Rozdział 1.
Zanim wystartujemy

Zanim na dobre rozpoczniemy operowanie takimi pojęciami, jak wspomniany we wstępie „algorytm", warto przedyskutować dokładnie, co przez nie rozumiemy[1].


Definicja

Algorytm to:

- skończony ciąg/sekwencja reguł, które aplikuje się na skończonej liczbie danych, pozwalający rozwiązywać zbliżone do siebie klasy problemów;
- zespół reguł charakterystycznych dla pewnych obliczeń lub czynności informatycznych.

Cóż, definicje powyższe wydają się klarowne i jasne, jednak obejmują na tyle rozległe obszary działalności ludzkiej, że daleko im do precyzji. Pomijając chwilowo znaczenie, samo pochodzenie terminu algorytm nie zawsze było do końca jasne. Dopiero specjaliści zajmujący się historią matematyki odnaleźli najbardziej prawdopodobny źródłosłów: termin ten pochodzi od nazwiska perskiego pisarza matematyka Muhammada ibn Musa al-Chuwarizmi[2] (żył w IX wieku n.e.). Jego zasługą jest dostarczenie klarownych reguł wyjaśniających krok po kroku zasady operacji arytmetycznych wykonywanych na liczbach dziesiętnych.

Słowo algorytm jest często łączone z imieniem greckiego matematyka Euklidesa (365 — 300 p.n.e.) i jego słynnym przepisem na obliczanie największego wspólnego dzielnika dwóch liczb `a` i `b` (NWD):

```
dane wejściowe: a i b, zmienne pomocnicze: c, res
   dopóki a>0 wykonuj:
      podstaw za c resztę z dzielenia a przez b;
      podstaw za b liczbę a;
      podstaw za a liczbę c;
      podstaw za res liczbę b;
      rezultat: res.
```

Oczywiście Euklides nie proponował swojego algorytmu dokładnie w ten sposób (w miejsce funkcji reszty z dzielenia stosowane były sukcesywne odejmowania), ale jego pomysł można zapisać w powyższy sposób bez szkody dla wyniku, który w każdym przypadku będzie taki sam. Nie jest to, rzecz jasna, jedyny algorytm, z którym mieliśmy w swoim życiu do czynienia. Każdy z nas z pewnością umie zaparzyć kawę:

---

[1] Definicja pochodzi ze słownika *Le Nouveau Petit Robert* (Dictionnaires le Robert — Paris 1994) — tłumaczenie własne.

[2] Jego nazwisko pisane było po łacinie jako *Algorismus*, pisownia w tym wydaniu książki jest zgodna z encyklopedią PWN.

- włączyć gaz;
- zagotować niezbędną ilość wody;
- wsypać zmieloną kawę do szklanki;
- zalać kawę wrzącą wodą;
- osłodzić do smaku;
- poczekać, aż odpowiednio naciągnie.

Powyższy przepis działa, ale zawiera kilka słabych punktów: co to znaczy „odpowiednia ilość wody"? Co dokładnie oznacza stwierdzenie „osłodzić do smaku"? Przepis przygotowania kawy ma cechy algorytmu (rozumianego w sensie zacytowanych wyżej definicji słownikowych), ale brak mu precyzji niezbędnej do wpisania go do jakiejś maszyny, tak aby w każdej sytuacji umiała ona sobie poradzić z poleceniem „przygotuj mi małą kawę". (Np. jak w praktyce określić warunek, że kawa „odpowiednio naciągnęła"?).

Jakie w związku z tym cechy powinny być przypisane algorytmowi rozumianemu w kontekście informatycznym? Dyskusję na ten temat można by prowadzić dość długo, ale przyjmując pewne uproszczenia, można zadowolić się następującymi wymogami wyszczególnionymi poniżej.

Każdy algorytm:

- posiada dane wejściowe (w ilości większej lub równej zero) pochodzące z dobrze zdefiniowanego zbioru (np. algorytm Euklidesa operuje na dwóch liczbach całkowitych);
- produkuje pewien wynik (niekoniecznie numeryczny);
- jest precyzyjnie zdefiniowany (każdy krok algorytmu musi być jednoznacznie określony);
- jest skończony (wynik algorytmu musi zostać kiedyś dostarczony — mając algorytm A i dane wejściowe D, powinno być możliwe precyzyjne określenie czasu wykonania $T(A)$);
- daje się zastosować do rozwiązywania całej klasy zagadnień, a nie tylko jednego konkretnego zadania.

Ponadto niecierpliwość każe nam szukać algorytmów efektywnych, tzn. wykonujących swoje zadanie w jak najkrótszym czasie i wykorzystujących jak najmniejszą ilość pamięci (do tej tematyki powrócimy jeszcze w rozdziale 3.). Zanim jednak pospieszymy do klawiatury, aby wpisywać do pamięci komputera programy spełniające powyższe założenia, popatrzmy na algorytmikę z perspektywy historycznej.

Jak to wcześniej bywało, czyli wyjątki z historii maszyn algorytmicznych

Cytowane na samym początku tego rozdziału imiona matematyków kojarzonych z algorytmiką rozdzielone są ponad tysiącem lat i mogą łatwo zasugerować, że ta gałąź wiedzy przeżywała w ciągu wieków istnienia ludzkości burzliwy i błyskotliwy rozwój. Oczywiście nijak się to ma do rzeczywistego postępu tej dziedziny, który był i ciągle jest ściśle związany z rewolucją techniczną dokonującą się na przestrzeni zaledwie ostatnich dwustu lat. Owszem, jeśli zechcemy traktować informatykę i algorytmikę jako pewną całość wywodzącą się naturalnie z systemów obliczeniowych, to warto wspomnieć o osiągnięciach ludów sumeryjskich, wynalazców tabliczek obliczeniowych, własnego kalendarza i sześćdziesiętnego systemu pomiarowego (24-godzinna doba to ich wynalazek). Znani są też Chińczycy, wynalazcy Abakusa, czyli najsłynniejszego liczydła w historii ludzkości, choć mało kto ma świadomość, że praktycznie każdy w miarę cywilizowany lud dopracowywał się jakiegoś systemu wspomagającego obliczenia i trudno tu oddawać komuś palmę pierwszeństwa. Ponadto, licytując się tego typu faktami, łatwo cofniemy się do okresu datowanego na 2 – 3 tysiące lat p.n.e., tylko czy to ma obecnie wartość inną niż ciekawostka?

Aby nie zamieniać tego rozdziału w podręcznik historii, pominę rozważania na temat maszyny do dodawania Blaise'a Pascala (ok. 1645) lub jej anglosaskiego odpowiednika, skonstruowanego w niemal tym samym czasie przez G. W. Leibniza. Popatrzmy jedynie na kilka charakterystycznych wydarzeń związanych z wynalazkami, które nie tylko ułatwiały obliczanie, ale i pozwalały na elementarne programowanie, czyli coś, co już jednoznacznie kojarzy nam się z komputerami i algorytmami.

— 1801 —

Francuz Joseph Marie Jacquard wynajduje krosno tkackie, w którym wzorzec tkaniny był „programowany" na swego rodzaju kartach perforowanych. Proces tkania był kontrolowany przez algorytm zakodowany w postaci sekwencji otworów wybitych w karcie. Sam pomysł był wynikiem wielu lat pracy Jacquarda i mógł ujrzeć światło dzienne dzięki przypadkowi, jakim było uczestnictwo w konkursie państwowym, na którym przedstawił maszynę do robienia sieci rybackich.

Koncepcja Jacquarda zainteresowała francuskiego matematyka, L.M. Carnota, który ściągnął go do Paryża w celu kontynuowania badań i pomógł w uzyskaniu stypendium rządowego. Pierwsze prace omal nie doprowadziły do śmierci wynalazcy, rozwścieczeni tkacze niemal utopili go w Rodanie, przeczuwając, że jego maszyna stanowi zagrożenie dla ich zatrudnienia (a dokładniej: dla zatrudnienia ich dzieci, które do tej pory służyły za pomocników podnoszących nitki, aby umożliwić utkanie lub nie odpowiedniego wzoru przez przesuwającą się poprzecznie cewkę z nitką — wynalazek Jacquarda eliminował pięć stanowisk pracy przy jednym krośnie!). Pomysł Jacquarda był dopasowany do ówczesnych możliwości technicznych, ale warto zauważyć, że karta perforowana z zakodowaną logiką dwustanową (dziurka lub jej brak oznaczał dla maszyny tkackiej podjęcie lub nie odpowiedniej akcji natury mechanicznej) jest prekursorem współczesnych pamięci, w których za pomocą liczb binarnych koduje się odpowiednie akcje algorytmu!

— 1833 —

Anglik Charles Babbage częściowo buduje maszynę do wyliczania niektórych formuł matematycznych. W czasach, w których żył Babbage, nastąpiła eksplozja zastosowań matematyki (astronomia, nawigacja), a nie istniały metody wspomagające obliczenia w sposób automatyczny. Babbage był autorem koncepcji tzw. *maszyny analitycznej*, zbliżonej do swego poprzedniego dzieła, ale wyposażonej w możliwość przeprogramowywania, jak w przypadku maszyny Jacquarda.

W pewnym uproszczeniu maszyna ta miała składać się z magazynu (odpowiednik pamięci realizowanej jako kolumny kół, później zastąpionej bębnem), młyna (jednostka obliczeniowa wykonująca operacje dzięki obrotom kół i przekładni) i mechanizmu sterującego wykorzystującego karty perforowane (Jacquard!). Czyż nie przypomina to schematu komputera?

Opisy maszyny Babbage'a były na tyle dokładne, że matematyczka Ada Lovelace, córka lorda Byrona, opracowała pierwsze teoretyczne „programy" dla tej nieistniejącej jeszcze maszyny, stając się pierwszą uznaną... programistką w historii informatyki[3]!

Wymagania natury mechanicznej, jakie stawiała ta maszyna, pozwoliły na skonstruowanie jej pierwszego prototypu dopiero w dwadzieścia lat od narodzin samej idei, a sama maszyna powstała dopiero w roku... 1992, oczywiście bardziej jako ciekawostka niż potrzeba naukowa.

— 1890 —

W zasadzie pierwsze publiczne i na dużą skalę użycie maszyny bazującej na kartach perforowanych. Chodzi o maszynę do opracowywania danych statystycznych, dzieło Amerykanina Hermana Holleritha, użyte przy dokonywaniu spisu ludności. (Na marginesie warto dodać, że

[3] Od jej imienia pochodzi nazwa języka programowania ADA.

przedsiębiorstwo Holleritha przekształciło się w 1911 roku w *International Business Machines Corp.*, bardziej znane jako IBM, będące do dziś czołowym producentem komputerów).

— lata trzydzieste —

Rozwój badań nad teorią algorytmów (plejada znanych matematyków: Turing, Gödel, Markow). Z tego okresu wywodzi się słynne zagadnienie postawione przez pruskiego[4] matematyka Dawida Hilberta, który w 1928 roku na międzynarodowym kongresie matematyków publicznie postawił pytanie, czy istnieje metoda pozwalająca rozstrzygnąć prawdziwość dowolnego twierdzenia matematycznego, w wyniku li tylko mechanicznych operacji na symbolach. Studentom informatyki bliskie będzie pojęcie tzw. maszyny Turinga, abstrakcyjnej maszyny obliczeniowej złożonej z głowicy czytająco-piszącej oraz nieskończonej taśmy zawierającej symbole (np. liczby lub operatory działań). Ten abstrakcyjny model matematyczny stworzył podwaliny pod współczesne komputery. W ramach tej książki Czytelnik powinien zapamiętać tylko, że to, co określa się nieco myląco terminem *maszyna*, jest wyłącznie *modelem* schematu działania wg zadanego algorytmu.

— lata czterdzieste —

Budowa pierwszych komputerów ogólnego przeznaczenia (głównie dla potrzeb obliczeniowych wynikłych w tym wojennym okresie: badania nad łamaniem kodów, początek „kariery" bomby atomowej).

Pierwszym urządzeniem, które można określić jako „komputer", był automatyczny kalkulator MARK 1, skonstruowany w 1944 roku (jeszcze na przekaźnikach, czyli jako urządzenie elektromechaniczne). Jego twórcą był Amerykanin Howard Aiken z uniwersytetu Harvarda. Aiken bazował na idei Babbage'a, która musiała czekać 100 lat na swoją praktyczną realizację! W dwa lata później powstaje pierwszy elektroniczny komputer ENIAC[5] (jego wynalazcy: J. P. Eckert i J. W. Mauchly z uniwersytetu Pensylwania), który miał oryginalnie wspomagać obliczenia balistyczne.

Powszechnie jednak za pierwszy komputer w pełnym tego słowa znaczeniu uważa się EDVAC[6] zbudowany na uniwersytecie w Princeton. Jego wyjątkowość polegała na umieszczeniu programu wykonywanego przez komputer całkowicie w jego pamięci, podobnie jak i pamięci do przechowywania wyników obliczeń. Autorem tej przełomowej idei był matematyk Johannes von Neumann (Amerykanin węgierskiego pochodzenia)[7].

— okres powojenny —

Prace nad komputerami prowadzone są w wielu krajach równolegle. W grę zaczyna wchodzić wkroczenie na nowo powstały obiecujący rynek komputerów (kończy się bowiem era budowania unikalnych uniwersyteckich prototypów). Na rynku pojawiają się kalkulatory IBM 604 i BULL Gamma 3, a następnie duże komputery naukowe, np. UNIVAC 1 i IBM 650. Zaczynającej się zarysowywać dominacji niektórych producentów usiłują przeciwdziałać badania prowadzone w wielu krajach (mniej lub bardziej systematycznie i z różnorakim poparciem polityków), ale... to już jest temat na osobną książkę!

---

[4] Urodzony w Königsbergu, obecnie zwanym Kaliningradem.

[5] Ang. *Electronic Numerical Interpreter And Calculator.*

[6] Ang. *Electronic Discrete Variable Automatic Computer.*

[7] Koncepcja komputera została opracowana w 1946 roku, jednak jego pierwsza realizacja praktyczna powstała dopiero w roku 1956.

— teraz —

Burzliwy rozwój elektroniki powoduje masową, do dziś trwającą komputeryzację wszelkich dziedzin życia. Komputery stają się czymś powszechnym i niezbędnym, wykonując tak różnorodne zadania, jak tylko każe im to wyobraźnia ludzka.

Jak to się niedawno odbyło, czyli o tym, kto „wymyślił" metodologię programowania

Zamieszczone w poprzednim paragrafie kalendarium zostało doprowadzone do momentu, w którym programiści zaczęli mieć do dyspozycji komputery z prawdziwego zdarzenia. Olbrzymi nacisk, jaki był kładziony na rozwój sprzętu, w istocie doprowadził do znakomitych rezultatów — efekt jest widoczny dzisiaj w praktycznie każdym biurze i w coraz większej liczbie domów prywatnych.

W latach sześćdziesiątych zaczęto konstruować pierwsze naprawdę duże systemy informatyczne — w sensie ilości kodu, głównie asemblerowego, wyprodukowanego na poczet danej aplikacji. Ponieważ jednak programowanie było ciągle traktowane jako działalność polegająca głównie na intuicji i wyczuciu, zdarzały się całkiem poważne wpadki w konstrukcji oprogramowania: albo szybko tworzone były systemy o małej wiarygodności, albo też nakład pieniędzy włożonych w rozwój produktu znacznie przewyższał szacowane wydatki i stawiał pod znakiem zapytania sens podjętego przedsięwzięcia. Brak było zarówno metod, jak i narzędzi umożliwiających sprawdzanie poprawności programowania. Powszechną metodą programowania było testowanie programu aż do momentu jego całkowitego „odpluskwienia"[8]. Zwróćmy jeszcze uwagę, że oba wspomniane czynniki: wiarygodność systemów i poziom nakładów są niezmiernie ważne w praktyce; informatyczny system bankowy musi albo działać stuprocentowo dobrze, albo nie powinien być w ogóle oddany do użytku! Z drugiej strony poziom nakładów przeznaczonych na rozwój oprogramowania nie powinien odbić się niekorzystnie na kondycji finansowej przedsiębiorstwa.

W pewnym momencie sytuacja stała się tak krytyczna, że zaczęto wręcz mówić o kryzysie w rozwoju oprogramowania! W 1968 roku została nawet zwołana konferencja NATO (Garmisch, Niemcy) poświęcona przedyskutowaniu zaistniałej sytuacji. W rok później w ramach IFIP (ang. *International Federation for Information Processing*) została utworzona specjalna grupa robocza pracująca nad tzw. metodologią programowania.

Z historycznego punktu widzenia dyskusja na temat udowadniania poprawności algorytmów zaczęła się jednak od artykułu Johna McCarthy'ego „A basis for a mathematical theory of computation", gdzie padło zdanie: „zamiast sprawdzania programów komputerowych metodą prób i błędów aż do momentu ich całkowitego odpluskwienia powinniśmy udowadniać, że posiadają one pożądane własności". Nazwiska ludzi, którzy zajmowali się teoretycznymi pracami na temat metodologii programowania, nie znikły bynajmniej z horyzontu: Dijkstra, Hoare, Floyd, Wirth itd. (będą oni jeszcze nieraz cytowani w tej książce!).

Krótka prezentacja, której dokonaliśmy w poprzednich dwóch paragrafach, ukazuje dość zaskakującą młodość algorytmiki jako dziedziny wiedzy. Warto również zauważyć, że nie jest to nauka, która powstała samorodnie. O ile obecnie należy ją odróżniać jako odrębną gałąź wiedzy, to nie sposób nie docenić wielowiekowej pracy matematyków, którzy dostarczyli algorytmice zarówno narzędzi opisu zagadnień, jak i wielu użytecznych teoretycznych rezultatów. (Powyższa uwaga tyczy się również wielu innych dziedzin wiedzy).

Teraz, gdy już zdefiniowaliśmy sobie głównego bohatera tej książki (bohatera zbiorowego: chodzi bowiem o algorytmy!), przejrzymy kilka sposobów używanych do jego opisu.

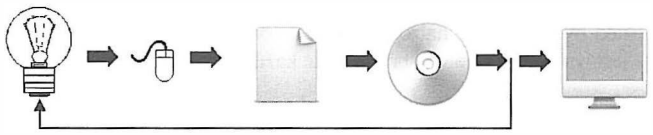
[8] Żargonowe określenie procesu usuwania błędów z programu.

Proces koncepcji programów

W poprzednim paragrafie wyszczególniliśmy kilka cech charakterystycznych, które powinien posiadać algorytm rozumiany jako pojęcie informatyczne. Szczególny nacisk położony został na precyzję zapisu. Wymóg ten jest wynikiem ograniczeń narzuconych przez współcześnie istniejące komputery i kompilatory — nie są one bowiem w stanie rozumieć poleceń nieprecyzyjnie sformułowanych, zbudowanych niezgodnie z wbudowanymi w nie wymogami syntaktycznymi.

Rysunek 1.1 w sposób uproszczony obrazuje etapy procesu programowania komputerów. Olbrzymia żarówka symbolizuje etap, który jest od czasu do czasu pomijany przez programistów (dodajmy, że zwykle z opłakanymi skutkami) — REFLEKSJĘ.

Rysunek 1.1. *Etapy konstrukcji programu*



Następnie jest tworzony tzw. tekst źródłowy nowego programu, mający postać pliku tekstowego, wprowadzanego do komputera za pomocą zwykłego edytora tekstowego. Większość istniejących obecnie kompilatorów posiada taki edytor już wbudowany, więc użytkownik w praktyce nie opuszcza tzw. środowiska zintegrowanego, grupującego programy niezbędne w procesie programowania. Ponadto niektóre środowiska zintegrowane zawierają zaawansowane edytory graficzne umożliwiające przygotowanie zewnętrznego interfejsu użytkownika praktycznie bez pisania ani jednej linii kodu. Pomijając już jednak tego typu szczegóły, generalnie efektem pracy programisty jest plik lub zespół plików opisujących w formie symbolicznej sposób zachowania się programu wynikowego. Opis ten jest kodowany w tzw. języku programowania, który stanowi na ogół podzbiór języka naturalnego[9]. Kompilator dokonuje mniej lub bardziej zaawansowanej analizy poprawności i, jeśli wszystko jest w porządku, produkuje tzw. *kod wykonywalny*, zapisany w postaci zrozumiałej dla komputera. Plik zawierający kod wykonywalny może być następnie wykonywany pod kontrolą systemu operacyjnego komputera (który to system notabene jest także złożony z wielu pojedynczych programów). Kod wykonywalny jest specyficzny dla danego systemu operacyjnego. W ostatnich latach rozpowszechnił się język Java, który pozwala budować programy niezależne od systemów operacyjnych, ale dzieje się to na zasadzie pewnego „oszustwa": kod wykonywalny nie jest uruchamiany bezpośrednio przez system operacyjny, tylko poprzez specjalne środowisko uruchomieniowe, które izoluje go od sprzętu i systemu, wprowadzając jako warstwę pośrednią opóźnienia niezbędne dla dokonania translacji kodu pośredniego na kod finalny.

Gdzie w tym procesie umiejscowione jest to, co stanowi tematykę książki, którą trzymasz, Czytelniku, w ręku? Otóż z całego skomplikowanego procesu tworzenia oprogramowania zajmiemy się tym, co do tej pory nie jest (jeszcze?) zautomatyzowane: koncepcją algorytmów, ich jakością i technikami programowania aktualnie używanymi w informatyce. Będziemy anonsować pewne problemy dające się rozwiązać za pomocą komputera, a następnie omówimy sobie, jak to zadanie wykonać w sposób efektywny. Tworzenie zewnętrznej otoczki programów, czyli tzw. *interfejsu użytkownika*, nie wchodzi w zakres tematyczny tej książki.

[9] W praktyce jest to język angielski.

Poziomy abstrakcji opisu i wybór języka

Jednym z delikatniejszych problemów związanych z opisem algorytmów jest sposób ich prezentacji zewnętrznej. Można w tym celu przyjąć dwie skrajne pozycje:

- zbliżyć się do maszyny (język asemblera: nieczytelny dla nieprzygotowanego odbiorcy);
- zbliżyć się do człowieka (opis słowny: maksymalny poziom abstrakcji zakładający poziom inteligencji odbiorcy niemożliwy aktualnie do wbudowania w maszynę[10]).

Wybór języka asemblera do prezentacji algorytmów wymagałby w zasadzie związania się z określonym typem maszyny, co zlikwidowałoby jakąkolwiek ogólność rozważań i uczyniłoby opis trudnym do analizy. Z drugiej zaś strony opis słowny wprowadza ryzyko niejednoznaczności, która może być kosztowna: program, po przetłumaczeniu go na postać zrozumiałą dla komputera, może nie zadziałać!

Aby zaradzić zaanonsowanym wyżej problemom, zwyczajowo przyjęło się prezentowanie algorytmów w dwojaki sposób:

- za pomocą istniejącego języka programowania;
- używając pseudojęzyka programowania (mieszanki języka naturalnego i form składniowych pochodzących z kilku reprezentatywnych języków programowania).

W niniejszym podręczniku można napotkać obie te formy i wybór którejś z nich został podyktowany kontekstem omawianych zagadnień. Przykładowo: jeśli dany algorytm jest możliwy do czytelnego zaprezentowania za pomocą języka programowania, wybór będzie oczywisty! Od czasu do czasu jednak napotkamy sytuacje, w których prezentacja kodu w pełnej postaci, gotowej do wprowadzenia do komputera, byłaby zbędna (np. zbliżony materiał był już przedstawiony wcześniej) lub nieczytelna (liczba linii kodu przekracza objętość jednej strony). W każdym jednak przypadku ewentualne przejście z jednej formy w drugą nie powinno stanowić dla Czytelnika większego problemu.

Już we wstępie zostało zdradzone, iż językiem prezentacji programów będzie C++. Pora zatem dokładniej wyjaśnić powody, które przemawiały za tym wyborem. C++ jest językiem programowania określanym jako *strukturalny*, co z założenia ułatwia pisanie w nim w sposób czytelny i zrozumiały. Związek tego języka z klasycznym C umożliwia oczywiście tworzenie absolutnie nieczytelnych listingów, będziemy tego jednak starannie unikać. W istocie częstokroć będą omijane pewne możliwe mechanizmy optymalizacyjne, aby nie zatracić prostoty zapisu. Najważniejszym jednak powodem użycia C++ jest fakt, iż ułatwia on programowanie na wielu poziomach abstrakcji. Istnienie klas i wszelkie obiektowe cechy tego języka powodują, że zarówno ukrywanie szczegółów implementacyjnych, jak i rozszerzanie już zdefiniowanych modułów (bez ich kosztownego „przepisywania") jest bardzo łatwe, a są to właściwości, którymi nie można pogardzić.

Być może cenne będzie podkreślenie usługowej roli, jaką w procesie programowania pełni wybrany do tego celu język. Wiele osób pasjonuje się wykazywaniem wyższości jednego języka nad drugim, co jest sporem tak samo jałowym, jak wykazywanie „wyższości świąt Wielkiej Nocy nad świętami Bożego Narodzenia" (choć zapewne mniej zabawnym...). Język programowania jest w końcu tylko narzędziem, ulegającym zresztą znacznej (r)ewolucji na przestrzeni ostatnich lat. Pracując nad pewnymi partiami tej książki, musiałem od czasu do czasu zwalczać silną pokusę prezentowania niektórych algorytmów w takich językach jak LISP czy PROLOG.

Uprościłoby to znacznie wszelkie rozważania o listach i rekurencji — niestety ograniczyłoby również potencjalny krąg odbiorców książki do ludzi profesjonalnie związanych wyłącznie z informatyką.

[10] Niemowlę radzi sobie bez trudu z problemami, nad którymi biedzą się specjaliści od tzw. sztucznej inteligencji, usiłujący je rozwiązywać za pomocą komputerów! (Chodzi o efektywność uczenia się, rozpoznawanie form, etc.).

Zdając sobie sprawę, że C++ może być pewnej grupie Czytelników nieznany, przygotowałem dla nich w dodatku A minikurs tego języka. Polega on na równoległej prezentacji struktur składniowych w C++ i Pascalu, tak aby poprzez porównywanie fragmentów kodu nauczyć się czytania listingów prezentowanych w tej książce. Kilkustronicowy dodatek nie zastąpi oczywiście podręcznika poświęconego tylko i wyłącznie C++, umożliwi jednak lekturę książki osobom pragnącym z niej skorzystać bez konieczności poznawania nowego języka.

Poprawność algorytmów

Wpisanie programu do komputera, skompilowanie go i uruchomienie jeszcze nie gwarantują, że kiedyś nie nastąpi jego załamanie (cokolwiek by to miało znaczyć w praktyce). O ile jednak w przypadku niewinnych domowych aplikacji nie ma to specjalnego znaczenia (w tym sensie, że tylko my ucierpimy), to w momencie zamierzonej komercjalizacji programu sprawa znacznie się komplikuje. W grę zaczyna wchodzić nie tylko kompromitacja programisty, ale i jego odpowiedzialność za ewentualne szkody poniesione przez użytkowników programu.

Od błędów w swoich produktach nie ustrzegają się nawet wielkie koncerny programistyczne — w miesiąc po kampanii reklamowej produktu X pojawiają się po cichu „darmowe" (dla legalnych użytkowników) uaktualnione wersje, które nie mają wcześniej niezauważonych błędów. Popularne systemy operacyjne, takie jak np. Windows lub Mac OS, posiadają wbudowane mechanizmy automatycznej aktualizacji przez Internet, które służą naprawianiu wadliwych funkcji systemu lub bieżącej reakcji na zagrożenia (np. wirusy).

Mamy tu do czynienia z pośpiechem, którego celem jest wyprzedzenie konkurencji, co usprawiedliwia wypuszczanie przez dyrekcje firm niedopracowanych produktów — ze szkodą dla użytkowników niemających żadnych możliwości obrony przed tego typu praktykami. Z drugiej jednak strony uniknięcie błędów w programach wcale nie jest problemem banalnym i stanowi temat poważnych badań naukowych[11]!

Zajmijmy się jednak czymś bliższym rzeczywistości typowego programisty: pisze on program i chce uzyskać odpowiedź na pytanie: „Czy będzie on działał poprawnie w każdej sytuacji, dla każdej możliwej konfiguracji danych wejściowych?". Odpowiedź jest tym trudniejsza, im bardziej skomplikowane są procedury, które zamierzamy badać. Nawet w przypadku pozornie krótkich w zapisie programów liczba sytuacji, które mogą zaistnieć w praktyce, wyklucza ręczne przetestowanie programu. Pozostaje więc stosowanie dowodów natury matematycznej, zazwyczaj dość skomplikowanych. Jedną z możliwych ścieżek, którymi można dojść do stwierdzenia formalnej poprawności algorytmu, jest stosowanie metody *niezmienników* (zwanej niekiedy metodą *Floyda*). Mając dany algorytm, możemy łatwo wyróżnić w nim pewne kluczowe punkty, w których dzieją się interesujące dla danego algorytmu rzeczy. Ich znalezienie nie jest zazwyczaj trudne: ważne są momenty inicjalizacji zmiennych, którymi będzie operować procedura, testy zakończenia algorytmu, pętla główna itd. W każdym z tych punktów możliwe jest określenie pewnych zawsze prawdziwych warunków — tzw. *niezmienników*. Można sobie zatem wyobrazić, że dowód formalnej poprawności algorytmu może być uproszczony do stwierdzenia zachowania prawdziwości niezmienników dla dowolnych danych wejściowych.

[11] Formalne badanie poprawności systemów algorytmicznych jest możliwe przy użyciu specjalistycznych języków stworzonych do tego celu.

Dwa typowe sposoby stosowane w praktyce to:

- sprawdzanie stanu punktów kontrolnych za pomocą *debuggera* (odczytujemy wartości pewnych ważnych zmiennych i sprawdzamy, czy zachowują się poprawnie dla pewnych reprezentacyjnych danych wejściowych[12]).
- formalne udowodnienie (np. przez indukcję matematyczną) zachowania niezmienników dla dowolnych danych wejściowych.

Zasadniczą wadą powyższych zabiegów jest to, że są one nużące i potrafią łatwo zabić całą przyjemność związaną z efektywnym rozwiązywaniem problemów za pomocą komputera. Tym niemniej Czytelnik powinien być świadom istnienia również i tej strony programowania. Jedną z prostszych (i bardzo kompletnych) książek, którą można polecić Czytelnikowi zainteresowanemu formalną teorią programowania, metodami generowania algorytmów i sprawdzania ich własności, jest **[Gri84]** — entuzjastyczny wstęp do niej napisał sam Dijkstra[13], co jest chyba najlepszą rekomendacją dla tego typu pracy. Inny tytuł o podobnym charakterze, **[Kal90]**, można polecić miłośnikom formalnych dowodów i myślenia matematycznego. Metody matematycznego dowodzenia poprawności algorytmów są prezentowane w tych książkach w pewnym sensie niejawnie; zasadniczym celem jest dostarczenie narzędzi, które umożliwią quasi-automatyczne generowanie algorytmów.

Każdy program wyprodukowany za pomocą tych metod jest automatycznie poprawny — pod warunkiem, że nie został po drodze popełniony jakiś błąd. Wygenerowanie algorytmu jest możliwe dopiero po jego poprawnym zapisaniu wg schematu:

{warunki wstępne[14]} **poszukiwany program** {warunki końcowe}

Przy pewnej dozie doświadczenia możliwe jest wyprodukowanie ciągu instrukcji, które powodują przejście z „warunków wstępnych" do „warunków końcowych" — wówczas formalny dowód poprawności algorytmu jest zbędny. Można też podejść do problemu z innej strony, mając dany zespół warunków wstępnych i pewien program: czy jego wykonanie zapewnia ustawienie pożądanych warunków końcowych?

Czytelnik może nieco się obruszyć na ogólnikowość powyższego wywodu, ale jest ona wymuszona przez rozmiar tematu, który wymaga w zasadzie osobnej książki! Pozostaje zatem tylko ponowić zaproszenie do lektury niektórych zacytowanych wyżej pozycji bibliograficznych — niestety w momencie pisania tej książki niedostępnych w polskich wersjach językowych.

---

[12] Stwierdzenia: „ważne zmienne", „poprawne" zachowanie programu, „reprezentatywne" dane wejściowe, etc. należą do gatunku bardzo nieprecyzyjnych i są ściśle związane z konkretnym programem, którego analizą się zajmujemy.

[13] Jeśli już jesteśmy przy tym autorze, to warto polecić przynajmniej pobieżną lekturę **[DF89]**, która stanowi dość dobry wstęp do metodologii programowania.

[14] Wartości zmiennych, pewne warunki logiczne je wiążące, etc.