

# Unicode Transformation Formats

## UTF-8 and Co.

The ISO 10646 Universal Character Set (UCS, Unicode) is a coded character set with more than 40'000 defined elements. It is expected that this cardinality will grow to more than 100'000 soon, through additional definitions for characters that do not yet have a coding, so that all the world's characters will be represented in Unicode. But how can you represent more than  $2^8 = 256$  characters with 8bit bytes? This chapter explains and discusses the concepts of coded character sets versus their encoding schemes as well as the various Unicode representation schemes along with their implementation level on Unix: most prominently UTF-8 beside its precursors EUC and UTF-1 and its alternatives UCS-4, UTF-16, UTF-7,5, UTF-7, SCSU, HTML, and JAVA.

Another fine web page describing these subjects is Jukka Korpela's tutorial on character code issues.

### CCS + CES

#### What is a Coded Character Set?

An **abstract character** is a unit of textual information like the U+0041 LATIN CAPITAL LETTER A ('A'). Exactly how the textual information is broken into units and which units are identical is in many cases open to scientific debate and standardization.

A **coded character set** (CCS) is a well-defined mapping from integer numbers (code space, code points, code positions, code values, character numbers) to abstract characters.

A small example to play with the terminology: Let  $ABC := \{(65,'A'),(66,'B'),(67,'C')\}$ . ABC would be a CCS by this definition because it is a mapping (table) in set notation. Incidentally, ABC is a very small subset of the ASCII code. The character value of 65 is  $ABC(65)='A'$ . The code value of 'A' is 65. In ABC the code points 65, 66, and 67 make up the "code space"  $\{65,66,67\}$  which is completely occupied with character definitions. Within that short range there is no more free code space because none of the integer numbers in  $\{65,66,67\}$  could be mapped to yet another character seeking asylum without deleting a definition from ABC.

To avoid confusion, the corresponding simple set of characters such as ABC's character range  $\{'A','B','C'\}$  is not called a character set but a **character repertoire**.

The UCS is supposed to contain all other coded character sets like ISO-8859-2 as **repertoire subsets** (also encodes all of their characters) but only US-ASCII and ISO-8859-1 and MES / SECS are real **code subsets** with identical character numbers.

The downgrading text translation to a smaller repertoire may require transliteration.

#### Why do People like 8bit Charsets?

A CCS like US-ASCII or ISO-8859-1 with 256 or less characters and no integer value above 255 can easily serve as a **single-byte 8bit charset** where each octet of 8 bits (byte) is taken as a binary number to look up the one coded character it represents: 01000001 -> 65 -> 'A'.

The notion of an **8bit byte** is deeply ingrained into today's computer systems and their communication protocols. Files and memory sizes are measured and addressed in bytes, strings are represented in `char[]` arrays, the `read()` and `write()` system calls and most network protocols such as TCP transmit 8bit bytes.

You will probably waste your energy and create lots of incompatibilities if you try to change the **byte size** to 16, 20, 32 or 64 bits even though today's microprocessors could handle such quantities.

# Unicode Transformation Formats

## UTF-8 and Co.

Besides the incompatibilities there is also the argument that it is wasteful to have one character occupy 16 or 32 bits instead of 8 bits because that would double or quadruple file sizes and memory images.

### How do Multibyte Charsets Work?

Any rich CCS with more than 256 characters (as seen in China, Japan and Korea) needs thus a more or less complex **character encoding scheme** (CES, encoding, multibyte encoding, transformation format) so that a byte sequence (octet stream, multibyte string) can represent a sequence of larger integers (wcs, wide character string) that can then be mapped through a CCS to a sequence of abstract characters called meaningful text.

As stated in RFC 2278, the combination of CCS + CES is labeled as "**charset**" in MIME context. Many charsets have the same underlying CCS and repertoire: ISO-2022-KR and EUC-KR both encode some linearization of the 94x94 Korean national standard table KS C 5601 and UTF-16, UTF-8, and SCSU all encode the linear CCS Unicode.

### UTF-16

The original Unicode design suggested to extend the character size to a fixed length of 16 bits just like ISO-8859-1 uses a fixed length of 8 bits. And there are indeed a couple of systems (namely Windows NT and CE) that frequently read and write 16bit characters.

A fixed length of 16 bits has the problem that only  $2^{16} == 65'536$  characters can be encoded. And the original estimates that this number would be big enough to hold everything the world needs are currently being proven wrong as you can see in Michael Everson's allocation roadmap and the Unicode allocation pipeline.

In order to soothe the code space scarcity ISO 10646 Amendment 1 redefined the range {U+D800..U+DFFF} of (formerly private-use) 16-bit characters as "surrogates" to reference characters from the 20-bit range {U-00010000..U-0010FFFF} which gives us 16 additional 16-bit "planes". (In ISO terminology, our 32-bit code space is divided into 256 Groups of 256 Planes of 256 Rows of 256 character cells each - a "plane" is a 16-bit code space). Plane 1 (U-0001xxxx) is going to hold ancient and invented scripts and musical symbols, while Plane 2 (U-0002xxxx) is reserved for additional Han ideographs, Plane 14 (U-000Exxxx) is going to start with some meta characters for language tagging and there are two entire bonus private-use planes: Plane 15 (U-000Fxxxx) and Plane 16 (U-0010xxxx).

The characters needed for writing today's living languages are still supposed to be placed in the original Unicode Plane 0 (U+xxxx, also known as Basic Multilingual Plane or BMP) so that simpler and pre-UTF-16 implementations don't necessarily have to decode the surrogate pairs.

The UTF-16 reduction of the 20 bit range to 16 bit surrogate pairs goes like this:

```
putwchar(c)
{
    if (c > 0xFFFF) {
        putwchar (0xD7C0 + (c >> 10));
        putwchar (0xDC00 | c & 0x3FF);
    }
}
```

# Unicode Transformation Formats

## UTF-8 and Co.

This has the effect that

```
\uD800\uDC00 = U-00010000
\uD800\uDC01 = U-00010001
\uD801\uDC01 = U-00010401
\uDBFF\uDFFF = U-0010FFFF
```

You can always distinguish the leading high surrogates {U+D800..U+DBFF} from the following low surrogates {U+DC00..U+DFFF} so that long stretches of UTF-16 surrogate pairs are **self-segregating** just like the stretches of UTF-8 multibyte characters.

However note that UTF-16 does have an **additive offset** to get `\uD800\uDC00` to encode U-00010000 instead of simply the 10 + 10 bitmask bitshift concatenation

```
[1101 10]00 0000 0000 [1101 11]00 0000 0000 != U-00000000
```

The simple UTF-8-like bitmask concatenation would have encoded a closed 20-bit space {U+0000...U-000FFFFFF}. For my taste, that would have sufficed. Now that UTF-16 has the additive offset to skip the first  $2^{16}$  characters, we get  $2^{16} - 2^{11} + 2^{20} = 1'112'064$  encodable characters which is a bit more than  $2^{20}$ , about  $2^{20.1}$ . That means that the private use characters of the 17th plane U-0010xxxx accessed through the high surrogates {U+DBC0..U+DBFF} need an odd 21st bit that may only be set if the following four bits are zero.

Another problem with 16-bit characters is that they are fine on 16-bit systems but suffer from **byte-order** problems whenever they have to be serialized into 8bit bytes for storage or transmission because there are little-endian processors that put the lower value byte at the lower address (first thus) and there are big-endian processors that put the higher byte first like we write our numbers.

### The Naïve Approach

```
unsigned short int unicodechar = 0x20AC;
write (1, &unicodechar, sizeof(unicodechar));
```

will output little-endian =AC=20 on an Intel PC or DECstation and big-endian network byte order =20=AC on a Sparc.

The Unicode Standard specifies that the more human-readable big-endian network byte order is supposed to be used for UTF-16. To ensure that byte-order, our UTF-16 `putwchar()` function thus ends like this:

```
else {
    putchar (c >> 8); putchar (c & 0xFF);
}
}
```

Annex F of ISO 10646-1:1993 and § 2.4 of Unicode 2.0 recommend that UTF-16 texts start with the no-op character U+FEFF ZERO WIDTH NO-BREAK SPACE as **byte-order mark** (BOM) to recognize byte-swapped UTF-16 text from haphazard programs on little-endian Intel or DEC machines from its =FF=FE signature (U+FFFE is guaranteed to be no Unicode character).

# Unicode Transformation Formats

## UTF-8 and Co.

The limited UTF-16 without the surrogate mechanism is called **UCS-2**, the two-byte subset which is identical with the Basic Multilingual Plane of the original Unicode.

With two bytes per ideograph or syllable, UTF-16 is considered a minimally **compact** encoding for East Asian or Ethiopic text. Many applications including Xlib, Yudit and Java use UCS-2 or UTF-16 as internal `wchar_t` because they don't want to waste four bytes per character as UCS-4 does.

But with two bytes instead of one per letter, UTF-16 is also considered a **bloated** size-doubling encoding for ASCII or ISO-8859 texts which is why you might prefer SCSU for storage or transmission.

The worst problem with UTF-16 is yet that its byte stream often contains **null bytes** (for example in front of every ASCII character) and values that misleadingly look like other meaningful ASCII characters. That means that you cannot simply send UTF-16 text through your mail server, C compiler, shell nor use it in filenames or any application using the C string functions such as `strlen()`.

Unix systems with their rich heritage of traditional software are simply based on US-ASCII or ASCII-compatible extensions like ISO-8859 or EUC. For example, they have no `stty pass16` or `stty cs16` command to simply expand the terminal's character size to 16 bits. Therefore people prefer ASCII-compatible 8-bit transformation formats for which they can simply say `stty pass8`. Or 7-bit ASCII representations of the Unicode characters.

### UCS-4

A bit more generous than the 16-bit format is the native 32-bit format UCS-4 recommended by ISO 10646. It looks like this:

```
putwchar(c)
{
    putchar (c >> 24 & 0xFF); /* group */
    putchar (c >> 16 & 0xFF); /* plane */
    putchar (c >>  8 & 0xFF); /* row   */
    putchar (c          & 0xFF); /* cell  */
}
```

The byte-order mark U+FEFF turns into the UCS-4 signature =00=00=FE=FF or little-endian byte-permuted into =FF=FE=00=00.

This format will be able to express all UCS character values even if they lie far beyond UTF-16's reach. The UCS-4 format is binary transparent and knows no illegal sequences. All it requires is that the string lengths are multiples of 4 bytes. UCS-4 can easily be processed on the 32-bit computers of the 1990s using the default data type `int`. This is the internal wide character data type `wchar_t` of most Unix boxes. Unfortunately, it has the same null-byte, space-waste and byte-order problems as UTF-16, only worsened by a much bigger number of redundant null bytes. That's why you will hardly ever get to see UCS-4 text out in the wild life.

### EUC

The forefather of most Unix transformation formats (UTF) is AT&T's famous EUC (Extended Unix Code) encoding scheme used to encode the Japanese JIS X 0208 (EUC-JP), Chinese GB 2312 (EUC-CN), Taiwanese CNS 11643 (EUC-TW) and Korean KS C 5601 (EUC-KR) all of which are

# Unicode Transformation Formats

## UTF-8 and Co.

national standards specifying squares of 94 rows by 94 cells filled with ideographs, syllables, symbols and letters, similar to Unicode. Ignoring the SS2 and SS3 single-shift extension to include additional squares or cubes, EUC goes like this:

```
putwchar (c)
{
  if (c < 0x80) {
    putchar (c);    /* ASCII character */
  }
  else {
    putchar (0xA0 + /* ku (row) */ (c >> 8));
    putchar (0xA0 + /* ten (cell) */ (c & 0x7F));
  }
}
```

ASCII characters thus represent themselves and nothing else. And the square characters are represented by a pair of 8bit bytes each which gave them the name double-byte characters. If you stuff your ASCII characters into fixed-width boxes that equal half of a 14x14, 16x16 or 24x24 pixel square, like cterm and kterm and many other applications do, you even get the impression that double-byte characters are also double-width. That is because the width stays proportional to the byte count in EUC in spite of the reduced character count. You can put some 40 square characters in one standard screen line that otherwise is 80 ASCII characters wide.

As the first and second byte of a double-byte character both use the same {=A1..=FE} range of values, you cannot easily tell the one from the other and recognize the character boundaries in the middle of a long stretch of 8bit bytes. You have to back up to the first preceding 7bit ASCII byte which might for example be the linefeed `\n`, use it as synchronization point and then start counting pairs forwards from there. A regular expression search for the `=BB=CC` square could detect a false positive within the square sequence `=AA=BB =CC=DD` and cause harm to it.

Another problem with the {=A1..=FE} range is that it only contains 94 values, so that the character set is limited to  $94 \times 94 = 8'836$  squares which is not enough for the Unicode repertoire.  $94 \times 94 \times 94 = 830'584$  would however be enough to cover 16bit Unicode. And EUC does allow three-byte characters. But I have never seen any serious proposal for such an EUC-UN. The transformation arithmetic from the linear Unicode axis to the odd 94-cube is perhaps considered too computation-intensive. It might have come different had Unicode not stopped the original draft UCS DIS-1 10646 of 1990 that was still trying to stick to the ISO 2022 code extension techniques for 7/8-bit coded character sets.

### UTF-1

The first transformation format for Unicode was the UTF-1 specified in Annex G of the original ISO 10646 of 1993. It avoided the control code values {=00...=20,=7F...=9F} for graphic characters and used the remaining 190 graphic code values like this:

```
#define T(z) (z<94?z+33:z<190?z+66:z<223?z-190:z-96)

putwchar(c)
{
  if (c < 160) {
    putchar(c);
  }
}
```

# Unicode Transformation Formats

## UTF-8 and Co.

```
}
else if (c < 256) {
    putchar(160);
    putchar(c);
}
else if (c < 16406) {
    putchar(161+(c-256)/190);
    putchar(T((c-256)%190));
}
else if (c < 233005) {
    putchar(246+(c-16406)/190/190);
    putchar(T((c-16406)/190%190));
    putchar(T((c-16406)%190));
}
else {
    putchar(252+(c-233005)/190/190/190/190);
    putchar(T((c-233005)/190/190/190%190));
    putchar(T((c-233005)/190/190%190));
    putchar(T((c-233005)/190%190));
    putchar(T((c-233005)%190));
}
}
```

This UTF-1 has a number of **serious disadvantages**:

- You cannot recognize the beginning of a multibyte character in the middle of a byte stream. If you have lost track since the beginning of the stream, then there is **no self-synchronization**. You get complete garbage if you start decoding in the middle of a multibyte character but your computer won't notice. You cannot use traditional programs like `grep` and `sed` to do string search and substitution because they could find false positives if they looked at the middle of a multibyte character.
- Non-slash characters like U+010E LATIN CAPITAL LETTER D WITH CARON are represented with a **slash** (`/` in `"i/"`). This makes certain characters unusable in filenames. Other ASCII characters with special functions create the same problem.
- The integer **division** operations make UTF-1 unnecessarily slow.

The use UTF-1 is deprecated. Annex G (informative) has formally been deleted from ISO-10646 although the page has probably not been ripped out at your local library. Glenn Adams' 1992 implementation `utf.c` is still floating around.

### UTF-8

UTF-1's disadvantages led to the invention of UTF-2 alias (filesystem-safe) FSS-UTF, now known as the standard 8bit transformation format **UTF-8**. UTF-8 goes like this:

```
putwchar(c)
{
    if (c < 0x80) {
        putchar (c);
    }
}
```

# Unicode Transformation Formats

## UTF-8 and Co.

```
}  
else if (c < 0x800) {  
    putchar (0xC0 | c>>6);  
    putchar (0x80 | c & 0x3F);  
}  
else if (c < 0x10000) {  
    putchar (0xE0 | c>>12);  
    putchar (0x80 | c>>6 & 0x3F);  
    putchar (0x80 | c & 0x3F);  
}  
else if (c < 0x200000) {  
    putchar (0xF0 | c>>18);  
    putchar (0x80 | c>>12 & 0x3F);  
    putchar (0x80 | c>>6 & 0x3F);  
    putchar (0x80 | c & 0x3F);  
}  
}
```

The binary representation of the character's integer value is thus simply spread across the bytes and the number of high bits set in the lead byte announces the number of bytes in the multibyte sequence:

bytes	bits	representation
1	7	0vvvvvvv
2	11	110vvvvv 10vvvvvv
3	16	1110vvvv 10vvvvvv 10vvvvvv
4	21	11110vvv 10vvvvvv 10vvvvvv 10vvvvvv

(Actually, UTF-8 continues to represent up to 31 bits with up to 6 bytes, but it is generally expected that the one million code points of the 20 bits offered by UTF-16 and 4-byte UTF-8 will suffice to cover all characters and that we will never get to see any Unicode character definitions beyond that.)

### UTF-8's Features

The UTF-8 design pays off with a number of attractive features:

#### + transparency and uniqueness for ASCII characters

The 7bit ASCII characters {U+0000...U+007F} pass through transparently as {=00..=7F} and all non-ASCII are represented purely with non-ASCII 8bit values {=80..=F7} so that you will not mistake any non-ASCII character for an ASCII character, particularly the string-delimiting ASCII =00 NULL only appears where there a U+0000 NULL was intended, and you can use your ASCII-based text processing tools on UTF-8 text as long as they pass 8bit characters without interpretation.

#### + self-synchronization

UTF-8 is self-segregating: you can always distinguish a lead byte 11vvvvvv from a fill byte 10vvvvvv and you will never be mistaken about the beginning or the length of a multibyte character. You can start parsing backwards at the end or in the middle of a multibyte string and will soon find a synchronization point. String searches (fgrep) for a multibyte character beginning with a lead byte will never match on the fill byte in the middle of an unwanted multibyte character.

# Unicode Transformation Formats

## UTF-8 and Co.

### + processor-friendliness

UTF-8 can be read and written quickly with simple bitmask and bitshift operations without any multiplication or division. And as the lead-byte announces the length of the multibyte character you can quickly tell how many bytes to skip for fast forward parsing.

### + reasonable compression

UTF-8 is a reasonably compact encoding: ASCII characters are not inflated, most other alphabetic characters occupy only two bytes each, no basic Unicode character needs more than three bytes and all extended Unicode characters can be expressed with four bytes so that UTF-8 is no worse than UCS-4.

### + canonical sort-order

Comparing two UTF-8 multibyte character strings with the old `strcmp(mbs1,mbs2)` function gives the same result as the `wcscmp(wcs1,wcs2)` function on the corresponding UCS-4 wide character strings so that the lexicographic sorting and tree-search order is preserved.

### + EOF and BOM avoidance

The octets `=FE` and `=FF` never appear in UTF-8 output. That means that you can use `=FE=FF` (U+FEFF ZERO WIDTH NO-BREAK SPACE) as an indicator for UTF-16 text and `=FF=FE` as an indicator for byte-swapped UTF-16 text from haphazard programs on little-endian machines. And it also means that means that C programs that haphazardly store the result of `getchar()` in a `char` instead of an `int` will no longer mistake U+00FF LATIN SMALL LETTER Y WITH DIAERESIS as end of file because `ÿ` is now represented as `=C3=BF`. The `=FF` octet was often mistaken as end of file because `/usr/include/stdio.h` `#defines` EOF as -1 which looks just like `=FF` in 8bit 2-complement binary integer representation.

### + byte-order independence

UTF-8 has no byte-order problems as it defines an octet stream. The octet order is the same on all systems. The UTF-8 representation `=EF=BB=BF` of the byte-order mark U+FEFF ZERO WIDTH NO-BREAK SPACE is not really needed as a UTF-8 signature.

### + detectability

You can detect that you are dealing with UTF-8 input with high probability if you see the UTF-8 signature `=EF=BB=BF` (ï»¿) or if you see valid UTF-8 multibyte characters since it is very unlikely that they accidentally appear in Latin1 text. You usually don't place a Latin1 symbol after an accented capital letter or a whole row of them after an accented small letter.

## UTF-8's Standardization History

The UTF-8 standardization is a success story that led to the abandonment of UTF-1 and made UTF-8 the prime candidate for external multibyte representation of Unicode texts:

Ken Thompson and Rob Pike (from the famous AT&T Bell Laboratories who had also invented Unix and C) describe in their 1993 Usenix presentation "Hello world or Καλημέρα κόσμε or こんにちは世界" (your browser supports internationalized HTML if you see some Greek and Japanese here) how the UTF-8 format was crafted (together with the X/Open Group who were writing the internationalized XPG4 portability guide, forefather of today's Unix98 specification) to suit the needs of their new Plan9 operating system. Ken Thompson provided his commented sample implementation `fss-utf.c` in 1992.

UTF-8 was formally adopted as normative Annex R in Amendment 2 to ISO-10646 in 1996. UTF-8 was also included in The Unicode Standard, Version 2.0, as Appendix A.2, accompanied by Mark

# Unicode Transformation Formats

## UTF-8 and Co.

Davis' CVTUTF implementation. Also in 1996, François Yergeau's RFC 2044 defined the MIME charset label "UTF-8" which is now on the Internet standards track as updated RFC 2279.

Newer Internet protocols that want to do without charset labeling tend to use UTF-8 as default text encoding, like RFC 1889 RTP, RFC 2141 URN Syntax, RFC 2192 IMAP URL Scheme, RFC 2218 IWPS, RFC 2229 Dictionary Server Protocol, RFC 2241 DHCP, RFC 2244 ACAP, RFC 2251..2255 LDAP, RFC 2261 SNMP, RFC 2284 PPP, RFC 2295 HTTP, RFC 2324 HTCPCP, RFC 2326 RTSP, RFC 2327 SDP, RFC 2376 XML, ...

The mail internationalization report produced by the Internet Mail Consortium recommends since 1998-08-01 that

*All mail-displaying programs created or revised after January 1, 1999, must be able to display mail that uses the UTF-8 charset. Another way to say this is that any program created or revised after January 1, 1999, that cannot display mail using the UTF-8 charset should be considered deficient and lacking in standard internationalization capabilities.*

### UTF-8's Usability

The last but not least argument in favor of UTF-8 is that there is a small but growing number of Unix tools with special UTF-8 handling today already:

#### Yudit

Gaspar Sinai's very charming Unicode editor Yudit for the X11 Window System uses UTF-8 as default encoding and can enter, display and print, and cut and paste UTF-8 text. To enable yourself to view UTF-8 mails, you can simply add the following entry to your mailcap file:

```
text/plain; xviewer yudit < %s; test=case %{charset} in \  
[Uu][Tt][Ff]-8) [ yes ] \;\ * ) [ UTF-8 = no ] \  
\; esac
```

Yudit also comes with a code conversion tool named uniconv that you can use as a replacement for Plan9's unsupported tcs and can be plugged into your .pinerc with:

```
display-filters=_CHARSET(UTF-8)_ /usr/bin/uniconv -I UTF8 -O JAVA,  
_CHARSET(UTF-7)_ /usr/bin/uniconv -I UTF7 -O JAVA
```

#### Australian Plan9 derivatives

9term, the Plan9 terminal emulator for X, can display and enter UTF-8 text in a simple bitmapped window without cursor control (it is thus no xterm replacement) with a somewhat unintuitive (means that I did not understand it) interface, the same goes for the Wily text environment which seems to be quite flexible but hard to get used to.

#### Lynx

The textual web browser Lynx can approximate properly labeled UTF-8 texts on your non-UTF-8 display or convert a whole bunch of charsets to or through UTF-8. Lynx 2.8 renders for instance the non-ASCII characters from <http://czyborra.com/> like this on an ASCII display:

```
Roman Czyborra (Latin)  
= 'rOman tSi"'bOra (IPA)  
= Roman CHibora (Cyrillic)  
= TiBoRa RoMaN6 (Japanese)  
= roman cibora (Ethiopic)  
= R+W+M+N% ZJ'J+B+W+R+H+ (Hebrew)  
= r+w+m+/'n+ t+sny+b+w+r+h+ (Arabic)
```

# Unicode Transformation Formats

## UTF-8 and Co.

= UB85CUB9CC UCE58¼ (Korean).

### Netscape

Netscape's Navigator 4 (Mozilla) can use the Japanese JIS X 0208 fonts present in X11 to display some UTF-8 text when labeled with

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html;charset=UTF-8">
```

Mozilla has brought encapsulated MIME and Unicode rendering capabilities to the masses. Together with some input methods, it might grow into a full-blown UTF-8 mail, news and WWW messaging system. Frank Tang's [unixprint.html](#) contains some thoughts about how to make Mozilla print more than ISO-8859-1 PostScript and links to useful documents.

### XML

James Clark's XML tools like the SP SGML parser support UTF-8 and his canonical XML most prominently chooses UTF-8 as the only accepted encoding.

### 2UTF

Ri&ccaron;ardas &Ccaron;epas' 2UTF converter can convert a MIME message to UTF-8 or the other way around and contains a useful ASCII approximation table.

### GNU recode

François Pinard's pre-release recode 3.4g can convert its 8bit charsets (no JIS, GB, or EUC) to and from UTF-\*, combined with linebreak conversion, MIME transfer encoding or numerical representation which can also be used as an input method:

```
echo 0x20AC | recode -f ucs2/x2..utf8/qp gives: =E2=82=AC=
```

### GNU libc

Ulrich Drepper's (see his 1996 paper on GNU internationalization) growing GNU C library glibc-2.0.6 contains implementations for some members of the ISO-9899-Amendment-1 family of functions (mbstowcs & Co.) using UTF-8 as multibyte encoding. GNU gettext internationalization is going to be based on UTF-8: the GNU maintainers are asked to support UTF-8 but not required to support UTF-16.

### Java

Sun's programming language and operating environment Java uses Unicode in UTF-16 form internally, and its network computer (NC) idea might have the potential to bring us Unicode terminals to replace the ASCII VT100s. Java's DataInputStream and DataOutputStream offer readUTF() and writeUTF() methods but UTF-8 does not seem to be the default interchange format. UTF-8 is only one of the supported I/O encodings.

### Ada

The GNU Ada95 compiler gnat-3.11 shall support UTF-8 in both source code (proper math symbols finally!) and processed Wide\_String text I/O.

### Tcl

John Ousterhout's popular Tool command language Tcl presented at various USENIX conferences has Unicode support since version 8.1 and represents all strings in UTF-8.

### Python

# Unicode Transformation Formats

## UTF-8 and Co.

The high-level programming language Python has some Unicode support through Martin von Löwis's `wstring` module with `utf8` operations but the Python browser Grail 0.3 still seems to be limited to Latin1 when I fed it a UTF-8 hypertext.

### Perl

You can create your own UTF-8 assembler or disassembler with any old Perl interpreter:

```
#!/usr/local/bin/perl -p
# assemble <U20AC> into â,¬
sub utf8 { local($_)=@_;
    return $_ < 0x80 ? chr($_) :
        $_ < 0x800 ? chr($_>>6&0x3F|0xC0) . chr($_&0x3F|0x80) :
            chr($_>>12&0x0F|0xE0).chr($_>>6&0x3F|0x80).chr($_&0x3F|0x80);
} s/<U([0-9A-F]{4})>/&utf8(hex($1))/gei;

#!/usr/local/bin/perl -p
# disassemble non-ASCII codes from UTF-8 stream
$format=$ENV{"UCFORMAT"}||'<U%04X>';
s/([\xC0-\xDF])([\x80-\xBF])/sprintf($format,
unpack("c",$1)<<6&0x07C0|unpack("c",$2)&0x003F)/ge;
s/([\xE0-\xEF])([\x80-\xBF])([\x80-\xBF])/sprintf($format,
unpack("c",$1)<<12&0xF000|unpack("c",$2)<<6&0x0FC0|unpack("c",$3)&0x03F)/ge;
s/([\xF0-\xF7])([\x80-\xBF])([\x80-\xBF])([\x80-\xBF])/sprintf($format,
unpack("c",$1)<<18&0x1C0000|unpack("c",$2)<<12&0x3F000|
unpack("c",$3)<<6&0x0FC0|unpack("c",$4)&0x003F)/ge;
```

The module `use utf8`; will let you simplify these tasks with future Perl releases. Larry Wall is adding native Unicode and XML support to Perl together with the `perl-unicode@perl.org` experts.

### Linux console

Markus Kuhn wrote an handy UTF-8 manual page for Linux in 1995 and also hacked UTF-8 support into the Linux console which can be used with Yann Dirson's `consoletools`. The standard escape sequence `ESC % G` (=1B=25=47) switches the console into UTF-8 mode. All Unicode characters listed as displayable by one of the two loaded VGA fonts (limited to 512 characters) in the corresponding screen font map are displayed as such and all others are rendered as the default `U+FFFD REPLACEMENT CHARACTER` (black box or question mark).

### uxterm

There has been a unicoded `xterm` called `uxterm` from Ms Ho Yean Fee's Multilingual Application Support Service (MASS) group at the Institute of Systems Science of the National University of Singapore who have now formed their own company Star + Globe Technologies but it is only available in binary form for AIX, HP-UX, IRIX, OSF, and SunOS and not in OpenSource so you cannot port it to Linux nor enhance it.

### AIX

AIX has some UTF-8 locales thanks to IBM Austin.

### Solaris 7

# Unicode Transformation Formats

## UTF-8 and Co.

Solaris 2.7 has a new en\_US.UTF-8 locale that employs a set of ISO 8859 and CJK fonts to render UTF-8 text and print it through a printing utility called xutops, see lenup Sung's paper. Solaris 2.6 already supported UTF-8 in its /usr/lib/iconv/\*.so conversion library.

### Pine

Pine 4 contains code to map any charset to UTF-8 in Mark Crispin's imap/c-client/src/utf-8.[ch] but as of Pine 4.02 pine does not seem to use those routines yet except for unicoded searching.

Yet more needs to be done:

### Emacs

GNU Emacs 20 includes the MULE multilingual extension which unfortunately avoids Unicode and uses some non-standard internal encoding. Otfried Cheong has already written a unicode.el and utf2mule module to use UTF-8 within GNU Emacs 20.3.

There is hope that Emacs 21 will turn to Unicode completely. Richard Stallman is being advised by the emacs-unicode@gnu.org circle. Since Emacs is not just an editor but a programmable Lisp environment its Unicode support will help a lot to read and write web pages, mails and news in UTF-8 and run shells and other tools in a UTF-8 environment. Emacs has already been using 24-bit-characters (atoms) internally for quite a while.

### rxvt

The lean and colorful xterm replacement rxvt does not suffer from xterm's C1 problem (it simply uses the ISO 6279 replacement escape sequences instead of 8bit controls and treats {=80..=9F} as simple graphic characters), can already display Japanese Kanji, and has UTF-8 support on top of its TODO list.

### curses

The Unix98 specification defines wide character functions for the termcap-based screen-I/O library **curses** which makes programs like less, lynx and pine work so wonderfully in xterms as well as via serial lines or telnet all over the world. I hear that the curses wchar functions are for instance implemented in AIX and DEC Unix but our free GNU implementation ncurses is only 8bit clean base-level so far.

### vi

Larry Wall says that he wants a UTF-8-capable version of the traditional Berkeley Unix visual editor vi.

### less

The convenient text pager GNU less (opposite of more) displays UTF-8 codes as something quoted like "<E2><82><AC>" when started with LESSCHARSET=ascii or latin1. It can be tricked into passing the binary UTF-8 codes to a UTF-8 terminal by setting LESSCHARSET=koi8-r. But then it still breaks its lines after 80 characters without any respect for the UTF-8 multibyte character boundaries or the Unicode 2.0 § 5.13 character separation rules.

### communication software

# Unicode Transformation Formats

## UTF-8 and Co.

PGP 5.0i and IRC II-4.4 still use Latin1 as their canonical text encoding instead of UTF-8: `{cp850,ebcdic}_to_latin1` in `pgp-5.0i/src/lib/pgp/helper/pgpCharMap.c` and `ircii-4.4/source/translat.c`

### a2ps

Akim Demaille whose a2ps can already help you format some non-Latin1 texts in PostScript for printing is looking into Unicode.

### troff

The Plan9 version of John Osanna's troff text formatter can format some UTF-8 text but has yet to be ported back to Unix. And James Clark's GNU troff formatter **groff** which pioneered in making ISO-8859-1 usable with troff is orphaned without a maintainer understanding all of its internal workings. Brian Kernighan says that it's not that difficult.

### TeX

Some Unicode functions for TeX can be found in Werner Lembergs CJK package for LaTeX 2e and Yannis Haralambous' multilingual Omega. I cannot tell you how to get either of them to work but they wrote very interesting articles about it in the TUGboat. Also check out what Richard Kinch and Norman Walsh have to say.

Other operating systems support UTF-8 as well. Plan9 and BeOS use it as their native representation, Windows NT knows UTF-8 as code page 1208.

## UTF-8's Problems

With all that hype about UTF-8 having so many advantages and being standardized and supported by implementations, you may find yourself asking what the catch is. And there are indeed things you can criticize about UTF-8:

### variable length

UTF-8 is a variable-length multibyte encoding which means that you cannot calculate the number of characters from the mere number of bytes and vice versa for memory allocation and that you have to allocate oversized buffers or parse and keep counters. You cannot load a UTF-8 character into a processor register in one operation because you don't know how long it is going to be. Because of that you won't want to use UTF-8 multibyte bitstrings as direct scalar memory addresses but first convert to UCS-4 wide character internally or single-step one byte at a time through a single/double/triple indirect lookup table (somewhat similar to the block addressing in Unix filesystem i-nodes). It also means that dumb line breaking functions and backspacing functions will get their character count wrong when processing UTF-8 text.

### extra byte consumption

UTF-8 consumes two bytes for all non-Latin (Greek, Cyrillic, Arabic, etc.) letters that have traditionally been stored in one byte and three bytes for all symbols, syllabics and ideographs that have traditionally only needed a double byte. This can be considered a waste of space and bandwidth which is even tripled when the 8bit form is MIME-encoded as quoted-printable ("`=C3=A4`" is 6 bytes for the one character ä). SCSU aims to solve the compression problem.

### illegal sequences

# Unicode Transformation Formats

## UTF-8 and Co.

Because of the wanted redundancy in its syntax, UTF-8 knows **illegal sequences**. Some applications even emit an error message and stop working if they see illegal input: Java has its UTFDataFormatException. Other UTF-8 implementations silently interpret illegal input as 8bit ISO-8859-1 (or more sophisticated defaults like your local 8bit charset or CP1252 or less sophisticated defaults like whatever the implementation happens to calculate) and output them through their putwchar() routines as correct UTF-8 which may first seem as a nice feature but on second thought it turns out as corrupting binary data (which then again you weren't supposed to feed to your text processing tools anyway) and throwing away valuable information. You cannot include arbitrary binary string samples in a UTF-8 text such as this web page. Also, by illegally spreading bit patterns of ASCII characters across several bytes you can trick the system into having filenames containing ungraspable nulls (=C0=80) and slashes (=C0=AF).

### 8bit characters

UTF-8 uses 8bit characters which are still being stripped by many mail gateways because Internet messages were originally defined to be 7bit ASCII only but their number is decreasing as the software of the 1990s tends to be 8bit clean. The 8bit problem has led to the invention of UTF-7.

### C1 controls

UTF-8 uses the 100vvvvv values {=80..=9F} in more than 50 % of its multibyte representations but existing implementations of the ISO 2022, 4873, 6429, 8859 systems mistake these as C1 control codes. These control codes regularly drive my xterm into an insane state when I dump UTF-8 text in it, older mainframe terminals suffer from the same condition. This puts a serious damper on UTF-8's alleged compatibility with traditional tools because it requires them all to be reprogrammed which could have been prevented with a more perfect UTF-7,5.

### EBCDIC incompatibility

UTF-8 is only compatible with ASCII but not with EBCDIC. Well, I was just kidding - this is no serious disadvantage :) - but there is a draft UTR #16 specifying an "EBCDIC-Friendly UCS Transformation Format".

### Latin1 incompatibility

While ISO-8859-1 (the widespread Latin1 that is HTML's historical default charset) is a code subset of Unicode, ISO-8859-1's 8bit encoding scheme is no code subset of UTF-8. Latin1 letters look quite different when transformed into UTF-8. Non-ASCII characters in UTF-8 output look illegible on Latin-1 terminals. The many existing and unlabeled Latin1 texts are no legal UTF-8 input. Contrary to UTF-8, SCSU, JAVA and HTML allow Latin1 text to pass through transparently without being limited to Latin1.

Latin1	UTF-1	UTF-8	UTF-7,5	UTF-7	JAVA	HTML
		Â	çà	+AKA-	\u00a0	&#160;
í	í	Âí	çá	+AKE-	\u00a1	&#161;
ç	ç	Âç	çâ	+AKI-	\u00a2	&#162;
£	£	Â£	çã	+AKM-	\u00a3	&#163;
¤	¤	Â¤	çä	+AKQ-	\u00a4	&#164;
¥	¥	Â¥	çå	+AKU-	\u00a5	&#165;
		Â	çæ	+AKY-	\u00a6	&#166;
§	§	Â§	çç	+AKc-	\u00a7	&#167;
¨	¨	Â¨	çè	+AKg-	\u00a8	&#168;
©	©	Â©	çé	+AKk-	\u00a9	&#169;
à	à	Âà	çê	+AKo-	\u00aa	&#170;

# Unicode Transformation Formats

## UTF-8 and Co.

<<	<<	Â<	çë	+AKs-	\u00ab	&#171;
¬	¬	Â¬	çi	+AKw-	\u00ac	&#172;
		Â	çi	+AK0-	\u00ad	&#173;
®	®	Â®	çi	+AK4-	\u00ae	&#174;
-	-	Â-	çi	+AK8-	\u00af	&#175;
°	°	Â°	çö	+ALA-	\u00b0	&#176;
±	±	Â±	çñ	+ALE-	\u00b1	&#177;
²	²	Â²	çò	+ALI-	\u00b2	&#178;
³	³	Â³	çó	+ALM-	\u00b3	&#179;
´	´	Â´	çô	+ALQ-	\u00b4	&#180;
µ	µ	Âµ	çõ	+ALU-	\u00b5	&#181;
¶	¶	Â¶	çö	+ALY-	\u00b6	&#182;
·	·	Â·	ç÷	+ALc-	\u00b7	&#183;
,	,	Â,	çø	+ALg-	\u00b8	&#184;
¹	¹	Â¹	çù	+ALk-	\u00b9	&#185;
º	º	Âº	çú	+ALo-	\u00ba	&#186;
»	»	Â»	çû	+ALs-	\u00bb	&#187;
¼	¼	Â¼	çü	+ALw-	\u00bc	&#188;
½	½	Â½	çý	+AL0-	\u00bd	&#189;
¾	¾	Â¾	çþ	+AL4-	\u00be	&#190;
¿	¿	Â¿	çÿ	+AL8-	\u00bf	&#191;
À	À	Â€	£À	+AMA-	\u00c0	&#192;
Á	Á	Â	£Á	+AME-	\u00c1	&#193;
Â	Â	Â,	£Â	+AMI-	\u00c2	&#194;
Ã	Ã	Âf	£Ã	+AMM-	\u00c3	&#195;
Ä	Ä	Â,,	£Ä	+AMQ-	\u00c4	&#196;
Å	Å	Â...	£Å	+AMU-	\u00c5	&#197;
Æ	Æ	Â†	£Æ	+AMY-	\u00c6	&#198;
Ç	Ç	Â‡	£Ç	+AMc-	\u00c7	&#199;
È	È	Â^	£È	+AMg-	\u00c8	&#200;
É	É	Â%	£É	+AMk-	\u00c9	&#201;
Ê	Ê	Â\$	£Ê	+AMo-	\u00ca	&#202;
Ë	Ë	Â<	£Ë	+AMs-	\u00cb	&#203;
Ì	Ì	Â€	£Ì	+AMw-	\u00cc	&#204;
Í	Í	Â	£Í	+AM0-	\u00cd	&#205;
Î	Î	ÂŽ	£Î	+AM4-	\u00ce	&#206;
Ï	Ï	Â	£Ï	+AM8-	\u00cf	&#207;
Ð	Ð	Â	£Ð	+ANA-	\u00d0	&#208;
Ñ	Ñ	Â\	£Ñ	+ANE-	\u00d1	&#209;
Ò	Ò	Â'	£Ò	+ANI-	\u00d2	&#210;
Ó	Ó	Â"	£Ó	+ANM-	\u00d3	&#211;
Ô	Ô	Â''	£Ô	+ANQ-	\u00d4	&#212;
Õ	Õ	Â•	£Õ	+ANU-	\u00d5	&#213;
Ö	Ö	Â-	£Ö	+ANY-	\u00d6	&#214;
×	×	Â-	£×	+ANc-	\u00d7	&#215;
Ø	Ø	Â~	£Ø	+ANg-	\u00d8	&#216;
Ù	Ù	Â™	£Ù	+ANK-	\u00d9	&#217;
Ú	Ú	Â\$	£Ú	+ANo-	\u00da	&#218;
Û	Û	Â>	£Û	+ANs-	\u00db	&#219;
Ü	Ü	Âœ	£Ü	+ANw-	\u00dc	&#220;
Ý	Ý	Â	£Ý	+AN0-	\u00dd	&#221;
Þ	Þ	Âž	£Þ	+AN4-	\u00de	&#222;
ß	ß	ÂŸ	£ß	+AN8-	\u00df	&#223;
à	à	Â	£à	+AOA-	\u00e0	&#224;
á	á	Â¡	£á	+AOE-	\u00e1	&#225;
â	â	Âç	£â	+AOI-	\u00e2	&#226;

# Unicode Transformation Formats

## UTF-8 and Co.

ã	ã	Ã£	fã	+AOM-	\u00e3	&#227;
ä	ä	Ã¤	fä	+AOQ-	\u00e4	&#228;
å	å	Ã¥	få	+AOU-	\u00e5	&#229;
æ	æ	Ã¦	fæ	+AOY-	\u00e6	&#230;
ç	ç	Ã§	fç	+AOc-	\u00e7	&#231;
è	è	Ã¨	fè	+AOg-	\u00e8	&#232;
é	é	Ã©	fé	+AOk-	\u00e9	&#233;
ê	ê	Ãª	fê	+AOo-	\u00ea	&#234;
ë	ë	Ã«	fë	+AOs-	\u00eb	&#235;
ì	ì	Ã¬	fì	+AOw-	\u00ec	&#236;
í	í	Ã	fí	+AO0-	\u00ed	&#237;
î	î	Ã®	fî	+AO4-	\u00ee	&#238;
ï	ï	Ã¯	fï	+AO8-	\u00ef	&#239;
ð	ð	Ã°	fð	+APA-	\u00f0	&#240;
ñ	ñ	Ã±	fñ	+APE-	\u00f1	&#241;
ò	ò	Ã²	fò	+API-	\u00f2	&#242;
ó	ó	Ã³	fó	+APM-	\u00f3	&#243;
ô	ô	Ã´	fô	+APQ-	\u00f4	&#244;
õ	õ	Ãµ	fõ	+APU-	\u00f5	&#245;
ö	ö	Ã¶	fö	+APY-	\u00f6	&#246;
÷	÷	Ã·	f÷	+APc-	\u00f7	&#247;
ø	ø	Ã¸	fø	+APg-	\u00f8	&#248;
ù	ù	Ã¹	fù	+APk-	\u00f9	&#249;
ú	ú	Ãº	fú	+APo-	\u00fa	&#250;
û	û	Ã»	fû	+APs-	\u00fb	&#251;
ü	ü	Ã¼	fü	+APw-	\u00fc	&#252;
ý	ý	Ã½	fý	+AP0-	\u00fd	&#253;
þ	þ	Ã¾	fþ	+AP4-	\u00fe	&#254;
ÿ	ÿ	Ã¿	fÿ	+AP8-	\u00ff	&#255;

### UTF-7,5

In 1997 Jörg Knappen suggested a better UTF-8 for ISO 4873 systems that can handle graphic characters in only 190 of 256 cells which he names UTF-7,5 (UTF-seven-decimal-five):

bytes	bits	representation
1	7	0vvvvvvv
2	10	1010vvvv 11vvvvvv
3	16	1011vvvv 11vvvvvv 11vvvvvv

```
putwchar(c)
{
    if (c < 0x80) {
        putchar (c);
    }
    else if (c < 0x400) {
        putchar (0xA0 | c>>6);
        putchar (0xC0 | c & 0x3F);
    }
    else if (c < 0x10000) {
        putchar (0xB0 | c>>12);
        putchar (0xC0 | c>>6 & 0x3F);
        putchar (0xC0 | c & 0x3F);
    }
    else if (c < 0x110000) {
```

# Unicode Transformation Formats

## UTF-8 and Co.

```
    putwchar (0xD7C0 + (c >> 10));
    putwchar (0xDC00 | c & 0x3FF);
}
}
```

This UTF shares most of UTF-8's nice and not-so-nice properties inclusive of ASCII and sort-order transparency, disambiguity, self-segregation, and 2/3-byte compactness and adds some Latin1 transparency without any of UTF-1's disadvantages. It avoids the C1 controls and simply uses Latin1 character sequences to represent all non-ASCII characters. That means that it will not mess up terminals that use the C1 controls and allows UTF strings to be cut and pasted between Latin1 applications. Besides that, the UTF representation of Latin1's accented letters contains the original code prefixed by a pound sign (£) which means that its readability is maintained in Latin1 applications.

The price paid for this is that the representations of Cyrillic, Armenian, Hebrew and Arabic letters {U+0400..U+07FF} grow from two to three bytes length, =FE and =FF are no longer avoided, and UTF-16 surrogates have to be used for characters beyond the 16 bit range which could be helped by a scheme like

```
16 | 1010vvvv 11vvvvvv 11vvvvvv
22 | 1011vvvv 11vvvvvv 11vvvvvv 11vvvvvv
```

Unlike UTF-8, UTF-7,5 is not officially registered as a standard encoding with ISO, Unicode, or IANA. The name "UTF-7,5" or "UTF-7½" is not even a legal charset name by RFC 2278, a better name would be "UTF-3", "UTF-5", "UTF-6", or "UTF-9". Because of the missing acceptance as a standard you are confined to using UTF-7,5 in your own backyard and with your own implementations although the Latin1 compatibility would just make it a prime candidate for information interchange with other systems. Had UTF-7,5 been suggested 4 years earlier, it could have taken the place of UTF-8 as the standard interchange encoding and we would have fewer problems with Unicode text. But now it is probably wiser to invest energy into making the affected ISO-8859-1 tools ready for the UTF-8 standard instead of trying to promote UTF-7,5 as a new UTF-9 standard.

### UTF-7

All of the above UTFs produce 8bit bytes that are not in ASCII and that will get stripped on any terminal that is still set to character size 7 or any mail gateway that ensures RFC 822's rule that mail messages have to be in ASCII. To solve that problem, David Goldsmith and Mark Davis invented a **mail-safe** transformation format **UTF-7**. It was first published in RFC 1642 in 1994, prominently included as Appendix A.1 in The Unicode Standard, Version 2.0, and now updated in RFC 2152. It makes partial use of the MIME base64 encoding and goes roughly like this:

```
char base64[] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

putwchar(c)
{
    if (c == '+') {
        putchar('+');
        putchar('-');
    }
    else if (c < 0x80) {
        putchar(c);
    }
}
```

# Unicode Transformation Formats

## UTF-8 and Co.

```
}
else if (c < 0x10000) {
    putchar('+');
    putchar(base64[c>>10&63]);
    putchar(base64[c>>4&63]);
    putchar(base64[c<<2&63]);
    putchar('-');
}
else if (c < 0x110000) {
    c = 0xD7C0DC00 + (c >> 10 << 16 | c & 0x3FF);
    putchar('+');
    putchar(base64[c>>26&63]);
    putchar(base64[c>>20&63]);
    putchar(base64[c>>14&63]);
    putchar(base64[c>>8&63]);
    putchar(base64[c>>2&63]);
    putchar(base64[c<<4&63]);
    putchar('-');
}
}
```

Except for the '+' escaping, ASCII text remains unchanged with UTF-7. In some situations, the trailing '-' is optional. And by joining a whole stretch of non-ASCII characters into a larger base64 block you can encode an average of 3 Unicode characters in 8 bytes which is much better than the 9 bytes "=E5=A4=A9" for 1 CJK ideograph 天 in quoted-printable UTF-8.

However, base64 or 8bit SCSU can achieve much better compression, and UTF-7 is a bad general-purpose processing format: its flickering base64 grouping is awkward to program, most ASCII values can stand for almost any character and there are many different possible UTF-7 encodings of the same character so that UTF-7 is practically **unsearchable** without conversion.

### SCSU

The Standard Compression Scheme for Unicode (SCSU) described in Unicode Technical Report 6 is of course also a feasible Unicode encoding scheme. It offers a good compaction with predefined and sliding windows and Latin1 transparency. But it is stateful and quite unrestricted and perhaps a bit more complicated to decode or produce than all the other encoding schemes listed here.

### Java's Unicode Notation

There are some less compact but more readable ASCII transformations the most important of which is the Java Unicode Notation as allowed in Java source code and processed by Java's native2ascii converter:

```
putwchar(c)
{
    if (c >= 0x10000) {
        printf ("\u%04x\u%04x" , 0xD7C0 + (c >> 10), 0xDC00 | c & 0x3FF);
    }
    else if (c >= 0x100) printf ("\u%04x", c);
    else putchar (c);
}
```

## Unicode Transformation Formats

### UTF-8 and Co.

The advantage of the `\u20ac` notation is that it is very **easy to type** it in on any old ASCII keyboard and easy to look up the intended character if you happen to have a copy of the Unicode book or the `{unidata2,names2,unihan}.txt` files from the Unicode FTP site or CD-ROM or know what U+20AC is the €.

What's not so nice about the `\u20ac` notation is that the small letters are quite unusual for Unicode characters, the backslashes have to be quoted for many Unix tools, the four hexdigits without a terminator may appear merged with the following word as in `\u00a333` for £33, it is unclear when and how you have to escape the backslash character itself, 6 bytes for one character may be considered wasteful, and there is no way to clearly present the characters beyond `\uffff` without `\ud800\udc00` surrogates, and last but not least the plain hexnumbers may not be very helpful.

JAVA is one of the target and source encodings of `yudit` and its `unicov` converter.

### HTML's Numerical Character References

A somewhat more standardized encoding option is specified by HTML. RFC 2070 allows us to reference just any Unicode character within any HTML document of any charset by using the decimal numeric character reference `&#12345`; as in:

```
putwchar(c)
{
    if (c < 0x80 && c != '&' && c != '<') putchar(c);
    else printf ("%#d", c);
}
```

Decimal numbers for Unicode characters are also used in Windows NT's Alt-12345 input method but are still of so little mnemonic value that a hexadecimal alternative `&#x1bc`; is being supported by the newer standards HTML 4.0 and XML 1.0. Apart from that, hexadecimal numbers aren't that easy to memorize either. SGML has long allowed symbolic character entities for some character references like `&acute`; for é and `&euro`; for the € but the table of supported entities differs from browser to browser.