helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Profesjonalne programowanie. Część 1. Zrozumieć komputer



Autor: Randall Hyde
Tłumaczenie: Tomasz Żmijewski
ISBN: 83-7361-859-7
Tytuł oryginału: Write Great Code. Volume 1: Understanding the Machine
Format: B5, stron: 392

Chcesz zostać programistą doskonałym?
Zacznij od poznania szczegółów działania komputera

- Zapis wartości liczbowych oraz arytmetyka zmiennoprzecinkowa i binarna
- Organizacja dostępu do pamięci komputera
- Proces wykonywania programu oraz operacje wejścia i wyjścia

Kod napisany przez profesjonalnego programistę jest wydajny i efektywny. Aby tworzyć wydajny kod, należy poznać architekturę komputera i sposób, w jaki program jest wykonywany. Zrozumienie tego, w jaki sposób komputer realizuje kolejne instrukcje programu i jak słowa kluczowe języków wysokiego poziomu są przenoszone na rozkazy procesora, jest kluczem do napisania kodu, który po skompilowaniu da szybko i bezbłędnie działający program.

„Profesjonalne programowanie. Część 1. Zrozumieć komputer" to pierwszy tom serii książek przeznaczonych dla tych programistów, którzy chcą podnieść swoje kwalifikacje. Przedstawia wewnętrzną architekturę komputera od strony, której znajomość jest niezbędna programiście. Opisuje sposoby zapisu wartości liczbowych i tekstów, działania na liczbach binarnych i zmiennoprzecinkowych oraz logikę Boole'a. Czytając tę książkę, dowiesz się, w jaki sposób procesor przetwarza rozkazy asemblera, jak odbywa się dostęp do danych zapisanych w pamięci oraz jak przesyłane są dane do i z urządzeń zewnętrznych.

- Zapis liczb w systemie binarnym, ósemkowym i szesnastkowym
- Działania na liczbach binarnych i zmiennoprzecinkowych
- Sposoby reprezentacji danych znakowych
- Organizacja pamięci i tryby adresowania
- Złożone typy danych
- Projektowanie układów cyfrowych i logika Boole'a
- Architektura procesora i rozkazy asemblera
- Operacje wejścia i wyjścia

Jeśli chcesz, aby napisane przez Ciebie oprogramowanie budziło podziw, koniecznie przeczytaj tę książkę.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Podziękowania **9**

Rozdział 1. Co trzeba wiedzieć o programowaniu doskonałym **11**
1.1. Seria książek „Profesjonalne programowanie” 11
1.2. O czym jest ta książka 12
1.3. Wymagania wobec Czytelnika 15
1.4. Cechy kodu doskonałego 15
1.5. Wymagane środowisko 16
1.6. Dalsze informacje 17

Rozdział 2. Zapis liczb **19**
2.1. Czym jest liczba? 19
2.2. Systemy liczbowe 20
2.2.1. Dziesiętny pozycyjny system liczbowy 21
2.2.2. Podstawa 22
2.2.3. Binarny system liczbowy 23
2.2.4. Szesnastkowy system liczbowy 24
2.2.5. Liczby ósemkowe 26
2.3. Konwersja między liczbą a tekstem 28
2.4. Zapis liczb całkowitych 29
2.4.1. Bity 29
2.4.2. Łańcuchy bitowe 30
2.5. Liczby ze znakiem i bez znaku 32
2.6. Pewne przydatne cechy liczb binarnych 34
2.7. Rozszerzenie znakiem, uzupełnienie zerami i zawężenie 35
2.8. Nasycenie 38
2.9. Zapis dziesiętny kodowany binarnie (BCD) 39
2.10. Zapis stałopozycyjny 40
2.11. Skalowane formaty liczbowe 43
2.12. Zapis wymierny 45
2.13. Więcej informacji 45

Rozdział 3. Arytmetyka binarna i działania na bitach **47**
3.1. Działania arytmetyczne na liczbach dwójkowych i szesnastkowych 47
3.1.1. Dodawanie wartości binarnych 48
3.1.2. Odejmowanie liczb binarnych 49
3.1.3. Mnożenie wartości binarnych 50
3.1.4. Dzielenie wartości binarnych 51
3.2. Operacje logiczne na bitach 52
3.3. Operacje logiczne na liczbach binarnych i ciągach bitów 54

3.4. Przydatne operacje bitowe ... 55
3.4.1. Sprawdzanie poszczególnych bitów łańcucha za pomocą operatora AND ... 55
3.4.2. Sprawdzanie, czy grupa bitów zawiera same zera ... 56
3.4.3. Porównywanie zestawu bitów z łańcuchem binarnym ... 56
3.4.4. Tworzenie liczników modulo n za pomocą operacji AND ... 57
3.5. Przesunięcia i rotacje ... 58
3.6. Pola bitowe i pakowanie danych ... 61
3.7. Pakowanie i rozpakowywanie danych ... 64
3.8. Więcej informacji ... 68

Rozdział 4. Zapis zmiennopozycyjny ... 69
4.1. Wprowadzenie do arytmetyki zmiennopozycyjnej ... 69
4.2. Formaty zmiennopozycyjne IEEE ... 74
4.2.1. Zmiennopozycyjny format pojedynczej precyzji ... 75
4.2.2. Format zmiennopozycyjny o podwójnej precyzji ... 77
4.2.3. Format zmiennopozycyjny zwiększonej precyzji ... 77
4.3. Normalizacja i wartości nienormalizowane ... 78
4.4. Zaokrąglanie ... 79
4.5. Specjalne wartości zmiennopozycyjne ... 80
4.6. Wyjątki obliczeń zmiennopozycyjnych ... 82
4.7. Działania zmiennopozycyjne ... 82
4.7.1. Zapis liczb zmiennopozycyjnych ... 83
4.7.2. Dodawanie i odejmowanie zmiennopozycyjne ... 83
4.7.3. Mnożenie i dzielenie zmiennopozycyjne ... 93
4.8. Więcej informacji ... 99

Rozdział 5. Dane znakowe ... 101
5.1. Dane znakowe ... 101
5.1.1. Zestaw znaków ASCII ... 101
5.1.2. Zestaw znaków EBCDIC ... 104
5.1.3. Dwubajtowe zestawy znaków ... 105
5.1.4. Zestaw znaków Unicode ... 106
5.2. Łańcuchy znakowe ... 108
5.2.1. Formaty łańcuchów znakowych ... 108
5.2.2. Rodzaje łańcuchów — statyczne, pseudodynamiczne i dynamiczne ... 112
5.2.3. Zliczanie odwołań do łańcucha ... 114
5.2.4. Łańcuchy w Delphi i Kyliksie ... 114
5.2.5. Tworzenie własnych formatów łańcuchów ... 115
5.3. Zbiory znaków ... 115
5.3.1. Zbiory znaków w formie zbioru przynależności ... 115
5.3.2. Listowa reprezentacja zbiorów znaków ... 117
5.4. Definiowanie własnego zestawu znaków ... 117
5.4.1. Tworzenie wydajnego zestawu znaków ... 118
5.4.2. Grupowanie znaków odpowiadających cyfrom ... 120
5.4.3. Grupowanie liter ... 120
5.4.4. Porównywanie liter ... 122
5.4.5. Inne grupowania znaków ... 123
5.5. Dodatkowe informacje ... 124

Rozdział 6. Organizacja pamięci i dostęp do niej ... 127
6.1. Podstawowe elementy systemu ... 127
6.1.1. Magistrala systemowa ... 128
6.2. Fizyczna organizacja pamięci ... 130
6.2.1. 8-bitowe magistrale danych ... 132
6.2.2. 16-bitowe magistrale danych ... 133
6.2.3. 32-bitowe magistrale danych ... 134

6.2.4. Magistrale 64-bitowe ... 136
6.2.5. Dostęp do pamięci w procesorach innych niż 80x86 ... 136
6.3. Kolejność bajtów ... 136
6.4. Zegar systemowy ... 141
6.4.1. Dostęp do pamięci a zegar systemowy ... 142
6.4.2. Stany oczekiwania ... 143
6.4.3. Pamięć podręczna ... 145
6.5. Dostęp procesora do pamięci ... 148
6.5.1. Bezpośredni tryb adresowania pamięci ... 149
6.5.2. Tryb adresowania pośredniego ... 149
6.5.3. Tryb adresowania indeksowanego ... 150
6.5.4. Skalowane indeksowane tryby adresowania ... 151
6.6. Dodatkowe informacje ... 151

Rozdział 7. Złożone typy danych i obiekty w pamięci ... 153
7.1. Typy wskaźnikowe ... 153
7.1.1. Implementacja wskaźników ... 154
7.1.2. Wskaźniki i dynamiczna alokacja pamięci ... 155
7.1.3. Działania na wskaźnikach — arytmetyka wskaźników ... 155
7.2. Tablice ... 159
7.2.1. Deklaracje tablic ... 160
7.2.2. Reprezentacja tablic w pamięci ... 162
7.2.3. Dostęp do elementów tablicy ... 163
7.2.4. Tablice wielowymiarowe ... 164
7.3. Rekordy (struktury) ... 170
7.3.1. Rekordy w Pascalu i Delphi ... 170
7.3.2. Rekordy w C i C++ ... 171
7.3.3. Rekordy w HLA ... 171
7.3.4. Zapis rekordów w pamięci ... 171
7.4. Unie ... 174
7.4.1. Unie w C i C++ ... 174
7.4.2. Unie w Pascalu, Delphi i Kyliksie ... 174
7.4.3. Unie w asemblerze HLA ... 175
7.4.4. Unie w pamięci ... 176
7.4.5. Inne zastosowania unii ... 176
7.5. Dodatkowe informacje ... 177

Rozdział 8. Logika boolowska i projektowanie cyfrowe ... 179
8.1. Algebra Boole'a ... 179
8.1.1. Operatory boolowskie ... 179
8.1.2. Prawa algebry boolowskiej ... 180
8.1.3. Priorytety operatorów boolowskich ... 181
8.2. Funkcje logiczne i tabele prawdy ... 182
8.3. Numery funkcji ... 183
8.4. Algebraiczne przekształcanie wyrażeń logicznych ... 185
8.5. Formy kanoniczne ... 186
8.5.1. Forma kanoniczna sumy termów minimalnych a tabele prawdy ... 187
8.5.2. Algebraiczne wyprowadzanie formy kanonicznej sumy termów minimalnych ... 189
8.5.3. Forma kanoniczna jako iloczyn termów maksymalnych ... 189
8.6. Upraszczanie funkcji logicznych ... 190
8.7. Ale co to wszystko ma wspólnego z komputerami? ... 197
8.7.1. Równoważność układów elektronicznych i funkcji logicznych ... 198
8.7.2. Obwody złożone ... 199
8.7.3. Sterowanie sekwencyjne i zegarowe ... 204
8.8. Dodatkowe informacje ... 208

Rozdział 9. Architektura procesora 209
9.1. Podstawy budowy procesora 209
9.2. Dekodowanie i wykonywanie instrukcji — logika przypadku a mikrokod 211
9.3. Wykonywanie instrukcji krok po kroku 213
9.3.1. Instrukcja mov 213
9.3.2. Instrukcja add 215
9.3.3. Instrukcja jnz 217
9.3.4. Instrukcja loop 218
9.4. Równoległość — klucz do przyspieszenia 218
9.4.1. Kolejka wstępnego pobrania 222
9.4.2. Okoliczności ograniczające wydajność kolejki wstępnego pobrania 225
9.4.3. Potoki — jednoczesne wykonywanie wielu instrukcji 226
9.4.4. Bufory instrukcji — wiele dróg do pamięci 230
9.4.5. Zagrożenia związane z potokami 231
9.4.6. Działanie superskalarne — równoległe wykonywanie instrukcji 233
9.4.7. Wykonywanie kodu bez zachowania kolejności 235
9.4.8. Zmiana nazw rejestrów 235
9.4.9. Architektura z bardzo długim słowem instrukcji (VLIW) 236
9.4.10. Przetwarzanie równoległe 237
9.4.11. Wieloprocesorowość 238
9.5. Dodatkowe informacje 239

Rozdział 10. Konstrukcja zbioru instrukcji 241
10.1. Dlaczego projekt zbioru instrukcji jest ważny 241
10.2. Podstawowe cele projektowe zestawu instrukcji 243
10.2.1. Dobór długości kodu instrukcji 245
10.2.2. Plany na przyszłość 247
10.2.3. Dobór instrukcji 247
10.2.4. Przypisywanie instrukcjom kodów 248
10.3. Hipotetyczny procesor Y86 248
10.3.1. Ograniczenia Y86 249
10.3.2. Instrukcje Y86 249
10.3.3. Tryby adresowania Y86 251
10.3.4. Kodowanie instrukcji Y86 252
10.3.5. Przykłady kodowania instrukcji Y86 254
10.3.6. Rozszerzanie zbioru instrukcji Y86 257
10.4. Kodowanie instrukcji 80x86 259
10.4.1. Kodowanie operandów instrukcji 260
10.4.2. Kodowanie instrukcji add — kilka przykładów 266
10.4.3. Stałe jako operandy 268
10.4.4. Kodowanie operandów 8-, 16- i 32-bitowych 269
10.4.5. Alternatywne kodowanie instrukcji 270
10.5. Znaczenie projektu zbioru instrukcji dla programisty 270
10.6. Więcej informacji 271

Rozdział 11. Architektura pamięci i jej organizacja 273
11.1. Hierarchia pamięci 273
11.2. Jak działa hierarchia pamięci 276
11.3. Względna wydajność podsystemów pamięci 277
11.4. Budowa pamięci podręcznej 279
11.4.1. Pamięć odwzorowywana bezpośrednio 280
11.4.2. Pamięć w pełni powiązana 281
11.4.3. Pamięć powiązana n-krotnie 281
11.4.4. Dobieranie schematu pamięci podręcznej do rodzaju dostępu do danych 282

11.4.5. Polityka wymiany zawartości wierszy ... 282
11.4.6. Zapis danych w pamięci ... 283
11.4.7. Użycie pamięci podręcznej i oprogramowanie ... 284
11.5. Pamięć wirtualna, ochrona i stronicowanie ... 285
11.6. Zaśmiecanie ... 288
11.7. NUMA i urządzenia peryferyjne ... 289
11.8. Oprogramowanie świadome istnienia hierarchii pamięci ... 290
11.9. Organizacja pamięci w trakcie działania programu ... 291
11.9.1. Obiekty statyczne i dynamiczne, wiązanie, czas życia ... 293
11.9.2. Kod, segmenty tylko do odczytu i segment stałych ... 294
11.9.3. Segment zmiennych statycznych ... 294
11.9.4. Segment danych niezainicjalizowanych (BSS) ... 295
11.9.5. Segment stosu ... 295
11.9.6. Segment sterty i dynamiczna alokacja pamięci ... 296
11.10. Dodatkowe informacje ... 301

Rozdział 12. Wejście i wyjście (I/O) ... 303
12.1. Połączenie procesora ze światem zewnętrznym ... 303
12.2. Inne sposoby łączenia portu z systemem ... 306
12.3. Mechanizmy wejścia-wyjścia ... 307
12.3.1. Odwzorowywanie w pamięci ... 307
12.3.2. Wejście-wyjście i buforowanie ... 308
12.3.3. Odwzorowywanie na I/O ... 308
12.3.4. Bezpośredni dostęp do pamięci (DMA) ... 309
12.4. Hierarchia szybkości wejścia-wyjścia ... 310
12.5. Szyny systemowe i szybkość transferu danych ... 311
12.5.1. Wydajność magistrali PCI ... 312
12.5.2. Wydajność magistrali ISA ... 313
12.5.3. Magistrala AGP ... 313
12.6. Buforowanie ... 314
12.7. Handshaking ... 315
12.8. Przekroczenia czasu w porcie I/O ... 316
12.9. Przerwania i próbkowanie I/O ... 317
12.10. Działanie w trybie chronionym i sterowniki urządzeń ... 318
12.10.1. Sterowniki urządzeń ... 318
12.10.2. Komunikacja ze sterownikami urządzeń i „plikami” ... 319
12.11. Omówienie poszczególnych urządzeń peryferyjnych ... 320
12.12. Klawiatura ... 320
12.13. Standardowy port równoległy ... 322
12.14. Porty szeregowe ... 323
12.15. Stacje dysków ... 324
12.15.1. Dyskietki ... 324
12.15.2. Dyski twarde ... 324
12.15.3. Systemy RAID ... 329
12.15.4. Napędy ZIP i inne miękkie napędy optyczne ... 330
12.15.5. Napędy optyczne ... 330
12.15.6. Napędy CD-ROM, CD-R, CD-RW, DVD, DVD-R, DVD-RAM i DVD-RW ... 331
12.16. Napędy taśmowe ... 333
12.17. Pamięć flash ... 334
12.18. Dyski RAM i dyski półprzewodnikowe ... 336
12.19. Urządzenia i sterowniki SCSI ... 337
12.20. Interfejs IDE/ATA ... 342

12.21. Systemy plików na urządzeniach pamięci masowej ... 344
12.21.1. Użycie mapy bitowej wolnej przestrzeni ... 346
12.21.2. Tablice alokacji plików ... 347
12.21.3. Pliki w formie list bloków ... 350
12.22. Oprogramowanie przetwarzające dane na urządzeniach pamięci masowej ... 353
12.22.1. Wydajność dostępu do plików ... 354
12.22.2. Wejście-wyjście synchroniczne i asynchroniczne ... 355
12.22.3. Znaczenie typu operacji wejścia-wyjścia ... 356
12.22.4. Pliki odwzorowywane w pamięci ... 356
12.23. Uniwersalna magistrala szeregowa (USB) ... 357
12.23.1. Projekt USB ... 358
12.23.2. Wydajność USB ... 359
12.23.3. Rodzaje transmisji USB ... 360
12.23.4. Sterowniki urządzeń USB ... 362
12.24. Myszy, trackpady i inne urządzenia wskazujące ... 363
12.25. Joysticki i urządzenia do gier ... 364
12.26. Karty dźwiękowe ... 365
12.26.1. Jak dźwiękowe urządzenia peryferyjne wytwarzają dźwięk? ... 366
12.26.2. Formaty audio i pliki MIDI ... 368
12.26.3. Programowanie urządzeń audio ... 369
12.27. Dalsze informacje ... 369

Myśl lokalnie, pisz globalnie ... 371

Dodatek A Zestaw znaków ASCII ... 373

Skorowidz ... 377

Rozdział 5.

Dane znakowe

Wprawdzie komputery są kojarzone głównie z możliwościami obliczeniowymi, ale tak naprawdę znacznie częściej używa się ich do przetwarzania danych znakowych. Jeśli weźmiemy pod uwagę, jak istotne we współczesnych komputerach jest przetwarzanie znaków, pojmiemy, że bycie programistą doskonałym wymaga gruntownego zrozumienia danych i łańcuchów znakowych.

Pojęcie *znaku* odnosi się do symbolu zrozumiałego dla człowieka lub dla maszyny, zwykle niebędącego liczbą. W zasadzie znak to symbol, który można wpisać z klawiatury i zobaczyć na monitorze. Pamiętajmy, że poza literami do danych znakowych zaliczają się też znaki przestankowe, cyfry, spacje, tabulatory, znaki powrotu karetki (klawisz *ENTER*), inne znaki kontrolne i znaki specjalne.

W tym rozdziale przyjrzymy się, jak w komputerze są zapisywane znaki, łańcuchy i zbiory znaków. Omówimy też różne operacje na tych typach danych.

5.1. Dane znakowe

Większość systemów do kodowania różnych znaków wykorzystuje pojedyncze bajty lub pary bajtów. Dotyczy to także systemów Windows i Linux wykorzystujących zestawy znaków ASCII lub Unicode, gdzie znaki zapisuje się w pojedynczym bajcie lub w dwubajtowym ciągu. Zestaw znaków EBCDIC używany w maszynach IBM *mainframe* i w minikomputerach to kolejny przykład jednobajtowego kodu znaków.

Omówimy tutaj wszystkie trzy zestawy znaków i ich zapis wewnętrzny. Powiemy też, jak stworzyć własny, dostosowany do specyficznych potrzeb zestaw znaków.

5.1.1. Zestaw znaków ASCII

Zestaw znaków ASCII (skrót od ang. *American Standard Code for Information Interchange* — Amerykański Kod Standardowy Wymiany Informacji) odwzorowuje 128 znaków na liczby całkowite bez znaku od 0 do 127 ($0..$7F). Wprawdzie dokładny

sposób tego odwzorowania jest w gruncie rzeczy dość przypadkowy i niezbyt istotny, ale istnienie takiego standardu pozwala na komunikowanie się rożnych programów i urządzeń peryferyjnych. Standardowe kody ASCII są przydatne, ponieważ niemal wszyscy ich używają. Wobec tego, jeśli użyjemy kodu ASCII 65 do zapisania znaku *A*, to dowolne urządzenie peryferyjne — na przykład drukarka — prawidłowo zinterpretuje tę wartość jako literę *A*.

Zestaw znaków ASCII zapewnia tylko 128 różnych znaków, więc pojawia się ważne pytanie — co z pozostałymi 128 wartościami ($80..$FF), które można zapisać na jednym bajcie? Odpowiedź brzmi: pomijamy te znaki. I tak właśnie będziemy postępować w niniejszej książce. Inna możliwość to rozszerzenie zestawu ASCII o tych 128 znaków. O ile jednak wszyscy nie będą zgodni co do dokładnego sposobu tego rozszerzenia[1], podważony zostanie sens używania standardowego zestawu znaków. A doprowadzenie do zgody wszystkich to niełatwe zadanie[2].

Mimo poważnych niedostatków, zestaw znaków ASCII jest standardem wymiany danych między programami i komputerami. Większość programów potrafi odczytać dane ASCII, a także je zapisywać. Jako że większość Czytelników zapewne w swoich programach używa znaków ASCII, dobrze byłoby przeanalizować ich układ i zapamiętać kluczowe znaki, takie jak *0*, *A*, *a* i im podobne. W tabeli A.1 dodatku A wymieniono wszystkie znaki z tego zestawu.

Znaki ASCII dzieli się na cztery grupy zawierające po 32 znaki. Pierwsze 32 znaki o kodach od $0 do $1F (od 0 do 31) to specjalny zbiór znaków niedrukowalnych nazywanych *znakami kontrolnymi*. Nazwa ta wzięła się stąd, że realizują one różne funkcje sterujące drukarką i monitorem, a nie są pokazywane jako takie. Przykładami znaków kontrolnych mogą być: *powrót karetki*, powodujący umieszczenie kursora na początku bieżącego wiersza[3]; *nowy wiersz*, przenoszący kursor wiersz niżej; *backspace*, który przesuwa kursor o jeden znak w lewo. Niestety, niektóre znaki kontrolne powodują różne działania różnych urządzeń wyjściowych. Standaryzacja w tym zakresie jest bardzo niewielka. Aby mieć pewność, że wiemy, co dany znak kontrolny wykonuje w używanym urządzeniu, trzeba sprawdzić to w dokumentacji.

Druga grupa 32 znaków ASCII to różne symbole przestankowe, znaki specjalne i cyfry. Najważniejsze znaki z tej grupy to spacja (kod ASCII $20) oraz cyfry (kody $30..$39).

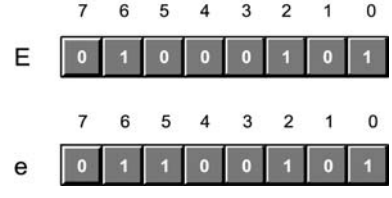
---

[1] Zanim spopularyzował się system Windows, produkty IBM obsługiwały na wyświetlaczach tekstowych 256-elementowy zestaw znaków. Chociaż zestaw ten funkcjonuje obecnie nawet we współczesnych komputerach PC, niewiele aplikacji czy urządzeń peryferyjnych nadal obsługuje te rozszerzone znaki.

[2] Powody do niegodzenia się na stosowanie wspomnianego przez Autora zestawu znaków mamy chociażby my, Polacy. Otóż zestaw ten, znany jako ISO-Latin1 lub ISO-8859-1, nie zawiera polskich liter, takich jak *ą* czy *ć*, i jest dostosowany do języków zachodnioeuropejskich. Istnieją alternatywne zestawy znaków dodatkowych, zawierające inne znaki; wszystkie one mają takie same 128 pierwszych znaków — ASCII. Przykładowo, zestaw ISO-Latin2 (inaczej ISO-8859-2) zawiera m.in. nasze znaki narodowe — *przyp. tłum.*

[3] Nazwa tego znaku pochodzi z czasów, kiedy chodziło o przesunięcie karetki maszyny do pisania. Powrót karetki polegał na fizycznym przesunięciu tej karetki tak, aby następny znak pojawił się przy lewym brzegu papieru.

Trzecia grupa 32 znaków zawiera wielkie litery. Kody liter od *A* do *Z* mieszczą się w zakresie $41..$5A. W alfabecie łacińskim jest tylko 26 liter, więc pozostałych 6 znaków zawiera różne symbole specjalne. Ostatnia grupa 32 znaków zawiera małe litery, pięć dodatkowych znaków specjalnych i jeszcze jeden znak kontrolny, *delete* (usuwający znak spod kursora). Zwróćmy uwagę na to, że małe litery wykorzystują kody ASCII $61..$7A. Jeśli zamieniamy małe litery na wielkie lub odwrotnie, różnica między małą a odpowiadającą jej wielką literą to tylko jeden bit. Na rysunku 5.1 pokazano przykładowo kody liter *E* i *e*.



Rysunek 5.1. *Kody ASCII liter E i e*

Oba znaki różnią się tylko piątym bitem. Wielkie litery mają piąty bit równy zero, małe — równy jeden. Można skorzystać z tego, aby szybko zamieniać małe litery na wielkie i odwrotnie; wystarczy przełączyć tylko jeden bit.

Bity piąty i szósty decydują o przynależności znaku do grupy (tabela 5.1). Wobec tego małe litery na wielkie (lub na znaki specjalne) można zamieniać, ustawiając bity piąty i szósty na zero.

Tabela 5.1. *Grupy znaków ASCII wyznaczone przez piąty i szósty bit*

Bit 6.	Bit 5.	Grupa znaków
0	0	znaki kontrolne
0	1	cyfry i znaki przestankowe
1	0	wielkie litery i znaki specjalne
1	1	małe litery i znaki specjalne

Przyjrzyjmy się przez chwilę kodom ASCII cyfr, pokazanym w tabeli 5.2. Dziesiętny zapis tych kodów niewiele nam mówi. Jednak zapis szesnastkowy ujawnia coś ważnego — młodszy półbajt kodu ASCII to binarny odpowiednik zapisywanej liczby. Jeśli odrzucimy starszy półbajt kodu (ustawimy go na zero), otrzymamy binarny zapis odpowiedniej cyfry. Z drugiej strony, łatwo możemy zamienić liczbę binarną z zakresu 0..9 na jej odpowiednik ASCII; wystarczy ustawić starszy półbajt na %0011, czyli dziesiętne 3. Zauważmy, że możemy użyć logicznej operacji AND, aby wymusić wyzerowanie starszych bitów; tak samo za pomocą logicznego OR możemy wymusić ustawienie starszych bitów na %0011 (dziesiętnie 3). Więcej informacji o konwersji łańcuchów na liczby Czytelnik znajdzie w rozdziale 2.

Mimo że ASCII jest standardem, kodowanie danych za jego pomocą nie gwarantuje nam pełnej przenośności między różnymi systemami. Chociaż literze *A* na jednej maszynie będzie odpowiadało *A* na innej, zakres standaryzacji znaków kontrolnych jest niewielki. Faktycznie spośród pierwszych 32 kodów kontrolnych ASCII uzupełnionych kodem *delete* z grupy ostatniej tylko cztery są powszechnie obsługiwane przez większość urządzeń i programów. Są to: *backspace* (BS), tabulator, powrót karetki (CR)

Tabela 5.2. *Kody ASCII cyfr*

Znak	Dziesiętnie	Szesnastkowo
0	48	$30
1	49	$31
2	50	$32
3	51	$33
4	52	$34
5	53	$35
6	54	$36
7	55	$37
8	56	$38
9	57	$39

i nowy wiersz (LF). Co gorsza, na różnych maszynach często te same kody kontrolne są różnie obsługiwane. Szczególnie nieprzyjemnym przypadkiem jest koniec wiersza. W systemach Windows, MS-DOS, CP/M i innych koniec wiersza oznacza się parą znaków CR-LF. Apple Macintosh i wiele innych systemów oznacza koniec wiersza pojedynczym znakiem CR. Z kolei Linux, BeOS i inne systemy z rodziny Unix oznaczają go pojedynczym znakiem LF.

Próba przekazania zwykłego pliku tekstowego między tymi systemami może być źródłem poważnej frustracji. Nawet jeśli używamy standardowych znaków ASCII, i tak musimy zawczasu skonwertować dane. Na szczęście konwersje te są proste na tyle, że wiele edytorów tekstu automatycznie obsługuje pliki z różnie kończącymi się wierszami. Istnieje też wiele darmowych programów wykonujących stosowne konwersje. Nawet jeśli musimy sami oprogramować taką konwersję, jest ona prosta — kopiujemy wszystkie znaki poza końcem wiersza, który odpowiednio zmieniamy.

5.1.2. Zestaw znaków EBCDIC

Wprawdzie nie ulega wątpliwości, że zestaw ASCII jest najpopularniejszym zestawem znaków, ale nie jest on jedyny. Na przykład IBM na wielu swoich komputerach *mainframe* i minikomputerach używa kodu EBCDIC. Kod ten pojawia się głównie na dużych komputerach, natomiast na osobistych spotyka się go bardzo rzadko, więc poświęcimy mu niewiele uwagi.

EBCDIC to skrót od angielskiego *Extended Binary Coded Decimal Interchange Code*, czyli *Rozszerzony kod wymiany danych dziesiętnych kodowanych binarnie*. Czy istniał kiedyś taki kod nierozszerzony? Owszem, kiedyś w komputerach IBM i w maszynach do dziurkowania kart używano zestawu znaków znanego jako BCDIC. Zestaw ten był oparty na kartach dziurkowanych i na zapisie dziesiętnym (wiązało się to ze stosowaniem zapisu dziesiętnego w starszych urządzeniach IBM-a).

Pierwsze, co trzeba powiedzieć o EBCDIC — nie jest to pojedynczy zestaw znaków, ale cała rodzina takich zestawów. Zestawy znaków EBCDIC mają wspólną część (na przykład litery zwykle są kodowane tak samo), ale różne wersje EBCDIC (nazywane *stronami kodowymi*) różnie kodują znaki przestankowe i specjalne. Na pojedynczym bajcie liczba możliwych kodowań jest ograniczona, w różnych stronach kodowych te same kody są używane do zapisu różnych znaków. Jeśli zatem mamy plik ze znakami EBCDIC i mamy go zapisać jako ASCII, szybko okaże się, że to nie jest wcale proste zadanie.

Zanim w ogóle zobaczymy zestaw znaków EBCDIC, musimy sobie zdać sprawę, że przodek EBCDIC, czyli BCDIC, istniał na długo przed pojawieniem się współczesnych komputerów. BCDIC powstał na potrzeby maszyn do dziurkowania i czytników kart. EBCDIC pomyślano jako proste rozszerzenie kodowania tak, aby umożliwić stosowanie w komputerach IBM rozszerzonego zestawu znaków.

Jednak EBCDIC odziedziczył po BCDIC pewne nietypowe, dzisiaj już archaiczne cechy. Na przykład kodowanie liter alfabetu nie jest ciągłe. Jest to prawdopodobnie bezpośredni efekt stosowania pierwotnie kodowania dziesiętnego (BCD). Początkowo litery zapewne były kodowane na kolejnych znakach. Jednak kiedy IBM rozszerzył zestaw znaków, użyto kombinacji binarnych niewystępujących w formacie BCD (wartości typu %1010..%1111). Takie wartości binarne występują między dwiema dotąd sąsiadującymi wartościami BCD, więc niektóre ciągi znaków (na przykład litery) nie są zapisywane w kodowaniu EBCDIC w sposób ciągły.

Niestety, z uwagi na nietypowość zestawu znaków EBCDIC wiele powszechnie stosowanych algorytmów działających na zestawie ASCII w przypadku EBCDIC po prostu nie działa. W rozdziale tym nie będziemy zajmowali się kodowaniem EBCDIC bardziej, niż tylko wspominając o tym standardzie tu i ówdzie. Trzeba jednak pamiętać, że wszystkie funkcje tego kodu mają swoje odpowiedniki w zestawie ASCII. Po szczegółowe informacje odsyłam Czytelnika do dokumentacji IBM.

5.1.3. Dwubajtowe zestawy znaków

Z uwagi na ograniczenia kodowania 8-bitowego (co oznacza maksymalnie 128 znaków) oraz na konieczność zapisania większej liczby znaków, w części systemów używa się specjalnych kodów potrzebujących do zapisania jednego znaku dwóch bajtów. Takie dwubajtowe zestawy znaków nie używają 16 bitów do zapisu każdego znaku; w większości przypadków używany jest jeden bajt, a tylko w niektórych — dwa bajty.

Typowy dwubajtowy zestaw znaków wykorzystuje standardowy zestaw ASCII z dodatkowymi znakami z zakresu $80..$FF. Niektóre wartości z tego zakresu to kody rozszerzenia informujące, że pojawi się drugi bajt. Każdy bajt rozszerzenia pozwala zapisać 256 dodatkowych znaków. Mając trzy wartości rozszerzające, można obsłużyć maksymalnie 1021 różnych znaków. Z każdego bajta rozszerzającego otrzymujemy 256 znaków, poza tym mamy 253 (256–3) znaków w standardzie jednobajtowym (minus trzy, gdyż tyle znaków służy jako znacznik rozszerzający, który w związku z tym nie powinien być liczony jako zwykły znak).

Dawniej, kiedy terminale i komputery wykorzystywały odwzorowanie w pamięci, dwubajtowe zestawy znaków były mało przydatne. W przypadku generatorów znaków wbudowanych w urządzenie każdy znak musiał mieć taką samą długość, ograniczano też liczbę obsługiwanych znaków. Jednak kiedy zaczęły dominować matrycowe wyświetlacze z programowymi generatorami znaków (Windows, Macintosh, Unix/XWindows), używanie dwubajtowych zestawów znaków stało się wygodne.

Wprawdzie zestawy dwubajtowe pozwalają zapisywać w zwartej formie wiele różnych znaków, ale przetwarzanie tekstu w takim formacie wymaga sporo mocy obliczeniowej. Jeśli na przykład mamy zakończone zerami łańcuchy znaków dwubajtowych (konstrukcja typowa w C i C++), określenie liczby znaków w łańcuchu może być nie lada wyzwaniem. Chodzi o to, że niektóre znaki potrzebują dwóch bajtów, a niektóre tylko jednego. Funkcja określająca długość łańcucha musi przeglądać znak po znaku, aby znaleźć znaki rozszerzające i odkryć w ten sposób miejsca, w których są znaki dwubajtowe. To podwójne porównanie podwaja czas określania długości łańcucha. Co gorsza, wiele algorytmów powszechnie stosowanych do obsługi danych łańcuchowych zawodzi w przypadku dwubajtowych zestawów znaków. Na przykład typowy sposób przemieszczania się w języku C po łańcuchu to zwiększanie lub zmniejszanie o jeden wskaźnika, na przykład `++ptrChar` lub `--ptrChar`. Wszystkie te mechanizmy nie zadziałają w przypadku kodowania dwubajtowego. Osoby korzystające z takich kodowań zwykle posiadają gotowe już funkcje biblioteczne, ale wiele innych funkcji napisanych przez nie lub przez kogoś innego nie zadziała. Z tego i innych powodów, jeśli potrzebne jest nam kodowanie większej niż 124 liczby znaków, powinniśmy nastawić się na korzystanie ze standardu Unicode, który zaraz zostanie opisany.

5.1.4. Zestaw znaków Unicode

Jakiś czas temu inżynierowie Apple Computer i Xeroksa stwierdzili, że ich nowe systemy z wyświetlaczami mozaikowymi i czcionkami wybieranymi przez użytkownika mogą wyświetlić o wiele więcej niż 256 znaków naraz. Choć można było zastosować kodowania dwubajtowe, stwierdzono, że występują poważne problemy związane z faktem, że znaki mają po dwa bajty, i poszukiwano innego rozwiązania. Okazał się nim zestaw znaków Unicode. Standard ten przyjął się na całym świecie i jest obsługiwany przez niemalże każdy system komputerowy i system operacyjny (Mac OS, Windows, Linux, Unix i wiele innych).

W Unicode do zapisu każdego znaku używa się 16-bitowych słów, więc można zapisać do 65 536 różnych znaków. Jest to oczywiście o wiele, wiele więcej niż dotychczasowe 256 możliwych znaków w 8-bitowym bajcie. Mało tego, Unicode jest zgodny z ASCII. Jeśli najstarszych 9 bitów[4] jest ustawionych na zero, młodszych 7 to kod ze standardowego zestawu ASCII. Jeśli 9 starszych bitów zawiera wartości niezerowe, całe 16 bitów tworzy znak rozszerzony (rozszerzony względem ASCII). Jeśli ktoś się jeszcze zastanawia, po co właściwie mamy aż tyle kodów znaków, wystarczy przypomnieć, że niektóre azjatyckie zestawy zawierają ich 4096 (tyle znaków mają ich podzbiory

[4] ASCII to kod 7-bitowy. Jeśli najstarszych 9 bitów 16-bitowej wartości Unicode jest zerami, pozostałych 7 bitów to kod ASCII danego znaku.

w Unicode). Zestaw znaków Unicode zawiera nawet kody, które mogą być używane jako znaki definiowane na potrzeby poszczególnych aplikacji. W chwili pisania tej książki zdefiniowano około połowę z 65 536 dostępnych znaków; pozostałe są zarezerwowane na rozszerzenia w przyszłości.

Obecnie wiele systemów operacyjnych i bibliotek do różnych języków programowania zawiera doskonałą obsługę Unicode. Na przykład system Microsoft Windows używa standardu Unicode wewnętrznie[5], co powoduje, że funkcje systemowe działają szybciej, jeśli przekaże im się łańcuchy Unicode, niż w przypadku przekazania łańcuchów ASCII (kiedy przekazujemy do nowszych wersji Windows łańcuch ASCII, system najpierw zamienia go z ASCII na Unicode i dopiero wtedy używa API funkcji systemowej). Analogicznie, kiedy Windows zwraca aplikacji łańcuch, jest to łańcuch Unicode; jeśli oczekuje ona łańcucha ASCII, Windows musi dodatkowo przeprowadzić stosowną konwersję.

Jednak Unicode ma też dwie poważne wady. Po pierwsze, dane znakowe zajmują dwa razy więcej miejsca niż w przypadku ASCII lub innych kodowań jednobajtowych. Chociaż obecnie komputery mają znacznie więcej pamięci niż kiedyś (dotyczy to tak pamięci RAM, jak i pamięci dyskowych), podwojenie rozmiaru plików tekstowych, baz danych i łańcuchów umieszczanych w pamięci (jak łańcuchy przetwarzane przez edytory i procesory tekstu) może znacząco wpłynąć na wydajność systemu. Co gorsza, skoro łańcuchy są obecnie dwukrotnie dłuższe, przetwarzanie łańcucha Unicode wymaga prawie dwukrotnie więcej instrukcji niż przetwarzanie łańcucha, w którym znaki są zapisywane na pojedynczych bajtach. Wobec tego funkcje obsługi łańcuchów mogą działać dwukrotnie wolniej niż funkcje przetwarzające dane jednobajtowe[6]. Druga wada Unicode jest taka, że większość istniejących gdziekolwiek na świecie plików z danymi jest zapisana jako ASCII lub EBCDIC, więc w przypadku korzystania w aplikacji z Unicode pewną ilość czasu trzeba poświęcić na konwersję między Unicode a innymi zestawami znaków.

Choć Unicode jest standardem powszechnie akceptowanym, nadal nie wydaje się, żeby był w powszechnym użyciu (choć stale zyskuje na popularności). Można się spodziewać, że już wkrótce przekroczy on „masę krytyczną" i stanie się wszechobecny. Jednak jest to nadal kwestia przyszłości, więc w większości przykładów w naszej książce nadal ograniczać się będziemy do znaków ASCII. Ale już niedługo zapewne w książce takiej jak ta trzeba będzie się skupić raczej na Unicode.

---

[5] Windows CE używa wyłącznie Unicode. Do funkcji tego systemu w ogóle nie ma możliwości przekazania łańcuchów ASCII.

[6] Można by twierdzić, że przetwarzanie łańcuchów Unicode za pomocą instrukcji przetwarzających słowa nie powinno trwać dłużej niż przetwarzanie bajtów za pomocą instrukcji przetwarzających bajty. Jednak zoptymalizowane funkcje przetwarzające łańcuchy przetwarzają zwykle podwójne słowa, a nawet większe jednostki danych. Takie funkcje mogą przetworzyć jednorazowo dwukrotnie mniej znaków Unicode niż znaków jednobajtowych, stąd do wykonania tej samej pracy trzeba dwukrotnie więcej instrukcji maszynowych.

5.2. Łańcuchy znakowe

Łańcuchy znakowe są prawdopodobnie drugim najbardziej powszechnie stosowanym typem danych, zaraz za liczbami całkowitymi. *Łańcuch znakowy* to ciąg znaków mający dwa atrybuty: *długość* i *dane znakowe*. Łańcuchy znakowe mają też inne atrybuty, na przykład *długość maksymalną*, jaką można zapisać w danej zmiennej, czy *liczbę odwołań* informującą, ile różnych zmiennych łańcuchowych odwołuje się do tego samego łańcucha. Wkrótce zajmiemy się tymi atrybutami i użyciem ich w programach, gdzie opiszemy różne formaty łańcuchowe oraz opowiemy o możliwych działaniach na łańcuchach.

5.2.1. Formaty łańcuchów znakowych

Różne języki różnie zapisują łańcuchy. Niektóre formaty łańcuchów wymagają mniej pamięci, inne pozwalają na szybsze przetwarzanie, jeszcze inne są wygodniejsze w użyciu, następne oferują dodatkowe funkcje programiście i systemowi operacyjnemu. Aby lepiej zrozumieć zasady rządzące łańcuchami znakowymi, dobrze jest przyjrzeć się pewnym typowym sposobom ich zapisu, występującym w różnych językach programowania wysokiego poziomu.

5.2.1.1. Łańcuchy zakończone zerem

Niewątpliwie *łańcuchy zakończone zerem* są najpopularniejszą obecnie wykorzystywaną formą zapisu. Jest to format natywny dla języków C, C++, Java i innych. Poza tym łańcuchy zakończone zerem spotyka się w programach pisanych w językach, które nie mają własnego formatu natywnego łańcuchów, takich jak asemblery.

Zakończony zerem łańcuch ASCII to ciąg zawierający zero lub więcej 8-bitowych znaków, kończący się bajtem zerowym (a w przypadku Unicode — ciąg mający zero lub więcej 16-bitowycyh kodów znaków zakończonych 16-bitowym zerowym słowem). Na przykład w C i C++ łańcuch ASCII „abc” wymaga czterech bajtów: po jednym na każdą z liter: *a*, *b* i *c* oraz jednego bajta zerowego.

Łańcuchy zakończone zerem mają pewne zalety, których pozbawione są inne formaty:

- Mogą zawierać praktycznie dowolnie długie ciągi znakowe, z narzutem zaledwie jednego bajta (lub dwóch bajtów w przypadku Unicode).
- Z uwagi na popularność języków C i C++ dostępne są bardzo dobrze zoptymalizowane biblioteki do ich obsługi.
- Są łatwe w implementacji. Faktycznie poza przypadkiem stałych literałów łańcuchowych języki C i C++ w ogóle nie zawierają obsługi takich łańcuchów. W tych językach są one po prostu tablicami znakowymi. Dlatego chyba projektanci C wybrali taki właśnie format — nie musieli troszczyć się o wzbogacanie języka o operatory łańcuchowe.
- Omawiany format pozwala zapisywać łańcuchy zakończone zerem w dowolnym języku umożliwiającym tworzenie tablic znaków.

Jednak łańcuchy zakończone zerem mają też wady, wobec czego nie zawsze są optymalną strukturą dla danych łańcuchowych. Wady ich używania są następujące:

- Funkcje do obsługi łańcuchów często działają bardzo niewydajnie w ich przypadku. Wiele operacji łańcuchowych musi znać długość łańcucha przed rozpoczęciem jego przetwarzania. Jedynym sposobem obliczenia długości łańcucha zakończonego zerem jest przejrzenie go od początku do końca. Im jest on dłuższy, tym wolniej działać będzie funkcja. Wobec tego łańcuchy zakończone zerem nie są najlepszym rozwiązaniem w przypadku przetwarzania długich łańcuchów.
- W ich przypadku trudno jest zapisać znak, którego kod jest zerem — na przykład znak ASCII NUL. Zwykle nie jest to jednak istotne.
- W takim łańcuchu nie ma nigdzie umieszczonej informacji, jak bardzo może on zostać wydłużony. Wobec tego w przypadku części funkcji do obsługi łańcuchów (takich jak konkatenacja) można dojść tylko do obecnego końca zmiennej lub ewentualnie sprawdzić przepełnienie, jeśli jawnie zostanie przekazana maksymalna dopuszczalna długość.

5.2.1.2. Łańcuchy poprzedzone długością

Drugi format łańcuchów znakowych, *łańcuchy poprzedzone długością*, pozwala uniknąć części problemów charakterystycznych dla łańcuchów zakończonych zerem. Łańcuchy poprzedzone długością są charakterystyczne dla języków takich jak Pascal; składają się zwykle z pojedynczego bajta określającego długość łańcucha, po którym następuje zero lub więcej 8-bitowych znaków. W tym wypadku napis „abc” składałby się z czterech bajtów: długości łańcucha ($03), a następnie znaków *a*, *b* i *c*.

Dzięki łańcuchom poprzedzonym długością można uniknąć dwóch problemów, na jakie natknęliśmy się przy poprzednim formacie ich zapisu. Po pierwsze, bez problemu można zapisywać znak NUL. Po drugie, szybciej działają operacje łańcuchowe. Inna ich zaleta jest taka, że jeśli spojrzeć na taki łańcuch jak na tablicę znakową, długość zwykle znajduje się na pozycji zerowej, więc indeksy samego łańcucha zaczynają się od jedynki. W przypadku wielu funkcji łańcuchowych indeksowanie znaków od jedynki jest znacznie wygodniejsze niż indeksowanie od zera (jak dzieje się w łańcuchach zakończonych zerem).

Łańcuchy poprzedzone długością mają też wady, z których najważniejszą jest narzucone ograniczenie długości do 255 znaków (o ile długość jest zapisywana w jednym bajcie). Można ominąć to ograniczenie, zapisując długość na 2 lub 4 bajtach, ale wtedy zwiększa się narzut na każdy łańcuch.

5.2.1.3. Łańcuchy siedmiobitowe

Ciekawy format działający na kodach 7-bitowych, takich jak ASCII, zakłada użycie najstarszego bita do oznaczenia końca łańcucha. Wszystkie znaki poza ostatnim w łańcuchu mają najstarszy bit wyzerowany (lub ustawiony, kwestia gustu), a ostatni znak ma ten bit ustawiony (lub odwrotnie, wyzerowany).

Oto wady łańcuchów siedmiobitowych:

- Aby określić ich długość, trzeba je w całości przejrzeć.
- W tym formacie nie można zapisać łańcucha o zerowej długości.
- Niewiele języków programowania pozwala zapisywać stałe literały jako łańcuchy 7-bitowe.
- Ograniczeni jesteśmy do 128 kodów znaków, choć w przypadku korzystania ze zwykłego ASCII nie stanowi to problemu.

Jednak ogromną zaletą łańcuchów 7-bitowych jest to, że nie wymagają żadnego narzutu na kodowanie długości. Do ich obsługi najlepszym wyborem jest asembler (przy czym definiuje się makro do tworzenia stałych literałów takich łańcuchów). Wynika to ze zwartości zapisu tych łańcuchów, co docenią przede wszystkim właśnie programiści asemblerowi. Oto makro asemblera HLA, które konwertuje literał łańcuchowy na łańcuch 7-bitowy:

```
#macro sbs( s );

    // Pobranie wszystkich znaków łańcucha poza ostatnim:

    (@substr( s, 0, @length(s) - 1) +

        // Złączenie ostatniego znaku z ustawionym najstarszym bitem:

        char( uns8( char( @substr( s, @length(s) - 1, 1))) | $80 ))

#endmacro
    ...
byte sbs( "Hello World" );
```

5.2.1.4. Łańcuchy HLA

Póki nie przeszkadza nam kilka dodatkowych bajtów narzutu, można stworzyć format łańcuchów łączących w sobie zalety łańcuchów poprzedzonych długością i zakończonych zerem, unikając jednocześnie wad jednych i drugich. Przykładem jest natywny format łańcuchów języka HLA[7].

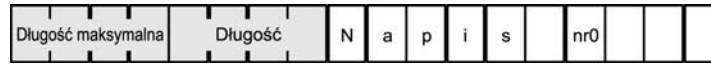
Najpoważniejszą wadą łańcuchów w formacie HLA jest duży narzut na każdy łańcuch. Procentowo może on być szczególnie istotny w środowiskach o ograniczonych zasobach i w przypadku korzystania z wielu krótkich łańcuchów. Łańcuchy HLA zawierają zarówno długość przed łańcuchem właściwym, jak i kończący bajt zerowy, a także inne informacje; łącznie narzut wynosi dziewięć bajtów na łańcuch[8].

[7] Zwróćmy uwagę na to, że HLA jest asemblerem, więc można — i to bez trudu — obsłużyć w nim praktycznie dowolny rozsądny format łańcuchów. Natywny format łańcuchów HLA wykorzystywany jest w stałych literałach; jest on też używany przez większość procedur z biblioteki standardowej HLA.

[8] Tak naprawdę, w związku z wymaganiami co do wyrównania danych w pamięci, narzut dla niektórych łańcuchów może sięgać 12 bajtów.

Format HLA wykorzystuje 4-bajtowy przedrostek opisujący długość; dzięki temu długość ta może być nieco większa niż cztery miliardy znaków (jest to oczywiście o wiele więcej, niż w ogóle znajduje praktyczne zastosowanie). HLA wstawia też na koniec danych łańcuchowych bajt zerowy, dzięki czemu łańcuchy HLA są zgodne funkcjami obsługującymi łańcuchy zakończone zerem (pod warunkiem, że funkcje te nie zmieniają długości łańcuchów). Następne cztery dodatkowe bajty w łańcuchu HLA zawierają maksymalną dopuszczalną długość łańcucha. Dzięki temu dodatkowemu polu funkcje obsługujące w HLA łańcuchy mogą w razie potrzeby sprawdzić, czy nie wystąpiło przepełnienie. Na rysunku 5.2 pokazano postać łańcucha HLA w pamięci.

Rysunek 5.2. *Format łańcuchów HLA*



Cztery bajty znajdujące się tuż przed pierwszym znakiem łańcucha to faktyczna jego długość. Cztery poprzednie bajty to maksymalna długość łańcucha. Za znakami wchodzącymi w skład tekstu znajduje się bajt zerowy. HLA gwarantuje, że cała struktura danych łańcucha będzie wielokrotnością czterech bajtów (jest to korzystne z uwagi na wydajność), więc na końcu takiego obiektu może znajdować się do trzech bajtów wypełnienia (aby struktura z rysunku 5.2 miała długość będącą wielokrotnością czterech bajtów, wystarczą dwa bajty wypełnienia).

Zmienne łańcuchowe HLA to wskaźniki zawierające adres bajta z pierwszym znakiem łańcucha. Aby sięgnąć do pól określających długość, trzeba załadować wartość wskaźnika do rejestru 32-bitowego. Do pola określającego długość sięgamy, podając offset równy –4, a do pola określającego długość maksymalną — offset równy –8. Oto przykład:

```
static
    s :string := "Hello World";
        ...
    mov( s, esi );          // Do esi wstawiamy adres litery 'H'
                            // z napisu "Hello World"
    mov( [esi-4], ecx );    // Długość łańcucha wstawiamy do ECX (dla "Hello
                            //  World" jest to 11).
        ...
    mov( s, esi );
    cmp( eax, [esi-8] );    // Sprawdzamy, czy długość z EAX przekracza
                            // maksymalną długość łańcucha.
    ja StringOverflow;
```

Miłą cechą zmiennych łańcuchowych HLA jest to, że (jako obiekty tylko do odczytu) są one kompatybilne z łańcuchami zakończonymi zerem. Na przykład, jeśli mamy funkcję języka C (lub innego) spodziewającą się jako parametru łańcucha zakończonego zerem, możemy wywołać ją, przekazując jej łańcuch HLA:

```
jakasFunkcjaC( zmiennaLancuchowaHLA );
```

Jedynym problemem jest to, że funkcja ta nie może w żaden sposób zmieniać długości łańcucha (gdyż kod C nie skorygowałby pola określającego długość). Oczywiście, przy powrocie zawsze można byłoby użyć funkcji C `strlen`, aby sprawdzić długość łańcucha i w razie potrzeby ją skorygować, ale modyfikowanie długości łańcuchów HLA z zewnątrz w zasadzie jest kiepskim pomysłem.

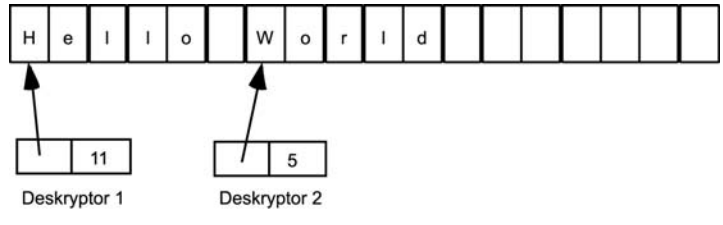
5.2.1.5. Łańcuchy z deskryptorem

Rozważane przez nas dotąd formaty łańcuchów miały wszystkie atrybuty łańcucha (długości i bajty kończące) wraz z samym łańcuchem. Być może nieco elastyczniejszym rozwiązaniem jest trzymanie informacji o maksymalnej i bieżącej długości łańcucha w strukturze zawierającej też wskaźnik do danych znakowych. Takie struktury nazywamy *deskryptorami*. Oto przykładowa struktura danych języka Pascal (lub Delphi czy Kyliksa):

```
type
    dString :record
        curLength  :integer;
        strData    :^char;
    end;
```

Zauważmy, że struktura ta nie zawiera danych znakowych, ale ma wskaźnik `strData` do pierwszego znaku łańcucha. Pole `curLength` określa bieżącą długość łańcucha. Oczywiście, można do takiego rekordu dodać dowolnie wiele innych potrzebnych pól, na przykład długość maksymalną, choć ta akurat nie jest tu potrzebna; większość formatów wykorzystujących deskryptory jest *dynamiczna* i ogranicza się tylko do przechowywania faktycznej długości.

Ciekawą cechą łańcuchów opartych na deskryptorach jest to, że same dane związane z deskryptorem mogą być częścią dłuższego łańcucha. W danych tych nie jest zapisywana długość łańcucha ani nie są wstawiane żadne bajty kończące, więc dane znakowe dwóch łańcuchów mogą na siebie nachodzić. Spójrzmy na przykład na rysunek 5.3. Mamy tu dwa łańcuchy: „Hello World” i „World”. Napisy ten nachodzą na siebie. Dzięki temu można zaoszczędzić nieco pamięci, poza tym niektóre funkcje (takie jak `substring`) mogą działać bardzo szybko. Oczywiście, kiedy mamy do czynienia z takim zachodzeniem na siebie łańcuchów, jak to pokazano, nie można modyfikować danych łańcucha, gdyż powodowałoby to modyfikowanie jednocześnie części drugiego łańcucha.



Rysunek 5.3. *Zachodzące na siebie łańcuchy wykorzystujące deskryptory*

5.2.2. Rodzaje łańcuchów — statyczne, pseudodynamiczne i dynamiczne

Wśród łańcuchów, które dotąd omówiliśmy, możemy wyróżnić trzy typy, jeśli podział ów będzie przebiegał według tego, kiedy system alokuje pamięć na dane znakowe. Chodzi o łańcuchy statyczne, pseudodynamiczne i dynamiczne.

5.2.2.1. Łańcuchy statyczne

Łańcuchy statyczne to takie, w których programista ustala maksymalną ich długość podczas pisania programu. Do tej grupy należą łańcuchy Pascala (i łańcuchy Delphi krótkie, ang. *short*). Należą tu też tablice znaków zawierające zakończone zerem łańcuchy C i C++. Weźmy pod uwagę następującą deklarację Pascala:

```
(* przykład statycznego łańcucha w Pascalu *)

var  pascalString :string(255);   // Maksymalna długość to zawsze 255 znaków.
```

A oto przykład w C i C++:

```
// przykład statycznego łańcucha w C i C++:

char cString[256];   // Maksymalna długość to zawsze 255 znaków (plus bajt zerowy).
```

Kiedy taki program działa, nie można w żaden sposób zwiększyć maksymalnej wielkości łańcuchów statycznych. Nie można też zmniejszyć ilości rezerwowanej na nie pamięci. Łańcuchy te zawsze zajmą 256 bajtów i kropka. Zaletą łańcuchów statycznych jest to, że kompilator może określić maksymalną ich długość na etapie kompilacji i niejawnie przekazać tę informację do funkcji obsługujących, dzięki czemu możliwe jest w trakcie działania programu kontrolowanie naruszeń długości.

5.2.2.2. Łańcuchy pseudodynamiczne

Łańcuchy pseudodynamiczne to takie, których długość jest ustawiana przez system w trakcie działania programu za pośrednictwem funkcji typu `malloc` alokujących pamięć na łańcuch. Jednak kiedy system już pamięć zaalokuje, wielkość maksymalna łańcucha jest ustalona. Tak właśnie działają łańcuchy HLA[9]. Programista HLA do alokowania pamięci na zmienną łańcuchową zwykle używa funkcji `stralloc`. Po jej wywołaniu obiekt łańcuchowy ma ustaloną długość, która nie może ulegać zmianom[10].

5.2.2.3. Łańcuchy dynamiczne

Systemy łańcuchów dynamicznych, zwykle wykorzystujące format oparty na deskryptorach, automatycznie alokują wystarczającą ilość pamięci przy każdym utworzeniu nowego łańcucha lub takiej modyfikacji, która zmienia długość łańcucha już istniejącego. W przypadku łańcuchów dynamicznych operacje takie, jak przypisanie czy pobranie części łańcucha, są banalnie proste — zwykle wystarczy tylko skopiować dane z deskryptora, więc operacje te działają szybko. Jednak przy korzystaniu z łańcuchów w ten sposób trzeba pamiętać, że nie wolno danych z powrotem wstawiać do tego samego obiektu, gdyż można zmodyfikować dane wykorzystywane też przez inny łańcuch.

---

[9] Ale przecież HLA jest asemblerem, więc można stworzyć w nim także łańcuchy czysto statyczne i w pełni dynamiczne.

[10] Tak naprawdę można wywołać `strrealloc` do zmiany wielkości łańcucha HLA, ale łańcuchy dynamiczne załatwiają to automatycznie, czego nie umożliwiają funkcje obsługi łańcuchów w przypadku wykrycia przepełnienia.

Rozwiązaniem tego problemu jest wykorzystanie techniki nazywanej *kopiowaniem przy zapisie*. Polega ona na tym, że kiedy trzeba zmienić coś w łańcuchu dynamicznym, najpierw wykonywana jest kopia tego łańcucha i na niej przeprowadzane są wszystkie konieczne zmiany. Badania wykazały, że w przypadku typowych programów kopiowanie przy zapisie może poprawić wydajność wielu aplikacji, gdyż operacje takie jak przypisanie i wybieranie podłańcucha są wykonywane znacznie częściej niż modyfikowanie danych wewnątrz łańcucha. Jedyną wadą tego mechanizmu jest to, że po kilku modyfikacjach danych w pamięci sterta może zawierać nieużywane już łańcuchy znakowe. Aby uniknąć takich *wycieków pamięci*, w systemach dynamicznych łańcuchów znakowych zwykle stosowane jest *oczyszczanie*, które przeszukuje obszar z danymi łańcuchów znakowych, wyszukując stare, nieużywane dane i przywracając je pamięci pozostającej do dyspozycji programu. Niestety, takie oczyszczanie może działać powoli.

5.2.3. Zliczanie odwołań do łańcucha

Zastanówmy się nad sytuacją, w której mamy dwa deskryptory łańcuchów (lub po prostu dwa wskaźniki) wskazujące te same dane w pamięci. Oczywiście, nie można takiej pamięci zwolnić (czyli przeznaczyć do ponownego wykorzystania), gdy jeden ze wskaźników przestaje być używany. Jednym (i, niestety, powszechnym) rozwiązaniem jest zrzucenie odpowiedzialności za obsługę tego typu sytuacji na programistę. W miarę jednak, jak rośnie stopień skomplikowania aplikacji, poleganie na programiście często prowadzi do pojawiania się błędnych wskaźników, wycieków pamięci i innych tego typu problemów. Lepszym rozwiązaniem jest pozwolenie programiście na zwolnienie takiej wielokrotnie wykorzystanej pamięci, z jednoczesnym wstrzymaniem fizycznego zwolnienia do chwili zwolnienia ostatniego wskaźnika wskazującego te same dane. Do śledzenia wskaźników i związanych z nimi danych używa się *liczników odwołań*.

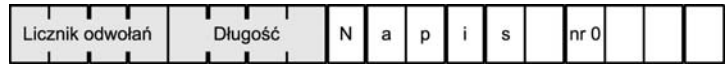
Licznik odwołań to liczba całkowita zliczająca wskaźniki odwołujące się do danych znakowych umieszczonych w pamięci. Przy każdym przypisaniu adresu łańcucha do jakiegoś wskaźnika licznik jest zwiększany o jeden. Przy każdym zwolnieniu któregoś ze wskaźników licznik jest zmniejszany o jeden. Faktyczne zwolnienie pamięci ma miejsce dopiero wtedy, gdy licznik zmniejszy się do zera.

Zliczanie odwołań świetnie się sprawdza, jeśli język obsługuje automatycznie szczegóły związane z przypisywaniem łańcuchów. Jeśli liczniki zechcemy zaimplementować ręcznie, jedyną trudnością będzie zapewnienie, że licznik zostanie zwiększony zawsze przy przypisywaniu wskaźnika innej zmiennej wskaźnikowej. Najlepiej nigdy nie przypisywać wskaźników bezpośrednio, ale skorzystać z funkcji lub makra aktualizujących liczniki. Jeśli licznik odwołań nie będzie prawidłowo korygowany, pojawią się nieprawidłowe wskaźniki lub dojdzie do wycieków pamięci.

5.2.4. Łańcuchy w Delphi i Kyliksie

Chociaż Delphi i Kylix obsługują „krótkie łańcuchy” zgodne z łańcuchami poprzedzonymi długością (znanymi z wcześniejszych wersji Delphi), w wersjach od 4.0 używane są łańcuchy dynamiczne. Wprawdzie format ten nie został opublikowany (i przez to może ulegać zmianom), ale eksperymenty z Delphi przeprowadzone przeze

mnie pokazały, że łańcuchy te są bardzo podobne do znanych z HLA. Delphi wykorzystuje zakończone zerem ciągi znaków poprzedzone długością łańcucha i licznikiem odwołań (zamiast długości maksymalnej używanej w HLA). Na rysunku 5.4 pokazano układ takiego łańcucha w pamięci.



Rysunek 5.4. *Format danych znakowych w Delphi i Kyliksie*

Tak jak w HLA, tak i w Delphi zmienne łańcuchowe to wskaźniki do pierwszego znaku faktycznych danych. Aby określić długość i liczbę odwołań, procedury Delphi i Kyliksa wykorzystują ujemne offsety –4 i –8. Jednak, jako że format ten nie został opublikowany, w aplikacjach nigdy nie należy sięgać do tych pól bezpośrednio. Delphi i Kylix zawierają funkcję określającą długość łańcucha, więc tak naprawdę nie ma powodu sięgać bezpośrednio do samego pola.

5.2.5. Tworzenie własnych formatów łańcuchów

Zwykle wykorzystuje się takie formaty łańcuchów, jakie udostępnia używany język programowania, chyba że mamy bardzo specyficzne wymagania. Jeśli tak, okazuje się, że większość języków programowania zawiera konstrukcje umożliwiające użytkownikom definiowanie własnych formatów łańcuchowych.

Jedyny problem, na jaki możemy się natknąć, to wymaganie, aby stałe literały łańcuchowe były zapisywane zawsze w jednym formacie. Jednak zwykle łatwo można napisać prostą funkcję konwersji, która zamieni takie literały na wybrany format.

5.3. Zbiory znaków

Kolejnym złożonym typem danych opartym na typie znakowym jest zbiór znaków rozumiany jako pojęcie matematyczne. Bycie elementem tego zbioru jest nałożoną nań relacją binarną. Znak albo należy do zbioru, albo nie. Nie można w zbiorze mieć wielu kopii tego samego znaku. Pojęcie ciągu (kiedy znaki występują w określonej kolejności, jak w łańcuchu znakowym) jest zbiorom znaków obce.

W tabeli 5.3 zestawiono wybrane typowe funkcje dotyczące zbiorów znaków, aby przybliżyć Czytelnikowi, jakie operacje wykonuje się na takich danych.

5.3.1. Zbiory znaków w formie zbioru przynależności

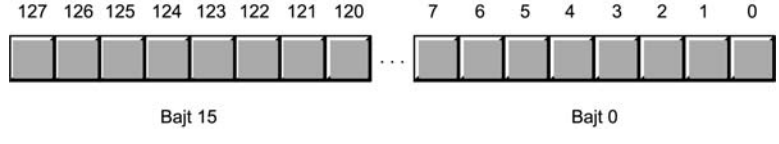
Zbiory znaków można zapisywać na różne sposoby. Niektóre języki implementują je jako tablice wartości logicznych (jedna wartość na jeden możliwy znak). Każda z tych wartości decyduje, czy odpowiadający jej znak jest w danym zbiorze dostępny; prawda oznacza dostępność znaku, zaś fałsz brak danego znaku w zestawie. Aby nie marnować

Tabela 5.3. *Typowe funkcje zbioru znaków*

Funkcja-operator	Opis
Przynależność	Pozwala sprawdzić, czy dany znak należy do zbioru (zwraca prawdę lub fałsz).
Przecięcie	Zwraca przecięcie dwóch zbiorów znaków (czyli zbiór składający się ze znaków należących do obu zbiorów pierwotnych).
Suma	Zwraca sumę dwóch zbiorów znaków (czyli wszystkie znaki będące elementami jednego lub obu zbiorów pierwotnych).
Różnica	Zwraca różnicę dwóch zbiorów (czyli znaki należące do jednego ze zbiorów, ale nie do drugiego).
Pobranie	Pobiera ze zbioru pojedynczy znak.
Podzbiór	Zwraca prawdę, jeśli jeden zbiór znaków jest podzbiorem drugiego.
Podzbiór właściwy	Zwraca prawdę, jeśli jeden zbiór znaków jest podzbiorem właściwym drugiego.
Nadzbiór	Zwraca prawdę, jeśli jeden zbiór znaków jest nadzbiorem drugiego.
Nadzbiór właściwy	Zwraca prawdę, jeśli jeden zbiór znaków jest nadzbiorem właściwym drugiego.
Równość	Zwraca prawdę, jeśli jeden zbiór znaków jest równy drugiemu.
Nierówność	Zwraca prawdę, jeśli jeden zbiór znaków nie jest równy drugiemu.

pamięci, zwykle każdemu znakowi ze zbioru przypisany jest pojedynczy bajt. Wobec tego taki zbiór znaków wymaga 16 bajtów (128 bitów) pamięci w przypadku 128 znaków lub 32 bajtów (256 bitów) w przypadku 256 możliwych znaków. Taki zapis zestawu znaków nazywamy *zbiorem przynależności*.

Język HLA wykorzystuje 16-bajtową tablicę do zapisu 128 możliwych znaków ASCII. Ta 128-bitowa tablica jest ułożona w pamięci tak, jak to pokazano na rysunku 5.5.



Rysunek 5.5. *Zapis zbioru znaków w języku HLA*

Zerowy bit zerowego bajta odpowiada znakowi ASCII o kodzie zero (znakowi NUL). Jeśli bit ten jest ustawiony, dany zestaw znaków zawiera znak NUL. Jeśli bit ten jest wyzerowany, zbiór nie zawiera znaku NUL. Analogicznie, bit pierwszy ósmego bajta odpowiada znakowi ASCII o kodzie 65, czyli wielkiej literze *A*. Bit 65. będzie zawierał jedynkę, jeśli w opisywanym zbiorze znaków znak *A* się pojawia — i zero w przeciwnym razie.

W Pascalu (w języku środowisk Delphi i Kylix) wykorzystuje się analogiczny schemat do zapisu zbiorów znaków. W Delphi zbiór znaków może zawierać ich do 256, więc takie struktury zajmują 256 bitów (czyli 32 bajty).

Zbiory znaków można zapisywać też na różne inne sposoby, ale zaletą opisanej tu postaci wektora bitowego (czyli tablicy bitowej) jest łatwość implementacji takich operacji na zbiorach, jak suma, przecięcie, różnica, porównanie i sprawdzenie przynależności.

5.3.2. Listowa reprezentacja zbiorów znaków

Czasami mapa bitowa zbioru przynależności nie sprawdza się. Jeśli na przykład zbiory są zawsze bardzo małe (zawierają nie więcej jak trzy czy cztery elementy), używanie 16 lub 32 bajtów do ich zapisu jest czystym marnotrawstwem. W przypadku bardzo małych zbiorów zwykle najlepszą metodą postępowania jest użycie list znaków[11]. Jeśli rzadko potrzebnych jest nam więcej niż kilka znaków, przeglądanie całego łańcucha w celu znalezienia potrzebnego znaku jest wystarczającym rozwiązaniem.

Poza tym, jeśli nasze zbiory mogą mieć wiele znaków, zapis w formie zbioru przynależności może być długi (na przykład taki zapis zbioru znaków Unicode wymagałby 8192 bajtów). Z tych (i nie tylko tych) powodów zapis w formie zbioru przynależności nie zawsze jest najlepszy. Wtedy z odsieczą przychodzą nam zapisy w formie list.

5.4. Definiowanie własnego zestawu znaków

W zestawach takich, jak ASCII, EBCDIC czy Unicode właściwie nie ma nic specjalnego. Ich główna siła tkwi w tym, że są standardami międzynarodowymi, więc bardzo wiele systemów z nich korzysta. Jeśli będziemy trzymać się jednego z tych standardów, bardzo prawdopodobne, że nie będziemy mieli problemów z wymianą danych z innymi. Po to właśnie tworzy się standardy kodowania.

Jednak kody te nie były projektowane z myślą o ułatwieniu przetwarzania znaków. ASCII i EBCDIC tworzono z myślą o urządzeniach, które dzisiaj mają wartość głównie muzealną. Zestaw ASCII miał pasować do mechanicznych klawiatur dalekopisów, a EBCDIC tworzono z myślą o starych systemach z kartami perforowanymi. Takie urządzenia trudno znaleźć dziś gdziekolwiek poza muzeum, więc układ znaków też nie pasuje do używanych obecnie komputerów. Gdybyśmy mogli dzisiaj stworzyć nowy zestaw znaków, najprawdopodobniej znacząco różniłby się on od ASCII i EBCDIC. Zapewne miałby wiele wspólnego z używanymi dzisiaj klawiaturami (zawierałby na przykład kody odpowiadające powszechnie używanym klawiszom, jak strzałki czy klawisze *PageUp* i *PageDown*). Kody byłyby tak ułożone, aby ułatwić wykonywanie typowych obliczeń.

Choć zestawy znaków ASCII i EBCDIC niezbyt szybko odejdą w zapomnienie, nikt nie broni nam zdefiniować własnego zestawu znaków. Zestawy znaków tworzone na potrzeby konkretnych aplikacji są — jak by to powiedzieć — dostosowane do potrzeb pojedynczych aplikacji, więc nie będziemy mogli potem wymieniać plików tekstowych z innymi aplikacjami, niedostosowanymi do naszego kodowania. Jednak dość łatwo zapewnić translację między różnymi zestawami znaków — wystarczy do tego słownik przekodowujący. Wtedy już prosta będzie zamiana kodowania wewnętrznego

[11] Jednak w tym wypadku to na programiście spoczywa odpowiedzialność za zachowanie semantyki zbiorów — czyli nie można dopuścić do tego, aby na liście którykolwiek znak pojawił się więcej niż raz.

na zewnętrzne (na przykład ASCII) na potrzeby funkcji wejścia i wyjścia. Jeśli dobierzemy dobre kodowanie, które pozwoli poprawić wydajność naszego programu, utrata wydajności związana z przekodowaniem wejścia-wyjścia będzie opłacalna. Jak zatem dobrać kodowanie do własnego zestawu znaków?

Pierwsze, nad czym można by się zastanowić, to ile znaków zamierzamy obsługiwać. Oczywiście, liczba znaków bezpośrednio wpłynie na wielkość danych znakowych. Typowym wyborem jest 256 znaków, gdyż to bajty są najpowszechniej stosowanym elementarnym typem danych. Jednak trzeba pamiętać, że jeśli nie potrzebujemy 256 znaków, nie powinniśmy tworzyć tak dużego zestawu. Jeśli na przykład wystarczy nam 128 znaków, albo nawet tylko 64, to nasze „pliki tekstowe" będą mniejsze. Analogicznie, przekazywanie danych przy użyciu specjalizowanych zestawów znaków będzie szybsze, jeśli będziemy musieli przekazać zawsze sześć albo siedem bitów zamiast ośmiu. Jeśli z kolei będziemy potrzebowali więcej niż 256 znaków, trzeba rozważyć zalety i wady stosowania wielu stron kodowych, korzystania ze znaków dwubajtowych, czyli 16-bitowych. Poza tym pamiętać trzeba, że zestaw Unicode umożliwia definiowanie własnych znaków. Jeśli zatem potrzebujemy więcej niż 256 znaków, warto pomyśleć nad użyciem Unicode i zawartych w nim znaków użytkownika, co pozwoli nam przynajmniej w jakimś stopniu pozostać w zgodzie ze standardami.

W tej części rozdziału zdefiniujemy 128-elementowy zestaw znaków wykorzystujący pełne, 8-bitowe bajty. Podstawowym naszym zadaniem będzie zmiana układu kodów ASCII tak, aby wygodniej było na nich robić pewne obliczenia; poza tym zmienimy znaczenie części kodów kontrolnych tak, aby były przydatne we współczesnych komputerach, a nie w archaicznych dalekopisach. Dodamy też kilka nowych znaków, niewystępujących w standardzie ASCII. Głównym celem tego ćwiczenia będzie uzyskanie większej wydajności obliczeniowej, a nie samo tworzenie nowych znaków. Nasz zestaw znaków nazwiemy zestawem HyCode.



Tworząc w tym rozdziale zestaw HyCode, nie staramy się stworzyć nowego standardu kodowania. HyCode to po prostu prezentacja tego, jak można stworzyć własny, dostosowany do specyficznych potrzeb zestaw znaków i w ten sposób udoskonalić swoje programy.

5.4.1. Tworzenie wydajnego zestawu znaków

Podczas tworzenia nowego zestawu znaków powinniśmy uwzględnić kilka spraw. Na przykład, czy będziemy musieli zapisywać łańcuchy znakowe w istniejącym formacie? Może to wpływać na sposób kodowania naszych łańcuchów. Jeśli na przykład będziemy chcieli używać funkcji bibliotecznych korzystających z łańcuchów zakończonych zerem, musimy w naszym zestawie zostawić zero jako znacznik końca łańcucha. Trzeba pamiętać też, że wiele funkcji obsługujących łańcuchy nie zadziała z naszym nowym zestawem niezależnie od tego, jak go zdefiniujemy. Przykładowo, funkcje typu `stricmp` działają tylko w przypadku liter ułożonych zgodnie ze standardem ASCII (lub innym powszechnie przyjętym standardem). Wobec tego nie powinniśmy tak bardzo przejmować się istniejącym formatem, gdyż wiele funkcji obsługujących łańcuchy i tak

będzie trzeba samemu napisać od nowa. Zestaw znaków HyCode nie rezerwuje jako końca łańcucha znaku zerowego — i to dobrze, gdyż, jak widzieliśmy, takie łańcuchy nie działają zbyt wydajnie.

Jeśli przyjrzymy się programom korzystających z funkcji do obsługi łańcuchów, stwierdzimy, że pewne funkcje występują szczególnie często:

- Sprawdzanie, czy znak jest cyfrą.
- Konwersja znaku cyfry na odpowiadającą mu liczbę.
- Konwersja liczby na odpowiadający jej znak cyfry.
- Sprawdzanie, czy znak jest literą.
- Sprawdzanie, czy znak jest małą literą.
- Sprawdzanie, czy znak jest wielką literą.
- Porównywanie dwóch znaków (lub łańcuchów) bez uwzględniania wielkości liter.
- Sortowanie zbioru łańcuchów (z uwzględnieniem i bez uwzględnienia wielkości liter).
- Sprawdzanie, czy dany znak jest znakiem alfanumerycznym.
- Sprawdzanie, czy dany znak może występować w identyfikatorach.
- Sprawdzanie, czy dany znak jest operatorem arytmetycznym lub logicznym.
- Sprawdzanie, czy znak jest nawiasem (czyli jednym ze znaków *(*, *)*, *[*, *]*, *{*, *}*, < lub >).
- Sprawdzanie, czy znak jest znakiem przestankowym.
- Sprawdzanie, czy znak jest białym znakiem (czyli spacją, tabulatorem lub znakiem nowego wiersza).
- Sprawdzanie, czy znak jest znakiem sterującym kursorem.
- Sprawdzanie, czy znak jest znakiem kontrolującym ekran (na przykład *PageUp*, *PageDown*, *Home* czy *End*).
- Sprawdzanie, czy znak odpowiada klawiszowi funkcyjnemu.

Nasz zestaw HyCode zdefiniujemy tak, aby opisane działania były tak szybkie i tak proste, jak tylko jest to możliwe. Znaczną przewagę nad kodem ASCII możemy uzyskać, jeśli wszystkie znaki wchodzące w skład jakiejś grupy zestawimy razem — na przykład wszystkie litery lub wszystkie znaki sterujące. Dzięki temu będziemy mogli robić dowolne z powyższych sprawdzeń za pomocą tylko dwóch porównań. Na przykład wygodnie byłoby sprawdzać, czy dany znak jest znakiem przestankowym, porównując go z wartościami ograniczającymi znaki przestankowe z góry i z dołu. Nie można w ten sposób obsłużyć wszystkich możliwych zakresów porównań, ale warto uwzględnić zakresy występujące najczęściej. Wprawdzie w ASCII pewne ciągi znaków są ułożone bardzo sensownie, ale można je jeszcze poprawić. Na przykład nie można tam sprawdzić, czy znak jest przestankowy, przy ograniczeniu się do dwóch porównań, gdyż znaki te są porozrzucane po całym zestawie.

5.4.2. Grupowanie znaków odpowiadających cyfrom

Weźmy pod uwagę pierwsze trzy funkcje z powyższej listy — najłatwiej będzie nam je realizować, jeśli znakom o kodach od *0* do *9* przypiszemy kolejne cyfry. Korzystając z pojedynczego porównania wartości bez znaku, sprawdzamy, czy dany znak jest cyfrą. Konwersja między cyfrą a jej wartością jest banalna, gdyż kod i wartość znaku są identyczne.

5.4.3. Grupowanie liter

Kolejnym typowym problemem związanym z obsługą znaków i łańcuchów znakowych jest obsługa liter. Zestaw ASCII, choć jest o niebo lepszy od EBCDIC, po prostu nie nadaje się do sprawdzania i przetwarzania liter. Oto problemy z literami ASCII, które rozwiążemy w HyCode:

- Litery zostały umieszczone w dwóch osobnych częściach. Sprawdzanie, czy znak jest literą, wymaga zrobienia czterech porównań.
- Małe litery mają kody ASCII większe od wielkich liter. Jeśli chodzi o porównania, bardziej intuicyjne jest traktowanie małych liter jako znajdujących się wcześniej niż wielkie.
- Wszystkie małe litery mają wartości większe od poszczególnych wielkich liter. Prowadzi to do niezgodnych z intuicją efektów, kiedy *a* jest większe od *B*, choć każde dziecko wie, że jest inaczej.

Kodowanie HyCode rozwiązuje te problemy na kilka ciekawych sposobów. Po pierwsze, do zapisu 52 liter używane są kody od $4C do $7F. HyCode wykorzystuje tylko 128 znaków ($00..$7F), z czego 52 to litery. Wobec tego, jeśli będziemy sprawdzać, czy dany znak jest literą, wystarczy porównać, czy jego kod jest większy lub równy $4C. W językach wysokiego poziomu wystarczy porównanie typu:

```
if( c >= 76) ...
```

Jeśli kompilator będzie obsługiwał HyCode, wystarczy zapis:

```
if( c >= 'a') ...
```

W asemblerze wystarczy wykorzystać dwie instrukcje:

```
cmp( al, 76 );
jnae NotAlphabetic;
    // Instrukcje wykonywane dla liter.

NotAlphabetic:
```

Kolejna zaleta HyCode (istotnie różniąca to kodowanie od innych) to przeplatanie wielkich i małych liter (czyli kolejno zapisywane są znaki *a*, *A*, *b*, *B*, *c*, *C* i tak dalej). Dzięki temu sortowanie i porównywanie łańcuchów jest bardzo łatwe, niezależnie od tego, czy uwzględniamy wielkość liter, czy nie. Przeplatanie powoduje, że najmłodszy

bit znaku wskazuje, czy znak jest małą literą (najmłodszy bit jest zerem), czy wielką (najmłodszy bit jest jedynką). HyCode wykorzystuje następujące kodowanie liter:

```
a:76, A:77, b:78, B:79, c:80, C:81, ..., y:124, Y:125, z:126, Z:127
```

Sprawdzanie, czy dany znak HyCode jest wielką, czy małą literą, jest nieco trudniejsze od sprawdzania, czy jest literą, ale w asemblerze jest to i tak prostsze od analogicznego sprawdzenia kodu ASCII. Aby sprawdzić, czy znak jest danej wielkości, robi się dwa porównania: najpierw sprawdza się, czy jest on literą, a potem określa jego wielkość. W C i C++ użylibyśmy do tego następujących instrukcji:

```
if( (c >= 76) && (c & 1))
{
    // kod wykonywany w przypadku wielkich liter
}

if( (c >= 76 && !(c & 1))
{
    // kod wykonywany w przypadku małych liter
}
```

Wyrażenie `(c & 1)` zwraca prawdę, jeśli najmłodszy bit `c` jest jedynką, czyli kiedy mamy do czynienia z wielką literą. Analogicznie `!(c & 1)` zwraca prawdę, jeśli najmłodszy bit `c` jest zerem, czyli kiedy `c` jest małą literą. W przypadku asemblera 80x86 można sprawdzić, czy znak jest wielką, czy małą literą, za pomocą trzech instrukcji maszynowych:

```
// Uwaga: ROR(1, AL) powoduje odwzorowanie małych liter na zakres $26..$3F (38..63),
//            a wielkich na zakres $A6..$BF (166..191). Wszystkie inne znaki są
//            odwzorowywane na mniejsze wartości z tych zakresów.

    ror( 1, al );
    cmp( al, $26 );
    jnae NotLower;      // Uwaga: obsługujemy wartości bez znaku!

        // Kod obsługujący małe znaki

NotLower:

// Zauważmy, że w instrukcja ROR tworzy kody z zakresu $A6..$BF będące
// ujemnymi wartościami 8-bitowymi. Są to przy okazji *najmniejsze* liczby
// ujemne, jakie ROR może wygenerować z zestawu znaków HyCode.

    ror( 1, al );
    cmp( al, $a6 );
    jge NotUpper      // Uwaga: obsługujemy wartości bez znaku!

        // Kod obsługujący wielkie litery.

NotUpper:
```

Niestety, niewiele języków programowania zawiera odpowiednik instrukcji `ror`, niewiele pozwala też traktować znaki jako wartości raz ze znakiem, raz bez niego. Wobec tego pokazany kod jest w zasadzie ograniczony w zastosowaniach do programów asemblerowych.

5.4.4. Porównywanie liter

Zastosowane w HyCode grupowanie liter oznacza, że kolejność słownikową możemy uzyskać prawie za darmo. Póki nie przeszkadza nam, że małe litery są mniejsze od odpowiadających im wielkich, sortowanie łańcuchów HyCode daje kolejność słownikową. Wynika to stąd, że w HyCode w przeciwieństwie do ASCII zachodzą następujące relacje między literami:

```
a < A < b < B < c < C < d < D < ... < w < W < x < X < y < Y < z < Z
```

Tak właśnie wygląda kolejność słownikowa, poza tym takiej kolejności użytkownicy oczekują intuicyjnie.

Porównywanie niezależne od wielkości liter jest tylko nieznacznie trudniejsze od porównywania z uwzględnieniem tej wielkości (i o wiele łatwiejsze od porównywania bez uwzględniania wielkości liter w ASCII). Kiedy porównujemy dwie litery, po prostu maskujemy ich najmłodsze bity (lub ustawiamy je na jeden) i automatycznie uzyskujemy porównanie niezależne od ich wielkości.

Aby zobaczyć, jakie korzyści daje HyCode przy porównywaniu bez uwzględnienia wielkości liter, przyjrzyjmy się takiemu porównaniu w C i C++ dla znaków ASCII:

```
if( toupper( c ) == toupper( d ))
{
    // kod obsługi c==d z dokładnością co do wielkości liter.
}
```

Kod ten wygląda całkiem przyzwoicie, ale przyjrzyjmy się teraz funkcji (a raczej makru) `toupper`[12].

```
#define toupper(ch) ((ch >= 'a' && ch <= 'z') ? ch & 05f : ch )
```

Teraz, kiedy mamy już takie makro, preprocesor języka C pokazany powyżej kod rozwinie następująco:

```
if
(
        ((c >= 'a' && c <= 'z') ? c & 0x5f : c )
     == ((d >= 'a' && d <= 'z') ? d & 0x5f : d )
)
{
    // kod obsługi c==d z dokładnością co do wielkości liter.
}
```

Oto przykład kodu 80x86 realizującego podobne funkcje:

```
    // Zakładamy, że c jest w cl, a d w dl

    cmp( cl, 'a');      // Sprawdzamy, czy c jest z zakresu 'a'..'z'.
    jb NotLower;
```

[12]Tak naprawdę w standardowej bibliotece C może być jeszcze gorzej: obecnie biblioteki te zawierają tablice przekodowań odwzorowujące zakresy znaków, które jednak tu pominiemy.

```
    cmp( cl, 'z' );
    ja NotLower;
    and( $5f, cl);      //Konwersja małych liter z c1 na wielkie
Notlower:

    cmp( dl, 'a' );      // Sprawdzamy, czy d jest w zakresie 'a'..'z'
    jb NotLower2;
    cmp( dl, 'z' );
    ja NotLower2;
    and( $5f, dl );      // Konwersja małego znaku z dl na wielki.
Notlower2:

    cmp( cl, dl );      // Porównanie znaków (jeśli były to litery,
                        //  teraz są wielkimi literami).
    jne NotEqual;       // Pomijamy kod obsługujący c==d, gdy c i d nie są
                        // sobie równe.

        // Kod obsługujący c==d bez uwzględniania wielkości liter.
NotEqual:
```

Kiedy używamy HyCode, porównanie bez uwzględniania wielkości liter jest znacznie prostsze. Oto odpowiedni kod asemblera HLA.

```
// Sprawdzamy, czy CL jest literą. Nie trzeba porównywać DL, gdyż jeśli
// DL nie jest literą, porównanie zawsze zawiedzie.

    cmp( cl, 76);       // Jeśli CL < 76 ('a'), nie jest to litera, więc oba
    jb TestEqual;       // znaki na pewno nie są sobie równe — nawet pomijając
                        // wielkość liter.

    or( 1, cl );        // CL to litera — niech będzie wielka.
    or( 1, dl );        // DL może zawierać literę lub nie. Wymuszamy jej zamianę
                        // na wielką.

TestEqual:
    cmp( cl, dl );      // Porównanie wielkich wersji przekazanych znaków. Jeśli
    jb TestEqual;       // nie są sobie równe, odrzucamy je.

TheyreEqual:
    // Kod obsługujący c==d przy porównaniu bez uwzględnienia wielkości liter.

NotEqual:
```

Jak widać, w ciągu HyCode do porównania dwóch znaków używa się połowy instrukcji.

5.4.5. Inne grupowania znaków

Skoro na jednym końcu zakresu znaków są litery, a na drugim cyfry, sprawdzenie, czy znak jest alfanumeryczny, wymaga dwóch porównań (to i tak lepiej niż cztery porównania konieczne w ASCII). Oto kod Pascala (Delphi, Kyliksa) do sprawdzania, czy znak jest alfanumeryczny:

```
if( ch < chr(10) or ch >= chr(76)) then ...
```

Istnieją programy — i to nie tylko kompilatory — które muszą skutecznie przetwarzać łańcuchy znaków będące identyfikatorami. W większości języków w identyfikatorach mogą występować znaki alfanumeryczne; my do ich identyfikacji potrzebujemy dwóch porównań.

W wielu językach w identyfikatorach mogą występować także podkreślenia, a czasami (na przykład w MASM i TASM) inne znaki, jak „małpa" (@) czy znak dolara ($). Wobec tego podkreśleniu przypiszemy wartość 75, dolarowi i „małpie" kody 73 i 74 — dzięki temu nadal znaki występujące w identyfikatorach będziemy mogli wskazać po dwóch porównaniach.

Z podobnych powodów w HyCode zgrupowano kilka innych klas znaków w ciągłe obszary. Na przykład zgrupowane są klawisze sterujące kursorem, białe znaki, nawiasy (okrągłe, kwadratowe, klamrowe i trójkątne), operatory arytmetyczne, znaki przestankowe i tak dalej. W tabeli 5.4 przedstawiono kompletny zestaw znaków HyCode. Analizując kody liczbowe poszczególnych znaków, zauważymy, że te kody pozwalają sprawnie wykonywać większość opisanych wcześniej funkcji.

5.5. Dodatkowe informacje

ASCII, EBCDIC i Unicode to standardy międzynarodowe. Więcej informacji o rodzinie zestawów znaków EBCDIC znaleźć można na witrynie firmy IBM, *http://www.ibm.com*. ASCII i Unicode to standardy ISO i jako takie są udokumentowane przez ISO. W zasadzie dokumentacja ta jest płatna, ale mnóstwo informacji o zestawach znaków ASCII i Unicode można znaleźć, podając ich nazwy w wyszukiwarce internetowej. O Unicode można też poczytać na witrynie *http://www.unicode.org*.

Osoby zainteresowane szerszym opisem znaków, łańcuchów, zestawów znaków i funkcji do obsługi tych struktur powinny zajrzeć do dokumentacji następujących języków:

- Język programowania awk.
- Język programowania Perl.
- Język programowania SNOBOL4.
- Język programowania Icon.
- Język programowania SETL.
- Asembler HLA.

Szczególnie asembler HLA zawiera szeroki zestaw funkcji do obsługi znaków, łańcuchów, zestawów znaków i do dopasowywania wzorców. Podręcznik biblioteki standardowej HLA dostępny jest pod adresem *http://webster.cs.ucr.edu*.

Tabela 5.4. *Zestaw znaków HyCode*

Binarnie	Szesnastkowo	Dziesiętnie	Znak	Binarnie	Szesnastkowo	Dziesiętnie	Znak
0000_0000	00	0	0	0001_1110	1E	30	End
0000_0001	01	1	1	0001_1111	1F	31	Home
0000_0010	02	2	2	0010_0000	20	32	PageDown
0000_0011	03	3	3	0010_0001	21	33	PageUp
0000_0100	04	4	4	0010_0010	22	34	w lewo
0000_0101	05	5	5	0010_0011	23	35	w prawo
0000_0110	06	6	6	0010_0100	24	36	w górę
0000_0111	07	7	7	0010_0101	25	37	w dół (nowy wiersz)
0000_1000	08	8	8	0010_0110	26	38	spacja nierozdzielająca
0000_1001	09	9	9	0010_0111	27	39	akapit
0000_1010	0A	10	klawiatura numeryczna	0010_1000	28	40	powrót karetki
0000_1011	0B	11	kursor	0010_1001	29	41	nowy wiersz (Enter)
0000_1100	0C	12	funkcja	0010_1010	2A	42	tabulator
0000_1101	0D	13	alt	0010_1011	2B	43	spacja
0000_1110	0E	14	control	0010_1100	2C	44	(
0000_1111	0F	15	polecenie	0010_1101	2D	45	)
0001_0000	10	16	len	0010_1110	2E	46	[
0001_0001	11	17	len128	0010_1111	2F	47	]
0001_0010	12	18	bin128	0011_0000	30	48	{
0001_0011	13	19	EOS	0011_0001	31	49	}
0001_0100	14	20	EOF	0011_0010	32	50	<
0001_0101	15	21	kontrola	0011_0011	33	51	>
0001_0110	16	22	break (przerwanie)	0011_0100	34	52	=
0001_0111	17	23	escape (cancel)	0011_0101	35	53	^
0001_1000	18	24	pauza	0011_0110	36	54	|
0001_1001	19	25	dzwonek	0011_0111	37	55	&
0001_1010	1A	26	tabulacja wstecz	0011_1000	38	56	-
0001_1011	1B	27	backspace	0011_1001	39	57	+
0001_1100	1C	28	delete				
0001_1101	1D	29	insert				
0011_1010	3A	58	*	0101_1101	5D	93	I
0011_1011	3B	59	/	0101_1110	5E	94	j

Tabela 5.4. *Zestaw znaków HyCode — ciąg dalszy*

Binarnie	Szesnastkowo	Dziesiętnie	Znak	Binarnie	Szesnastkowo	Dziesiętnie	Znak
0011_1100	3C	60	%	0101_1111	5F	95	J
0011_1101	3D	61	~	0110_0000	60	96	k
0011_1110	3E	62	!	0110_0001	61	97	K
0011_1111	3F	63	?	0110_0010	62	98	l
0100_0000	40	64	'	0110_0011	63	99	L
0100_0001	41	65	.	0110_0100	64	100	m
0100_0010	42	66	:	0110_0101	65	101	M
0100_0011	43	67	;	0110_0110	66	102	n
0100_0100	44	68	"	0110_0111	67	103	N
0100_0101	45	69	'	0110_1000	68	104	o
0100_0110	46	70	`	0110_1001	69	105	O
0100_0111	47	71	\	0110_1010	6A	106	p
0100_1000	48	72	#	0110_1011	6B	107	P
0100_1001	49	73	$	0110_1100	6C	108	q
0100_1010	4A	74	@	0110_1101	6D	109	Q
0100_1011	4B	75	-	0110_1110	6E	110	r
0100_1100	4C	76	a	0110_1111	6F	111	R
0100_1101	4D	77	A	0111_0000	70	112	s
0100_1110	4E	78	b	0111_0001	71	113	S
0100_1111	4F	79	B	0111_0010	72	114	t
0101_0000	50	80	c	0111_0011	73	115	T
0101_0001	51	81	C	0111_0100	74	116	u
0101_0010	52	82	d	0111_0101	75	117	U
0101_0011	53	83	D	0111_0110	76	118	v
0101_0100	54	84	e	0111_0111	77	119	V
0101_1010	55	85	E	0111_1000	78	120	w
0101_0110	56	86	f	0111_1001	79	121	W
0101_0111	57	87	F	0111_1010	7A	122	x
0101_1000	58	88	g	0111_1011	7B	123	X
0101_1001	59	89	G	0111_1100	7C	124	y
0101_1010	5A	90	h	0111_1101	7D	125	Y
0101_1011	5B	91	H	0111_1110	7E	126	z
0101_1100	5C	92	i	0111_1111	7F	127	Z