technologię CGI.

W trzeciej linii tworzymy zmienną o nazwie `zapytanie` (w języku Perl jest to znak `$` umieszczony przed odpowiednio skonstruowanym ciągiem literowo-cyfrowym oznacza, że mamy do czynienia ze zmienną). Zmienna ta umożliwi nam dostęp do funkcji upraszczających prostą obsługę zapytań prowadzonych z wykorzystaniem interfejsu CGI.

W kolejnej linii tworzymy zmienną, która z danych wysłanych do skryptu wyłuska zmienną o nazwie `"imie"`. Warto tutaj dodać, że zmienna ta to najczęściej nic innego jak zawartość pola o nazwie „imie", które znajdowało się w formularzu, przy pomocy którego komunikujemy się z serwerem.

Piąta linia powyższego programu spowoduje utworzenie nagłówka poprzedzającego zawartość poprawnej odpowiedzi programu CGI.

Ostatnia zaś, szósta linia wyświetli dla przykładu napis `"Twoje imię to: Piotr"`. W kodzie tym użyto funkcji: `escapeHTML($imie)`, której zadaniem jest usunięcie z wprowadzonych danych znaczników języka HTML. Zabezpiecza to taki skrypt przed atakami typu HTML Injection.

Podsumowując powyższy skrypt, choć może niezbyt olśniewający swoim działaniem, został napisany w sposób zapewniający jego względnie poprawne działanie. Weryfikację danych prowadzi się tu w niewidoczny sposób, przerzucając tę robotę na podprogramy zgromadzone w module wpierającym CGI. Skrypt nie powoduje żadnych niebezpiecznych akcji, bo jego jedynym zadaniem jest pobranie danych dostarczonych przez przeglądarkę, wyciągnięcie z nich imienia użytkownika i stworzenie prostej strony internetowej z odpowiedzią. Ponieważ jest to pełnoprawny program napisany w Perlu, więc dałoby się go uruchomić ręcznie z wiersza poleceń, żeby jednak mógł działać jako część oprogramowania serwera, trzeba go umieścić w odpowiednim katalogu, którym zwykle jest katalog `cgi-bin` w głównym katalogu programu serwera.

Nieco inne podejście do tego tematu prezentują tak zwane serwlety. Serwlet to napisana według wymagań wymuszonych przez mechanizm dziedziczenia (patrz rozdział o językach programowania) klasa języka Java. Klasa ta działa po stronie serwera, rozszerzając jego możliwości. Uruchamia się ją w bezpiecznym środowisku serwera aplikacji (np. `GlassFish`) albo kontenera webowego (np. `Apache Tomcat`). Serwlety mają dostęp do wszystkiego, co w swym bogactwie oferuje język Java.

Życie serwletu zaczyna się w momencie, w którym programista skompiluje jego program i umieści go w odpowiednim katalogu serwera. Uruchomienie serwera nie jest jednoznaczne z uruchomieniem serwletu, który ładowany jest do pamięci dopiero w momencie, w którym będzie naprawdę potrzebny, czyli najczęściej wtedy, gdy pojawi się pierwsze żądanie użytkownika.

Każde następne żądanie będzie powodowało jedynie utworzenie kopii (tzw. instancji) tej klasy i związanego z nią wątku obsługującego to konkretne żądanie. Każdy nowo stworzony serwlet wywołuje metodę `service()` interfejsu

Servlet, która rozpoznaje, jakiego typu są żądania nadchodzące od użytkownika i wywołuje odpowiednie funkcje w celu ich obsłużenia.

Ostatecznie, po obsłużeniu całego żądania, na kopii serwletu wywoływana jest metoda `destroy()`, która zwalnia wszystkie zasoby systemowe, z których on korzystał. Poniżej znajduje się przykład serwletu, którego efekt działania jest mniej więcej taki sam jak efekt działania opisanego wcześniej skryptu CGI.

```
 1  public class Przyklad extends HttpServlet
 2  {
 3    private String imie = null;
 4
 5    public void init()
 6    {
 7      imie = this.getInitParameter("imie");
 8    }
 9
10   public void doGet(HttpServletRequest request,
                               HttpServletResponse response)
11   {
12     response.setContentType("text/html");
13     PrintWriter out = response.getWriter();
14     out.println("Twoje imie to: " + imie);
15     out.close();
16   }
17 }
```

Powyższy program nie jest kompletny. Usunięto z niego między innymi fragmenty kodu odpowiedzialne za załadowanie do pamięci klas, z których korzysta ten serwet i za obsługę tzw. wyjątków. Należy to mieć na uwadze podczas próby przeniesienia tego programu z papieru do pamięci komputera.

Przyjrzyjmy się teraz powyższemu programowi. W linii trzeciej zadeklarowano tam zmienną o nazwie `"imie"`, do której od razu została przypisana wartość `null`. Znaczy to, że zmienna `"imie"` nie zawiera teraz żadnej sensownej wartości i jest jakby pustym obiektem.

W linii piątej znajduje się początek metody `init()`, od której rozpoczyna się działanie serwletu. Cała akcja działania tej metody skupia się w linii siódmej, w której z danych nadesłanych przez przeglądarkę wyciągamy informację o imieniu użytkownika.

W linii dziesiątej rozpoczyna się metoda obsługująca jeden z typów żądań internetowych, czyli metodę GET. Kod zawarty w linii dwunastej sprawia, że odpowiedź udzielona przez serwet będzie miała formę tekstową, która być może będzie miała coś wspólnego z językiem HTML. W linii trzynastej tworzymy obiekt o nazwie out. Obiekt ten pozwoli nam na wysłanie odpowiedzi do przeglądarki, z której pochodzą dane do obrobienia. W linii 14 wykorzystując

stworzony przed chwilą obiekt, wysyłamy do przeglądarki napis `"Twoje imię to: Piotr"` (o ile użytkownik tak ma na imię), a w piętnastej linii zamykamy dostęp do możliwości dopisywania czegoś do danych kierowanych do przeglądarki. W tym miejscu serwlet zrobił już, co miał zrobić i należy po nim posprzątać. Służy do tego metoda `destroy()`, która nie jest widoczna w powyższym kodzie, dlatego że polegamy tutaj na jej wersji odziedziczonej po przodku klasy Przykład, czyli klasie HttpServlet.

Powstaje teraz pytanie, jaki jest sens używania serwletów, skoro kompletny program, który robi to samo napisany w Perlu, jest kilka razy krótszy? Odpowiedź jest prosta. To, co wygodne dla człowieka, nie zawsze jest wygodne dla komputera. Powyższy serwet, mimo że na oko jest dużo większy od opisanego wcześniej skryptu CGI, wymaga mniejszych ilości zasobów systemowych do tego, by działać w poprawny sposób. Dopóki więc strona, którą tworzymy, nie będzie nawiedzana przez dziesiątki tysięcy ludzi dziennie, to wybór technologii jest kwestią sporną. Potem lepiej zainteresować się bardziej wydajnymi sposobami serwowania dynamicznych treści internetowych, do których należy użycie serwletów.

Oba przedstawione powyżej sposoby mają jedną wspólną cechę. W jednym i drugim przypadku zaprezentowano jedynie kod odpowiedzialny za przetwarzanie danych. Aby można było w wygodny sposób je wprowadzać, należałoby stworzyć pliki z formularzami odwołującymi się do tych skryptów.

W pierwszym przypadku mieliśmy tu do czynienia z pewną dowolnością w doborze języka programowania, w którym miało być tworzone nasze oprogramowanie. W drugim mogła to być tylko Java. Co jednak zrobić, kiedy ktoś czuje awersję do tego języka, gdyż na przykład nie lubi programować obiektowo? Rozwiązaniem w tym przypadku może być pomysł Microsoftu, czyli ASP.NET. ASP (ang. *Active Server Pages*) oparte na platformie .NET umożliwia wybranie do tego celu jednego z języków dostępnych w platformie .NET. Oprócz tego technologia ta umożliwia łączenie ze sobą w jednym miejscu opisu formularza i programu, który ma przetworzyć zawarte w nim dane. Poniżej znajduje się przykład takiego właśnie połączenia. Pobieżna lektura tego kodu powinna zaowocować przynajmniej niejasnym przekonaniem, że jest to coś znajomego, bo jak nietrudno się domyślić, jest to kolejny przykład programu, który pobiera imię użytkownika i wyświetla proste zdanie z jego użyciem.

```
 1  <%@ Page Language="VB" %>
 2
 3  <script runat="server">
 4    Sub obsluga(obj As Object, e As EventArgs)
 5    Response.Write("Twoje imie to: ")
 6    Response.Write(imie.Value)
 7    End Sub
 8  </script>
 9
10  <html>
```

```
11 <body>
12 <form runat="server">
13 <input type="text" size="25" id="imie" runat="server" />
14 <asp:button id="ok" Text="Ok" runat=server OnClick="obsluga"/>
15 </form>
16 </body>
17 </html>
```

W powyższym kodzie dosyć wyraźnie zarysowują się trzy części. W pierwszej linijce pojawia się informacja na temat języka, w jakim zostanie napisane oprogramowanie przetwarzające dostarczone mu dane. W tym przypadku jest to Visual Basic, a wiadomo to stąd, że w odpowiednim miejscu pierwszej linii użyto jego skrótowej nazwy.

Między trzecią a ósmą linią znajduje się kod odpowiedzialny za przetwarzanie danych wysłanych przez przeglądarkę, a pomiędzy linią dziesiątą i siedemnastą można znaleźć opis, jak będzie wyglądała strona z formularzem umożliwiającym wpisanie imienia.

Do tej pory rozdział ten traktował jedynie o możliwościach przetwarzania informacji. Temat nie został wyczerpany, bo wciąż pojawiają się nowe pomysły na komunikowanie się przeglądarki z serwerem w celu uzyskania interaktywnych stron internetowych. A to onacza, że zamiast zajmować się przetwarzaniem informacji, pora przejść do opisu możliwości ich magazynowania.

Właściwie nie ma jednej dobrej odpowiedzi, w jaki sposób należy to robić, wszystko bowiem zależy od tego, co chcemy osiągnąć. W przypadku, gdy chcemy magazynować dane o bardzo niewyszukanej formie (np. jedynie pary login - hasło) oraz mamy pewność, że do tego zestawu już nigdy nic nie zostanie dodane, może się okazać, że najlepszym pomysłem będzie rozwiązanie oparte na przykład na zapisywaniu haseł w plikach o nazwach odpowiadających loginom. Przy takim założeniu sprawdzenie, czy dany użytkownik ma już swoje konto na naszym serwisie, jest równoznaczne ze sprawdzeniem, czy plik o nazwie będącej jego loginem już istnieje. Podobnie też, chcąc zweryfikować poprawność wprowadzonego przez niego hasła, musimy jedynie odczytać hasło zawarte w jego pliku z tym, które podał. Jednakże pomimo tego, że operacje takie wymagają wolnych działań plików na systemie, to w przypadku ogromnej liczby użytkowników mogą się nieźle sprawdzić, prowadząc do oszczędności czasowych.

Skoro to takie dobre rozwiązanie, to czemu go nie stosować powszechnie? Odpowiedź jest prosta: bo warunki, pod którymi będzie się je dało wykorzystać, zdarzają się niezwykle rzadko. Zwykle jest tak, że ilość danych, które odnoszą się do konkretnego użytkownika i które trzeba przechowywać na serwerze, rośnie wraz z popularnością serwisu. Bywa też i tak, że dane te nie łączą się ze sobą w tak oczywisty sposób, a ich zestaw trzeba przeszukiwać w znacznie bardziej wyrafinowane sposoby. W takich sytuacjach z pomocą przychodzą tzw. bazy danych.

Pierwsze systemy zarządzania bazami danych zostały opracowane w latach sześćdziesiątych ubiegłego wieku. Powstały wtedy dwa modele danych: hierarchiczny i sieciowy. Dane gromadzone w modelu hierarchicznym przypominają swoim wyglądem strukturę katalogów na dysku twardym. W przypadku danych zgromadzonych na dysku twardym dotarcie do konkretnych danych wymaga przejścia przez pewną liczbę podkatalogów i otworzenia pewnego pliku. W modelu sieciowym przechodzimy podobną drogę od tzw. rekordów nadrzędnych do rekordów podrzędnych aż w końcu odnajdziemy poszukiwane dane.

Model sieciowy stanowił już znaczący postęp w stosunku do modelu hierarchicznego. W modelu tym dane grupowane były w sieć, dzięki czemu można było znacznie szybciej przemieszczać się pomiędzy nimi, odnajdując te informacje, które nas interesują.

Prawdziwym jednak przełomem okazała się idea wprowadzona przez Edgara. F. Codda, który zaproponował tzw. relacyjny model danych. Model ten miał się stać lekiem na krytykowane przez Codda mieszanie opisu struktury informacyjnej z opisami mechanizmów dostępu do informacji, które było powszechne w bazach danych opartych na obu poprzednich modelach. Pomysł ten był jednak na tyle wizjonerski, że musiał odczekać kilka lat nim pojawiły się techniczne możliwości jego realizacji. Dopiero w 1976 roku pojawiły się dwa prototypowe rozwiązania oparte na modelu relacyjnym, a były to Ingres Michaela Stonebrakera i System R firmy IBM. Dopiero w roku 1980 pojawiły się pierwsze komercyjne bazy danych stosujące opisywany model, a były to Oracle oraz DB2.

Co takiego jest w tym modelu, że okazał się na tyle dobrym pomysłem, że opiera się na nim większość dzisiejszych sensownych baz danych? Odpowiedzią na to pytanie jest genialny w swojej prostocie sposób grupowania danych. Dane te są przechowywane w tabelach, z których każda ma ustaloną liczbę kolumn i dowolną liczbę wierszy. Każda z tych tabel ma też zdefiniowany klucz danych, czyli wyróżniony zestaw atrybutów, które jednoznacznie identyfikują dany wiersz tabeli.

14.1 Przykład

Załóżmy, że chcemy stworzyć spis członków naszej rodziny. Możemy w tym celu zbudować następującą tabelę:

Opis
Wujek Stasiek
Ciocia Lusia
Kuzyn Maruda

Jak widać, nazwy członków rodziny się nie powtarzają, więc każda z nich jest atrybutem w jednoznaczny sposób określającym wiersz tabeli. Co jednak

zrobić, gdy nasza rodzina obfituje np. w wujków Staśków? Rozwiązaniem takiego problemu, jest wprowadzenie dodatkowego atrybutu, którym może być na przykład PESEL albo tworzony w sposób automatyczny numer identyfikacyjny.

Opis	Numer identyfikacyjny
Wujek Stasiek	1
Ciocia Lusia	2
Wujek Stasiek	3
Kuzyn Maruda	4

Trzeba przyznać, że sam pomysł jest dobry, lecz bez odpowiedniego wsparcia w postaci narzędzia ułatwiającego przeszukiwanie zgromadzonych w ten sposób danych nie na wiele by się zdał. Wsparciem takim są serwery baz danych, które oferują wiele mechanizmów, wydatnie ułatwiających życie poszukiwaczom danych. Do wymagań narzucanych wartym uwagi bazom danych należy zaliczyć:

- zapewnienie możliwości zbierania, administrowania i utrwalania danych w plikach,
- zapewnienie bezpieczeństwa i spójności gromadzonych danych,
- zapewnienie sprawnego i wygodnego dostępu do danych, co zwykle realizuje się poprzez język zapytań. Obecnie jednym z najpopularniejszych jest język SQL, który należy do grupy języków deklaratywnych, co znaczy, że zwykle wie lepiej od nas, jak zrobić coś, co chcemy zrobić z bazą danych,
- zapewnienie w miarę łatwego dostępu do gromadzonych danych dla innych języków programowania,
- zapewnienie jednoczesnego dostępu wielu użytkownikom,
- regulowanie dostępu do danych, czyli krótko mówiąc autoryzację,
- zapewnienie możliwości odtworzenia zawartości bazy danych po awarii,
- optymalizowanie użycia pamięci oraz czasu dostępu do danych (zwykle robi się to poprzez indeksowanie danych),
- zapewnienie wsparcia dla współdziałania w środowiskach rozproszonych (chodzi tu na przykład o sieci komputerowe wspólnie realizujące jakieś zadanie).

Błyskawiczny rozwój sieci komputerowych z internetem na czele sprawia, że dzisiejsze bazy danych działają zwykle w trybie klient - serwer. Klientami są tutaj programy pytające o pewien zakres danych, a serwerem jest system zarządzania bazą danych. W taki sposób działają między innymi systemy Microsoft SQL Server, Oracle czy niezwykle popularna ze względu na możliwość bezpłatnego użytkowania baza MySQL.

Jednak prawdziwym strzałem w dziesiątkę jest połączenie obu opisywanych wyżej technologii, czyli stworzenie interaktywnego serwisu internetowego

wykorzystującego bazy danych. Takie rozwiązanie legło u podstaw działania banków internetowych, serwisów społecznościowych, a nawet jednego z najbardziej rozpoznawanych produktów na świecie, czyli wyszukiwarki Google.

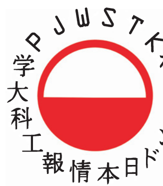
Nawiasem mówiąc, liczba mniej lub bardziej potrzebnych serwisów internetowych działających w ten sposób stale rośnie. Mechanizm ten wykorzystywany jest na równych prawach w serwisach randkowych, lokalizatorach internetowych czy bazach wiedzy takich jak wikipedia.pl czy planetmath.org. Nie da się ukryć, że upowszechnianie się wiedzy na temat tego typu rozwiązań ma swoje odzwierciedlenie w mentalności społeczeństwa. Ludzie coraz częściej polegają na możliwościach serwowanych im przez maszyny, zapominając o tym, że to samo da się osiągnąć przy pomocy ludzkiego umysłu.

Alfred Nobel wynajdując dynamit, nie spodziewał się, że ktoś odważy się go wykorzystać w niecnych celach. My również nie jesteśmy w stanie przewidzieć wszystkich konsekwencji ustawicznego zwiększania się ilości informacji, które gromadzone są w Internecie na temat każdego z nas. Dlatego pozostaje jedynie żywić nadzieję na to, że opisywane tu rozwiązania pchną ludzkość na nowe ścieżki rozwoju, a nie przyczynią się do jej głębszego ogłupienia połączonego ze zniewoleniem opisywanym niejednokrotnie przez Orwella.

Literatura

[1.] „HTML i XHTML" Jennifer Niederst Robbins; Helion 2006.
[2.] „PHP5 - księga eksperta" John Coggeshall; Helion 2005.
[3.] „Algorytmy od podstaw" Simon Harris, James Ross; Helion 2006.
[4.] „Piękny kod - tajemnice mistrzów programowania" praca zbiorowa pod redakcją Andy'ego Orana i Grega Wilsona; Helion 2008.
[5.] „Opowieści matematyczne" Michał Szurek; Wydawnictwa Szkolne i Pedagogiczne 1987.
[6.] „Przez rozrywkę do wiedzy - rozmaitości matematyczne" Stanisław Kowal; Wydawnictwa Naukowo - Techniczne 1986.
[7.] „Atlas fizjologii" Santiago Ferrandiz, Fanny Font, Lluis Serra; Wydawnictwo Wiedza i Życie 1992.
[8.] „Mała encyklopedia logiki" praca zbiorowa pod redakcją Witolda Marciszewskiego; Ossolineum 1988.
[9.] „Java - kompendium programisty" Herbert Schildt; Helion 2005.

ポーランド日本情報工科大学



POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Jedna z najlepszych uczelni w Polsce – wyróżniana przez pracodawców, studentów i media.

Informatyka

Studia inżynierskie, magisterskie uzupełniające, podyplomowe, studia doktoranckie, uprawnienia habilitacyjne, studia przez internet.

Specjalizacje:
animacja 3D, bazy danych, eksploracja www, inteligentne systemy przetwarzania danych, inżynieria oprogramowania i baz danych, multimedia, programowanie aplikacji biznesowych, programowanie gier, programowanie systemowe i sieciowe, robotyka, sieci urządzeń mobilnych, systemy rozproszone i równoległe.

Akredytacja Państwowej Komisji Akredytacyjnej

Architektura Wnętrz
Studia licencjackie

Kultura Japonii
Studia licencjackie

Sztuka Nowych Mediów (Grafika)
Studia licencjackie, magisterskie uzupełniające

Zarządzanie Informacją
Studia inżynierskie

Kursy:
Akademia Sieciowa CISCO; LPI Linux; Microsoft

Akademickie Liceum Ogólnokształcące przy PJWSTK

02-008 Warszawa, ul. Koszykowa 86
tel.: 022 58 44 500, fax: 022 58 44 501
e-mail: pjwstk@pjwstk.edu.pl
www.pjwstk.edu.pl

PJWSTK w Bytomiu
Informatyka
Sztuka Nowych Mediów, Grafika
Studia inżynierskie, licencjackie
41-902 Bytom, Aleja Legionów 2
tel.: 032 387 16 60
e-mail: bytom@pjwstk.edu.pl
www.bytom.pjwstk.edu.pl

PJWSTK w Gdańsku
Informatyka
Studia inżynierskie
80-045 Gdańsk, ul. Brzegi 55
tel.: 058 683 59 75
e-mail: gdansk@pjwstk.edu.pl
www.gdansk.pjwstk.edu.pl

ISBN 978-83-89244-86-4