3.

(Ne)podpora Unicode v PHP

Jazyk XML jako znakovou sadu používá Unicode. Současná verze PHP však standard Unicode nepodporuje. V této kapitole se proto nejprve seznámíme s problematikou znakových sad a kódování a pak se podíváme na to, jak lze obejít chybějící podporu Unicode.

3.1 Znakové sady, kódování a Unicode

Než se podíváme na možnosti PHP ohledně práce se znakovými sadami a kódováními, bude užitečné si vysvětlit pár základních pojmů.

3.1.1 Znaková sada

Jestliže chceme, aby počítač uměl pracovat s textem, musíme nadefinovat, jaké znaky se v textu mohou vyskytovat. A protože počítače vnitřně reprezentují veškeré informace pomocí čísel, musíme těmto znakům přiřadit číselné kódy, které je budou zastupovat. *Znaková sada* je právě taková množina znaků, kde má každý znak přiřazený číselný kód. Znaková sada nám vymezuje repertoár znaků, které je možné v textu používat.

Historický vývoj dal vzniknout desítkám a možná i stovkám různých znakových sad. Nejznámější z nich je asi znaková sada ASCII, která definovala 128 znaků, jež zahrnovaly písmena anglické abecedy, číslice, interpunkční znaménka a pár dalších speciálních znaků. Znaková sada ASCII tak byla vhodná pro zápis textů v angličtině, pro psaní kódů programů apod. Nešlo v ní však psát například české texty, protože chyběly znaky s diakritikou. Vzniklo tak mnoho znakových sad, které jsou rozšířením ASCII – definují celkem 256 znaků. Prvních 128 znaků je shodných s ASCII a druhých 128 může být použito právě pro národní znaky. Pro češtinu takových znakových sad existovalo několik, v dnešní době se používají zejména znakové sady ISO Latin 2 (ISO 8859-2) a Windows CP 1250.

Tabulka 3.1 ukazuje, že v různých znakových sadách mohou mít znaky různé kódy nebo nemusí být vůbec definovány.

Tabulka 3.1: Ukázka definice kódu znaku pro různé znakové sady

Znak	Kód v ASCII	Kód v ISO Latin 2	Kód ve Windows CP 1250
A	65	65	65
á	–	225	225
Ž	–	174	142

3.1.2 Kódování

Znaková sada definuje repertoár dostupných znaků a jejich číselných kódů. Abychom mohli text v nějaké znakové sadě uložit do souboru nebo přenést po síti, musíme jednotlivé kódy znaků převést na posloupnost bajtů, protože právě bajty jsou základní jednotkou, do které se ukládají informace v souborech nebo posílají po síti. Způsobu převodu číselného kódu znaku do posloupnosti bajtů se říká *kódování*. Protože historické znakové sady jako ASCII, ISO Latin 2 nebo Windows CP 1250 obsahovaly maximálně 256 znaků, šlo jako kódování použít obyčejnou identitu. Číselný kód znaku se rovnou zapsal jako jeden bajt, a bylo vystaráno.

3.1.3 Unicode

Problém znakových sad jako ISO Latin X a CP 125X byl v tom, že byly navrženy pro ukládání textů v omezené množině jazyků. Kdybychom měli například text, který míchá třeba češtinu s ruštinou (která používá azbuku), neměli bychom k dispozici žádnou znakovou sadu, která by zahrnovala jak české znaky s diakritikou, tak i azbuku. S postupující globalizací a rozšiřováním počítačů vůbec začal být tento stav neudržitelný. Logickým východiskem proto bylo vytvoření univerzální znakové sady, která by zahrnovala znaky všech běžně používaných jazyků. Při jejím použití by pak bylo možné v textu používat libovolný v ní definovaný znak a bez problému tak míchat texty i v zcela odlišných jazycích. Tato univerzální znaková sada vznikla na začátku 90. let minulého století a jmenuje se *Unicode*. Je navržena tak, aby byla schopná reprezentovat více jak jeden milión znaků, i když v dnešní době je definováno „jen“ 96 382 znaků.[1]

Protože byl jazyk XML navrhován jako univerzální a musí být schopný reprezentovat informace v libovolném jazyce, používá se v něm pro zápis textu právě znaková sada Unicode.

[1] Standard Unicode se neustále vyvíjí a jsou do něj podle potřeby přidávány nové znaky. Informace o počtu znaků se vztahuje k verzi Unicode 4.0.

Podobně je definován i jazyk HTML 4.0 – jako znakovou sadu používá Unicode, takže uvnitř dokumentu lze použít libovolný znak definovaný v rámci Unicode. Při studiu specifikací možná narazíte na to, že místo o Unicode hovoří o znakové sadě ISO 10646. Nenechte se tím zmást – jedná se o totožnou znakovou sadu, která definuje stejné znaky a přiřazuje jim stejné číselné kódy. Jediný rozdíl je v tom, že standardy Unicode a ISO 10646 jsou formálně vydávány různými organizacemi.

Na obrázku 3.1 se můžeme podívat, jak vypadají definice znaků a jim odpovídajích číselných kódů ve standardu. Z tabulky je například patrné, že znak „ě“ má kód 11B v šestnáctkové soustavě (283 v desítkové). Pro označení znaku s určitým číselným kódem se často používá syntaxe U+*«XXXX»*, kde *«XXXX»* je právě kód znaku vyjádřený v šestnáctkové soustavě. Písmeno „ě“ tak můžeme v této notaci zachytit jako U+011B.[2]

Jako u jakékoliv jiné znakové sady, i číselné kódy znaků Unicode musíme pro účely převodu do počítačové reprezentace převést na posloupnost bajtů. Vzhledem k velkému počtu znaků není už příliš vhodné používat kódování v podobě identity, kdy se kód znaku zapisuje přímo beze změn. Takové kódování existuje a jmenuje se UTF-32, ale prakticky se nepoužívá. Jeden znak zapíše do čtyř bajtů, ve kterých přímo reprezentuje číselný kód znaku. Jedná se tedy o velice neúspornou reprezentaci textu.

Existují proto úspornější kódování UTF-8 a UTF-16. S nimi se v praxi setkáme často, například proto, že se jedná o preferovaná kódování pro dokumenty XML.

3.1.3.1 UTF-8

Podívejme se nejprve podrobněji na kódování UTF-8, které se dnes v prostředí internetu nejčastěji používá pro kódování textu v Unicode. Zvláštností UTF-8 je to, že jeden znak se může zakódovat do proměnlivého počtu bajtů (od jednoho do čtyř). Navíc je UTF-8 navrženo tak, aby bylo zpětně kompatibilní s ASCII. Máme-li tedy v Unicode text obsahující jen znaky z ASCII a zapíšeme jej v kódování UTF-8, vypadá výsledný soubor stejně, jako by byl rovnou vytvořen v ASCII. Je to možné díky tomu, že prvních 128 znaků Unicode bylo převzato z ASCII a UTF-8 je přímo kóduje jako odpovídající jednobajtovou hodnotu.

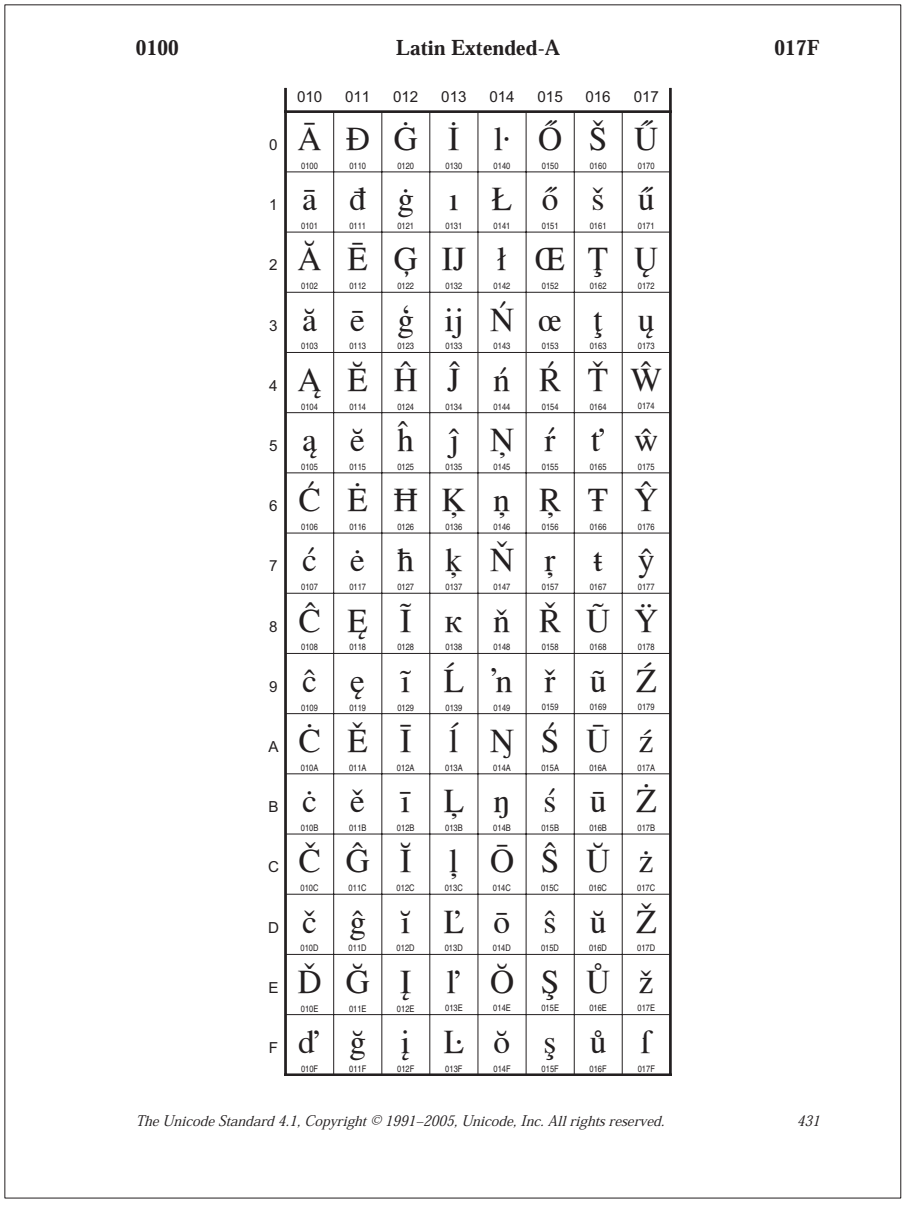
Obsahuje-li text znaky Unicode s kódy většími než 128, jsou tyto znaky reprezentovány jako několik bajtů, mezi které se po jednotlivých bitech rozdělí hodnota číselného kódu znaku. Jak probíhá převod kódu znaku na sekvenci bajtů v UTF-8, zachycuje tabulka 3.2.

Tabulka 3.2: Princip kódování v UTF-8

Rozsah kódů	Číselný kód Unicode	Výsledný počet bajtů	Výsledné zakódování do UTF-8
U+0000–U+007F	00000000 00000000 0xxxxxxx	1	0xxxxxxx
U+0080–U+07FF	00000000 00000yyy yyxxxxxx	2	110yyyyy 10xxxxxx
U+0800–U+FFFF	00000000 zzzzyyyy yyxxxxxx	3	1110zzzz 10yyyyyy 10xxxxxx
U+10000–U+10FFFF	000uuuuu zzzzyyyy yyxxxxxx	4	11110uuu 10uuzzzz 10yyyyyy 10xxxxxx

Jak lze tuto tabulku využít pro kódování textu do UTF-8, si ukážeme na jednoduchém příkladě. Předpokládejme, že chceme do UTF-8 uložit text „á —“ (malé písmeno „á“, me-

[2] Jazyk XML však používá vlastní notaci pro označení znaku s určitým číselným kódem. Písmeno „ě“ můžeme do dokumentu XML vložit mimo jiné pomocí zápisu `ě` nebo `ě`.



0100 **Latin Extended-A** **017F**

	010	011	012	013	014	015	016	017
0	Ā 0100	Đ 0110	Ġ 0120	İ 0130	l· 0140	Ő 0150	Š 0160	Ű 0170
1	ā 0101	đ 0111	ġ 0121	ı 0131	Ł 0141	ő 0151	š 0161	ű 0171
2	Ă 0102	Ē 0112	Ģ 0122	IJ 0132	ł 0142	Œ 0152	Ţ 0162	Ų 0172
3	ă 0103	ē 0113	ģ 0123	ij 0133	Ń 0143	œ 0153	ţ 0163	ų 0173
4	Ą 0104	Ĕ 0114	Ĥ 0124	Ĵ 0134	ń 0144	Ŕ 0154	Ť 0164	Ŵ 0174
5	ą 0105	ĕ 0115	ĥ 0125	ĵ 0135	Ņ 0145	ŕ 0155	ť 0165	ŵ 0175
6	Ć 0106	Ė 0116	Ħ 0126	Ķ 0136	ņ 0146	Ŗ 0156	Ŧ 0166	Ŷ 0176
7	ć 0107	ė 0117	ħ 0127	ķ 0137	Ň 0147	ŗ 0157	ŧ 0167	ŷ 0177
8	Ĉ 0108	Ę 0118	Ĩ 0128	ĸ 0138	ň 0148	Ř 0158	Ũ 0168	Ÿ 0178
9	ĉ 0109	ę 0119	ĩ 0129	Ĺ 0139	ʼn 0149	ř 0159	ũ 0169	Ź 0179
A	Ċ 010A	Ě 011A	Ī 012A	Í 013A	Ŋ 014A	Ś 015A	Ū 016A	ź 017A
B	ċ 010B	ě 011B	ī 012B	Ļ 013B	ŋ 014B	ś 015B	ū 016B	Ż 017B
C	Č 010C	Ĝ 011C	Ĭ 012C	ļ 013C	Ō 014C	Ŝ 015C	Ŭ 016C	ż 017C
D	č 010D	ĝ 011D	ĭ 012D	Ľ 013D	ō 014D	ŝ 015D	ŭ 016D	Ž 017D
E	Ď 010E	Ğ 011E	Į 012E	ľ 013E	Ŏ 014E	Ş 015E	Ů 016E	ž 017E
F	ď 010F	ğ 011F	į 012F	Ŀ 013F	ŏ 014F	ş 015F	ů 016F	ſ 017F

The Unicode Standard 4.1, Copyright © 1991–2005, Unicode, Inc. All rights reserved.

Reprodukce tabulky ze standardu Unicode byla do knihy zařazena s laskavým svolením Unicode, Inc.

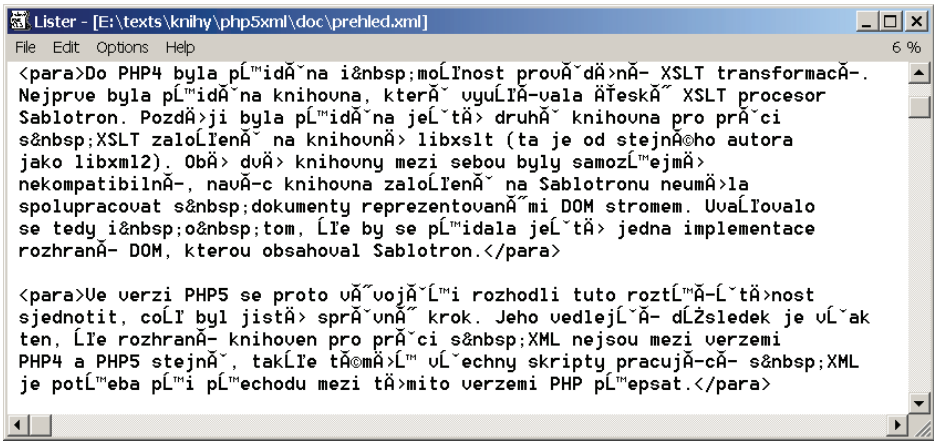
Obrázek 3.1: Ukázka definice znaků ze standardu Unicode

zeru a pomlčku). Nejprve musíme zjistit, jaké jsou kódy těchto znaků v Unicode. Zjistíme, že se jedná o znaky s kódy U+00E1, U+0020 a U+2014. Už z toho je patrné, že první znak se zakóduje do dvou bajtů, druhý do jednoho a třetí dokonce do tří bajtů. Nyní si stačí kódy znaků převést do dvojkové soustavy a podle tabulky provést zakódování do UTF-8:

3. (Ne)podpora Unicode v PHP

```
á (U+00E1) = 00000000 11100001 → 11000011 10100001 = C3 A1
  (U+0020) = 00000000 00100000 → 00100000 = 20
— (U+2014) = 00100000 00010100 → 11100010 10000000 10010100 = E2 80 94
```

Vidíme, že text „á —“ se do UTF-8 zakóduje jako posloupnost bajtů C3 A1 20 E2 80 94. Dekódování probíhá přesně opačným způsobem. Uvidíme-li v nějaké aplikaci podivný text, kde se budou vyskytovat znaky jako „Ă“, „Ä“ a „Ĺ“ následované dalším nesmyslným znakem, znamená to obvykle, že si soubor uložený v kódování UTF-8 prohlížíme pomocí programu, který neumí provést správné dekódování sekvencí bajtů UTF-8 zpět na Unicode znaky a tyto znaky následně zobrazit.



Obrázek 3.2: Vzhled textu kódovaného v UTF-8 zobrazeného v režimu bez podpory UTF-8

3.1.3.2 UTF-16

Kódování UTF-16 je podstatně jednodušší než UTF-8. Je-li kód znaku menší než 65536, zapíše se v kódování UTF-16 jako jedno 16bitové slovo (2 bajty). Je-li kód znaku vyšší, zapíše se jako dvě 16bitová slova, do kterých se rozdělí bity z původní hodnoty podobně jako u UTF-8.

Tabulka 3.3: Princip kódování v UTF-16

Rozsah kódů	Číselný kód Unicode	Počet bajtů	Výsledné zakódování do UTF-16
U+0000–U+FFFF	xxxxxxxx xxxxxxxx	2	xxxxxxxx xxxxxxxx
U+10000–U+10FFFF	000uuuuu xxxxxxxx xxxxxxxx	4	110110ww wwxxxxx 110111xx xxxxxxxx[a]

[a] Platí přitom, že wwww = uuuuu – 1.

Zajímavé je, že Unicode nedefinuje žádné znaky s kódy mezi U+D800–U+DFFF, takže nedojde k nejednoznačnosti při kódování znaků do dvou 16bitových slov. Případům, kdy se znak kóduje do dvou 16bitových slov, se říká náhradní páry (surrogate pairs).

Pojďme si nyní ukázat, jak by se do UTF-16 zakódoval text „á —“. Protože žádný z těchto tří znaků nemá kód vyšší než 65535, použije se vždy uložení do jednoho 16bitového slova.

```
á (U+00E1) → 00 E1
  (U+0020) → 00 20
— (U+2014) → 20 14
```

Text tedy bude uložen jako posloupnost bajtů 00 E1 00 20 20 14. Jenže ve skutečnosti to není tak jednoduché. Historicky existují dva způsoby ukládání 16bitových hodnot do paměti počítače. Jedné se říká velký endián (big endian) a v paměti jsou dva bajty 16bitové hodnoty uloženy v pořadí, které odpovídá jejich lidskému zápisu. Tento zápis je použit v příkladu.

Druhá metoda označovaná jako malý endián (little endian) ukládá nejdříve méně významný bajt 16bitové hodnoty a pak ten více významný. V tomto případě by byl náš text zakódován jako E1 00 20 00 14 20. Aby aplikace, které čtou text v UTF-16, mohly mezi těmito dvěma variantami rozlišit, vkládá se na začátek souborů uložených v UTF-16 speciální znak U+FEFF, tzv. BOM (Byte Order Mark). Podle toho, zda soubor začíná sekvencí bajtů FE FF, nebo FF FE, můžeme snadno poznat, o jakou z variant UTF-16 se jedná. Není-li BOM v textu přítomen, musí být pořadí bajtů patrné z názvu kódování – hovoříme pak o kódování UTF-16BE, resp. UTF-16LE.

Některé aplikace vkládají znak BOM i do textu uloženého v UTF-8, kde je to zcela zbytečné, protože UTF-8 dovoluje jen jedno pořadí bajtů. Vkládání tohoto znaku má však nepříjemné vedlejší účinky na interpret PHP (viz část 3.3.1).

Kódování UTF-16 je sice jednodušší než UTF-8, nicméně pro přenos a ukládání dat se v internetu příliš nepoužívá, protože není zpětně kompatibilní s ASCII. Navíc pro texty v angličtině a v jazycích, které nepoužívají příliš často znaky nad rámec ASCII, je zápis v UTF-16 skoro dvakrát delší než v UTF-8. Nicméně UTF-16 je zase snazší na zpracování, a proto se v mnoha prostředích používá pro interní reprezentaci řetězců. UTF-16 používají vnitřně například Windows, Java nebo .NET.

3.1.3.3 Zapomeňte na staré znakové sady

Jak jsme viděli, pro zápis textu v Unicode je nejlepší použít kódování UTF-8 nebo UTF-16. Existují však situace, kdy musíme používat nějaké starší softwarové nástroje, které tato kódování nepodporují. Ani pak však není nic ztraceno. To, čemu se v dobách před Unicode říkalo znaková sada – např. ISO Latin 2, Windows CP 1250 – lze dnes chápat jen jako speciální kódování znakové sady Unicode. Tato kódování se vyznačují tím, že jeden znak vždy ukládají do jednoho bajtu a jsou schopna reprezentovat jen vybraných 256 znaků z celého repertoáru Unicode. Převod kódu Unicode na hodnoty bajtů ve vybraném kódování je pak dán tabulkou, která jednotlivým znakům Unicode přiřazuje hodnoty.[3] Například pro kódování Windows CP 1250 bychom zjistili, že znak „á“ U+00E1 je reprezentován jako bajt s hodnotu E1. Znak dlouhé pomlčky „—“ U+2014 je reprezentován jako bajt s hodnotou 97.

Zapisujeme-li dokument XML v jiném kódování než UTF-16 nebo UTF-8, musíme použité kódování identifikovat v deklaraci XML pomocí jeho oficiálního názvu. Pro Windows CP 1250 má tento identifikátor tvar `windows-1250`, takže deklarace XML pak vypadá:

```
<?xml version="1.0" encoding="windows-1250"?>
```

[3] Překódovací tabulky pro všechna běžně používaná kódování lze získat na adrese *ftp://ftp.unicode.org/Public/MAPPINGS/*.

Zcela analogicky vypadá deklarace při použití kódování ISO 8859-2:

```
<?xml version="1.0" encoding="iso-8859-2"?>
```

Takový dokument XML jako znakovou sadu pořád používá Unicode, ale přímo do něj můžeme zapsat jen znaky, které jsou podporovány použitým kódováním. Jakékoliv další znaky Unicode lze vložit pomocí jejich číselného kódu s využitím syntaxe `&#«kód_znaku»;`. Kód je přitom možné zapsat buď v desítkové soustavě, nebo v šestnáctkové. K odlišení zápisů v šestnáctkové soustavě se používá písmeno ‚x' před kódem znaku. Například znak copyrightu © (U+00A9) můžeme do dokumentu XML vložit několika různými způsoby:

```
© - přímý zápis
&#xA9; - zápis pomocí kódu v šestnáctkové soustavě
&#XA9; - zápis pomocí kódu v šestnáctkové soustavě
&#x00A9; - zápis pomocí kódu v šestnáctkové soustavě
&#169; - zápis pomocí kódu v desítkové soustavě
```

Protože je dnes většina datových formátů (XML a HTML nevyjímaje) postavena výhradně na znakové sadě Unicode, je pro zjednodušení terminologie nejjednodušší považovat původní znakové sady jako ISO-8859-2 nebo Windows-1250 jen za kódování schopná reprezentovat podmnožinu Unicode.

Když už nyní víme, co je to znaková sada, kódování a Unicode, můžeme se konečně podívat na to, jak to vše souvisí s PHP. Zdála-li se vám úvodní exkurze do problematiky znakových sad a kódování příliš složitá, věřte, že i tak byla hodně zjednodušena pro snazší uchopení. [15]

3.2 PHP a práce s řetězci

Velký problém PHP5 je v tom, že i přes dávné sliby vývojářů[4] vůbec neobsahuje podporu Unicode. Zatímco jiná prostředí pro tvorbu aplikací jako Java nebo .NET mají přímo datový typ znak Unicode a řetězec takových znaků, PHP nic takového nenabízí. PHP totiž ve skutečnosti nepracuje s textovými řetězci složenými ze znaků, ale s posloupnostmi binárních dat složených z bajtů. Jak se to projevuje prakticky? Představme si, že do proměnné pozdrav uložíme řetězec „čau":

```
$pozdrav = "čau";
```

Jeden by si mohl myslet, že proměnná `$pozdrav` obsahuje textový řetězec skládající se ze tří znaků „č", „a" a „u". Ale ve skutečnosti se řetězec skládá z bajtů, které v souboru se skriptem tyto tři znaky reprezentovaly. Ty budou jiné, bude-li skript uložen v kódování windows-1250 nebo iso-8859-2. Jiné budou i v kódování UTF-8 a navíc to už nebudou bajty tři, ale dokonce čtyři, protože písmeno „č" je v UTF-8 reprezentováno dvěma bajty. Na první pohled si tohoto drobného nedostatku nevšimneme, protože třeba při vypisování hodnot to nevadí. Použijeme-li příkaz:

```
echo $pozdrav;
```

tak se do výstupu zkrátka zapíší bajty odpovídající řetězci „čau" v kódování skriptu, a pokud je kódování generované stránky stejné, prohlížeč zobrazí korektní text.

[4] *http://www.root.cz/clanky/zend-neni-jen-staroveky-iransky-jazyk/*

První problém, na který narazíme, bude použití klasických řetězcových funkcí. Dejme tomu, že chceme ve skriptu zjišťovat délku nějakého textového řetězce – třeba při kontrole, zda není moc dlouhý. Příklad 3.1 ukazuje jednoduchý skript, který vypíše délku tří řetězců uložených v poli.

Příklad 3.1: *Skript vypisující délku řetězců (skript je v kódování windows-1250) – `unicode/retezec-windows-1250.php`*

```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'>
<html lang="cs">
  <head>
    <meta http-equiv="content-type" content="text/html;charset=windows-1250">
    <title>Demonstrace špatné podpory Unicode &ndash; stránka v kódování ▶
windows-1250</title>
  </head>
  <body>
<?php

$text[] = "Text bez diakritiky";
$text[] = "Český text";
$text[] = "Text se speciálními znaky - (pomlčka), „uvozovky“";

foreach ($text as $retezec)
{
  echo "Řetězec: " . $retezec . "<br>\n";
  echo "Délka řetězce: " . strlen($retezec) . "<br>\n";
}

?>
  </body>
</html>
```

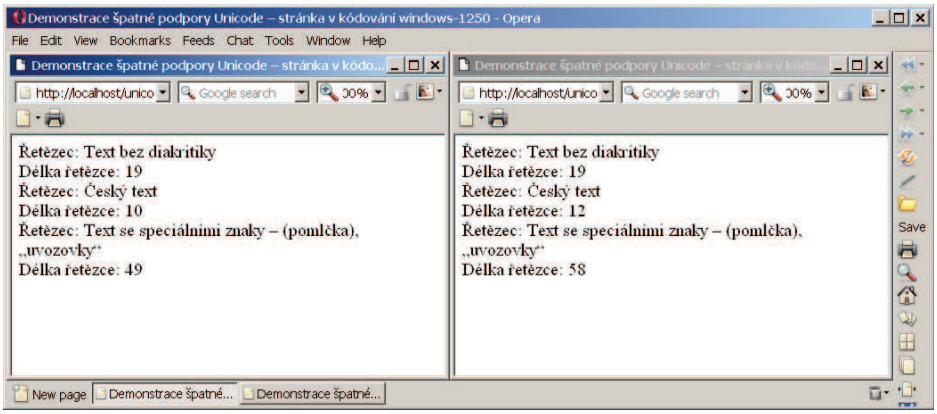
Nyní skript beze změny uložíme v kódování UTF-8 a jen změníme identifikaci kódování v meta tagu.

Příklad 3.2: *Skript vypisující délku řetězců (skript je v kódování utf-8) – `unicode/retezec-utf-8.php`*

```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'>
<html lang="cs">
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
   <title>Demonstrace špatné podpory Unicode &ndash; stránka v kódování utf-8</▶
title>
  </head>
  <body>
<?php

$text[] = "Text bez diakritiky";
$text[] = "Český text";
$text[] = "Text se speciálními znaky - (pomlčka), „uvozovky“";

foreach ($text as $retezec)
{
```

```
  echo "Řetězec: " . $retezec . "<br>\n";
  echo "Délka řetězce: " . strlen($retezec) . "<br>\n";
}

?>
  </body>
</html>
```

Člověk by očekával, že výstup z těchto skriptů bude identický. Jak je však vidět na obrázku 3.3, skript uložený v kódování UTF-8 nadhodnocuje skutečnou délku řetězců. Výsledek je špatně, protože funkce `strlen()` nepočítá počet znaků, ale bajtů uložených v řetězci. A při uložení skriptu v kódování UTF-8 jsou znaky s českou diakritikou reprezentovány jako dva bajty, a ne jako jeden bajt při použití windows-1250.



Obrázek 3.3: Funkce `strlen()` nefunguje společně s UTF-8

Nulová podpora Unicode ze strany PHP je velice nepříjemná, v praxi ji však lze obejít. Musíme se však postarat o to, abychom měli všechny texty v očekávaném kódování a aplikovali na ně funkce, které pro dané kódování fungují správně.

3.2.1 Ruční překódování

Jednou z možností, jak obejít limity PHP, je převést si řetězce zakódované v UTF-8 do nějakého jednobajtového kódování jako windows-1250. Jednomu znaku pak bude odpovídat jeden bajt a řetězcové funkce pak budou pracovat správně. Uvnitř skriptu můžeme pro překódování použít například knihovnu iconv, která nabízí funkci `iconv()`. Funkce převede zadaný řetězec ze zdrojového do cílového kódování.

```
echo strlen(iconv("utf-8", "windows-1250", "čau"));   // vypíše 3
```

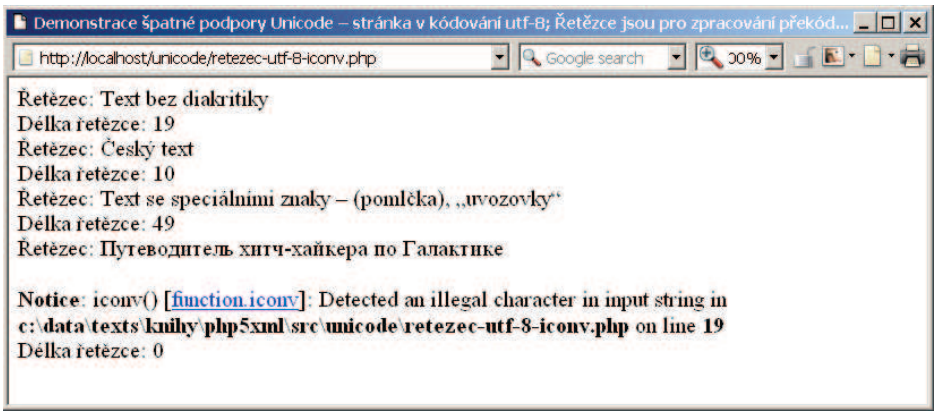
Tento přístup však může fungovat jen pro řetězce, ve kterých se vyskytují pouze znaky z cílového kódování. Kdyby vstupní řetězec obsahoval nějaké unicodové znaky, které nejde ve windows-1250 reprezentovat, nešlo by konverzi provést. Ilustruje to následující skript, který kromě textů v češtině obsahuje i text v ruštině.

Příklad 3.3: *Ruční překódování pomocí iconv – `unicode/retezec-utf-8-iconv.php`*

```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'>
<html lang="cs">
```

```
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
   <title>Demonstrace špatné podpory Unicode &ndash; stránka v kódování utf-8;
          Řetězce jsou pro zpracování překódovány do windows-1250</title>
  </head>
  <body>
<?php

$text[] = "Text bez diakritiky";
$text[] = "Český text";
$text[] = "Text se speciálními znaky – (pomlčka), „uvozovky“";
$text[] = "Путеводитель хитч-хайкера по Галактике";

foreach ($text as $retezec)
{
  echo "Řetězec: " . $retezec . "<br>\n";
  echo "Délka řetězce: " . strlen(iconv("utf-8", "windows-1250", $retezec))
       . "<br>\n";
}

?>
  </body>
</html>
```

Jak vidíme na obrázku 3.4, české texty se do kódování windows-1250 dají převést bez problémů. Toto kódování však neumí reprezentovat znaky azbuky, a proto funkce ohlásí chybu při konverzi textu v ruštině.



Obrázek 3.4: Při konverzi obecného řetězce do jednobajtového kódování může dojít k chybě

Tento přístup proto můžeme použít pouze v případě, kdy máme jistotu, že text bude obsahovat znaky reprezentovatelné ve zvoleném jednobajtovém kódování. Psát aplikace tímto způsobem je však poměrně krátkozraké, protože nikdy dopředu nevíme, kdy vyvstane požadavek na zápis textů v jiných jazycích.

Rozhodneme-li se psát aplikace takto omezeně, je pak otázkou, proč se trápit s konverzí mezi UTF-8 a dalšími kódováními. Nebylo by jednodušší psát skripty rovnou třeba v kódování windows-1250 nebo iso-8859-2 a nestarat se o žádné konverze? Možná bylo, ale ne

v případě, kdy pracujeme s dokumenty XML. Všechny knihovny pro práci s XML v PHP se totiž chovají tak, že textové řetězce obsažené v dokumentech XML mají v paměti reprezentovány v kódování UTF-8. Nezáleží přitom na tom, v jakém kódování byl původní dokument uložen. Tato konverze se provádí, protože kódování UTF-8 je schopné reprezentovat libovolný znak z Unicode.

Kdybychom tedy například měli jednoduchý dokument XML uložený v kódování windows-1250 a jednoduchý skript v témže kódování, který data z XML čte, dostali bychom na výstupu nesmyslné znaky, jak ukazuje obrázek 3.5.

Příklad 3.4: *Dokument XML v kódování windows-1250 – `data/konik.win.xml`*

```
<?xml version="1.0" encoding="windows-1250"?>
<dokument>
  <veta>Příliš žluťoučký kůň úpěl ďábelské ódy.</veta>
</dokument>
```

Příklad 3.5: *Skript pro čtení XML dokumentů uložený v kódování windows-1250 – `unicode/text-xml-windows-1250.php`*

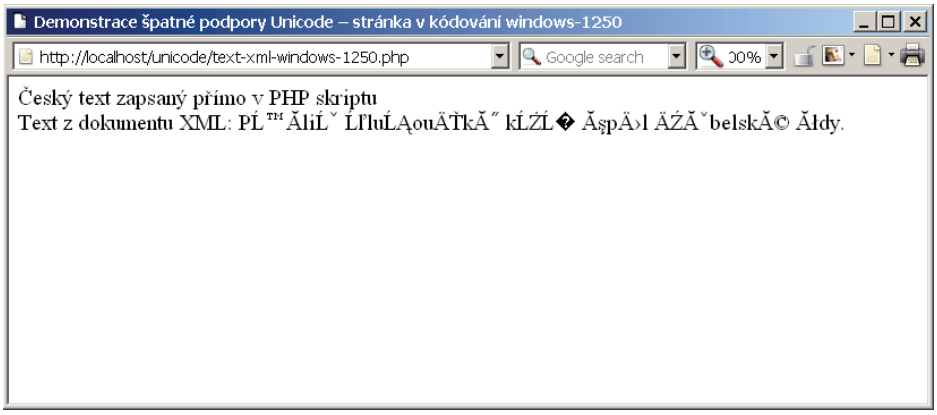
```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'>
<html lang="cs">
  <head>
    <meta http-equiv="content-type" content="text/html;charset=windows-1250">
    <title>Demonstrace špatné podpory Unicode &ndash;
           stránka v kódování windows-1250</title>
  </head>
  <body>
<?php

  echo "Český text zapsaný přímo v PHP skriptu<br>\n";

  $xml = simplexml_load_file("../data/konik.win.xml");
  echo "Text z dokumentu XML: ";
  echo $xml->veta;

?>
  </body>
</html>
```

Výstup skriptu je nesmyslný, protože dokument XML se automaticky během načítání pomocí knihovny SimpleXML překóduje do UTF-8. Když pak tento text bez úprav vypíšeme mezi ostatní výstupy skriptu, které jsou v kódování windows-1250, dostaneme samozřejmě nesmysly. V tomto případě je proto potřeba zase využít funkci `iconv()`, tentokráte pro převod řetězců z UTF-8 do kódování použitého skriptem.

Příklad 3.6: *Překódování řetězců z dokumentu XML do kódování použitého skriptem – `unicode/text-xml-prekodovani-windows-1250.php`*

```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'>
<html lang="cs">
  <head>
    <meta http-equiv="content-type" content="text/html;charset=windows-1250">
    <title>Demonstrace špatné podpory Unicode &ndash;
           stránka v kódování windows-1250</title>
```


3.2.1 Ruční překódování



Obrázek 3.5: Řetězce pocházející z XML jsou v PHP automaticky překódovány do UTF-8

```
  </head>
  <body>
<?php

  echo "Český text zapsaný přímo v PHP skriptu<br>\n";

  $xml = simplexml_load_file("../data/konik.win.xml");
  echo "Text z dokumentu XML: ";
  echo iconv("utf-8", "windows-1250", $xml->veta);

?>
  </body>
</html>
```

Vidíme, že překódování jedním i druhým směrem skript zbytečně komplikuje. V dnešní době je proto nejlepší, zvláště pracujeme-li hodně s XML, psát skripty výhradně v kódování UTF-8. V tomto kódování je pak nutné také generovat i výstup a načítat všechny vstupy – např. dokumenty XML a informace z databáze. To, že standardní řetězcové funkce PHP nepracují dobře s textem v UTF-8, musíme vyřešit jinak, nejlépe pomocí knihovny mbstring.

3.2.2 Knihovna mbstring

Pro práci s řetězci, které mohou mít jeden znak uložen ve více než jednom bajtu, je v PHP určena knihovna mbstring. Úvodní písmena „MB“ v názvu knihovny jsou přitom zkratkou z anglického „multi-byte“ a vyjadřují tak účel knihovny. Ta původně vznikla pro potřeby japonských uživatelů a jimi používaných znakových sad a kódování. Podporuje však i Unicode a jeho nejběžnější kódování jako UTF-8 a UTF-16.

Knihovna definuje mnoho nových funkcí, mezi kterými je mnoho ekvivalentů standardních řetězcových funkcí rozšířených o podporu různých kódování. Například místo funkce `strlen()` můžeme použít funkci `mb_strlen()` a určit, v jakém kódování je řetězec uložen. Např.:

```
echo mb_strlen("čau", "utf-8"); // vypíše 3, je-li skript uložen v UTF-8
```

Abychom mohli funkce knihovny použít, musíme si PHP buď s podporou této knihovny zkompilovat nebo načíst odpovídající modul v `php.ini`:

```
extension=php_mbstring.dll
```

Příklad 3.7: *Korektní přístup ke zpracování řetězců využívá knihovnu mbstring – `unicode/mbstring-utf-8.php`*

```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'>
<html lang="cs">
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
   <title>Demonstrace špatné podpory Unicode &ndash; stránka v kódování utf-8, ▶
explicitní využití knihovny mb_string</title>
  </head>
  <body>
<?php

$text[] = "Text bez diakritiky";
$text[] = "Český text";
$text[] = "Text se speciálními znaky – (pomlčka), „uvozovky“";

foreach ($text as $retezec)
{
  echo "Řetězec: " . $retezec . "<br>\n";
  echo "Délka řetězce: " . mb_strlen($retezec, "utf-8") . "<br>\n";
}

?>
  </body>
</html>
```

Představa, že budeme ve všech skriptech před název řetězcových funkcí doplňovat text `mb_` a přídavný parametr s informací o kódování, není moc lákavá. Naštěstí lze knihovnu mbstring provozovat v režimu, kdy předefinuje standardní řetězcové funkce. Pro využití této funkcionality stačí do `php.ini` přidat následující dvě řádky:

```
mbstring.internal_encoding = utf-8
mbstring.func_overload = 7
```

Používáme-li webový server Apache, můžeme toto nastavení definovat zvlášť pro jednotlivé adresáře pomocí souboru `.htaccess`:

```
php_value mbstring.internal_encoding "utf-8"
php_value mbstring.func_overload 7
```

První direktiva říká, v jakém kódování jsou interně ve skriptu uchovávány řetězce. V našem případě tak knihovna očekává, že skripty budou uloženy v kódování UTF-8 a rovněž data pocházející z dokumentů XML nám odpovídající knihovny pro práci s XML vrátí v UTF-8.

Druhá direktiva určuje, které standardní funkce PHP budou nahrazeny ekvivalentní funkcí knihovny mbstring. Hodnotu získáme jako součet konstant 1 (má se nahradit funkce `mail()`), 2 (mají se nahradit základní řetězcové funkce) a 4 (mají se nahradit funkce pro práci s re-

3.2.2 Knihovna mbstring

gulárními výrazy). Kompletní přehled funkcí, které se dají nahradit funkcemi knihovny mbstring, přináší tabulka 3.4.

Tabulka 3.4: Seznam funkcí, které mohou být nahrazeny funkcí knihovny mbstring

Původní funkce	Funkce mbstring	Hodnota pro `mbstring.func_overload`
`mail()`	`mb_send_mail()`	1
`strlen()`	`mb_strlen()`	2
`strpos()`	`mb_strpos()`	2
`strrpos()`	`mb_strrpos()`	2
`substr()`	`mb_substr()`	2
`ereg()`	`mb_ereg()`	4
`eregi()`	`mb_eregi()`	4
`ereg_replace()`	`mb_ereg_replace()`	4
`eregi_replace()`	`mb_eregi_replace()`	4
`split()`	`mb_split()`	4

Výše uvedené nastavení knihovny mbstring umožňuje poměrně transparentní práci s řetězci zapsanými v UTF-8. Konečně můžeme ve skriptu uloženém v kódování UTF-8 napsat:

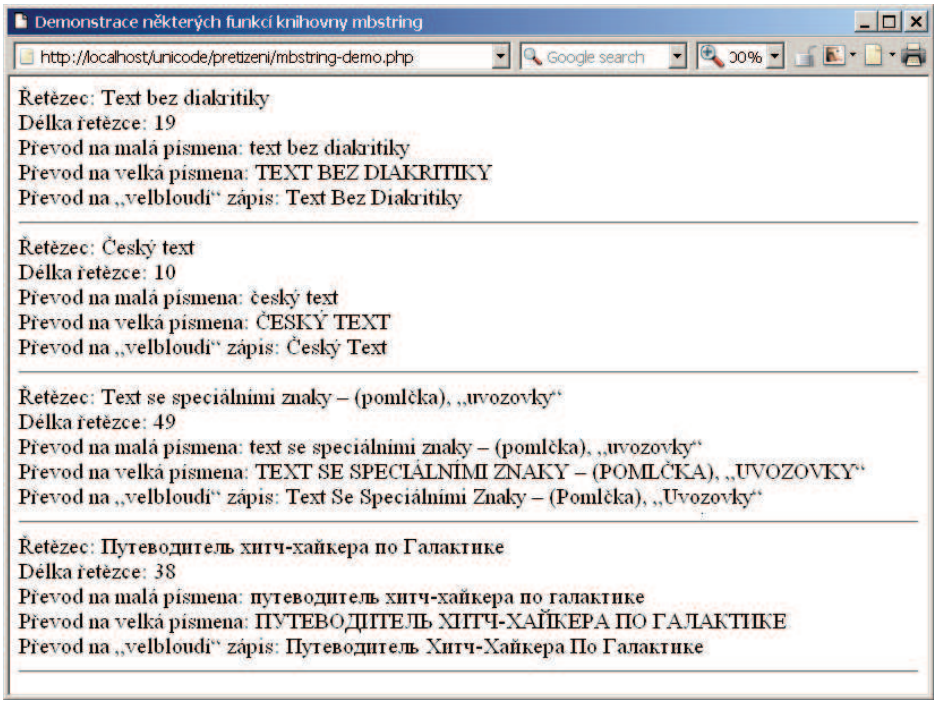
```
echo strlen("čau");
```

a skript nám vypíše hodnotu 3. Téměř všechny skripty v knize (s výjimkou několika málo skriptů, které demonstrovaly problémy s kódováním) jsou proto uloženy v kódování UTF-8 a předpokládají aktivaci knihovny mbstring.

Příklad 3.8: *Demonstrace některých funkcí knihovny mbstring – `unicode/pretizeni/mbstring-demo.php`*

```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'>
<html lang="cs">
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Demonstrace některých funkcí knihovny mbstring</title>
  </head>
  <body>
<?php

$text[] = "Text bez diakritiky";
$text[] = "Český text";
$text[] = "Text se speciálními znaky – (pomlčka), „uvozovky“";
$text[] = "Путеводитель хитч-хайкера по Галактике";

foreach ($text as $retezec)
{
  echo "Řetězec: " . $retezec . "<br>\n";
  echo "Délka řetězce: " . strlen($retezec) . "<br>\n";
  echo "Převod na malá písmena: " . strtolower($retezec) . "<br>\n";
```

```
  echo "Převod na velká písmena: " . strtoupper($retezec) . "<br>\n";
  echo "Převod na „velbloudí“ zápis: " . mb_convert_case($retezec, ▶
MB_CASE_TITLE) . "<br>\n";
  echo "<hr>\n";
}

?>
  </body>
</html>
```



Obrázek 3.6: Demonstrace knihovny mbstring

Knihovna mbstring nabízí mnoho dalších funkcí pro práci s řetězci. Umí provádět i transparentní překódování dat požadavku a výsledku skriptu. Podrobnější informace o těchto funkcích knihovny naleznete v PHP manuálu.[5]

3.3 Další problémy

Možná vám už nyní připadá, že se problematice znakových sad a kódování věnujeme zbytečně mnoho. Ale opak je pravdou. Mnoho věcí jsme záměrně zjednodušili a nyní se ještě podíváme na několik zvláštností, na které můžete při práci s PHP a Unicode narazit.

[5] *http://www.php.net/manual/en/ref.mbstring.php*

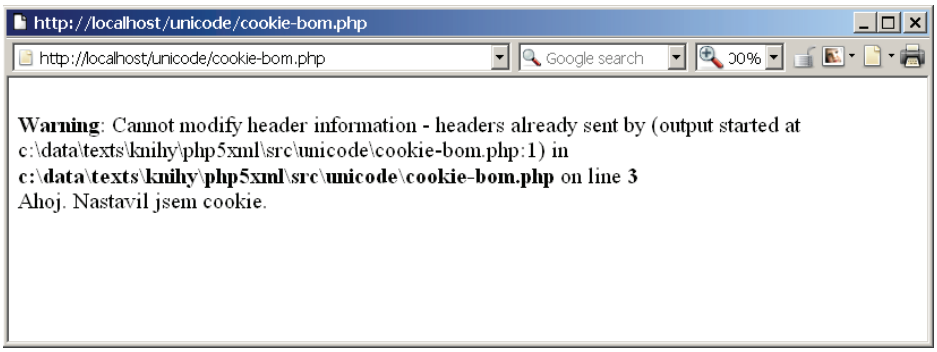
3.3.1 BOM a UTF-8

Představme si, že ukázkový skript z příkladu 3.9 napíšeme v nějakém jednoduchém editoru, třeba v Poznámkovém bloku, a uložíme v kódování UTF-8. Skript by měl nastavit cookie a poté do prohlížeče odeslat řádku textu, která se zobrazí.

Příklad 3.9: *Skript generující hlavičky HTTP uložený v UTF-8 s BOM – `unicode/cookie-bom.php`*

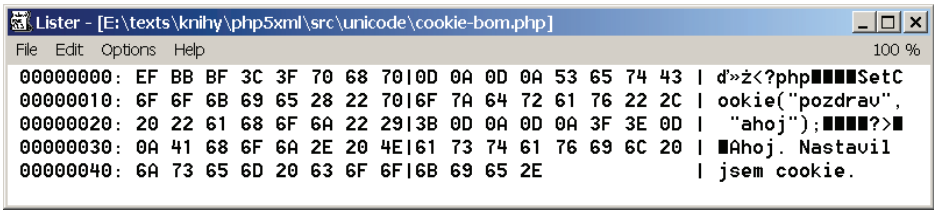
```
<?php

setcookie("pozdrav", "ahoj");

?>
Ahoj. Nastavil jsem cookie.
```

Při pokusu o načtení skriptu do prohlížeče však získáme nepříjemné chybové hlášení (viz obrázek 3.7). Takové hlášení se objevuje v případech, kdy skript generuje nějaké hlavičky HTTP – pomocí funkcí `header()` nebo `setcookie()` – a před těmito hlavičkami je vygenerován nějaký výstup. Hlavičky HTTP pak již nejde odeslat, protože musí předcházet samotnému tělu odpovědi.



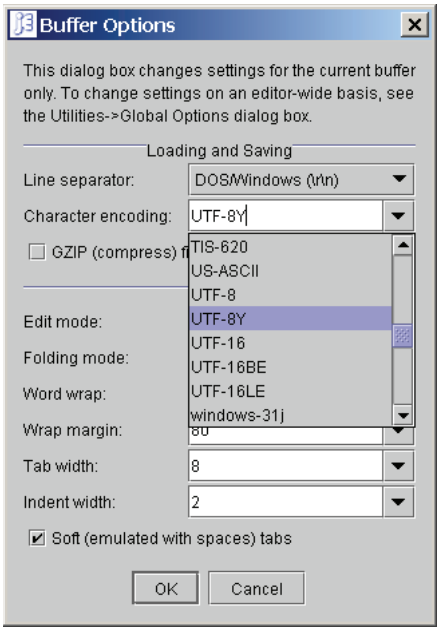
Obrázek 3.7: BOM na začátku souboru znemožní odeslání hlaviček HTTP

Jak to, že se však odeslala nějaká data na výstup skriptu, když skript začíná uvozením PHP kódu `<?php`, za kterým se hned volá funkce `setcookie()`? Podívejme se na soubor se skriptem jako na binární soubor. Vidíme, že první tři bajty souboru obsahují hodnoty EF, BB a BF. Kdybychom tuto sekvenci bajtů dekódovali algoritmem UTF-8, zjistili bychom, že reprezentují znak U+FEFF neboli tzv. BOM. Jak jsme si již dříve řekli, tento znak má význam při použití kódování UTF-16. Jeho použití společně s UTF-8 je zcela zbytečné, protože zde existuje jen jedna možnost jak skládat bajty za sebe. Některé aplikace (například již zmíněný Poznámkový blok) však tento znak ukládají i na začátek souborů kódovaných v UTF-8 jako signaturu signalizující, že se jedná o soubor v kódování UTF-8. Z pohledu PHP jsou tak před znakem „<“ uvozujícím kód skriptu další tři znaky, které se zkrátka pošlou na výstup. Bohužel se tam pošlou ještě před odesíláním hlaviček HTTP.



Obrázek 3.8: Jak objevit BOM na začátku souboru

Řešení problému s BOM jsou dvě. Prvním řešením je neukládat tento znak na začátek skriptu. Musíme pak skripty vytvářet v editoru, který BOM společně s UTF-8 nepoužívá nebo umí jeho generování vypnout. Většina lepších editorů umí toto nastavení provést. Například populární editor jEdit[6] standardně BOM neukládá, a pokud jej chceme do souboru uložit, musíme použít kódování s názvem `UTF-8Y`.



Obrázek 3.9: Výběr kódování v editoru jEdit

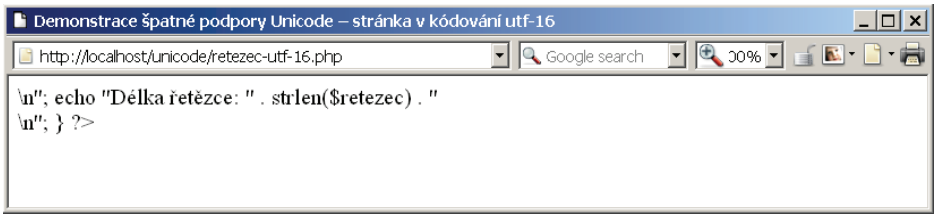
Druhou možností je použití bufferovaného výstupu, kdy se výstup skriptu před odesláním klientovi ukládá do vyrovnávací paměti. V této paměti pak může interpret PHP předřadit hlavičky HTTP před tělo odpovědi, i když je tělo odpovědi již částečně nebo zcela vygenerováno. Pro použití bufferovaného výstupu je potřeba přidat do konfiguračního souboru `php.ini` následující direktivu:

```
output_buffering = On
```

[6] *http://www.jedit.org*

3.3.2 PHP a UTF-16

Vzhledem k tomu, že PHP nijak speciálně nepodporuje Unicode, mohou být skripty zapsány pouze v kódování, které je zpětně kompatibilní s ASCII. V opačném případě interpret nerozpozná sekvenci znaků `<?php`, resp. `<?` a nezahájí provádění příkazů PHP. Tím pádem nejde psát skripty v kódování UTF-16, protože mezi ASCII znaky jsou v tomto kódování nulové bajty, které oddělují jednotlivé znaky sekvence uvozující PHP kód. Interpret PHP příkazy v tomto skriptu neprovede a dostaneme nesmyslný výstup (v podstatě zdrojový kód skriptu).



Obrázek 3.10: Skripty uložené v kódování UTF-16 se neinterpretují

3.3.3 Unicode a porovnávání řetězců

Použití znakové sady Unicode v sobě skrývá ještě jednu záludnost, se kterou by se robustní a kvalitně napsaná aplikace měla korektně vypořádat. Zápis některých znaků je možné v Unicode provést několika způsoby. Týká se to například znaků s diakritikou. Písmeno „á“ můžeme zapsat dvěma různými způsoby:

á (U+00E1) ➔ á
a (U+0061) + ´ (U+0301) ➔ á

Vidíme, že znak lze zapsat buď přímo, anebo jako kombinaci dvou znaků – písmene „a“ a samostatného akcentu čárky nad písmenem, který se s předchozím znakem zkombinuje dohromady.

Nepříjemným důsledkem dvojí možnosti zápisu je obtížné porovnávání řetězců. Když budeme například hledat nějakou hodnotu v databázi na základě dotazu uživatele zadaného do webového formuláře, musíme mít jistotu, že se používá stejný způsob zápisu znaků s akcenty, jinak nemusíme hledaný text najít.

Tento problém jde snadno obejít tím, že texty před jejich dalším zpracováním znormalizujeme. Můžeme si vybrat, zda za normalizovanou podobu budeme považovat dekomponovaný text obsahující samostatné akcenty, nebo naopak komponovaný text, kde jsou rovnou zapsány znaky s akcenty. Standard Unicode definuje několik způsobů normalizace řetězců[7]. Implementace těchto algoritmů existují i pro PHP[8].

Můžeme například používat třídu `I18N_UnicodeNormalizer`, která je součástí PEAR. Pro její instalaci stačí spustit následující příkaz.

```
$ pear install I18N_UnicodeNormalizer-1.0.0
downloading I18N_UnicodeNormalizer-1.0.0.tgz ...
```

[7] *http://www.unicode.org/reports/tr15/*
[8] *http://pear.php.net/package/I18N_UnicodeNormalizer*

```
Starting to download I18N_UnicodeNormalizer-1.0.0.tgz (2,154,776 bytes)
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.............done: 2,154,776 bytes
install ok: channel://pear.php.net/I18N_UnicodeNormalizer-1.0.0
```

Třída `I18N_UnicodeNormalizer` obsahuje několik metod, které dokáží provádět několik různých způsobů normalizace pro řetězce uložené v kódování UTF-8. Například metoda `toNFC` převede řetězec do komponované podoby a metoda `toNFD` zase do dekomponované:

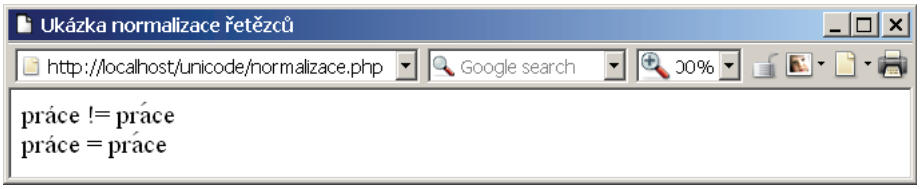
```
echo I18N_UnicodeNormalizer::toNFC("a´"); // vypíše jeden znak "á"
echo I18N_UnicodeNormalizer::toNFD("á"); // vypíše dva znaky "a´"
```

Aplikujeme-li tuto metodu před porovnáváním na oba řetězce, provede se jejich porovnání bez ohledu na to, jestli byly v zápisu řetězců použity odlišné způsoby zápisu znaků s akcenty. Praktické použití demonstruje následující příklad.

Příklad 3.10: *Porovnávání normalizovaných řetězců – unicode/normalizace.php*

```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'>
<html lang="cs">
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Ukázka normalizace řetězců</title>
  </head>
  <body>
<?php

$text1 = "práce";      // řetězec v komponované podobě
$text2 = "pra´ce";     // řetězec v dekomponované podobě

// "hloupé" porovnání řetězců
echo $text1 . ($text1 == $text2 ? " = " : " != ") . $text2 . "<br>\n";

// načtení knihovny pro provádění normalizace řetězců
require_once "I18N/UnicodeNormalizer.php";

// "chytré" porovnání normalizovaných řetězců
echo $text1 . (I18N_UnicodeNormalizer::toNFC($text1) == ▶
I18N_UnicodeNormalizer::toNFC($text2) ? " = " : " != ") . $text2 . "<br>\n";

?>
  </body>
</html>
```


3.3.3 Unicode a porovnávání řetězců



Obrázek 3.11: Ukázka porovnání řetězců před a po provedení normalizace

Nově připravovaná verze PHP 6 by již měla mít v sobě podporu Unicode integrovánu. Tak doufejme, že se této verze dočkáme v dohledné době a že podpora Unicode bude začleněna tak, aby nedošlo k porušení zpětné kompatability a nová verze PHP se tak začala brzy používat na web serverech.