1. Wstęp

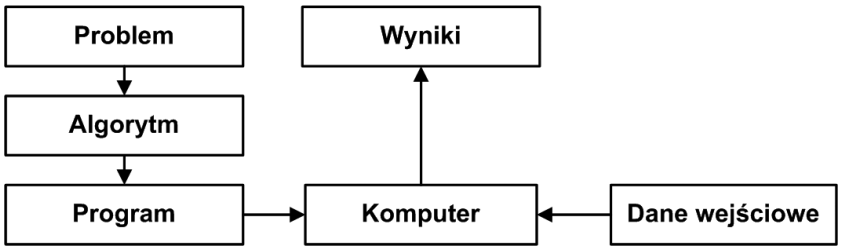
1.1. Zaproszenie

Podręcznik niniejszy przeznaczony jest dla czytelników, którzy chcieliby nauczyć się programowania, czyli sposobu tworzenia programów wykonywanych przez komputery. Początkowe rozdziały przedstawiają pojęcia podstawowe dotyczące komputerów i programowania, dalsza część zawiera opis popularnego języka programowania C. Podręcznik ten przeznaczony jest dla szerokiego kręgu czytelników – korzystanie z niego nie wymaga żadnej wstępnej wiedzy specjalistycznej.

1.2. Pojęcia podstawowe

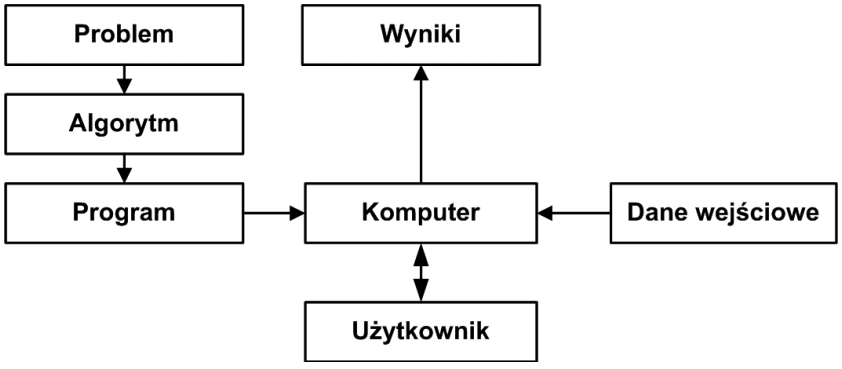
Komputer to urządzenie do automatycznego przetwarzania informacji, którego działanie polega na wykonywaniu *programów*. Ta krótka definicja podkreśla dwa istotne fakty. Po pierwsze, komputery służą do przetwarzania informacji, czyli do przetwarzania danych, bowiem *dane* to zapisane w sposób trwały informacje. Proces ten polega na wyznaczaniu nowych danych (wyników, rezultatów), na podstawie danych uprzednio dostępnych (danych wejściowych, początkowych). Po drugie, przebieg procesu przetwarzania danych jest zadawany za pomocą programu, czyli precyzyjnego i jednoznacznego wykazu działań, które komputer ma automatycznie wykonać. Najbardziej nowoczesny komputer jest więc bezużyteczny, o ile nie jest dla niego dostępny żaden program. Z drugiej strony działanie wszystkich komputerów sprowadza się do jednej, zawsze takiej samej czynności – do wykonywania programów.

Abstrakcyjny sposób rozwiązania dowolnego problemu nazywany jest *algorytmem*. Określenie *abstrakcyjny* oznacza w tym przypadku sposób wymyślony przez pewną osobę (lub grupę osób) i pozostający na razie wyłącznie w jej umyśle (w ich umysłach). Algorytm może zostać zapisany, postać tego zapisu zależy najczęściej od rodzaju rozwiązywanego problemu. Na przykład algorytm ugotowania grochówki można zapisać w języku naturalnym, jako przepis kulinarny. Komputery służą, jak już powiedziano, do przetwarzania danych, stąd program wykonywany przez komputer jest zapisem algorytmu rozwiązującego pewien problem przetwarzania danych.



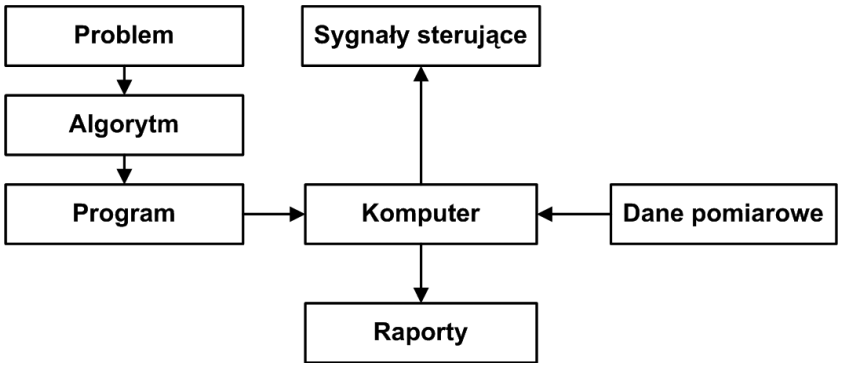
Rys.1.1. Podstawowy schemat wykorzystania komputera

Algorytm, będący rozwiązaniem pewnego problemu, został zapisany jako program (rys. 1.1). Program ten jest wykonywany przez komputer, podczas realizacji programu komputer korzysta z danych wejściowych. Efektem pracy komputera są wyniki przekazywane użytkownikowi (np. wyświetlane na ekranie monitora lub drukowane za pomocą drukarki). Taki schemat użytkowania komputera był przede wszystkim stosowany w początkowym okresie rozwoju informatyki. Obecnie bardzo często użytkownik bierze aktywny udział w wykonaniu programu, dostarczając dodatkowych danych i decydując o kolejnych etapach procesu przetwarzania (rys. 1.2). Typowym przykładem takiego interakcyjnego sposobu wykorzystania komputera są programy edycyjne, umożliwiające przygotowanie, poprawianie, przechowywanie i drukowanie różnych dokumentów.



Rys. 1.2. Interakcyjna współpraca z komputerem

Komputery mogą być również wykorzystywane do sterowania różnymi procesami. Zwane w tym przypadku *mikrokontrolerami* lub *mikroprocesorami* sterującymi nadzorują działanie bardzo wielu popularnych urządzeń, takich jak na przykład telefony komórkowe, telewizory, samochody (np. sterowanie zapłonem, blokada antypoślizgowa ABS), pralki czy urządzenia klimatyzacyjne. Bardzo szeroko komputery są również wykorzystywane do sterowania procesami przemysłowymi. W takich zastosowaniach schemat wykorzystania komputera jest nieco inny (rys. 1.3).



Rys. 1.3. Wykorzystanie komputera do sterowania

Program, będący zapisem pewnego algorytmu sterowania, jest wykonywany przez komputer, który na bieżąco otrzymuje dane pomiarowe. Wynikiem pracy programu są sygnały sterujące i ewentualnie raporty, gromadzone w pamięci trwałej komputera. Na przykład dla wspomnianego układu antypoślizgowego ABS samochodu, dane pomiarowe to ciąg odczytywanych co stały okres czasu wartości prędkości obrotowej koła – wykrycie przez program zbyt gwałtownego zmniejszenia prędkości obrotowej powoduje wygenerowanie sygnału sterującego, zmniejszającego nacisk szczęk hamulcowych, związanych z danym kołem.

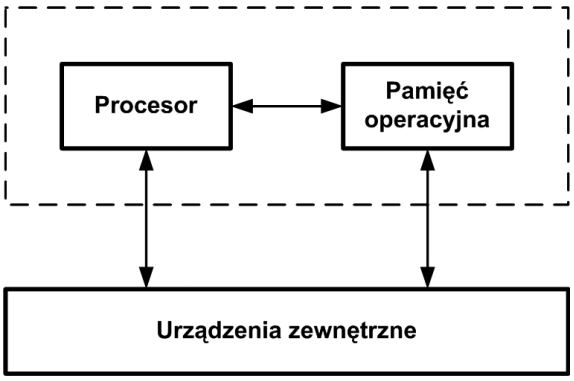
W chwili obecnej komputery to urządzenia elektroniczne (pomijając nieliczne konstrukcje eksperymentalne). Oznacza to, że dane zapisywane są za pomocą odpowiednich stanów układów elektromagnetycznych, wykorzystując ich pojemność, ładunek elektryczny czy namagnesowanie. Przetwarzanie tak zapisanych danych możliwe jest dzięki wykorzystaniu zjawisk elektromagnetycznych – przepływu prądu elektrycznego czy zmiany charakteru namagnesowania. Taki sposób odwzorowania informacji preferuje wykorzystanie do zapisu wartości liczbowych *systemu binarnego (dwójkowego)*, w którym występują tylko dwie cyfry: 0 i 1, zwane *bitami*. Cyfra 0 może być reprezentowana przez brak przepływu prądu lub przez namagnesowanie typu N, cyfrę 1 może reprezentować wystąpienie przepływu prądu lub namagnesowanie typu S. Komputery przechowują więc i przetwarzają wyłącznie ciągi cyfr binarnych (*ciągi zerojedynkowe*, np. 10010101). Oczywiście dla wygody użytkownika dane wejściowe i wyniki pracy programów zapisywane są z użyciem systemu dziesiętnego.

Współpraca użytkownika z komputerem odbywa się za pośrednictwem wyspecjalizowanego programu zwanego *systemem operacyjnym.* System ten zarządza wszystkimi zasobami komputera oraz akceptuje i realizuje polecenia wydawane przez użytkownika. Polecenia te mogą dotyczyć uruchamiania wskazanych programów, zarządzania plikami dyskowymi czy też wykonywania różnych operacji administracyjnych. Początkowo systemy operacyjne udostępniały jedynie interfejs tekstowy – poszczególne polecenia miały postać komend z odpowiednimi parametrami (np. *dir*, *cd E:*, *ipconfig*). Aby uruchomić dany program należało wpisać jako pole-

cenie jego nazwę i podać ewentualne parametry (np. *KopiaPliku DanePrywatne.txt*). Obecnie większość systemów operacyjnych oferuje użytkownikowi interfejs graficzny – poszczególne programy uruchamia się wybierając myszką reprezentujące je znaki graficzne (ikony). Niemniej nadal możliwe jest tekstowe sterowanie pracą komputera za pomocą *okna konsoli* (tzw. *wiersz poleceń*). Programy przykładowe zamieszczone w niniejszym podręczniku są przeznaczone do uruchamiania właśnie w oknie konsoli.

1.3. Budowa komputera

Podstawowe elementy komputera przedstawione zostały na rysunku 1.4. Centralną częścią każdego komputera jest *procesor* – układ elektroniczny wykonujący programy. Programy te oraz wymagane przez nie dane, pobierane są z *pamięci operacyjnej*. Wprowadzanie i wyprowadzanie danych, ich przechowywanie czy przesyłanie umożliwiają *urządzenia zewnętrzne*.



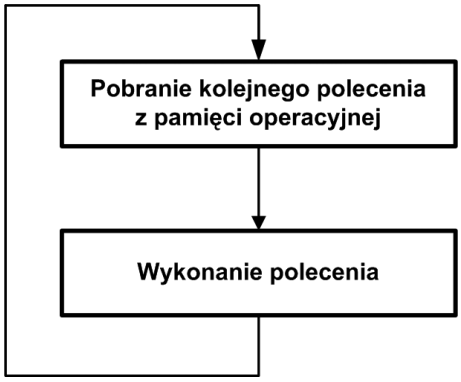
Rys. 1.4. Budowa komputera

1.3.1. Procesor

Zadaniem *procesora* jest wykonanie programu. Każdy typ procesora posiada swój charakterystyczny i niepowtarzalny zestaw poleceń (rozkazów), które potrafi wykonywać. Aby program mógł być wykonany przez procesor musi więc mieć postać ciągu akceptowanych przez niego poleceń. Polecenia te są bardzo proste – można je podzielić na następujące grupy:

- polecenia przepisywania danych pomiędzy procesorem a pamięcią operacyjną,
- polecenia wykonania podstawowych operacji arytmetycznych (+, -, *, /),
- polecenia umożliwiające podejmowanie decyzji i przechodzenie do innych fragmentów programu,
- polecenia współpracy z urządzeniami zewnętrznymi.

O programie będącym ciągiem poleceń procesora mówi się, iż jest on zapisany w *języku wewnętrznym* lub w *języku asemblera*. Na szczęście programiści bardzo rzadko muszą zapisywać programy bezpośrednio w języku wewnętrznym – dla ich wygody wprowadzono znacznie dogodniejszy sposób zapisu programów, za pomocą tzw. *języków proceduralnych* (zwanych dawniej *językami programowania wysokiego poziomu*). Ponieważ procesor może wykonywać wyłącznie program zapisany w języku wewnętrznym konieczne jest przekształcenie na tę postać programów zapisanych w językach wyższego poziomu.



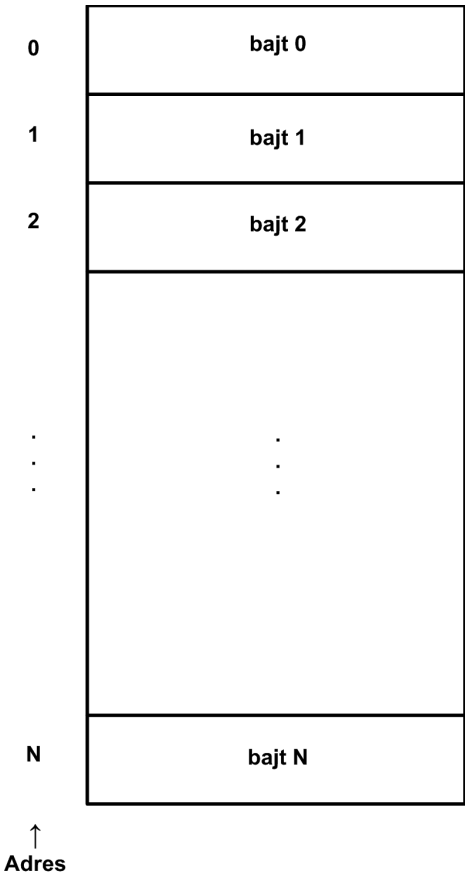
Rys. 1.5. Podstawowy cykl pracy komputera

Programy będące ciągami poleceń i dane przechowywane są w pamięci operacyjnej. Z tej pamięci procesor pobiera więc do wykonania kolejne polecenie i następnie je wykonuje. Podczas wykonywania polecenia z pamięci operacyjnej mogą być pobierane odpowiednie dane (rys. 1.5).

1.3.2. Pamięć operacyjna

Pamięć operacyjna komputera służy do przechowywania programów (zapisanych jako ciągi poleceń) oraz danych. Przymiotnik operacyjna oznacza, iż procesor posiada do niej bezpośredni dostęp. Przesyłanie danych pomiędzy procesorem i pamięcią operacyjną powinno odbywać się z maksymalną, możliwą do osiągnięcia prędkością, ponieważ prędkość ta ma decydujący wpływ na ostateczną prędkość pracy komputera.

Pamięć operacyjna składa się z ponumerowanych *komórek*, służących do przechowywania ciągów binarnych (rys. 1.6). Od wielu już lat obowiązuje niepisana umowa, iż w pojedynczej komórce pamięci przechowywany jest ciąg binarny o długości 8 bitów, zwany *bajtem*. Stąd popularnie mówi się o pamięci złożonej z bajtów. Numer komórki (bajtu) nazywany jest jej *adresem*, współczesne komputery wyposażane są w pamięci operacyjne o wielkości Gigabajtów (1 Gigabajt = 1024^3 bajtów).



Rys. 1.6. Budowa pamięci operacyjnej

1.3.3. Urządzenia zewnętrzne

Liczne rodzaje urządzeń zewnętrznych dołączanych do komputera można podzielić na następujące grupy:

- urządzenia do ręcznego wprowadzania danych,
- urządzenia do wyprowadzania danych,
- pamięci zewnętrzne,
- urządzenia do przesyłania danych do/z sieci komputerowej oraz do/z innych urządzeń elektronicznych
- urządzenia specjalistyczne.

W ramach prezentowanego w tym podręczniku kursu programowania przedstawiony zostanie sposób korzystania z urządzeń należących do pierwszych trzech z wymienionych grup. Będzie to *klawiatura alfanumeryczna*, *monitor ekranowy* i *pamięć dyskowa* (rozdział 9.).

1.4. Reprezentacja danych w pamięci komputera

W pamięci komputera (operacyjnej lub zewnętrznej) mogą być przechowywane dane różnych rodzajów, np.:

- liczby,
- teksty,
- obrazy,
- dźwięki.

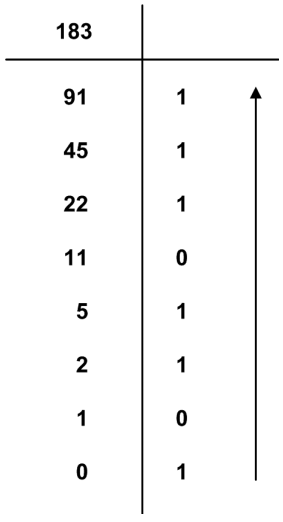
Ale, jak już powiedziano, pamięć komputera przechowuje wyłącznie ciągi zerojedynkowe – stąd wszystkie te rodzaje danych muszą zostać zapisane właśnie za pomocą takich zerojedynkowych ciągów. Przedstawiony teraz zostanie sposób reprezentowania w pamięci komputera liczb i tekstów (sposoby reprezentowania obrazów i dźwięków omawiane są w podręcznikach dotyczących technik multimedialnych).

1.4.1. Liczby

W przypadku liczb całkowitych należy podać sposób przekształcenia (konwersji) liczb zapisanych w powszechnie stosowanym systemie dziesiętnym na równe co do wartości liczby zapisane w systemie binarnym i odwrotnie.

Konwersja dziesiętno-binarna

Aby dokonać tej konwersji należy całkowitą, dodatnią liczbę dziesiętną dzielić przez 2 i zapisywać w następnej linii wartość całkowitą ilorazu oraz resztę. Proces ten kończy się po osiągnięciu ilorazu równego 0. Odczytując reszty w kolejności od ostatniej wyznaczonej do pierwszej otrzymuje się liczbę binarną o wartości równej wyjściowej liczbie dziesiętnej (rys. 1.7).



Rys. 1.7. Konwersja dziesiętno-binarna

Konwersja binarno-dziesiętna

Dla dokonania zamiany całkowitej, dodatniej liczby binarnej na liczbę dziesiętną wykorzystuje się wzór na wartość liczby zapisanej w wagowym systemie pozycyjnym:

$$W = a_0 p^0 + a_1 p^1 + a_2 p^2 + ... \qquad [1]$$

gdzie:

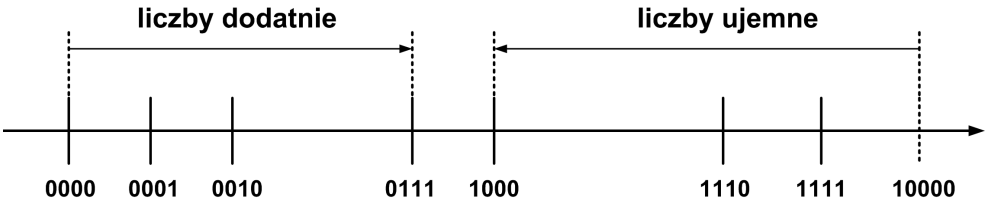
W – wartość liczby,

a_i - wartość i-tej cyfry,

p – podstawa systemu zapisu.

Dla systemu dziesiętnego podstawa p = 10, a kolejne potęgi p to 1, 10, 100,... W przypadku systemu binarnego p = 2, a kolejne potęgi p to 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ... Aby więc liczbę całkowitą binarną B = 10110111 zamienić na liczbę D o takiej samej wartości, zapisaną w systemie dziesiętnym, należy kolejne cyfry liczby binarnej i kolejne wartości potęg binarnych podstawić do wzoru [1] (pamiętając, że cyfra znajdująca się na prawym końcu liczby binarnej ma indeks 0):

D = 1*1 + 1*2 + 1*4 + 0*8 + 1*16 + 1*32 + 0*64 + 1*128
czyli D = 183.



Rys. 1.8. Uzupełnienie do 2

Liczby całkowite ze znakiem (ujemne albo dodatnie) przechowywane są w pamięci komputera w formacie zwanym *uzupełnieniem do 2* (*Uzp2*). Na rys. 1.8 przedstawiono ten format dla liczb 4-bitowych. Jak widać kolejne liczby dodatnie o wartościach dziesiętnych od 0 do 7 zapisywane są w sposób naturalny z wykorzystaniem 3 bitów (najstarszy bit jest równy 0). Liczby ujemne o wartościach dziesiętnych od –1 do –8 zapisywane są licząc od wartości binarnej 10000 (ta wartość reprezentuje w tym przypadku 2, występujące w nazwie formatu). Dla liczb ujemnych najstarszy bit ma zawsze wartość 1. W tabeli 1.1 przedstawiono ciągi binarne, odpowiadające liczbom o wartościach dziesiętnych od –8 do +7. Taki sposób reprezentowania liczb dodatnich i ujemnych wydaje się skomplikowany – jest on jednak bardzo wygodny przy realizacji dodawania i odejmowania tych liczb.

W programowaniu stosowane są również liczby o podstawie 8 (*liczby ósemkowe* lub *oktalne*) i 16 (*liczby szesnastkowe* lub *heksadecymalne*). Przyczyną wykorzystywania takich liczb jest łatwość dokonania ich konwersji na liczbę binarną i odwrotnie. Dla obliczenia binarnej reprezentacji dowolnej liczby całkowitej ósemkowej/szesnastkowej należy każdą jej cyfrę zamienić na 3/4 bitowy odpowiednik (tab. 1.2). Konwersja liczby binarnej na postać ósemkową/szesnastkową polega na podziale ciągu binarnego na segmenty 3/4 bitowe i zastąpienie każdego segmentu odpowiednią cyfrą ósemkową/szesnastkową. Cyfry ósemkowe to cyfry 0 ... 7. Dla zapisu cyfr szesnastkowych (których jest 16) wykorzystuje się cyfry 0 ... 9 oraz litery A ... F (stosowane są też litery a ... f).

Tabela 1.1. Liczby czterobitowe dodatnie i ujemne

0	0000	–8	1000
+1	0001	–7	1001
+2	0010	–6	1010
+3	0011	–5	1011
+4	0100	–4	1100
+5	0101	–3	1101
+6	0110	–2	1110
+7	0111	–1	1111

Tabela 1.2. Cyfry szesnastkowe

ósemkowe

0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Liczby niecałkowite (rzeczywiste) przechowywane są w pamięci komputera jako tzw. *liczby zmiennopozycyjne.* Wartość liczby przechowywana jest w postaci wykładniczej jako

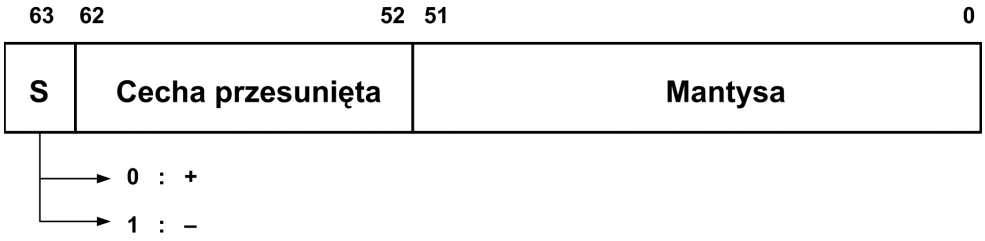
$$\pm m * 2^{\pm c}$$

gdzie:

m – *mantysa* liczby

c – *cecha* liczby

Mantysa zapisywana jest jako binarny ułamek właściwy, cecha jako liczba całkowita binarna (tzw. cecha przesunięta). Istnieje kilka rodzajów liczb zmiennopozycyjnych, różniących się zakresami przechowywanych wartości. Format najpopularniejszej liczby zmiennopozycyjnej (liczba podwójna – *double*) przedstawiono na rys. 1.9. Formaty te są zdefiniowane w zaleceniu IEEE 754.



Rys 1.9. Format zapisu liczby zmiennopozycyjnej typu *double*

1.4.2. Teksty

Aby utworzyć reprezentację tekstu w pamięci komputera należy zdefiniować sposób reprezentowania każdego znaku graficznego, występującego w alfabecie języka, w którym tekst ten został zapisany. Na przykład dla języka polskiego musi istnieć sposób jednoznacznej reprezentacji liter a ÷ ź, A ÷ Ź oraz cyfr 0 ÷ 9 (traktowanych jako znaki graficzne), znaków przestankowych i różnych znaków specjalnych (np. znaku @). Ponieważ w pamięci komputera przechowywane są jedynie ciągi binarne, których najprostszą interpretacją są liczby całkowite, należy każdemu znakowi graficznemu alfabetu przypisać unikatową liczbę całkowitą. Takie przypisanie nosi nazwę *kodowania*, liczba związana z danym znakiem graficznym zwana jest jego *kodem.* Pełen wykaz kodów dla znaków graficznych alfabetu to *tabela kodowania* lub po prostu *kod*.

Opracowano wiele tabel kodowania dla znaków graficznych (znaków alfanumerycznych) należących do alfabetów języków naturalnych. Do końca lat 80-tych XX wieku powszechnie stosowanym kodem był kod ASCII (Dodatek A). Kod ten przypisuje każdej literze, cyfrze i popularnym znakom specjalnym ciągi binarne

o długości 1 bajta. Wynika z tego, iż za pomocą tego kodu można reprezentować 256 różnych znaków graficznych. Nie jest to liczba wystarczająca, biorąc pod uwagę wielość znaków specjalnych i liter diakrytyzowanych (np. ą, ł, Ź, ä, õ, Ę) występujących w różnych językach naturalnych. Stąd w kodzie ASCII wyróżniono 128 znaków podstawowych (litery łacińskie, cyfry, znaki sterujące i podstawowe znaki specjalne) – znakom tym przypisano kody od 0 do 127. Pozostałe znaki graficzne (w tym polskie znaki diakrytyzowane) podzielono na grupy po 128 znaków – do kodowania każdej takiej grupy użyto takiego samego zakresu kodów od 128 do 255. Rezultatem wprowadzenia takiego rozwiązania jest niejednoznaczność kodowania. Nie wiadomo np. jakiemu znakowi graficznemu odpowiada kod 205 – trzeba wiedzieć do jakiej grupy znaków go odnieść, czyli jaką *stronę kodową* zastosować. Problem ten istnieje do chwili obecnej – zdarza się niekiedy, że np. zamiast polskiej litery *ó* wyświetlany jest jakiś dziwny symbol. Większość programów edycyjnych umożliwia wybranie sposobu kodowania, czyli wskazanie strony kodowej, która ma być stosowana do przetwarzanego tekstu. Poprawne wyświetlanie i drukowanie polskich liter zapewnia np. kodowanie *Eastern Europe*.

Niedostatki kodu ASCII spowodowały opracowanie w latach 90-tych XX wieku nowego, jednoznacznego standardu kodowania, zwanego *Unicode*. Kodowanie to przewiduje, w maksymalnym wariancie, stosowanie do reprezentowania znaków graficznych ciągów 32-bitowych. Dzięki temu można zakodować jednoznacznie ponad 4 mld znaków graficznych. Jest to liczba wystarczająca dla objęcia jednoznacznym kodowaniem wszystkich znaków graficznych należących do wszystkich języków naturalnych jakie istnieją na naszej planecie (włączając w to znaki języka chińskiego i znaki języków martwych) oraz tysięcy znaków specjalnych.

Nie można jednak było z dnia na dzień zarzucić stosowania kodu ASCII – za pomocą tego kodowania do końca XX wieku zapisano olbrzymie ilości danych, zostało ono też zastosowane w milionach różnych programów. Opracowano więc różne warianty systemu Unicode. Najprostszy z nich i najczęściej stosowany to wariant UTF-8. Zakłada on, że 128 podstawowych znaków graficznych jest kodowanych tak, jak w kodzie ASCII, za pomocą kodu jednobajtowego. Pozostałe znaki wymagają kodów o długości 2, 3 lub 4 bajtów. Dzięki takiemu rozwiązaniu można nadal korzystać z danych zapisanych w kodzie ASCII (w zakresie znaków podstawowych) i równocześnie kod jest jednoznaczny, czyli na podstawie liczbowej wartości kodu można zawsze wskazać znak graficzny, który on reprezentuje.

2. Programowanie

2.1 Zmienne i struktury danych

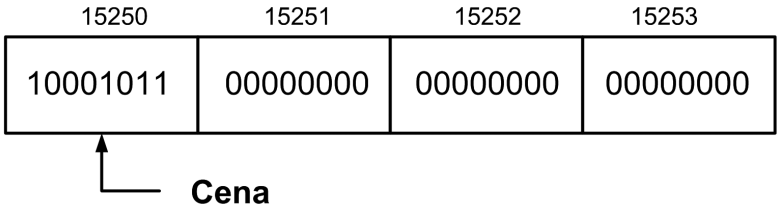
Program, który ma zostać wykonany przez komputer, jest zapisem algorytmu rozwiązującego pewien problem dotyczący przetwarzania danych. Dane w ramach programu są reprezentowane przez:

- zmienne,
- struktury danych.

Zmienne służą do reprezentowania danych pojedynczych, nie powiązanych ze sobą żadną relacją porządkującą. Każda zmienna może być uważana za parę:

{ *identyfikator* , *wartość* }

gdzie *identyfikator* jest unikatową nazwą zmiennej, czyli pojedynczym wyrazem danego języka. *Wartość zmiennej* zależy od jej rodzaju – większość zmiennych to zmienne liczbowe (numeryczne), ich wartościami są liczby całkowite lub zmiennopozycyjne. Identyfikatorem zmiennej programista posługuje się w programie, natomiast wartość zmiennej jest przechowywana w pamięci komputera za pomocą odpowiedniej liczby bajtów. I tak np. jeżeli identyfikatorowi `Cena` zostanie w programie nadana wartość 139, to w pamięci komputera zarezerwowane zostaną 4 bajty dla przechowywania binarnej postaci liczby 139 (rys. 2.1, dlaczego akurat 4 bajty zostanie wyjaśnione w rozdziale 3).



Rys. 2.1. Identyfikator i wartość zmiennej

Adres pierwszego z tej czwórki bajtów zostaje powiązany z identyfikatorem `Cena` i kolejne zmiany wartości tej zmiennej będą powodowały zmianę ciągu binarnego, w tych bajtach przechowywanego. Jak widać na rysunku 2.1 najmłodsza część ciągu binarnego reprezentującego liczbę dziesiętną 139 została zapisana w bajcie

o najmniejszym adresie (w bajcie najbliższym początku pamięci). Taki sposób zapisu danych liczbowych nosi nazwę mało-końcowego (ang. *little endian*) i jest powszechnie stosowany przy zapisie liczb w pamięci komputera. Operacja nadania nowej wartości zmiennej jest zapisywana na różne sposoby, na razie będzie w tym celu używany znak strzałki w lewo.

```
Cena ← 139
. . . . . .
Cena ← 99
```

Nadanie nowej wartości zmiennej powoduje usunięcie jej poprzedniej wartości. W programach często występują zapisy o postaci:

```
Cena ← Cena + 14
```

Zapis ten nie zawiera sprzeczności, jak to by się mogło wydawać na pierwszy rzut oka. Wskazuje on, że należy odczytać dotychczasową wartość zmiennej Cena (czyli 99), powiększyć ją o 14 i wynik zapisać jako nową wartość zmiennej Cena (113).

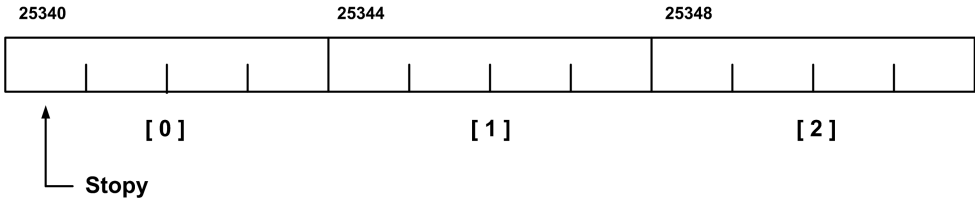
Jeżeli pomiędzy pewną liczbą danych występuje powiązanie kolejnościowe lub powiązanie przynależnościowe, to taki zestaw danych zapisuje się za pomocą *struktury danych.*

Najprostszą strukturą danych jest *tablica*, zawierająca ciąg uporządkowanych kolejno wartości takiego samego typu. Wartości te to *elementy tablicy.* W najprostszym przypadku (*tablicy jednowymiarowej*) tworzą one ciąg ponumerowany za pomocą kolejnych liczb całkowitych dodatnich. Numeracja ta rozpoczyna się od wartości 0. Każda tablica posiada unikatowy identyfikator. Numer elementu tablicy (czyli jego *indeks*) zapisywany jest w nawiasach prostokątnych.

```
Wymiary [ 0 ]          // początkowy element tablicy Wymiary
Wymiary [ 15 ]         // element tablicy Wymiary o indeksie 15
Wymiary [ ind_1 ]      // element tablicy Wymiary o indeksie
                       // równym wartości zmiennej ind_1
```

Jeżeli w programie użyta zostanie np. tablica Stopy zawierająca 3 elementy, będące liczbami całkowitymi, to w pamięci komputera zarezerwowany zostanie spójny obszar o długości 3 * 4 = 12 bajtów dla przechowywania wartości elementów tej tablicy (rys. 2.2). Z identyfikatorem Stopy powiązany zostanie adres początkowego bajtu, początkowego elementu tej tablicy (czyli elementu Stopy[0]).

Tablice jednowymiarowe są też wykorzystywane do przechowywania danych tekstowych – elementy takiej tablicy to kody znaków graficznych pewnego alfabetu. Niekiedy, dla zaznaczenia końca tekstu, jako ostatni dodaje się element o wyróżnionej wartości (punkt 3.7).

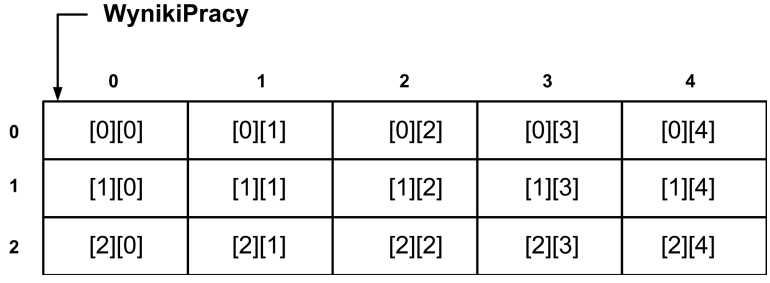


Rys. 2.2. Tablica jednowymiarowa

Tablice dwuwymiarowe są reprezentacją powszechnie stosowanych tabelek złożonych z wierszy i kolumn (i również znanych z algebry *macierzy*). Aby wskazać element takiej tablicy należy więc podać numer wiersza (liczony od 0) i numer kolumny (również liczony od 0) w kolejności: numer_wiersza, numer_kolumny.

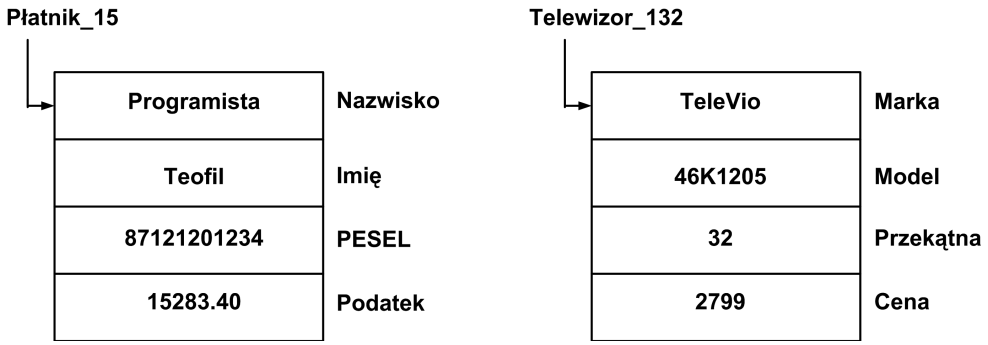
```
Wynagrodzenia [12] [15]      // wiersz numer 12, kolumna numer 15
Wynagrodzenia [x] [y]        // wiersz o numerze będącym wartością x
                             // kolumna o numerze będącym wartością y
```

Gdy w programie wystąpi np. tablica Wyniki_Pracy o 3 wierszach i 5 kolumnach, której elementami są liczby całkowite, to w pamięci komputera rezerwowany jest spójny obszar o długości 3 * 5 * 4 = 60 bajtów (rys. 2.3). Z identyfikatorem WynikiPracy powiązany zostanie adres początkowego bajtu elementu z początkowego wiersza i początkowej kolumny (czyli elementu WynikiPracy [0] [0]).



Rys. 2.3. Tablica dwuwymiarowa

Tablice stosowane są w przypadkach, gdy dane powiązane są kolejnościowo (w jednym lub dwu wymiarach). Wszystkie elementy tablicy muszą być takiego samego typu. W procesie przetwarzania danych często występują również dane powiązane przynależnościowo, to jest dane różnych typów opisujące pewien przedmiot czy pewną osobę. Struktura danych, grupująca dane będące opisem przedmiotu czy osoby, nosi nazwę *rekordu* lub po prostu *struktury*. Elementy struktury to pola, każde pole może być innego typu (rys. 2.4).



Rys. 2.4. Struktury

Struktura Płatnik_15 składa się z 4 pól. Pierwsze dwa z nich (Nazwisko i Imię) to pola tekstowe (czyli jednowymiarowe tablice kodów znaków graficznych), pole PESEL zawiera liczbę całkowitą, a pole Podatek liczbę zmiennopozycyjną. Podobnie pola struktury Telewizor_132 zawierają dane różnych typów.

Istnieje wiele innych rodzajów struktur danych, niektóre z nich zostaną przedstawione w punktach 6.3 i 6.4 .

2.2. Zapisywanie algorytmów

2.2.1. Schematy blokowe

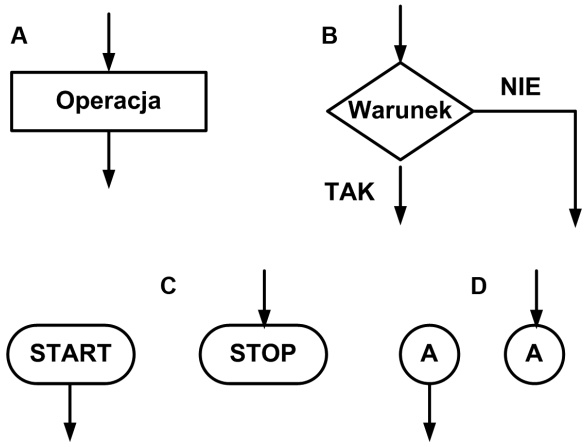
Jednym z prostszych sposobów zapisywania algorytmów (czyli pisania programów) jest użycie do tego celu notacji graficznej, zwanej językiem *schematów blokowych* (lub *siecią działań*). Taki zapis umożliwia nadanie algorytmowi trwałej formy (czyli formy programu), nie umożliwia on jednak przekazania tak przygotowanego programu do wykonania przez komputer (pomijając niektóre systemy edukacyjne). Niemniej rysowanie schematów blokowych pozwala na zapoznanie się z podstawowymi składnikami, z których budowane są programy.

Dla potrzeb tego podręcznika wystarczające są 4 rodzaje bloków (rys. 2.5). Pełen zestaw bloków dla języka schematów blokowych definiuje norma PN-75/E-01226.

Blok A zawiera opis dowolnej operacji przetwarzania danych. Może to być operacja bardzo prosta (np. Numer ← Numer + 1) lub bardzo złożona (np. Wyznacz rozwiązania równania różniczkowego R_1). Ważne jest, że blok operacji zawsze posiada jedno wejście i jedno wyjście.

Blok B to blok warunkowy, czyli tzw *predykat,* umożliwiający zapisanie w programie operacji podejmowania decyzji. Posiada on jedno wejście i dwa wyjścia. Jeżeli warunek wpisany w bloku warunkowym jest prawdziwy, to nastąpi przejście wyjściem TAK, przeciwnie, gdy warunek nie jest prawdziwy, wyjściem NIE. Położenie graficzne wyjść TAK/NIE można zmieniać, ale zawsze musi istnieć jedno wejście do bloku warunkowego i dwa z niego wyjścia. Przykładowy warunek może mieć postać

Alfa > 0. Jeżeli po wejściu do takiego bloku zmienna Alfa ma wartość większą od 0, to nastąpi wyjście drogą TAK, przeciwnie, gdy Alfa jest mniejsza lub równa 0, wybrana zostanie droga NIE.



Rys. 2.5. Bloki podstawowe

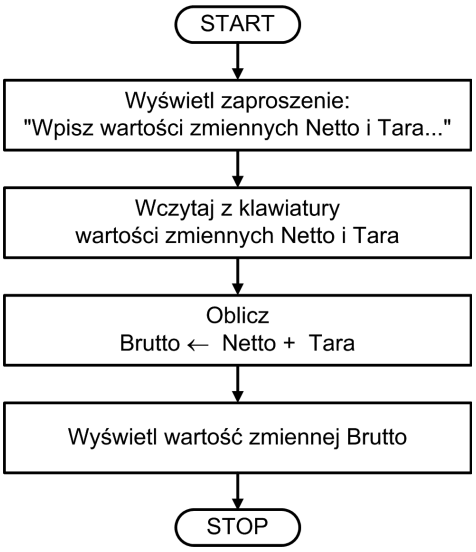
Bloki z grupy C służą do rozpoczynania i kończenia schematu blokowego. W każdym schemacie powinien wystąpić dokładnie jeden blok START i przynajmniej jeden blok STOP.

Ostatnia grupa bloków (D) zawiera proste bloki służące do łączenia części schematu blokowego, które znajdują się na różnych stronach.

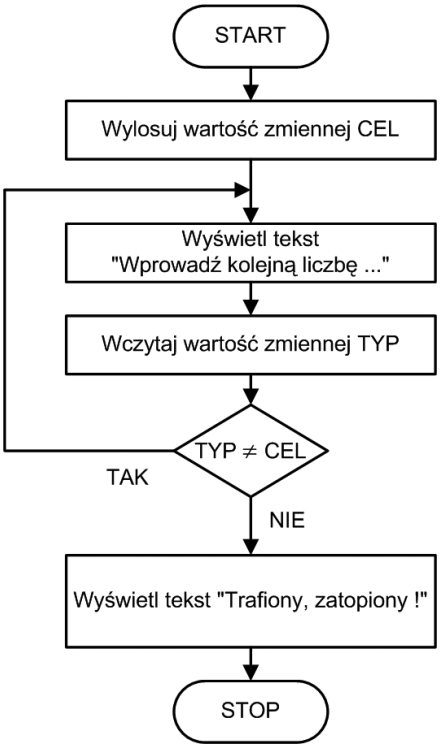
Budowanie schematu blokowego będącego zapisem pewnego algorytmu przetwarzania danych rozpoczyna się od bloku START. Następnie można wprowadzać i łączyć ze sobą dowolną liczbę bloków operacji i bloków warunkowych. Koniec programu zapisanego jako schemat blokowy następuje po osiągnięciu bloku STOP.

Rysunek 2.6 zawiera przykład konkretnego programu obliczającego wartość Brutto ← Netto + Tara. Przygotowując taki program należy uczynić go uniwersalnym, czyli takim, który wyliczy wartość Brutto do dowolnych, aktualnych wartości Netto i Tara. Konkretne liczbowe wartości aktualne, które mają zostać wykorzystane w danym wykonaniu programu, powinny zostać podane przez użytkownika podczas wykonywania programu. Program ten ma charakter liniowy – nie występują w nim żadne bloki warunkowe.

Jak już powiedziano programy zapisane za pomocą schematów blokowych nie bardzo nadają się do wykonania przez komputer. Program tak zapisany można jedynie wykonać ręcznie, przechodząc od startu do stopu, wykonując kolejne operacje (obliczenia) i notując na kartce kolejne wartości zmiennych. Takie wykonanie programu jest działaniem wysoce kształcącym, pozwala na dokładne zrozumienie procesu przetwarzania danych.



Rys. 2.6. Program liniowy



Rys. 2.7. Program z rozgałęzieniem

Na rysunku 2.7 pokazano prosty program realizujący grę Zgadnij_Liczbę. Po wylosowaniu wartości, którą gracz powinien zgadnąć, jest on proszony o podanie swojej kandydatki. W bloku warunkowym następuje sprawdzenie, czy gracz wprowadził poprawną wartość, jeżeli nie, to następuje przejście z powrotem do wyświetlania zaproszenia do dalszej gry. Gdy TYP = CEL następuje wyświetlenie informacji o trafieniu i koniec programu.

2.2.2. Języki programowania

Aby program mógł zostać wykonany przez komputer musi on zostać zapisany za pomocą odpowiedniego języka programowania. W rozdziale 1. wspomniano o języku wewnętrznym, w którym program ma postać ciągu poleceń realizowanych bezpośrednio przez procesor. Znacznie wygodniejsze jest użycie do zapisu programu jednego z licznych *języków proceduralnych* (zwanych dawniej *językami programowania wysokiego poziomu* – język wewnętrzny jest językiem niskiego poziomu). Języki takie należą do języków sztucznych o dokładnie zdefiniowanej składni. Pozwalają one między innymi na:

- deklarowanie zmiennych i struktur danych, które mają być użyte w programie,
- proste zapisywanie wyrażeń arytmetycznych i logicznych,
- stosowanie instrukcji sterujących, zawierających proste słowa języka angielskiego, do określania kolejności wykonania operacji (np. while, if),
- podział programu na podprogramy i ich wywoływanie,
- korzystanie z obszernych bibliotek zawierających między innymi podprogramy obliczania wartości funkcji arytmetycznych czy trygonometrycznych, podprogramy umożliwiające korzystanie z urządzeń zewnętrznych itp.

Języki proceduralne zaczęto definiować w latach 50-tych XX wieku (pierwszy masowo stosowany język programowania nosi nazwę FORTRAN). Spośród setek opracowanych do dnia dzisiejszego języków programowania tylko kilka jest na większą skalę stosowanych w praktyce. Wymienić tu można:

- języki programowania strukturalnego: C, Pascal, Ada, PL1,
- języki programowania obiektowego: C++, C#, Object Pascal, Java
- języki opisu dokumentów i stron WWW: HTML, XML, PHP, JAVA SCRIPT.

Ponadto zdefiniowano wiele języków specjalistycznych, np. LISP – do przetwarzania struktur listowych, SIMULA – do zapisu programów symulacyjnych itp.

Wyjaśnienia wymagają jeszcze terminy *programowanie strukturalne* i *programowanie obiektowe*. Zasadę programowania strukturalnego wprowadzono dla ograniczenia chaosu, jaki łatwo wprowadzić do programu, stosując przejścia pomiędzy jego dowolnymi miejscami. Beztroscy programiści tworzyli programy bardzo zagmatwane, sprawdzenie ich poprawności było często prawie niemożliwe. Zasada programowania strukturalnego ogranicza tę samowolę, przejście pomiędzy różnymi

miejscami programu może odbywać się wyłącznie w sposób ustandaryzowany, za pomocą struktur sterowania (punkt 2.3.3). Wykazano, iż dowolny algorytm można zapisać stosując zasady programowania strukturalnego.

Koncepcja programowania obiektowego pojawiła się w latach 60-tych, a na szeroką skalę została wprowadzona w latach 90-tych XX wieku. Metoda ta zakłada, że w programie tworzone będą *obiekty* będące odwzorowaniem odpowiednich wycinków rzeczywistości. Obiekt grupuje pewną liczbę danych i struktur danych oraz zestaw operacji dane te przetwarzających. Ograniczenia, które można w programowaniu obiektowym narzucić na możliwości dostępu do poszczególnych grup danych, pozwalają na tworzenie programów o znacznie wyższej odporności na błędy powstające podczas ich wykonywania. Języki programowania obiektowego są obecnie najczęściej stosowanymi językami programowania.

2.2.3. Poprawność programów

Przyjmując, że algorytm jest poprawny, czyli rzeczywiście umożliwia uzyskanie poprawnych rozwiązań danego problemu przetwarzania danych, należy zastanowić się, czy poprawny jest też program będący tego algorytmu zapisem. Programowanie jest procesem, w którym łatwo popełnić błędy, a z drugiej strony wykazanie poprawności programu nie jest zadaniem łatwym. Dla prostych programów można stosować formalne metody dowodzenia poprawności, są one jednak całkowicie niepraktyczne w zastosowaniu do wielkich programów profesjonalnych. Jedyne praktyczne możliwości oceny poprawności programu to przeglądanie tekstu programu (najlepiej dokonywane przez osobę nie będącą tego programu autorem) i staranne testowanie programu. Obydwie te metody są niestety zawodne. Osoba przeglądająca cudzy program dość szybko ulega zmęczeniu i nie dostrzega prostych błędów – sposób ten ma więc ograniczone zastosowanie. Podstawową metodą pozostaje więc testowanie programu, czyli wielokrotne jego wykonywanie dla najróżniejszych zestawów danych wejściowych. Proces testowania dużych programów jest realizowany zazwyczaj przez specjalną grupę pracowników (testerów). Na początku procesu testowania błędy wykrywane są (i usuwane) dość często, później czas potrzebny na wykrycie kolejnego błędu rośnie. Jeżeli przez pewien okres czasu (np. dwa tygodnie) testowanie nie doprowadziło do wykrycia żadnego błędu, to przyjmuje się (być może optymistycznie), że program jest poprawny. Do procesu testowania włączani są często przyszli użytkownicy tworzonego programu – producent udostępnia im tzw. wersję beta, licząc na współpracę w odnajdywaniu dalszych błędów. Bardzo nawet staranne testowanie nie prowadzi niestety do uzyskania pewności co do poprawności programu (nie jest dowodem poprawności). Przekonujemy się o tym na co dzień – sztandarowe produkty wybitnych firm informatycznych często wymagają poprawek już po krótkim czasie bardziej powszechnego użytkowania. Podobnie, w pozornie poprawnych programach, używanych od wielu lat, niespodziewanie pojawiają się błędy.

Uwagi te prowadzą do następujących wniosków:

- nigdy nie można mieć pewności, że dany program będzie działał poprawnie,
- programowanie jest działalnością bardzo odpowiedzialną, szczególnie gdy w grę wchodzi życie, zdrowie i pieniądze ludzi,
- programować należy z najwyższą starannością, przestrzegając zaleceń i dobrych praktyk wypracowanych przez *inżynierię oprogramowania*.

2.3. Podstawowe składniki programu

2.3.1. Deklarowanie danych i struktur danych

Dane występujące w rzeczywistości są w programie reprezentowane za pomocą zmiennych i struktur danych (punkt 2.1). Jeżeli program zapisywany jest jako schemat blokowy, to nie potrzeba uprzedzająco zgłaszać potrzeby użycia jakiejś zmiennej czy struktury danych. Pierwsze wystąpienie identyfikatora zmiennej czy struktury danych równoważne jest z wprowadzeniem jej do programu. Zasada ta stosuje się również do niektórych innych, beztrosko zdefiniowanych języków programowania.

W profesjonalnych językach programowania potrzebę użycia zmiennej czy też struktury danych należy zgłosić za pomocą odpowiedniej deklaracji. Korzystanie z niezadeklarowanych zmiennych czy struktur danych traktowane jest jako błąd.

Dla zmiennych deklaracja taka określa identyfikator i typ wartości zmiennej, co pozwala na zarezerwowanie odpowiedniej liczby bajtów pamięci komputera dla przechowywania wartości zmiennej.

```
int LiczbaKrzeseł = 15 ;
```

Deklaracja ta, zapisana w języku C, wskazuje, iż zmienna LiczbaKrzeseł będzie miała wartości całkowitoliczbowe, dodatnie lub ujemne, zapisane w pamięci komputera za pomocą 4 bajtów (parametry te wynikają z definicji użytego typu danych int). Początkowa wartość tej zmiennej będzie wynosiła 15.

W przypadku struktur danych deklaracja określa: identyfikator struktury danych, typ danych w niej występujących oraz liczbę tych danych.

```
double Przychody [ 12 ] = { 0.0 } ;
```

Tablica jednowymiarowa Przychody, której deklaracja również jest zapisana w języku C, będzie miała 12 elementów, każdy z nich to liczba zmiennopozycyjna, ośmiobajtowa. Wartości początkowe wszystkich tych elementów będą równe 0.0 .

2.3.2. Instrukcje

Instrukcje określają jakie operacje (obliczenia) i w jakiej kolejności zostaną zrealizowane podczas wykonania programu. Do ustalania kolejności wykonywania

operacji służą *instrukcje sterowania*, omówione szczegółowo w punkcie następnym. Pozostałe rodzaje instrukcji to najczęściej:

- instrukcja nadania wartości zmiennej lub elementowi struktury danych,
- instrukcja wywołania podprogramu,
- instrukcje wprowadzania i wyprowadzania danych.

Podstawową rolę pełni instrukcja nadawania nowej wartości zmiennej (*instrukcja przypisania*). Zwyczajowo identyfikator zmiennej jest zapisywany na początku (po lewej stronie) takiej instrukcji. Potem występuje operator przypisania wartości (jego postać zależy od konkretnego języka programowania), a po prawej stronie tego operatora występuje wyrażenie arytmetyczne lub logiczne, którego wartość w takcie wykonywania instrukcji zostaje obliczona i stanie się nową wartością zmiennej.

```
alfa = beta + 3 * delta – 1 ;                    // język C

delta := delta – beta + 5 ;                      // język Pascal
```

Jak widać w języku programowania C rolę operatora przypisania wartości pełni pojedynczy znak równości (=). W języku Pascal operator ten zbudowany jest z dwu znaków: dwukropka i znaku równości (:=). Wartości zmiennych znajdujących się w wyrażeniu po prawej stronie operatora przypisania nie ulegają zmianie, o ile są to inne zmienne, niż zmienna, której nadawana jest nowa wartość.

```
int licznik = 5 ;                                // język C

int składnik = 7 ;

licznik = 10 * licznik + składnik ;              // linia 3
```

W podanym przykładzie zadeklarowane zostały dwie zmienne całkowitoliczbowe: licznik o początkowej wartości 5 i składnik o wartości 7. W trzeciej linii znajduje się instrukcja przypisania, która określa nową wartość zmiennej licznik. Wartość ta jest obliczana w sposób następujący: dotychczasowa wartość zmiennej licznik (czyli 5) zostaje pomnożona przez 10 i do wyniku zostanie dodana wartość zmiennej składnik (czyli 7). Ostatecznie wartość zmiennej licznik wyniesie 57, a wartość zmiennej składnik nie ulegnie zmianie.

Instrukcje wywołania podprogramu omówione zostaną w punkcie 2.3.4, natomiast instrukcje wprowadzania i wyprowadzania danych nie występują w języku C, który jest przedmiotem wykładu w tym podręczniku. W języku C operacje wprowadzania i wyprowadzania danych realizowane są za pomocą odpowiednich funkcji (podprogramów) bibliotecznych.

2.3.3. Struktury sterowania

Wykonywanie programu rozpoczyna się od wykonania jego początkowej instrukcji. Po wykonaniu n-tej instrukcji wykonywana jest instrukcja o numerze n+1. Od tej zasady kolejnego wykonywania instrukcji istnieje tylko jedno odstępstwo, dotyczące

rozgałęzień, dzięki którym można w programie podejmować decyzje co do dalszego jego przebiegu. Wykazano, że takie ograniczenie sposobu określania kolejności wykonywania instrukcji nie ogranicza uniwersalności programów – każdy algorytm można zapisać za pomocą programu, w którym reguły te są przestrzegane.

2.3.3.1. Rozgałęzienia

W punkcie dotyczącym schematów blokowych (2.2.1) omówiono blok warunkowy, który posiada jedno wejście i dwa wyjścia. Wybór wyjścia, którym dalej postępować będzie proces wykonywania programu zależny jest od spełnienia (lub nie spełnienia) warunku zapisanego w bloku warunkowym. Warunek jest spełniony, gdy reprezentujące go wyrażenie logiczne jest prawdziwe. W językach programowania wprowadzono prawie identyczne rozwiązanie zwane *instrukcją warunkową*. Definiuje się warunek – gdy jest spełniony, to wykonywany jest określony ciąg instrukcji. Gdy natomiast warunek nie jest spełniony, to albo wykonuje się inny, przewidziany na tę okoliczność ciąg instrukcji, albo nie wykonuje się żadnej dodatkowej instrukcji i następuje przejście do instrukcji następnej.

```
int baza, kwota = 0 ;              // język C
// .....                           // fragment A
// .....
if  ( baza > 127 )                 // linia N
      kwota = 745 ;
// .....                           // fragment B
// .....
if ( baza < 23 )                   // linia M
      kwota = 35 ;
else
      kwota = 89 ;
```

W przykładzie tym występuje zmienna baza, której wartość determinuje spełnienie czy też niespełnienie kolejnych warunków. Po wykonaniu instrukcji z fragmentu A, w linii N występuje instrukcja warunkowa, rozpoczynająca się od słowa if (jeżeli). Gdy w trakcie wykonania fragmentu A zmiennej baza zostanie nadana wartość większa od 127, to warunek z linii N będzie spełniony i wykonana zostanie instrukcja przypisania, nadająca zmiennej kwota wartość 745. Jeżeli natomiast po osiągnięciu linii N zmienna baza będzie miała wartość mniejszą lub równą 127, to przypisanie nowej wartości zmiennej kwota nie zostanie wykonane i nastąpi bezpośrednie przejście do fragmentu B. Tego, czy warunek z linii N będzie spełniony nie można przewidzieć z góry, ponieważ we fragmencie A program mógł pobrać dane zewnętrzne, od których konkretnych wartości mogła być uzależniona wartość nadana w tym fragmencie zmiennej baza.

Instrukcja warunkowa z linii M zawiera poza warunkiem i słowem if dodatkowe słowo else (przeciwnie), które rozgranicza instrukcje wykonywane, gdy warunek jest spełniony od instrukcji przewidzianych na wypadek, gdyby warunek spełniony nie był. Tak więc, zależnie od aktualnej wartości zmiennej baza zmienna kwota otrzyma wartość 35 albo 89.

2.3.3.2. Pętle

Bardzo często w algorytmie rozwiązującym jakiś problem występuje konieczność wielokrotnego wykonania takiej samej operacji dla kolejnych wartości pewnych danych. Przykładem może być problem obliczenia sumy liczb 1 ... N, czy też problem obliczenia podatku od wynagrodzeń pracowników, które to wynagrodzenia są zebrane w tablicy jednowymiarowej. Pierwszy z tych przykładów należy do grupy obliczeń krokowych, w których wykonanie kolejnego obliczenia wymaga znajomości wyniku obliczenia poprzedniego. Drugi przykład to przetwarzanie kolekcji danych (zawartych w pewnej strukturze danych) – każdy element kolekcji jest przetwarzany w taki sam sposób.

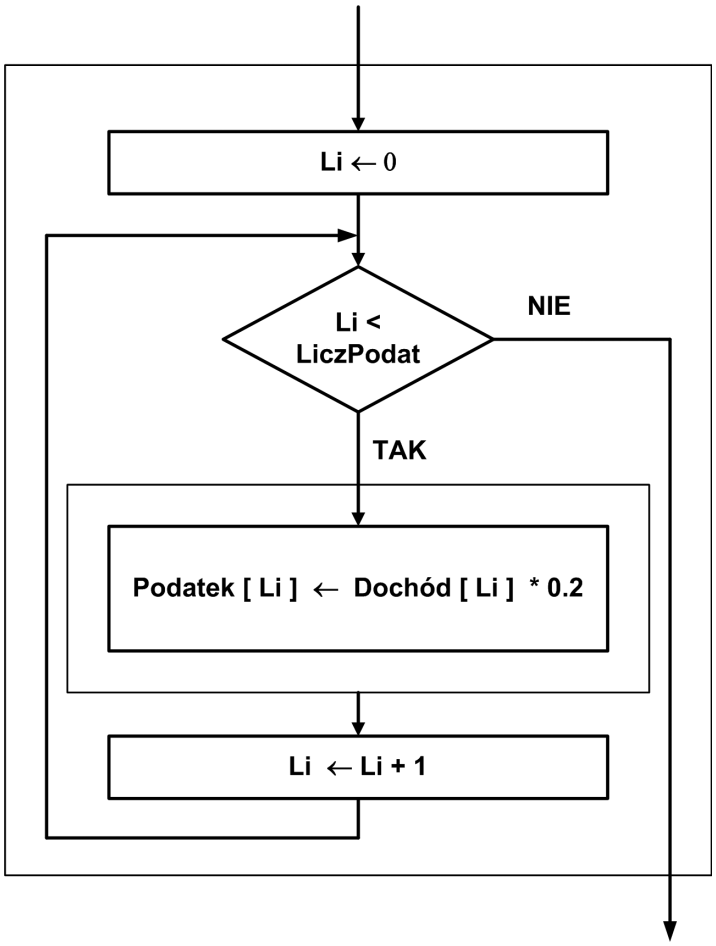
W programach, będących zapisami algorytmów, wielokrotne wykonanie takiej samej operacji jest realizowane za pomocą konstrukcji zwanej *pętlą*. W zapisie pętli istotne są:

- ustalenie warunków początkowych,
- definicja operacji powtarzanej w pętli (*treści pętli*),
- sformułowanie warunku zakończenia wykonywania pętli (*warunek wyjścia*).

Istnieją dwa podstawowe typy pętli:

- pętle wykorzystujące licznik – po każdym wykonaniu treści pętli licznik jest zmieniany, po czym sprawdza się, czy nie osiągnął wartości granicznej,
- pętle, których wykonanie kończy się po osiągnięciu przez przetwarzane dane zadanych wartości.

Na rysunku 2.8 przedstawiono schemat blokowy pętli obliczającej podatek od wynagrodzeń. Przed rozpoczęciem wykonywania tej pętli wartości wynagrodzeń wszystkich pracowników zapisane być muszą w tablicy Dochód. Ustalona też być musi wartość zmiennej LiczPodat, która określa liczbę elementów tej tablicy oraz liczbę elementów wynikowej tablicy Podatek. Dla każdego i-tego elementu tablicy Dochód obliczany jest stosowny podatek i jego wartość wpisywana jest jako i-ty element tablicy Podatek. Konstrukcja tej pętli rozpoczyna się od bloku ustalającego początkową wartość licznika Li (wynoszącą 0, ponieważ elementy tablic numerowane są od 0). Następnie, w bloku warunkowym sprawdza się, czy licznik Li ma jeszcze wartość mniejszą niż wartość zmiennej LiczPodat. Gdy ten warunek jest spełniony wykonuje się treść pętli (wyróżnioną odrębną ramką), składającą się w tym przypadku z jednego bloku zawierającego operację obliczenia podatku i wpisania jego wartości jako i-tego elementu tablicy Podatek. Kolejny element pętli to blok zwierający operację zmiany wartości licznika – w tym przypadku wartość licznika Li jest powiększana o 1. Treść pętli wykonywana jest wielokrotnie, ale cała konstrukcja pętli posiada tylko jedno wejście i jedno wyjście – pętla może więc być zawarta w pojedynczym bloku operacyjnym (co zostało na rys. 2.8 zaznaczone zewnętrznym obramowaniem z jednym wejściem i jednym wyjściem).



Rys. 2.8. Pętla wykorzystująca licznik

Ta sama pętla obliczająca podatki od wynagrodzeń może zostać zapisana w języku C w sposób następujący.

```
double Dochód [ 100 ] ;
double Podatek  [ 100 ] ;
int LiczPodat ;
int Li ;
// .....                                          // fragment A
for ( Li = 0 ; Li < LiczPodat ; Li = Li + 1)
      Podatek [ Li ] = Dochód [ Li ] * 0.2;       // 20%
// .....                                          //  fragment B
```

Po zadeklarowaniu tablic i zmiennych wykonywany jest fragment A, w którym ustalana jest wartość zmiennej LiczPodat i odpowiednie wartości wpisywane są do

tablicy Dochód. Pętla rozpoczyna się od słowa for (dla), następnie w nawiasach okrągłych zawarte są trzy składniki konstrukcji pętli (oddzielone średnikami). Pierwszy składnik to ustalenie wartości początkowej licznika. Składnik drugi to wyrażenie logiczne zwane warunkiem wyjścia – zakończenie wykonywania pętli następuje, gdy wyrażenie to będzie miało wartość *fałsz* (nieprawda, że). Ostatni składnik to instrukcja określająca sposób zmiany wartości licznika. W następnej linii znajduje się treść pętli. Wykonanie takiej pętli przebiega następująco:

a) ustalenie wartości początkowej licznika,
b) sprawdzenie warunku wyjścia, gdy wyrażenie ma wartość *fałsz*, następuje opuszczenie pętli i przejście do fragmentu B,
c) wykonanie treści pętli,
d) zmiana wartości licznika,
e) przejście do punktu b.

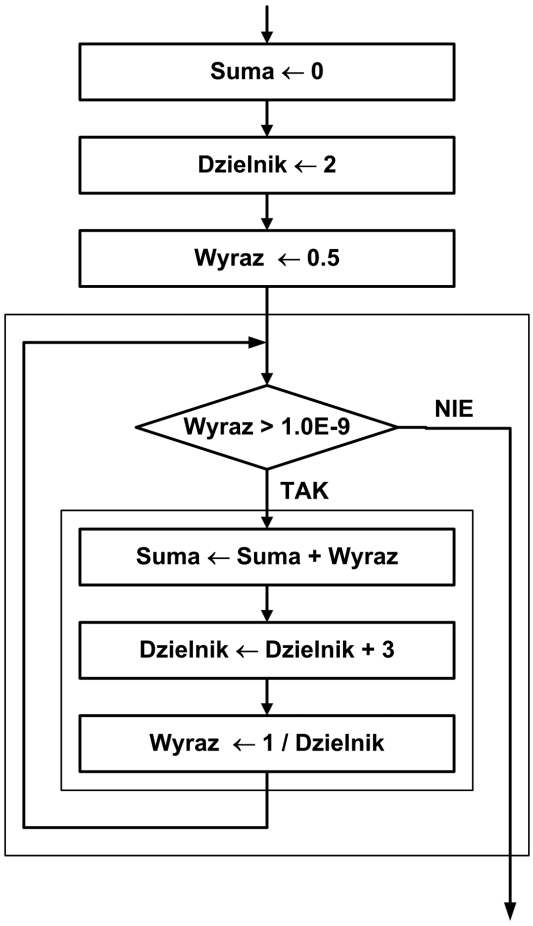
W prezentowanym przykładzie treść pętli była prosta – sprowadzała się do pojedynczej operacji. W bardziej złożonych programach treść pętli może być bardzo rozbudowana, w szczególności w treści pętli może być zawarta (*zagnieżdżona*) następna pętla. Zawsze jednak treść pętli musi dać się sprowadzić do pojedynczego bloku operacyjnego, posiadającego jedno wejście i jedno wyjście. Przestrzeganie tej reguły jest podstawowym wymogiem programowania strukturalnego.

Drugi rodzaj pętli to *pętla dopóki*, w której nie przewiduje się bezpośredniego stosowania licznika wykonań. W warunku wyjścia takiej pętli sprawdza się natomiast, czy przetwarzane zmienne (jedna lub wiele) osiągnęły odpowiednie wartości (*wartości graniczne*). Jako przykład może posłużyć program obliczający sumę wyrazów szeregu:

$$S = \frac{1}{2} + \frac{1}{5} + \frac{1}{8} + \frac{1}{11} + \cdots$$

Sumowanie ma być prowadzone dopóty, dopóki kolejny wyraz będzie większy od 10^{-9}.

Pierwsze trzy bloki przedstawione na rys. 2.9 zawierają operacje ustalające warunki początkowe wykonania pętli. Sama pętla (zawarta w dodatkowym obramowaniu) rozpoczyna się od sprawdzenia warunku wyjścia – gdy wyrażenie logiczne reprezentujące ten warunek ma wartość *fałsz*, to następuje zakończenie wykonywania pętli. W treści pętli (również ograniczonej dodatkowym obramowaniem) następuje dodanie aktualnej wartości wyrazu do sumy, zwiększenie dzielnika o 3 i obliczenie wartości kolejnego wyrazu ciągu.



Rys. 2.9. Pętla dopóki

W języku C pętla obliczająca sumę wyrazów rozważanego szeregu może zostać zapisana w sposób następujący.

```
double Suma, Dzielnik, Wyraz ;
Suma = 0 ;
Dzielnik = 2 ;
Wyraz = 0.5 ;
while ( Wyraz > 1.0E-9 )
{
      Suma = Suma + Wyraz ;
      Dzielnik = Dzielnik + 3 ;
      Wyraz = 1 / Dzielnik ;
}
// .....                          // fragment A
```

Pierwsze cztery linie programu zawierają deklarację zmiennych i przypisanie im wartości początkowych. Pętla rozpoczyna się od słowa while (dopóki), za którym w nawiasach okrągłych wpisane zostało wyrażenie logiczne reprezentujące warunek wyjścia. Zakończenie wykonywania pętli i przejście do fragmentu A programu nastąpi, gdy wartość zmiennej Wyraz stanie się mniejsza lub równa od 10^{-9} . Treść pętli składa się w tym przykładzie z trzech instrukcji, zostały one połączone w instrukcję złożoną, za pomocą nawiasów klamrowych. Kolejne instrukcje wykonują dodanie aktualnej wartości wyrazu do sumy, zwiększenie dzielnika o 3 i obliczenie następnej wartości wyrazu sumowanego ciągu.

2.3.4. Podprogramy

Modularyzacja programu, czyli jego podział na części, zwane podprogramami lub modułami, jest nieodzowna w przypadku nieco już większych programów. Istnieją dwa powody dokonywania takiego podziału. Po pierwsze, w ramach wyróżnionego podprogramu można zawrzeć operacje przetwarzające, które będą wielokrotnie stosowane w rożnych miejscach programu. Przykładem może być podprogram wyświetlający na ekranie tabelę. W każdym miejscu programu, w którym dane maja zostać zaprezentowane w formie tabeli, można będzie z takiego podprogramu skorzystać. Drugim, daleko ważniejszym powodem, jest potrzeba ograniczenia liczby instrukcji, które programista ma napisać w ramach pojedynczego modułu i których poprawność należy potem sprawdzić. Zweryfikowanie poprawności programu złożonego z tysięcy linii (a taką liczbę linii posiada wiele profesjonalnych programów) jest zadaniem prawie niemożliwym do wykonania. Znacznie łatwiej rozwiązać ten problem dokonując modularyzacji takiego wielkiego programu. Poszczególne moduły (podprogramy) mogą mieć np. po kilkadziesiąt linii – sprawdzenie poprawności takiego modułu jest znacznie łatwiejsze i niekiedy może zostać zautomatyzowane.

Po napisaniu podprogramu staje się on dostępny do wykorzystania – użycie go następuje po dokonaniu jego *wywołania.* Podprogramy często posiadają parametry (zwane również *argumentami*) – wartości tych parametrów są przekazywane w momencie wywołania podprogramu. Przykładem może być prosty podprogram arytmetyczny o nazwie Pierwiastek, który oblicza wartość pierwiastka kwadratowego z liczby, która została mu przekazana jako argument w momencie wywołania. Ogólnie podprogram ten definiowany jest jako:

```
Pierwiastek ( X )
```

Każde wywołanie zawierać natomiast powinno konkretną wartość, dla której pierwiastek ma zostać wyliczony (wartość ta staje się w danym wywołaniu wartością argumentu X).

```
Pierwiastek ( 145 )          // pierwiastek z liczby 145
Pierwiastek ( alfa )         // pierwiastek z aktualnej wartości
                             // zmiennej alfa
```

Modularyzując program wyróżnia się program główny, od którego rozpoczyna się wykonanie całego programu i podprogramy, które są z programu głównego wywoływane. Oczywiście poszczególne podprogramy mogą się również nawzajem wywoływać.

Budując schemat blokowy podprogramu korzysta się z takich samych bloków jak przy tworzeniu programu głównego, jedynie w początkowym bloku START można podać nazwę podprogramu i jego argumenty. Podobnie w bloku STOP można określić wynik działania podprogramu.

W językach programowania określone są konstrukcje składniowe umożliwiające definiowanie i wywoływanie podprogramów, zwanych w tych językach raczej *funkcjami* lub *procedurami*. Ogólna zasada określa jako funkcje takie podprogramy, które przekazują pojedynczy wynik swojego działania (będący najczęściej liczbą). Przykładem funkcji jest więc omawiany podprogram Pierwiastek. Procedury nie mają pojedynczego wyniku, ich działanie może prowadzić do zmiany wartości wielu danych lub do wykonania różnych czynności organizacyjnych (np. wyświetlenia wyników na ekranie monitora). Sposób definiowania i wywoływania funkcji w języku C przedstawiono w rozdziale 7.

2.5. Przygotowanie i wykonanie programu

Do przygotowania i wykonania programu zazwyczaj stosowane są obecnie zintegrowane systemy programowania (IDE – *Integrated Development Environment*), które wspomagają programistę w niełatwym zadaniu napisania i sprawdzenia poprawności programu. Systemy te są niekiedy przeznaczone do programowania z użyciem tylko jednego języka programowania (np. języka C czy C++) albo umożliwiają wybór dowolnego z wielu udostępnianych języków. Przykładem tego ostatniego rodzaju systemu programowania jest system Microsoft Visual Studio.NET®. Ten właśnie system posłużył do przygotowania i przetestowania przykładowych fragmentów programów zawartych w tym podręczniku.

W fazie uczenia się programowania programista otrzymuje najczęściej jakieś zadanie z zakresu przetwarzania danych i przede wszystkim musi wymyślić algorytm rozwiązujący postawiony problem, a następnie zapisać ten algorytm jako program, wykorzystując wybrany język programowania. Wymyślony algorytm powinien uwzględniać możliwości stosowanego języka programowania. Jeżeli np. w wybranym języku nie jest dostępna operacja potęgowania liczb, to trzeba układając algorytm przewidzieć, w jaki sposób to potęgowanie będzie realizowane. Języki programowania, jak to już było omawiane, udostępniają zazwyczaj skromny zestaw struktur danych i jeszcze uboższy zestaw struktur sterowania. Tworząc algorytm rozwiązujący pewien problem przetwarzania danych należy więc brać pod uwagę możliwości języków programowania i kolejne kroki algorytmu formułować tak, aby było możliwe ich zapisanie w wybranym języku.

2.5.1. Edycja tekstu

Zapisywanie algorytmu, czyli pisanie programu w wybranym języku programowania polega na ręcznym wpisywaniu poszczególnych deklaracji i instrukcji (proces ten jest niekiedy wspomagany przez automatyczne generowanie fragmentów programu). Zintegrowane systemy programowania zawierają zazwyczaj specjalizowane edytory tekstowe, przystosowane właśnie do wprowadzania i modyfikowania tekstu programu (lub jak się niekiedy mówi *kodu programu* lub *kodu źródłowego*). Edytory te, poza standardowymi funkcjami dostępnymi w normalnych edytorach tekstu, oferują szereg udogodnień ułatwiających pisanie programu. Należą do nich między innymi:

- *kolorowanie składni*: edytor rozpoznaje niektóre standardowe fragmenty programu i nadaje im odrębny kolor, np. identyfikatory wyróżnione (if, while,...) – niebieskie, teksty – brązowe, itp.
- automatyczne stosowanie wcięć kolejnych linii programu, zależnie od stopnia ich zależności od linii poprzedniej, np.

```
if ( wart > 34 )                // język C
        ilo = 5 ;               // początek linii przesunięty w prawo (wcięcie)
```

- automatyczne generowanie fragmentów programów, np. szkieletów wybranych instrukcji:

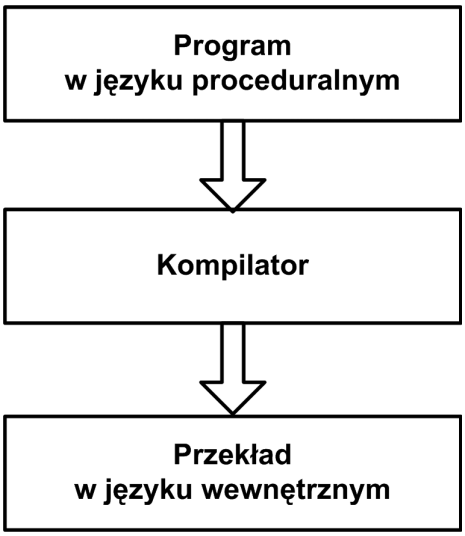
```
for (    ;    ;    )
{                }
```

- odszukiwanie nawiasu, odpowiadającego nawiasowi wskazanemu (bardzo użyteczne przy mocno zagnieżdżonych wyrażeniach czy instrukcjach).

Program źródłowy, przygotowany za pomocą edytora, jest wpisywany do pliku dyskowego o odpowiednim rozszerzeniu. Dla języka programowania C są to zazwyczaj rozszerzenia *.c* lub *.cpp*.

2.5.2. Kompilacja

Program zapisany w proceduralnym języku programowania ma zostać wykonany przez komputer. Ale, jak to przedstawiono w poprzednim rozdziale, procesor (czyli element komputera realizujący program) może wykonywać jedynie programy będące ciągami prostych poleceń, czyli programy zapisane w języku wewnętrznym. Należy więc tekst programu zapisany w języku proceduralnym przetłumaczyć na program w języku wewnętrznym, równoważny co do realizowanego algorytmu. Zadanie to wykonują specjalne programy (rys. 2.10) zwane *kompilatorami*.



Rys. 2.10. Kompilacja programu

Kompilator, opracowany dla danego proceduralnego języka programowania, traktuje tekst programu zapisany w tym języku (czyli program źródłowy) jako swoje dane wejściowe, odczytywane ze wskazanego pliku dyskowego. Kompilacja programu źródłowego składa się z następujących faz.

- Analiza leksykalna oddzielnych elementów programu źródłowego, czyli sprawdzenie, czy są one poprawnie zapisane (błędem leksykalnym jest np. zapisanie liczby całkowitej jako -34a12).
- Analiza składniowa deklaracji i instrukcji programu źródłowego, czyli sprawdzenie, czy elementy programu występują w sekwencji zgodnej z odpowiednią regułą gramatyczną – błąd składniowy występuje np. w następującym zapisie:

  ```
  for (beta < 1) else delta = 2 ;          // język C
  ```

  Poszczególne elementy tej linii są poprawne leksykalnie, ale sekwencja tych elementów jest niepoprawna.
- Analiza semantyczna, czyli sprawdzenie poprawności znaczenia poszczególnych wyrażeń programu źródłowego. Wykrywane są na tym etapie np. błędy wynikające z braku uprzedzających deklaracji identyfikatorów zmiennych czy funkcji, prób wykonywania operacji dla nieodpowiednich typów danych (np. dodawanie liczby całkowitej do tekstu) itp.
- Generacja przekładu – gdy wszystkie poprzednie analizy nie wykryją błędów w programie źródłowym, kompilator generuje przekład w języku wewnętrznym, czyli ciągi poleceń przeznaczonych do bezpośredniego wykonania przez procesor. Przekład ten jest zapisywany do pliku wykonywalnego.

Jeżeli użytkownik korzysta z systemu operacyjnego należącego do rodziny MS Windows®, to plik wykonywalny otrzyma rozszerzenie *.exe* i będzie miał strukturę wewnętrzną, odpowiednią dla tego typu plików.

Kompilator jest często składnikiem zintegrowanego systemu programowania i w związku z tym może być wykorzystywany do sprawdzania każdej wprowadzonej za pomocą edytora linii programu źródłowego. Pomoc kompilatora jest również niezbędna edytorowi do realizacji niektórych jego funkcji jak np. do odszukiwania nawiasu odpowiadającego nawiasowi zaznaczonemu.

2.5.3. Wykonanie programu

Po uzyskaniu pliku wykonywalnego, zawierającego przekład programu źródłowego, można przystąpić do jego wykonania. Jeżeli użytkownik korzysta ze zintegrowanego systemu programowania, to może uruchomić przekład za pomocą odpowiedniego polecenia tego systemu. W podręczniku niniejszym omawiane są wyłącznie programy korzystające z konsoli (punkt 1.2) – okno konsoli zostanie automatycznie otwarte przez system programowania, po wybraniu polecenia uruchomienia przekładu programu. Korzystając z tego okna można wprowadzać dane niezbędne wykonywanemu programowi i obserwować wyprowadzane wyniki.

Zintegrowane systemy programowania umożliwiają zazwyczaj wspomaganie procesu uruchamiania programu (ang. *debugging*). Poprawność leksykalna, składniowa i semantyczna programu, stwierdzona przez kompilator, nie świadczy niestety o jego pełnej poprawności, czyli nie gwarantuje, że przetłumaczony program jest poprawną realizacją wyjściowego algorytmu. Programując bardzo łatwo popełnić błąd, który w procesie kompilacji nie może zostać wykryty (kompilator nie otrzymuje żadnej informacji o algorytmie, który tłumaczony program ma realizować). Prostym przykładem takiej sytuacji jest zapisanie w programie instrukcji:

```
WartośćPartiiTowaru = LiczbaSztuk – CenaJednejSztuki ;                    // język C
```

Algorytm na pewno przewiduje dla obliczenia łącznej wartości partii towarów pomnożenie ich liczby przez cenę pojedynczej sztuki, użycie znaku odejmowania jest oczywistym błędem, niestety takiego błędu kompilator nie jest w stanie wykryć.

Kompilator, analizując program źródłowy i generując jego przekład może zapisać informacje dodatkowe, umożliwiające późniejsze wspomaganie uruchamiania tego programu. Korzystając z tych informacji system programowania pozwala na wykonywanie między innymi następujących operacji:

- zatrzymanie wykonywania programu we wskazanym jego miejscu (np. po osiągnięciu wskazanej linii),
- zatrzymanie wykonywania programu po wykonaniu zadanej liczby powtórzeń wskazanej pętli,
- zatrzymanie wykonywania programu, gdy spełniony zostanie zadany warunek (np. `Debet > 10000`),

- wznowienie wykonywania programu do końca lub do osiągnięcia następnego miejsca zatrzymania,
- wyświetlenie aktualnych wartości zmiennych i elementów struktur danych (wartości obliczonych do chwili zatrzymania wykonywania programu),
- zmiana wartości zmiennych i elementów struktur danych w miejscu zatrzymania – po wznowieniu wykonywania program będzie korzystał z nowo wprowadzonych wartości.

Wszystkie te operacje ułatwiają usunięcie błędów logicznych popełnionych podczas programowania. Gdy programista stwierdzi, że istnieje duże prawdopodobieństwo, iż jego program jest całkowicie poprawny, to można dokonać kompilacji tego programu raz jeszcze, uzyskując przekład docelowy, przeznaczony do normalnego użytkowania. Taka postać przekładu nie zawiera żadnych dodatkowych informacji wspomagających uruchamianie, dzięki czemu zajmuje mniejszy obszar pamięci i wykonuje się szybciej.

Przekład programu źródłowego może też być oczywiście wykonywany bez nadzoru systemu programowania. Należy w tym celu otworzyć okno konsoli (np. wybrać opcję systemu operacyjnego *Wiersz polecenia*), przejść do katalogu, w którym znajduje się plik z przekładem (plik o rozszerzeniu *.exe*) i wywołać program, wpisując jako polecenie nazwę tego pliku.

2.5.4. Interpretacja programu

W poprzednich dwu punktach przedstawiono sposób wykonania programu zapisanego w języku proceduralnym, polegający na jego przetłumaczeniu na postać wykonywalną (kompilacja) i wykonaniu tego przekładu. Nie jest to jedyny dostępny sposób wykonywania takich programów. Druga możliwość to posłużenie się *programem interpretującym* (*interpreterem*). Interpreter nie generuje przekładu, ale traktując program źródłowy jako swoje dane wejściowe, analizuje go i krok po kroku wykonuje zadane w nim operacje przetwarzania danych. Na przykład napotykając w programie źródłowym instrukcje:

```
Inwestycja = 1500 ;

Zysk = Inwestycja * 1.5 ;
```

interpreter zapisze w swoich strukturach wewnętrznych informację, że wartość zmiennej Inwestycja wynosi aktualnie 1500, a wartość zmiennej Zysk to 2250.

Korzystanie z programów interpretujących ma liczne wady, np. :

- analiza poprawności programu źródłowego jest mniej dokładna, niż w przypadku kompilacji – kompilator analizuje cały tekst programu źródłowego i dostarcza informacji o wszystkich znalezionych błędach, interpreter rozpoczyna wykonywanie programu źródłowego i grzęźnie z powodu prostych niekiedy błędów lub zbacza na trudne do przeanalizowania manowce,

- wykonanie programu źródłowego w trybie interpretacji trwa zdecydowanie dłużej, niż wykonanie przekładu tego programu,
- wielokrotne wykonanie programu źródłowego wymaga wielokrotnego uruchamiania interpretatora.

Jedyną zaletą interpreterów wydaje się być zaoszczędzenie czasu potrzebnego na kompilację programu źródłowego. Bilans wad i zalet interpretacyjnego wykonywania programów zapisanych w językach proceduralnych wskazuje na zdecydowana przewagę kompilacji i wykonywania przekładu. Toteż rzadko się już spotyka programy interpretujące przeznaczone dla profesjonalnych języków proceduralnych (np. dla języków C# czy Java). Interpretery pozostały jako narzędzie uruchamiania programów zapisanych w tzw. *językach skryptowych*. Języki te udostępniają zestaw standardowych poleceń (mogących posiadać parametry). Program zapisywany jest jako ciąg takich poleceń. Języki skryptowe stosowane są niekiedy do tworzenia aplikacji internetowych – przykładem może tu być pleniący się powszechnie język PHP.

2.6. Język programowania C

Język programowania C to *imperatywny*, proceduralny język programowania, który został zdefiniowany na początku lat 70-tych XX wieku. Jego autorem jest Denis Ritchie – pracujący w tym czasie w firmie Bell Labs. Język C bardzo szybko zdobył dużą popularność, między innymi dzięki temu, że był podstawowym językiem programowania dla nowego wówczas systemu operacyjnego Unix. Moduły programowe tego systemu operacyjnego zapisywane były w języku C (poza najbardziej wewnętrznymi modułami, które musiały być zapisane z użyciem języka wewnętrznego). Ponadto składnikiem systemu Unix był wbudowany kompilator języka C, umożliwiający tworzenie oprogramowania użytkowego. W krótkim czasie opracowano wiele kompilatorów języka C dla różnych typów komputerów. Autorzy tych kompilatorów interpretowali niekiedy definicję języka C na swój sposób i czasami wprowadzali do języka własne rozszerzenia. Skutkiem takich praktyk były coraz większe trudności z przenoszeniem programów zapisanych w języku C pomiędzy różnymi kompilatorami. Dla ujednolicenia definicji języka C i wprowadzenia jednej, powszechnie stosowanej wersji opublikowano w roku 1989 standard języka C, tzw. ANSI C. Po 10-ciu latach, w roku 1999, powstał następny standard tego języka, określany jako C99.

Język C stał się również punktem wyjścia do definiowania następnych generacji języków programowania. W latach 80-tych XX wieku opracowano definicję języka programowania C++, który jest rozszerzeniem języka C o mechanizmy programowania obiektowego. Dokonując tego rozszerzenia wprowadzono również dodatkowe konstrukcje do części nieobiektowej języka C++. W końcowej dekadzie tamtego wieku język C++ był najpowszechniej stosowanym na świecie językiem programowania. Kolejnym krokiem było wprowadzenie w roku 2002 obiektowego

języka programowania C#, który również wywodzi się z języków C i C++, ale jest językiem nieco wyższego poziomu (nie udostępnia mechanizmów, które być może przyspieszają wykonanie programu lecz umożliwiają równocześnie wprowadzanie do programu wielu trudnych do wykrycia błędów). Jak widać te trzy języki programowania powiązane są literą C, występującą w ich nazwie. Nie jest to tylko ukłon w stronę szacownych przodków – wiele definicji składniowych wprowadzonych w języku C bez zmian przeniesiono do języków C++ i C#. Ponadto język C stał się podstawą wielu specjalizowanych języków programowania (np. języków do zapisu algorytmów sterowania, języków wspomagających dla pakietów matematycznych itp.).

Twórca kompilatora danego języka programowania decyduje ostatecznie o tym, jakie elementy leksykalne i jakie konstrukcje składniowe mogą występować w programach źródłowych. Tworząc kompilator podejmuje się również decyzję o sposobie generowania przekładu instrukcji języka programowania, występujących w programie źródłowym, na sekwencje poleceń języka wewnętrznego. Czyli ostateczną definicję leksykalną, składniową i semantyczną języka programowania ustala autor kompilatora, przeznaczonego dla danego języka programowania. Przestrzeganie wspomnianych standardów języka C powinno doprowadzić do sytuacji, w której program uznany za poprawny przez jeden z dostępnych kompilatorów będzie również poprawny po przeanalizowaniu przez dowolny, inny kompilator tego języka. Rzeczywistość nie jest jednak tak idealna – pomimo istnienia obowiązujących standardów twórcy kompilatorów języka C często ulegają pokusie pomijania niektórych fragmentów standardu i dodawania własnych rozwiązań niestandardowych. Szczególnie często nie jest w pełni realizowany standard C99 – przyczyną jest być może fakt spadku zainteresowania językiem C, wypartym z powszechnego stosowania przez języki obiektowe.

Systemy programowania w języku C umożliwiają niekiedy wybór wersji języka C, która będzie akceptowana przez kompilator. Lista możliwości to na przykład: ANSI C, C99, EXT C (ten ostatni, to autorski pomysł twórców jednego z systemów programowania). Często można również, zadając odpowiednie opcje, definiować sposób interpretacji niektórych elementów języka. Możliwości te są szczególnie rozbudowane w kompilatorach z rodziny GNU C.

Powstaje więc pytanie, jaką wersję języka C przedstawić w dalszej części tego podręcznika. Język ANSI C nie jest już raczej stosowany, C99 nie zyskał szerszej popularności. Racjonalnym rozwiązaniem wydaje się w tej sytuacji ucieczka do przodu, polegająca na omówieniu nieobiektowej części języka C++. Ta nieobiektowa część języka C++ powstała przez dołączenie szeregu nowych elementów do języka ANSI C. Te dodatkowe możliwości bardzo się już upowszechniły i są szeroko stosowane w praktyce programistycznej. Taki wybór przedmiotu dalszego wykładu ułatwia również przyszłą naukę programowania w języku C++ czy C#.

Pozostałe rozdziały tego podręcznika są poświęcone omówieniu języka programowania C, rozumianego jako nieobiektowa część języka C++.