

Unicode

Reuven M. Lerner

(Reprinted from Linux Magazine)

Unicode is necessary for international web development but poses a few pitfalls.

Growing up in the northeastern United States, I never had to use a language other than English. I read in English, spoke in English, wrote in English and conducted business in English. This was also true of the engineers who created ASCII back in 1968, who made sure the 128 ASCII characters would suffice for English-language documents. So long as you stuck with the standard set of ASCII characters, you were guaranteed the ability to move files from one computer to another without having to worry about them getting garbled.

ASCII was fine in its day, but people who spoke French, Spanish and other Western European languages quickly discovered that it was insufficient for their needs. After all, people who write in these languages on a computer want to use the correct accent marks. So over the course of time, the 7-bit ASCII code became the 8-bit extended ASCII code, including a number of special letters and symbols necessary for displaying Western European text.

But because extended ASCII was never declared a standard, a number of different, incompatible extensions to the base ASCII code became widespread. Windows had its own extensions, as did the Macintosh and NeXTSTEP operating systems. So although you could write a document in French using Windows, you would need to translate it when moving it to the Macintosh. Otherwise, the bytes would be interpreted on the receiving machine incorrectly, turning your otherwise superb French screenplay into something more akin to French toast.

International standards finally prevailed, at least somewhat, with a standard known formally as ISO-8859-1 and informally as Latin-1. Computer manufacturers could then exchange Western European documents without having to worry about things becoming garbled. Of course, this meant we were using all eight bits of each character's byte, doubling the number of available characters from 128 to 256.

However, this didn't solve all of the problems. For example, Hebrew speakers have their own standard, ISO-8859-8, which is identical to Latin-1 for characters 0-127 and quite different from 128-256. A document written in Hebrew but displayed on a computer using Latin-1 will look like a letter substitution puzzle using letters from the wrong alphabet.

Practically speaking, this means you cannot write a document that contains English, Hebrew and French using the ISO-8859 series of standards. And indeed, this makes sense given the fact that we have only 256 characters to play with in a single 8-bit byte. But it raises some serious questions and issues for those of us who work with more than two languages. Things get especially hairy if you want to display a page in English, French, Hebrew and Chinese. After all, there are tens of thousands of ideographs in Chinese, not to mention Japanese and other languages.

Enter Unicode, the ASCII table for the next century. Like ASCII, Unicode assigns a number to each letter, number and symbol. Unlike ASCII, Unicode contains enough space for every written symbol ever created by humans. This means that a Unicode document can contain any number of characters from any number of languages, without having to worry about clashes between them. Unicode also handles a number of issues that ASCII never dreamed about, including combining characters (for accents and other diacritical marks) and directional issues (for languages that do not read from left to right).

Unicode has been around for about a decade, but it is only now becoming popular and supported for web applications. This month, we take a look at Unicode as it affects web developers. What should

Unicode

Reuven M. Lerner

(Reprinted from Linux Magazine)

you consider? What do you need to worry about? And, how can you get around the problems associated with Unicode?

Introduction to Unicode

Unicode, like ASCII, assigns a unique number to each letter, number, symbol and control character. As indicated above, though, Unicode extends through each of the symbols and character sets ever created. So using Unicode, you can create a document that uses English, Russian, Japanese and Arabic, in which each character is clearly distinct from the others.

How do we turn these unique numbers—known as code points in the Unicode universe—into bits and bytes? The encoding for ASCII is very straightforward; with only 127 characters (or 256, if you include the various extensions), each ASCII character will fit into a single byte. And indeed, C programmers know that the char data type is an 8-bit integer.

The most obvious solution is to assign a fixed multibyte encoding for our Unicode characters. And indeed, UCS-2 is such an encoding, using two bytes to describe all of the basic 65,536 Unicode characters. (There are some extended characters that require additional bytes, but we won't go into that.) UCS-2 assigns a single 2-byte code to each of these characters. Documents are thus equally long regardless of the language in which they are written, and programs can easily calculate the number of bytes they need by doubling the number of characters. Microsoft's modern operating systems use UCS-2, as you might have noticed if you exchange any documents with users of those systems.

But there is a basic problem with UCS-2, namely its incompatibility with ASCII. If you have 100,000 documents written in ASCII, you will have to translate them into UCS-2 in order to read them accurately. Given that most modern programs work with ASCII, this lack of backward compatibility is quite a problem.

Enter UTF-8, which is a variable-length Unicode encoding. Just as Roman and Arabic numerals represent the same numbers differently, UTF-8 and UCS-2 are simply different encodings for the same underlying Unicode character set. But whereas every UCS-2 character requires two bytes, a UTF-8 character might require anywhere from one to four bytes. One-byte UTF-8 characters are the same as in ASCII, which means that a legal ASCII document is also a legal UTF-8 document. However, Latin-1 and other 8-bit character sets are incompatible with UTF-8; existing Latin-1 documents will not only need to be transformed but could potentially double in size.

UTF-8 is the preferred encoding on UNIX and Linux systems, as well as in most of the standards and open-source software that I tend to use. Perl, Python, Tcl and Java all encode strings in UTF-8. PostgreSQL has supported UTF-8 for years, and Unicode support has apparently been added to MySQL 4.1, which will be released in alpha in the coming months.

Adding Unicode support to an existing system is a Herculean task for which the various developers should be given great praise. Not only do developers need to add support for multibyte characters, but databases and languages also need to support regular expressions and sorting operators, neither of which is easy to do.

Unicode and HTTP

Now that we have gotten the basics out of the way, let's consider how Unicode documents are transferred across the Web. The basic problem is this: when your browser receives a document, how does it know if it should interpret the bytes as Latin-1, Big-5 Chinese or UTF-8?

Unicode

Reuven M. Lerner

(Reprinted from Linux Magazine)

The answer lies in the Content-type HTTP header. Every time an HTTP server sends a document to a browser, it identifies the type of content it is sending using a MIME-style designation, such as text/html, image/png or application/msword. If you receive a JPEG image (image/jpeg), there is only one way to represent the image. But if you receive an HTML document (text/html), the Content-type header must indicate the character set and/or encoding that is being used. We do this by adding a charset= designation to the end of the header, separating the type from the charset. For example:

```
Content-type: text/html; charset=utf-8
```

Purists rightly say that UTF-8 is an encoding and not a character set. Unfortunately, it's too late to do anything about this. This is similar to the fact that the word "referrer" is misspelled in the HTTP specification as "referer"; everyone knows that it's wrong but is afraid to break existing software.

If no Content-type is specified, it is assumed to be Latin-1. Moreover, if no Content-type is specified, individual documents can set (or override) the value within a metatag. Metatags cannot override an explicit setting of the character set, however.

As you begin to work with different encodings, you will undoubtedly discover an HTTP server that has not been configured correctly and that is announcing the wrong character set in the Content-type header. An easy way to check this is to use Perl's LWP (library for web programming), which includes a number of useful command-line programs for web developers, for example:

```
$ HEAD http://yad2yad.huji.ac.il/
```

Typing the above on my Linux box returns the HTTP response headers from the named site:

```
200 OK
Cache-Control: max-age=0
Connection: close
Date: Tue, 10 Dec 2002 08:38:37 GMT
Server: AOLserver/3.3.1+ad13
Content-Type: text/html; charset=utf-8
```

As you can see, the Content-type header is declaring the document to be in UTF-8.

Mozilla and other modern browsers allow the user to override the explicitly stated encoding. Although this should not normally be necessary for end users, I often find this functionality to be useful when developing a site.

Unicode and HTML

Although it's nice to know we can transfer UTF-8 documents via HTTP, we first need some UTF-8 documents to send. Given that ASCII documents are all UTF-8 documents as well, it's easy to create valid UTF-8 documents, so long as they contain only ASCII characters. But what happens if you want to create HTML pages that contain Hebrew or Greek? Then things start to get interesting and difficult.

There are basically two ways to include Unicode characters in an HTML document. The first is to type the characters themselves using an editor that can work with UTF-8. For example, GNU Emacs allows me to enter text using a variety of keyboard options and then save my document in the encoding of my choice, including UTF-8. If I try to save a Chinese document in the Latin-1 encoding, Emacs will refuse to comply, warning me that the document contains characters that do not exist in

Unicode

Reuven M. Lerner

(Reprinted from Linux Magazine)

Latin-1. Unfortunately, for people like me who want to use Hebrew, Emacs doesn't yet handle right-to-left input.

A better option, and one which is increasingly impressive all of the time, is Yudit, an open-source UTF-8-compliant editor that handles many different languages and directions. It can take a while to learn to use Yudit, but it does work. Yudit, like Emacs, allows you to enter any character you want, even if your operating system or keyboard does not directly support all of the desired languages.

Both Emacs and Yudit are good options if you are working on Linux, if you are willing to tinker a bit, and if you don't mind writing your HTML by hand. But nearly all of the graphic designers I know work on other platforms, and getting them to work with HTML editors that use UTF-8 has been rather difficult.

Luckily, Mozilla comes with not only a web browser but a full-fledged HTML editor as well. As you might expect, Mozilla's composer module is a bit rough around the edges but handles most tasks just fine.

Another option is to use HTML entities. The best-known entities are <, > and & which make it possible to insert the <, > and & symbols into an HTML document without having to worry that they will be interpreted as tags.

Modern browsers not only understand entities such as © (the copyright symbol) but also include the full list of Unicode characters. Thus, you can refer to Unicode characters by inserting &#XXXX; in your document, entering the character's decimal code instead of the XXXX. For example, the following HTML document displays my name in Hebrew, using Unicode entities:

```
<html>
  <head><title>Reuven's name</title></head>
  <body><p>&#1512;&#1488;&#1493;&#1489;&#1503;</p>
  </body>
</html>
```

Creating the above document does not require a Unicode-compliant editor, and it will render fine in any modern browser, regardless of the Content-type that was declared in the HTTP response headers. However, editing a file that uses entities in this way is tedious and difficult at best. Unfortunately, the save-as-HTML feature in the international editions of Microsoft Word uses this extensively, which makes it easy for Word users to create Unicode-compliant documents but difficult for people to edit them later.

Pitfalls

As I indicated earlier, Unicode is a complex standard, and it has taken some time for different languages and technologies to support it. For example, Perl 5.6.x used Unicode internally, but input and output operations couldn't easily use it, which made such support basically useless. Perl 5.8 by contrast has excellent Unicode support, allowing developers to write regular expressions that depend on Unicode properties.

There are still some problems, however. A major problem that developers have to deal with is the issue of input encoding vs. storage encoding, such as when your terminal might use Latin-1 but the back end might use UTF-8. This sort of arrangement means you can continue to use your old (non-

Unicode

Reuven M. Lerner

(Reprinted from Linux Magazine)

Unicode) terminal program and fonts but connect to and use your Unicode-compliant back-end program.

Various implementations also have some holes, which might not be obvious when you first start to work on a project. For example, I recently worked on a J2EE project that used PostgreSQL on its back end and stored all of the characters in Unicode. Everything was fine until we decided to compare the user's input string with text in the database in a case-insensitive fashion. Unfortunately, the PostgreSQL function we used doesn't handle case insensitivity correctly for Unicode strings. We found a workaround in the end, but it was both embarrassing and frustrating to encounter this.

Collating, or sorting, is also a difficult issue—one that has bitten me on a number of occasions. Unicode defines a character set, but it does not indicate in which order the characters in that set should be sorted. Until recently, for example, “ch” was sorted as its own separate letter in Spanish-speaking countries; this was not true for speakers of English, German and French. The sort order thus depends not only on the character set, but on the locale in which the character set is being applied. You may need to experiment with the LANG and LC_ALL environment variables (among others) to get things to work the way you expect.

Conclusion

Unicode is clearly the way of the future; most operating systems now support it to a certain degree, and it is becoming an entrenched standard in the computer world. Unfortunately, Unicode requires unlearning the old practice of equating characters and bytes and handling a great deal of new complexities and problems.

If you only need to use a single language on your web site, then consider yourself lucky. But if you want to use even a single non-ASCII character, you will soon find yourself swimming in the world of Unicode. It's worth learning about this technology sooner rather than later, given that it is slowly but surely making its way into nearly every open-source system and standard.