Kodowania znaków

Komputer przechowuje w swojej pamięci tylko bity, które przyjmują wartość `0` lub `1`. Aby wyświetlać obrazy albo litery potrzebujemy *schematu kodowania*. Na przykład:

```
01100010 01101001 01110100 01110011
b        i        t        s
```

Ciąg `01100010` odpowiada tu znakowi `b`, ciąg `01101001` znakowi `i`, itd. W *kodowaniu* każdy znak jest reprezentowany przez ciąg znaków. Powyższe odpowiadają kodowaniu `ASCII`, używającemu jednego bajtu, czyli 8-bitowych ciągów, do kodowania znaków. Jeszcze jeden przykład:

```
01000001  A
01000010  B
01000011  C
01000100  D
01000101  E
01000110  F
```

Kod `ASCII` zawiera 95 drukowalnych symboli, takich jak małe i duże litery angielskiego alfabetu, cyfry, znaki interpunkcyjne, dolar, niawiasy, itp. Zawiera też 33 znaki niedrukowalne, takie jak tabulator, backspace, i znak null (zazwyczaj koniec pliku). Łącznie `ASCII` opisuje 128 znaków, do ich kodowania używa wszystkich możliwych kombinacji 7 bitów. Ponieważ komputery operują na 8 bitach dodatkowy 1 bit bywa używany do rozszerzania kodowania `ASCII` do 256 znaków. Przykład "Witaj w TCS" w `ASCII`:

```
01010111  01101001  01110100  01100001  01101010
          00100000  01110111  00100000
          01010100  01000011  01010011

57 69 74 61 6a 20 77 20 54 43 53
```

Znaki końca linii

Specjalne znaki kodu `ASCII` mają różne zastosowania. Zwrócimy teraz uwagę na dwa z nich:

(i) znak `CR` (od "carriage return", `\r`, `0x0D`, kod dziesiętny 13),
(ii) znak `LF` (od "line feed", `\n`, `0x0A`, kod dziesiętny 10).

Systemy operacyjne kodują w różny sposób koniec linii w plikach. `Microsoft Windows`, rodzina systemów `DOS`, i inne starsze systemy, używają dwóch znaków `CR+LF`. Geneza tych

dwu znaków występujących po sobie pochodzi od imitacji sytuacji złamania linii na maszynie do pisania, gdzie jednym ruchem (pionowym) musimy przewinąć kartkę do następnej linii, a drugim ruchem musimy cofnąć głowicę na początek linii. Systemy `Unix`, `Linux`, `OS X` ale także np. `Amiga` kodują koniec linii jednym znakiem `LF`. Systemy `Commodore` , `ZX Spectrum` i wczesne wersje `MAC OS` kodują koniec linii znakiem `CR`. Są systemy kodujące koniec linii jeszcze inaczej ale autorzy tego opracowania jeszcze nie mieli z nimi nigdy do czynienia.

Większość tekstowych protokołów internetowych (`HTML`, `SMTP`, `FTP`) używa sekwencji `CR+LF` jako koniec linii ale rekomenduje aplikacjom tolerowanie także kodowania jednym znakiem `LF`.

W szczególności przenosząc plik z systemu `Windows` do systemu `Linux` możemy mieć problemy. Ponieważ znak `CR` jest niedrukowalny różnicy nie zauważymy przy wyświetlaniu pliku w edytorze bądź na konsoli. Ale już skrypty mogą byc błędnie interpretowane. Znak `\r` na końcu linii uniemożliwia interpretację basha.

Poniżej lista kilku najważniejszych znaków specjalnych `ASCII`:

skrót znaku	escape code	nazwa	kod dziesiętny
`CR`	`\r`	Carriage Return	13
`LF`	`\n`	New line Feed	10
`BS`	`\b`	Backspace	8
`NUL`	`\0`	NULL	0

Wielobajtowe kodowania i Unicode

`ASCII` pokrywa zbiór angielskich znaków, a co jeśli chodzi o inne języki? Jak polski (`ąćęłńóśżź`), czy niemiecki (`äöüß`)? Można uzyć nadmiarowego 8-mego bitu i rozszerzyć kodowanie do tych znaków, ale w przypadku języków takich jak vhiński i japoński to dalej nie wystarcza.

Ponieważ jeden bajt to za mało, stworzono kodowania używające dwóch bajtów (16 bitów) mogące przechowywać 65536 róznych znaków. To dużo, bo dzięki temu takie kodowanie jak `BIG-5` zawiera tradycyjne chińskie znaki, a `GB18030` zawiera zarówno tradycyjny jak i uproszczony zbiór chińskich znaków.

Pomimo rozszerzenia zbioru znaków, żaden z powyższych standardów nie zawiera wszystkich możliwych znaków. Zatem tworzenie dokumentów wykorzystujących znaki z wielu języków w tych kodowaniach dalej jest utrudnione, albo niemożliwe. Jako rozwiązanie tego problemu stworzono standard *Unicode* zawierający zbiór 1114112 znaków i symboli, a więc jest w stanie pomieścić także prehistoryczne, jak i zapewne przyszłe zbiory znaków.

A więc ile bitów używa *Unicode* do przechowywania tych znaków? Ani jednego, ponieważ *Unicode* nie jest kodowaniem. Wszystkie znaki *Unicode* mieszczą się w 3 bajtach. Jednak

komputery wykorzystują raczej kody o długości będącej potęgi dwójki, więc znaki przechowywane jest to w 4 bajtach. Ponieważ nie zawsze potrzebujemy kodować chińskie znaki, nadmiarowe bajty niepotrzebnie zajmowały by pamięć, stąd powstało kilka kodowań tego zbioru znaków. `UTF-8`, `UTF-16` i `UTF-32` odpowiednio 8, 16 i 32-bitowe kodowanie. Ciekawą własnością `UTF-8` i `UTF-16` jest to, że są to kodowania zmiennej długości, mogące rozszerzać, w razie potrzeby, liczbę wykorzystywanych bitów. Przykład:

znak	kodowanie	bity
A	`UTF-8`	`01000001`
A	`UTF-16`	`00000000 01000001`
A	`UTF-32`	`00000000 00000000 00000000 01000001`
あ	`UTF-8`	`11100011 10000001 10000010`
あ	`UTF-16`	`00110000 01000010`
あ	`UTF-32`	`00000000 00000000 00110000 01000010`

Podsumowując, dowolny znak może być zakodowany na wiele różnych sposobów, a ciąg bitów interpretowany jako różne znaki, zależnie od używanego kodowania.

bity	kodowanie	znaki
`11000100 01000010`	Windows Latin 1	ÄB
`11000100 01000010`	Mac Roman	fB

znaki	kodowanie	bity
Føö	Windows Latin 1	`01000110 11111000 11110110`
Føö	Mac Roman	`01000110 10111111 10011010`
Føö	UTF-8	`01000110 11000011 10111000 11000011 10110110`

Narzędzia do przeglądania zawartości pliku (binarnej) i zmiany jego kodowania

Z powodu takich właśnie różnic i różnych kodowań uzywanych przez edytory i systemy operacyjne, potrzebne są narzędzia do ustawiania i zmiany kodowania plików.

Po pierwsze jednak, jak już ustaliliśmy, czasami nie widać w zwykłym edytorze tekstu wszystkich znaków zawartych w pliku. Potrzebujemy narzędzia przeglądania zawartości pliku w formie kodów znaków.

```
hexdump witaj.txt

0000000 6957 6174 206a 2077 4354 0a53 656a 7473
0000010 6d65 7420 7475 6a61 6a0a 7365 6574 206d
0000020 6174 0a6d 610a 6161 000a
0000029
```

`hexdump`, w powyższym domyślnym użyciu, drukuje kody kolejnych znaków z pliku w

systemie szesnatskowym, pogrupowane po 8 grup dwucyfrowych (16-bitowych) w linii. Ponadto każda linia zaczyna się od *przesunięcia* (*offsetu*) pierwszego znaku w linii względem początku pliku.

Uwaga: `hexdump` drukuje bajty w kolejności interpretowanej przez procesor `Intel x86` (konwencja Intela, czy też *little endian byte ordering*). Jest to dominująca obecnie interpretacja (dualna do niej nazywana jest *big endian*)

`hexdump -C witaj.txt`
albo `hd witaj.txt` (`hd` jest synonimem `hexdump -C`)

```
00000000  57 69 74 61 6a 20 77 20  54 43 53 0a 6a 65 73 74  |Witaj w TCS.jest|
00000010  65 6d 20 74 75 74 61 6a  0a 6a 65 73 74 65 6d 20  |em tutaj.jestem |
00000020  74 61 6d 0a 0a 61 61 61  0a                       |tam..aaa.|
00000029
```

Opcja `-C` drukuje także podgląd pliku ze znakami drukowalnymi `ASCII` i kropkami w miejsce niedrukowalnych.

Inne opcje `hexdump`:
`-n 13` —drukuje jedynie 13 pierwszych bajtów
`-s 15` —drukuje bajty startując od piętnastego

Dobrym testerem (zgadywaczem) kodowania znaków użytego w pliku jest komenda `file`:

`file witaj.txt`
`file -i witaj.txt`
`file -b witaj.txt` (różne opcje, które nieznacznie modyfikują tekst drukowany na wyjściu)

`iconv` to program zmieniający kodowanie znaków danego pliku na inne. Przykładowo, zmiana z `ISO-8859-1` na `UTF-8` wygląda następująco:
`iconv -f iso-8859-1 -t utf-8 <input.txt >output.txt`
Listę dostępnych kodowań można zobaczyć po wywołaniu:
`iconv --list`