Getting Ready for Unicode:

A Day In The Life Of an Open Source Developer

David Eger

Georgia Institute Tech.
eger@theboonies.us

Rényi Institute,
Hungarian Academy of Sciences
Budapest, 1364. Pf. 127.
Advisers: Ms. Andréka Hajnal
Mr. István Németi

I came to Budapest with high hopes of using mathematics–universal algebra, lattice theory, and logic–to better handle software development. Many of the problems of software development aren't really so hard, though. From getting software to "do the right thing" with internationalized text to getting software talk to hardware, most software issues are basically engineering problems: decipher an existing system and design a solution. I solve software problems in the context of the Open Source Community. The projects that excite me are ones that solve problems for people. Because Open Source Software is in large part a volunteer effort, there never seems to be much shortage of problems to solve. Today I will talk about one such problem: the transition to Unicode. But before I begin, I'll try to explain a little bit about this community and why I get so much from my interactions with the people there.

The Free and Open Source Software Community

The Free and Open Source Software Community is rooted in the desire to help people. It has its roots as far back as the 1970s, when users of Digital computers decided to make a common library of programs they had written that anyone could check out and use. Computers cost millions of dollars, and there was no such thing as shrink-wrapped software. IBM or Digital Equipment provided a machine and perhaps the basic operating system, and from there on you needed a team of expert programmers working full-time to get any good use out of your machine. Creating

a common library helped everyone. Instead of having to rewrite all of the software in-house, you could send $5 to the user's library and receive a tape in the mail containing any program they had. For the researchers and teachers with access to the few computers in the world, this was a godsend.

Of course, software is always troublesome. There are bugs. There are things that a program does that don't quite fit your needs. With the Digital user's library, you could make improvements on the software and then distribute your copy so others could benefit.

At its most basic level, the Free Software community is based on this idea: that you can make software better and help your friends by giving it to them. But on another level, open source means empowering the community as a whole with open standards.

For an example of a closed standard, let's take Microsoft Word. Microsoft Word is a good program: it makes it easy to write and format documents, and to check your spelling and grammar. However, Microsoft doesn't publish the Word file format. This means that if David Chapman uses Microsoft Word, and he decides to send me a copy of a play he is writing, then I can't read it. My options are something like these:

I could try to save up enough forints to buy a copy of Word.

I have several nice text editors and word processors on my system. I don't really want to buy Word, I just want to read my friend's play.

I could ask David for a copy of Microsoft Word. They tell me that David is not supposed to give me a copy of Word. Something about copyright law?

I could e-mail David and ask him to please, send me a copy in a published format (e.g., PDF, plain 7-bit ASCII or Unicode text, Rich Text Format, HTML, XHTML).

This is a pretty good option. If there is a published reference for a file format, then it's not too difficult to find a program for my computer that can read it. And if no such program exists, I can write one. All of the above formats I can easily read on my computer. Nonetheless, David is a busy person, and it would really be best if his software just saved his documents in a standards-compliant format by default. I don't like bothering David about technicalities of his computer. He writes *plays*: it's *literature* that's important to David, not technical issues.

> *I could hope that a dozen open source developers reverse-engineer the new Word format every time Microsoft publishes a new version of Office, and then write good import filters for OpenOffice and AbiWord.*
>
> Because so many people use Word, this actually happens. Nonetheless, this reverse-engineering is an error-prone and time-consuming task. It means making lots of guesses about what the data in a .DOC file actually means, because there is no published reference where you can look it up. So the conversion is always a bit rough.

The standards that make up the Internet work only because they are open. Proprietary infrastructure is easy to create, but it only hinders communication.

The core of the Internet is based on a long list of open standards that everyone can read and work from: IP, TCP, HTTP, XML, HTML, FTP, SSH, and SSL to name a few. These standards arose in an environment of intellectual freedom, openness, and co-operation. Ideas for how to make computer networks work were posted as "Request For Comments" in a community mostly composed of university researchers. Today, those who continue to develop the Internet stan-

dards interact in public organizations, among them the Internet Engineering Task Force, the World Wide Web Consortium (W3C), and the Unicode Consortium.

The Internet and open source have grown up together. The people writing the Internet protocols published their source code so everyone could benefit: so they could get comments on their ideas and implementation, and so everyone could have the software to connect their computers together. As the years went by, hardware vendor would attend conferences like Network InterOp where they would show off how well their equipment worked, even with the equipment of other vendors.

The traditions of openness and community continue today in the Free and Open Source Software Community. Organizations like The Open Group and the IEEE POSIX committee serve as forums where standards can be ironed out and interoperability between systems improved.

Users of GNU/Linux software[154] worldwide have local Users' Groups where they help newcomers solve issues they have with their software. Open Source developers from Perth to Brussells communicate over mailing lists to solve problems and develop new software.

For me, Linux provides a powerful platform for software development and research. I always find interesting problems to solve. But more than that, I find a community that motivates me and makes the world seem smaller. I work with people all over the world: Benjamin Herrenschmidt in Australia, Geert Uytterhoeven in Belgium, and Michel Dänzer in Switzerland. Benjamin does a lot of work to make sure Linux runs on my laptop, so I ask him what I can do. This Spring, that consisted of writing the acceleration functions of the radeonfb framebuffer driver. It was a fun project, and now my code is in the mainline Linux Kernel, used by all Linux users who use an ATI Radeon video card.

Occasionally, Open Source Developers go to conferences where they can meet face-to-face, have a beer, talk about new projects, and figure out what to work on next. Actually seeing thousands of members of the community all gathered in one place is an exciting experience. Everyone is friendly; people are excited about working together to make a future where everyone has a good base of software that lets them be creative and lets them share what they do with others.

Now that I've given an overview of the community I work in, let's look at a specific technical issue that affects everyone using a computer today: Unicode.

Getting Ready For Unicode

My friend Michel Dänzer has a funny-looking character in his name. If you look at his email address, it's spelled differently: michel@daenzer.net. Software has been slow to converge on a standard for non-English text, and especially so on UNIX[155]. It has been traditionally difficult to send e-mail, read web pages, and share documents with accented characters. The Hungarian o (?) is exceptionally rare to find in electronic correspondence. In e-mail, most Americans will default to writing Hungarian with simple umlauts or without accents at all. To figure out why sending that ? is so difficult, we will first have to take a look at how text is encoded in computers.

154 GNU/Linux: a popular open source version of the UNIX operating system. There are many other versions of UNIX around today: AIX, HP-UX, NetBSD, and FreeBSD to name a few.

155 UNIX: A 30-year old Operating System, still alive and kicking. Why? Keep on Reading.

How Characters are Encoded in Computers

Number of Bits	Possible Values
8	0-255
16	0-65, 535
32	0-4, 294, 967, 295

Digital computers are constructed from millions of microscopic devices that are very good at remembering whether they are in one of two states: variously called off/on, low/high, or zero/one. The unit of information that signifies which of these two states the device is in is called a bit. Groups of eight bits are called bytes, and bytes are typically the smallest addressable unit of a computer's memory. In a sense, strings of binary are the only real information that a computer understands?all graphics, documents, and music are internally represented as binary numbers.

To encode text, designers of computer systems have traditinally placed one character in each byte. The mapping from the numeric value in a byte to the represented character is called the *character set* (charset) or *code page* in use. Almost all computers use some extension of the 7-bit ASCII character set (the main exception being EBCDIC-encoded text on IBM mainframes). ASCII reserves thirty-three of the 128 values (0x00-0x1F, 0x7F) for non-printing control characters and maps the rest of the values (0x20-0x7E) as follows:

The Printable 7-bit ASCII Character Set

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

ASCII contains the fifty-two latin characters, ten numerals, and assorted punctuation: in short, what you need for an English-language only shop. So what about Hungarian? Well, since there are eight bits in a byte, there are 128 values totally untouched by the ASCII standard: those bytes whose top bit is 1. Computer producers targetting locales needing just a few extra characters (not Chinese) typically have kept the simplicity of having one byte per character, and assigned the extra needed symbols?like the Hungarian ??to the range 0x80-0xFF.

Problem solved, no? Computers running with one of these character sets work quite well. The IBM-made cash registers at your local SMATCH display Oké tejföl 249 Ft quite nicely. In fact, this solution was *so popular* that OEMs couldn't design encodings fast enough; so we now have *several* eight-bit codings each for Russia, Central Europe, the United States, and Israel. So if you're in central europe using Windows 95 to open a text file with directions to *H?sök tér* (sent to you by your friend, who used her computer's default encoding, ISO Latin-2), the text dis-

ISO-8859-1 (Latin-1)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
xa		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
xb	°	±	²	³	´	µ	¶	•	¸	¹	º	»	¼	½	¾	¿
xc	À	Á	Â	Ä	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
xd	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	ø	Ù	Ú	Û	Ü	Ý	Þ	ß
xe	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
xf	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

played on your screen will likely be *H§s÷k tÚr* (the rendering of said bytes with the Central European DOS codepage). With the default Linux console font, my (ISO Latin-1) address book entry for *Rónyai Lajos* at *Lágymányosi u. 11.* shows up as *R≤nyai* at *Lßgymßnyosi u. 11.* (codepage 437). What a mess!

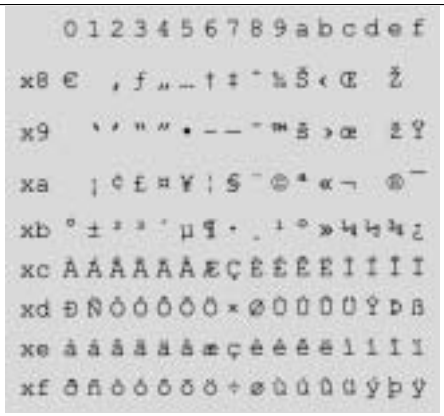
The worst part about this situation is that it is impossible to determine, simply from the data in a file which character set was used to encode it. To a computer, the file is just a rather long sequence of values each of which is between 0 and 255. One attempt to solve this problem would be to label every file with the character set used to make it. Such charset tags are a part of the MIME e-mail standard and the HTTP and HTML standards. Nonetheless, even if everyone agrees to label all of their files with charset information, what happens when you want to write a document that uses both Cyrillic *and* Hungarian?

To solve this problems, two standards bodies–The International Organization for Standardization (ISO), and the Unicode Porject–came together to create the Universal Character Set: ISO Standard 10646. ISO-10646 is defined as a 31-bit character set. The subset of ISO-10646 used for living languages is called the Basic Multilingual Plane and uses only the space from U+0000-U+FFFF. This notation: U+XXXX is a reference to what the Unicode standard calls a

ISO-8859-1 (Latin-2)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
xa		Ą	˘	Ł	¤	Ľ	Ś	§	¨	Š	Ş	Ť	Ź		Ž	Ż
xb	°	ą	˛	ł	´	ľ	ś	ˇ	¸	š	ş	ť	ź	˝	ž	ż
xc	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
xd	Đ	Ń	Ň	Ó	Ô	Ő	Ö	x	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
xe	ŕ	á	â	ă	ä	í	ć	ç	č	é	ę	ë	ě	í	î	ď
xf	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	˙

Windows 1252



codepoint: the numeric value associated with a certain unicode character. The number portion of such a reference is written in hexadecimal.

In Unicode, the first 256 codepoints agree with the ISO Latin-1 charset, leaving the 7-bit ASCII subset as-is U+0000-U+007F and leaving the 32 code positions U+0080-U+009F specifically unassigned. Above that, you find for instance the Hungarian ? at U+0151, Cyrillic from U+0400-U+052F, Arabic from U+0600-06FF, general punctuation from U+2000-U+206F and Katakana from U+30A0-U+30FF

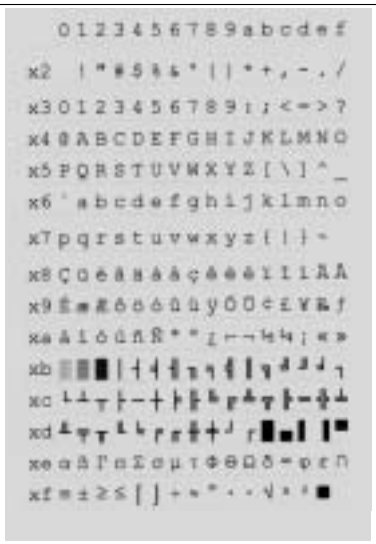
Beyond being just a big character table, the Unicode Standard specifies levels of support wherein implementations are required to perform increasingly complex typesetting foo, as complicated scripts require complicated typesetting: compositional rendering is essential for languages like Thai which need nontrivial composition of diacritical marks, and contextual shaping is needed for the proper rendering of Indic and Arabic scripts.

Making It Happen: Some Software Required

Having a big, thick standard to bring the world together is a wonderful thing. Actually having the world come together, though, is a bit tougher. To make it happen, several stacks of software need to be written: a stack to translate peoples' keystrokes to the proper unicode characters, another stack of software to handle the rendering of said text on a screen or printer, and enough glue to make sure people still can communicate with people who are still running yesterday's software.

KOI8-R

Codepage 437 (OEM US)



To explain some of the issues that arise, I'll need to digress a bit and talk about software development on Linux, a free and popular UNIX clone.

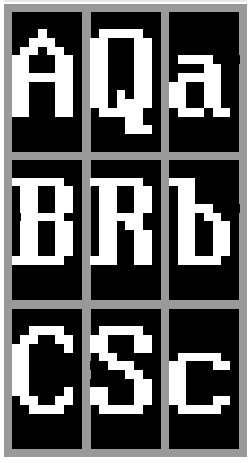
Where Have All the Computer Scientists Gone? The World of UNIX

Unix and C are the ultimate computer viruses.
–Richard Gabriel, *The Rise of "Worse is Better"*

In terms of Operating Systems, the UNIX family is ancient, dating from 1969 at Bell Labs. UNIX is a programming environment written by and for developers who wanted to get a moderately powerful programming environment with a minimal amount of effort. In software development one often finds that the last 20% takes 80% of the effort. The fathers of UNIX watched an attempt to create The Perfect Operating System™ die a painful crib death, so they decided to write instead a system that took 10% of the effort and yielded 50% of the value. To begin with, they designed a programming language called C. C is little more than portable assembly language; but, due to its simplicity, flexibility, and marriage with UNIX, it still dominates much of today's software development.

Also in the vein of keeping it simple, fonts on UNIX were implemented trivially as large bitmap tables. Using a bitmapped font means the glyph for each character in the font is the same size, say eight by sixteen pixels. This setup makes text rendering algorithms trivially simple to write. The letters on the screen form a grid, and it's easy to refer to text by line and column. On UNIX, users typically interact with the computer through a little window of text which runs a program emulating the terminals of early computing. A program called a "shell" runs on top of the terminal emulator, continually asking the user to enter new commands.

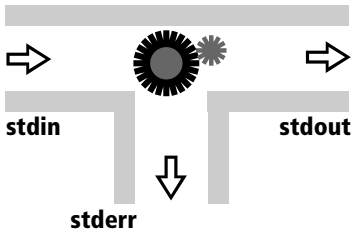
Part of an 8x16 Bitmap Font



All of this seems horribly primitive; rendering text well seems the last thing that this system is built for. It is. Nonetheless, UNIX does make a very nice system for programming (more on this later). Further, because the implementation is *so trivial*, it is easy to port it to a new machine; and because it is *good enough*, and

because creating a fundamentally better system is so hard, it is still the favourite of academics and programmers today. Nonetheless, the shortcomings of UNIX do get to its users, so little-by-little they improve it. The things that are interesting to computer scientists – like designing packet-switched network protocols–usually get implemented first on UNIX. The things that are interesting to most normal computer users–like high quality interactive typesetting, i.e., WYSIWYG editors–usually get implemented first on other systems.

The UNIX Information Flow Architecture



So why do software developers like UNIX so much? In short, Doug McIlroy's concept of pipes. (Almost) all of the files on UNIX are plain text, and (almost) all of the basic UNIX tools create or filter streams of text. When you put these together, you get a very powerful information flow architecture. This architecture is embedded into the C programming language: each program has a byte stream for input (stdin), a byte stream for output (stdout), and a byte stream it can use to tell the user something unexpected has happened (stderr).

The beauty in this setup really comes out when an experienced UNIX guru sits down to do some text manipulation. Instead of writing a long and complicated program in C, he types a few lines at his shell and suddenly has the output he wants. He can do this in large part because the tools fit together so well: the output of one command (filter) can be "pipe"d directly into the input of the next command.

To show an example, you first need to know what some of these UNIX tools do.

seq - output a sequence of (decimal) numbers
sort - sort lines of text
wc - word count; count the characters, words and lines of a file
uniq - drop duplicated lines of text
printf - print data according to a given format string
grep - read a set of files and print out lines matching a given pattern
awk - break up lines into pieces separated by whitespace
tr - perform substitution of one set of characters for another throughout the text
sed - perform simple editing tasks on text, such as, search and replace
2unicode - translate code point numbers to their UTF-8 encoded form (more on this later)
iconv - convert text from one charset to another
lynx - a text based web browser
telnet - open a pipe with a remote machine

Let's look at some examples of pipelines in use. Ildikó tells me that this paper ought to be between 20,000 and 40,000 characters. I would like to know how much more I ought to write, so I try the following command:

```
$ wc day_in_the_life.html
```

The "$" above is my shell's prompt for a new command. day_in_the_life.html is the name of this file. I receive in return the following output: 710 4769 46212

46,212 characters seem a bit high. Then I realize: HTML has a lot of extra characters that are just formatting. So to get a better estimate, I ask lynx to convert the file to just text; and give its output to wc:

```
$ lynx -dump day_in_the_life.html | wc
 413 3298 21271
```

This is much closer to how far we are through the report. The vertical bar "|" is the special character that tells the shell to take the output of lynx and give it to wc.

Let's look now at a slightly longer example: figuring out how many different charsets are used for e-mail. For this, I will take a look in my INBOX and ask what character sets the e-mails I've received over the last few years have come encoded as:

```
$ grep "charset=" /var/spool/mail/eger | grep -v
    "=3D" \
| tr "[A-Z]; \t<>" "[a-z]\n\n\n\n\n" \
| grep charset | tr -d "\"" | sort | uniq -c | sort -n
2 charset=iso-8859-15
2 charset=koi8-r
2 charset=unicode-1-1-utf-7
2 charset=windows-1251
6 charset=x-unknown
8 charset=iso-8859-2
46 charset=windows-1252
87 charset=utf-8
1362 charset=iso-8859-1
2194 charset=us-ascii
```

To get a better feel for how the above does its magic, take a look at Appendix B.

Technical Issues: Complications of Putting Unicode into Practice

Several factors conspire to create difficulty in implementing Unicode. The first such difficulty is that UNIX pipes were *so* popular that many of the basic Internet protocols were based around the following concept: a UNIX user opens a pipe to the email or ftp or web server and starts typing a few commands to it, interactively. For example, to send an email (taken straight from RFC 822) a user would interact as follows:

```
R: 220 BBN-UNIX.ARPA Simple Mail Transfer Ser-
    vice Ready
S: HELO USC-ISIF.ARPA
R: 250 BBN-UNIX.ARPA

S: MAIL FROM:<Smith@USC-ISIF.ARPA>
R: 250 OK

S: RCPT TO:<Jones@BBN-UNIX.ARPA>
R: 250 OK

S: DATA
R: 354 Start mail input; end with <CRLF>.<CRLF>
S: Hey Ron,
S:
S: Are you free for lunch tuesday at MerriMac's?
S:
S: -Joe
S: .
R: 250 OK

S: QUIT
R: 221 BBN-UNIX.ARPA Service closing transmis-
    sion channel
```

The first thing to note is that the end of an email is indicated by a period on a line by itself. This means that the five byte sequence 0d 0a 2e 0d 0a is never allowed to occur in the body of an internet message. If you try to type the sequence <CRLF>.<CRLF> in most

e-mail programs, the e-mail program will insert a space (ASCII 0x20) in front of the period.

Further, since these protocols were designed so you could type into them directly, any local character set that didn't match on the remote machine would appear differently; so all of the base protocol (for example the line FROM: <Smith@UCF-ISIF.ARPA>) have to be done in the character set that everyone agrees on: ASCII. This is one of the many reasons that Michel can only have michel@daenzer.net and not michel@dänzer.net as his e-mail address.

The limitations of the Internet protocols aren't, however, the biggest obstacles to putting Unicode into use on UNIX and the Internet, but it does reinforce that, with text you send over the Internet, either you *will* have to restrict your character set via some standard transformation (base64, uuencode, MIME extensions etc.) *or* be careful that the text you send doesn't contain any of the byte sequences which would otherwise trip-up the software in use today.

The UNIX World Decides UTF-8 is the Only Sane Way to Deal with a Multilingual World

"UTF-8 just happens to be the only sane policy for encoding complex characters into a byte stream." – Linus Torvalds, 16 Feb 2004

The biggest problem UNIX users have in transitioning to Unicode isn't that it will break the Internet protocols (there have been tricks to send non-ASCII files over the Internet for years, though the proper way to do this "natively" is under debate). The biggest problems are those centered around a number of invariants that UNIX users have always counted on: strings end with a zero-byte, a byte is a character, data is processed as a byte stream, and the amount of space a string will take on the screen is the same as the number of bytes it occupies in memory.

The Windows community made the transition to Unicode with Windows NT and Windows 2000 by deciding that all characters would simply be sixteen-bit numbers corresponding to Unicode codepoints. This encoding of Unicode is called UCS2. On the one hand, most of the algorithms remain the same, though the basic data-type changes:

```
int strlen(char *s)
{
  int i=0;
  while (*s++) { i++; }
  return i;
}
```

Old Code

```
int strlen(wchar_t *s)
{
  int i=0;
  while (*s++) { i++; }
  return i;
}
```

New Code

On the other hand, if you send new strings through old code, everything breaks. Therefore, making this change to 16-bit characters requires a "flag-day" type transition: everyone with relevant software has to change all of their software at the same time, or else things stop working. Old windows programs all had to be replaced at once with new ones, or even the most trivial programs would fail miser-

```
ASCII: 57 68 61 74 20 72 65 73 75 6d 65 3f 00          |What resume?|
UTF-8: 57 68 61 74 20 72 65 73 75 6d c3 a9 3f 00       |What resum.é?|
UCS2:  00 57 00 68 00 61 00 74 00 20 00 72 00 65 00 73 |.W.h.a.t. .r.e.s|
       00 75 00 6d 00 e9 00 3f 00 00                   |.u.m.é.?|
UCS4:  00 00 00 57 00 00 00 68 00 00 00 61 00 00 00 74 |...W...h...a...t|
       00 00 00 20 00 00 00 72 00 00 00 65 00 00 00 73 |... ...r...e...s|
       00 00 00 75 00 00 00 6d 00 00 00 e9 00 00 00 3f |...u...m...é...?|
       00 00 00 00
```

ably. There's also the issue that ASCII strings take up twice as much memory in the UCS2 encoding. Other popular Unicode encodings include UCS4 (four bytes per character) and UTF-8 (an encoding where characters take variable number of bytes to store).

Unfortunately for the Windows developers, after they decided to go with UCS2, the Unicode standards bodies decided that Unicode would be larger than 16 bits to make room for historical scripts (Aztec, Mayan, and Egyptian heiroglyphics, Aramaic, Old Persian, etc.), extended asian scripts, and artificially created alphabetic scripts like Shavian. Though not terribly important for most people, if Windows users ever want to name a file in Aramaic, they will have to wait for the Microsoft to have yet another flag day.

The UNIX community decided not to use UCS2: everything down to the defintion of the C programming language and the concept of a pipe assume that strings are zero-terminated sequences of bytes. Changing such a fundamental notion with the UNIX crowd isn't politically feasible, and would come too with a technical cost: the byte-oriented nature of UNIX in large part shields it from endian issues. The political infeasibility derives from the fact that there is no central command structure to the UNIX world as there is for the Windows world.

To bridge the technical and political gaps for the UNIX community, The Open Group called upon two of Bell Labs' gurus–Ken Thompson and Rob Pike–to find a solution everyone could live with. What they came up with, on a placemat in a New Jersey diner, is now known as the UTF-8 encoding of Unicode, detailed in Annex P of ISO-10646.

Unicode Codepoints	UTF-8 Encoded form
0x00000000 - 0x0000007F:	0xxxxxxx
0x00000080 - 0x000007FF:	110xxxxx 10xxxxxx
0x00000800 - 0x0000FFFF:	1110xxxx 10xxxxxx 10xxxxxx
0x00010000 - 0x001FFFFF:	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0x00200000 - 0x03FFFFFF:	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0x04000000 - 0x7FFFFFFF:	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

The UTF-8 Encoding of Unicode

The advantages of the UTF-8 are many:

UTF-8 is backwards compatible with 7-bit ASCII
UTF-8 is byte-stream oriented (no endian problems)
The entire 31-bit Unicode Character Set can be encoded in UTF-8
Any 7-bit ASCII byte will appear in UTF-8 only as itself
The C string library functions continue to "just work" on UTF-8 strings
Resynchronizing a UTF-8 byte-stream is trivial

The main disadvantage is that UTF-8 *is* a multibyte encoding: one codepoint may take between one and six bytes to encode, so programs that have to understand non-ASCII now have to constantly transform the byte stream to and from 16- or 32-bit Unicode. Nonetheless, old programs which look for the byte (ASCII 2F: '/') won't be confused by the presence of Kangxi Radicals (2F00-2FEF). Similarly, the end of e-mail marker 0d 0a 2e 0d 0a will only ever occur as itself in UTF-8 text.

Getting the World on the Same Page, One Step at a Time

Of course, agreeing on the Unicode standard and upon an encoding of Unicode are just the first steps towards bringing the world together with internationalized text. Developers have to be convinced that using simple 8-bit encodings is no longer acceptable for a multilingual world; software has to be readied for the new standards, and documents have to be converted. Text editors have to be told what internal encoding to use for text: that backspace means "delete the previous character," not "erase the previous byte." Web browsers have to be taught to identify charset tags in HTML and choose their fonts accordingly.

All of this work means that there lots of places that anyone can help. I got motivated to help fix Unicode-related issues when I noticed that the new version of X[156] made it easy to type non-ASCII characters: when I pressed AltGr-Shift-A, the letter 'á' appeared in my window.

Assuming all was well, I changed the text in my documents and web pages with these accented characters and looked up the W3C standards to make sure my web pages would show up properly to other people. Soon, however, I noticed that my documents showed up differently under X then they did at the console. After much reading, I amassed the knowledge that I've just shared with you about character sets and Unicode, and determined that the 'á' I saw was unfortunately an ISO-8859-1 encoded 'á' and not the UTF-8 encoded 'á'.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
 <head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <meta http-equiv="content-language" content="en-us" />
  <title>My Title Here</title>
 </head>
```

156 X/X Windows/X11/X11R6: This is the most common Graphical User Interface for UNIX

These W3C XHTML headers to tell the world that your web page is encoded in UTF-8

To fix my mistake, I read the standards and wrote a program which you'll find in Appendix C: utf8lint. utf8lint will process text files, ensuring that they are properly encoded in UTF-8, so when you share them with friends they will see what you intended and not gibberish.

While I was cleaning up my own files, I ran my filter on the Linux Kernel Sources: for if the most publicisized Open Source project isn't keeping to UTF-8, who will be? What I found was text encoded mostly in 7-bit ASCII, but also portions in ISO Latin-1, EUC-JP, and ISO-2022-JP. Almost all of the non-ASCII characters were accents on people's names. Occasionally there were non-English comments (as the Japanese in the excerpt above), and a few instances were uses of the

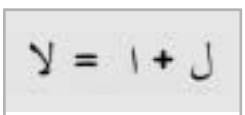
```
#ifndef CONFIG_ROM_KERNEL
        /* The early chip we have is buggy, and writing the interrupt
           vectors into low RAM may screw up, so for non-ROM kernels, we
           only rely on the reset vector being downloaded, and copy the
           rest of the interrupt vectors into place here.  The specific bug
           is that writing address N, where (N & 0x10) == 0x10, will _also_
           write to address (N - 0x10).  We avoid this (effectively) by
           writing in 16-byte chunks backwards from the end.  */

        AS85EP1_IRAMM = 0x3;    /* 内蔵命令RAMは [ write-mode ] になります */

        src = (u32 *)(((u32)&_intv_copy_src_end - 1) & ~0xF);
```

linux/arch/v850/kernel/as85ep1.c : *Software development isn't just in English any more*

degrees and micro symbols. After talking to the authors and determining the original character sets, I submitted four patches to bring the encodings of 276 files in line with UTF-8: clean-up takes place one file at a time.

Now For... Actually Looking at the Text

After fixing your data formats, your programs, and your data, there is still the non-trivial issue of rendering multilingual text. This takes several passes, the first to identify and tag each of the scripts used (Hebrew, Arabic, Latin, Japanese, Indic, etc.) and determine whether each of them is written right-to-left or left-to-right. After that one must choose an appropriate font: it's unusual, for instance, for one font to cover both Japanese and Devanagari. Finally, one must have program adequate to render the script. For a Latin script, rendering ligatures is nice but not necessary: bitmapped fonts will do. This is not the case for scripts like Arabic, which need more advanced rendering algorithms.



An Arabic Ligature

Delving into the details of rendering multilingual text is beyond the scope of this talk, however. For more information you can look into the web sites of the Pango Project at http://www.pango.org/ and SIL International at http://www.sil.org/.

Open Source Development, or Having an Itch and Scratching It

The information above about Unicode and UTF-8 I tracked down and present here because I was scratching an itch: I wanted to be able to type Hungarian or French without resorting to TEX or similarly specialized tools. For the most part, Windows *has* had this supported for three years, and it's a bit of an embarrassment for the Linux community that support is still lagging. Nonetheless, the community is made better by each person doing his part: it's a different model from paying a company to do the work for you. I find diving into technical details, and interacting with other developers quite fulfilling; and I hope the software I make is useful to the community as a whole.

References / Further Reading:

The Internet Standards, developed by The Internet Engineering Taskforce. All the information you need to make the Internet work, including a RFC archive and contact information for many of the people who have made modern communication possible. Here is a small selection of the RFCs relevant to this talk

RFC 1345: Character Mnemonics & Character Sets
RFC 2821: Simple Mail Transfer Protocol
RFC 2822: Internet Message Format
RFC 2045: MIME (part one): Format of Internet Message Bodies

Gabriel, Richard. The Rise of "Worse is Better". http://www.jwz.org/doc/worse-is-better.html

Hall, Jon "maddog". "Stories of the Early Days of Open Source: FOSDEM 2004 closing speech"

Kuhn, Markus. The Unicode and UTF-8 FAQ. http://www.cl.cam.ac.uk/~mgk25/unicode.html

Pike, Rob. "UTF-8 History". 30 April 2003. http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt

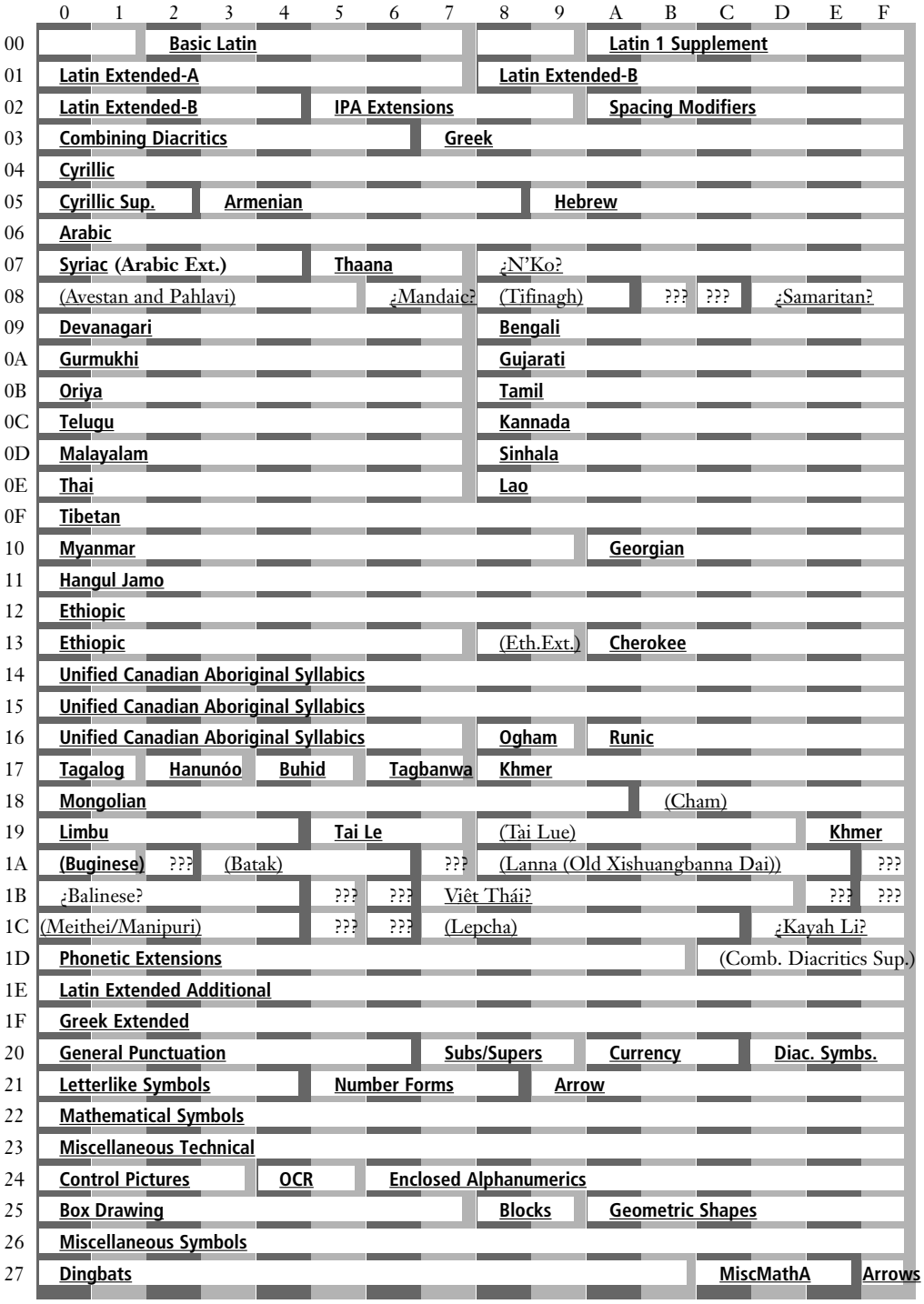
SIL International. Examples of Complex Rendering

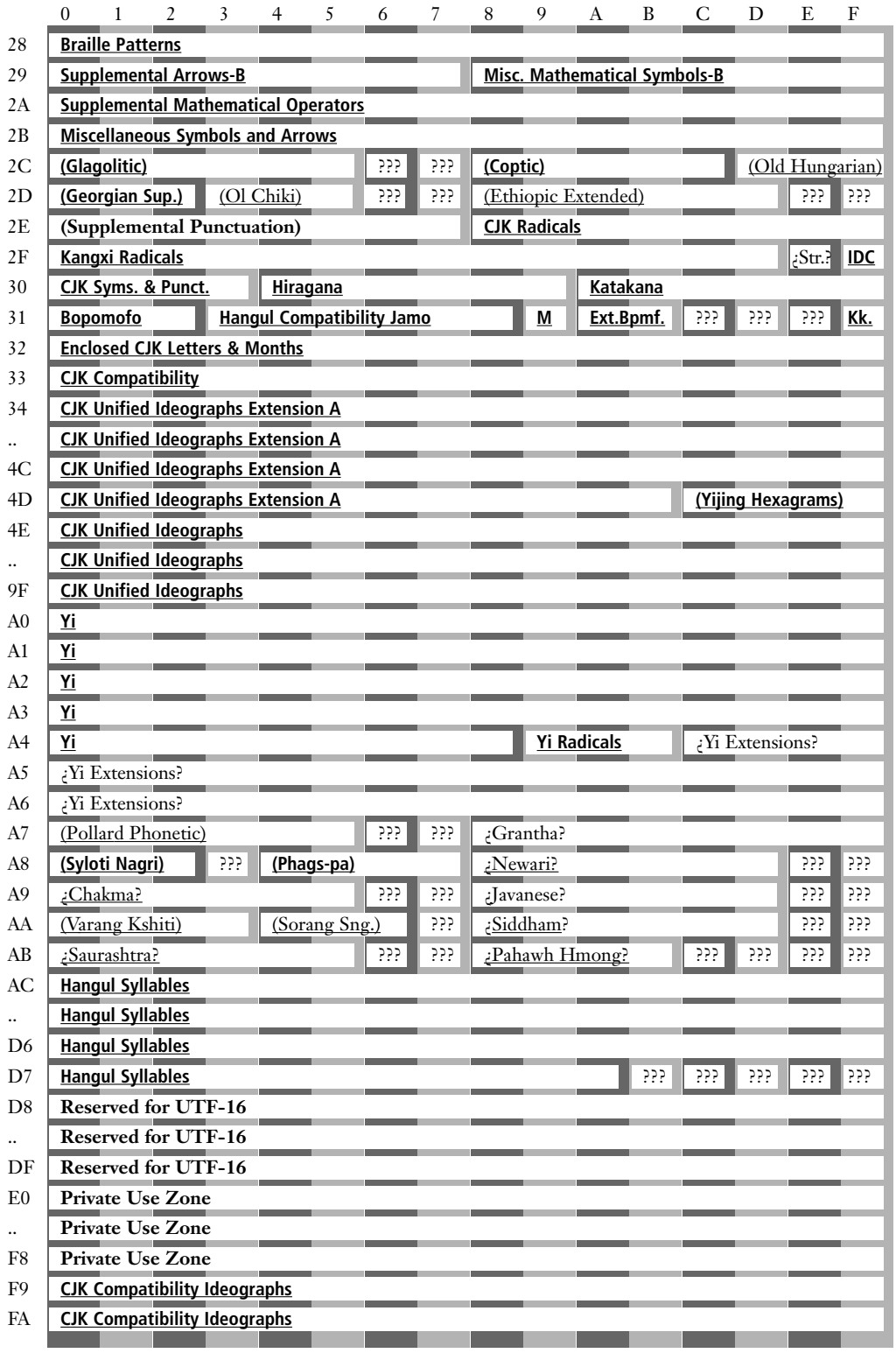
Thompson, Ken, Dennis Ritchie, Doug McIlroy. "The Creation of the UNIX Operating System" http://www.bell-labs.com/history/unix/

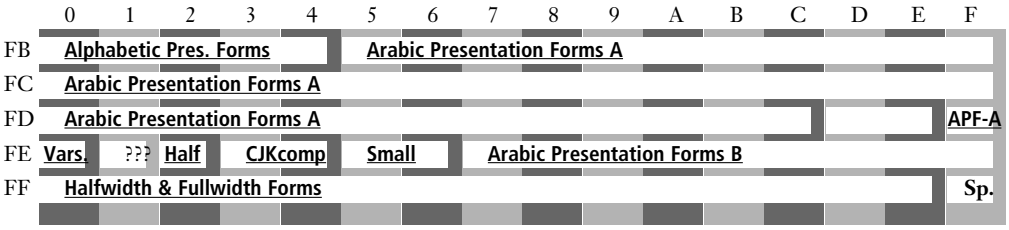
The Unicode Consortium. "The Unicode Standard, Version 4.0" Addison-Wesley, 2003. and http://www.unicode.org/

Appendix A:

The Unicode Basic Multilingual Plane, and Some Sample Code Charts




0 1 2 3 4 5 6 7 8 9 A B C D E F
28 Braille Patterns
29 Supplemental Arrows-B Misc. Mathematical Symbols-B
2A Supplemental Mathematical Operators
2B Miscellaneous Symbols and Arrows
2C (Glagolitic) ??? ??? (Coptic) (Old Hungarian)
2D (Georgian Sup.) (Ol Chiki) ??? ??? (Ethiopic Extended) ??? ???
2E (Supplemental Punctuation) CJK Radicals
2F Kangxi Radicals ¿Str.? IDC
30 CJK Syms. & Punct. Hiragana Katakana
31 Bopomofo Hangul Compatibility Jamo M Ext.Bpmf. ??? ??? ??? Kk.
32 Enclosed CJK Letters & Months
33 CJK Compatibility
34 CJK Unified Ideographs Extension A
.. CJK Unified Ideographs Extension A
4C CJK Unified Ideographs Extension A
4D CJK Unified Ideographs Extension A (Yijing Hexagrams)
4E CJK Unified Ideographs
.. CJK Unified Ideographs
9F CJK Unified Ideographs
A0 Yi
A1 Yi
A2 Yi
A3 Yi
A4 Yi Yi Radicals ¿Yi Extensions?
A5 ¿Yi Extensions?
A6 ¿Yi Extensions?
A7 (Pollard Phonetic) ??? ??? ¿Grantha?
A8 (Syloti Nagri) ??? (Phags-pa) ¿Newari? ??? ???
A9 ¿Chakma? ??? ??? ¿Javanese? ??? ???
AA (Varang Kshiti) (Sorang Sng.) ??? ¿Siddham? ??? ???
AB ¿Saurashtra? ??? ??? ¿Pahawh Hmong? ??? ??? ??? ???
AC Hangul Syllables
.. Hangul Syllables
D6 Hangul Syllables
D7 Hangul Syllables ??? ??? ??? ??? ???
D8 Reserved for UTF-16
.. Reserved for UTF-16
DF Reserved for UTF-16
E0 Private Use Zone
.. Private Use Zone
F8 Private Use Zone
F9 CJK Compatibility Ideographs
FA CJK Compatibility Ideographs



Appendix B:

Extracting Information From Your Inbox on UNIX

As an example of the sort of power that UNIX's pipeline abstraction gives to the user, let's examine a script that reads and summarizes the character sets used in the e-mail messages of my inbox.

```
$ grep "charset=" /var/spool/mail/eger | grep -v "=3D"
    | tr "[A-Z]; \t<>" "[a-z]\n\n\n\n\n" \
    | grep charset | tr -d "\"" | sort | uniq -c | sort -n
    2 charset=iso-8859-15
    2 charset=koi8-r
    2 charset=unicode-1-1-utf-7
    2 charset=windows-1251
    6 charset=x-unknown
    8 charset=iso-8859-2
   46 charset=windows-1252
   87 charset=utf-8
 1362 charset=iso-8859-1
 2194 charset=us-ascii
```

My inbox is in the common UNIX mbox format: a big text file named /var/spool/-mail/eger; mail messages are delivered by simply appending them to the end of this file. According to RFC 2046 (one of the documents that specify the workings of e-mail over the internet), the character set used for a MIME-compliant e-mail has to be specified in the Content-Type: message header, as in:

```
Content-type: text/plain; charset=iso-8859-1
```

Let's take a snapshot of the output of the first command in our pipeline and see how the stream of text is transformed by each filter:

```
grep "charset=" /var/spool/mail/eger
Content-Type: text/plain; charset=us-ascii
Content-Type: text/x-vcard; charset=us-ascii;
        charset="iso-8859-1"
Content-Type: text/plain; charset=us-ascii
Content-Type: text/x-vcard; charset=us-ascii;
        charset="iso-8859-1"
        charset="iso-8859-1"
Content-Type: text/plain; charset=us-ascii
        charset="iso-8859-1"
        charset="iso-8859-1"
charset=3Diso-8859-1">
        charset="iso-8859-1"
        charset="iso-8859-1"
Content-Type: text/plain; charset="us-ascii";
format=flowed
        charset="iso-8859-1"
Content-Type: TEXT/PLAIN; charset=US-ASCII
Content-Type: TEXT/PLAIN; charset=US-ASCII
        charset="iso-8859-1"
        charset="iso-8859-1"
charset=3Diso-8859-1">
        charset="iso-8859-1"
        charset="iso-8859-1"
<META content=3D"text/html; charset=3Diso-8859-1" =
Content-Type: text/plain; charset="us-ascii";
format=flowed
Content-Type: TEXT/PLAIN; charset=US-ASCII
Content-Type: text/plain; charset=us-ascii
Content-Type: text/x-vcard; charset=us-ascii;
        charset="iso-8859-1"
```

```
grep -v "=3D"
```

do a "reverse" grep *- print all the lines that do not contain the string "=3D"*

```
Content-Type: text/plain; charset=us-ascii
Content-Type: text/x-vcard; charset=us-ascii;
        charset="iso-8859-1"
Content-Type: text/plain; charset=us-ascii
Content-Type: text/x-vcard; charset=us-ascii;
        charset="iso-8859-1"
        charset="iso-8859-1"
Content-Type: text/plain; charset=us-ascii
        charset="iso-8859-1"
        charset="iso-8859-1"
        charset="iso-8859-1"
```

charset="iso-8859-1"
Content-Type: text/plain; charset="us-ascii";
format=flowed
charset="iso-8859-1"
Content-Type: TEXT/PLAIN; charset=US-ASCII
Content-Type: TEXT/PLAIN; charset=US-ASCII
charset="iso-8859-1"
charset="iso-8859-1"
charset="iso-8859-1"
charset="iso-8859-1"
Content-Type: text/plain; charset="us-ascii";
format=flowed
Content-Type: TEXT/PLAIN; charset=US-ASCII
Content-Type: text/plain; charset=us-ascii
Content-Type: text/x-vcard; charset=us-ascii;
charset="iso-8859-1"

tr "[A-Z]; \t<>" "[a-z]\n\n\n\n\n"

translate the strings to lower case, and split lines at semicolons, spaces, tabs, and less-than and greater-than symbols.

content-type:
text/plain

charset=us-ascii
content-type:
text/x-vcard

charset=us-ascii
charset="iso-8859-1"
content-type:
text/plain

charset=us-ascii
content-type:
text/x-vcard

charset=us-ascii

charset="iso-8859-1"

charset="iso-8859-1"
content-type:
text/plain

charset=us-ascii

charset="iso-8859-1"

charset="iso-8859-1"

charset="iso-8859-1"

charset="iso-8859-1"
content-type:
text/plain

charset="us-ascii"

format=flowed

charset="iso-8859-1"
content-type:
text/plain

charset=us-ascii
content-type:
text/plain

charset=us-ascii

charset="iso-8859-1"

charset="iso-8859-1"

charset="iso-8859-1"

charset="iso-8859-1"
content-type:
text/plain

charset="us-ascii"

format=flowed
content-type:
text/plain

charset=us-ascii
content-type:
text/plain

charset=us-ascii
content-type:
text/x-vcard

charset=us-ascii

charset="iso-8859-1"

grep charset

print only the lines containing the text "charset"

charset=us-ascii
charset=us-ascii
charset="iso-8859-1"
charset=us-ascii
charset=us-ascii
charset="iso-8859-1"
charset="iso-8859-1"
charset=us-ascii
charset="iso-8859-1"

charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“us-ascii“
charset=“iso-8859-1“
charset=us-ascii
charset=us-ascii
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“us-ascii“
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=“iso-8859-1“

grep charset

print only the lines containing the text “charset“

charset=us-ascii
charset=us-ascii
charset=“iso-8859-1“
charset=us-ascii
charset=us-ascii
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=us-ascii
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“us-ascii“
charset=“iso-8859-1“
charset=us-ascii
charset=us-ascii
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“iso-8859-1“
charset=“us-ascii“
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=“iso-8859-1“

tr -d “\“”

delete all of the quotes (yes, the quoting methods of the UNIX shell get very strange)

charset=us-ascii
charset=us-ascii
charset=iso-8859-1
charset=us-ascii
charset=us-ascii
charset=iso-8859-1
charset=iso-8859-1
charset=us-ascii
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=us-ascii
charset=iso-8859-1
charset=us-ascii
charset=us-ascii
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=iso-8859-1
sort
    charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=iso-8859-1
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
charset=us-ascii
uniq -c
delete duplicate lines, but print how many of a line occurred

    13 charset=iso-8859-1
    12 charset=us-ascii
sort -n
sort the lines in increasing numeric order

    13 charset=iso-8859-1
    12 charset=us-ascii

Appendix C:
utf8lint.c

```
/* read and canonicalize a UTF-8 text file from stdin;
 * transformations made:
 *  + convert too-long encodings to proper UTF-8
 *  + convert things that look like Latin-1 to UTF-8
 *  + give warnings about control characters
 * if there are problems, describe them to stderr;
 * write canonicalized form to stdout
 *
 * efficiency is not a concern in this code
 */
#include <stdio.h>
#include <stdarg.h>

typedef unsigned int u32;

#define MAX_PRINTED_ERRORS 20

void utf8log(int errors, FILE * fderr, const char * s, ...)
{
        va_list argp;
#if defined(QUELL_LOGGING)
        static int quelled = 0;
        if( errors >= MAX_PRINTED_ERRORS && !quelled) {
                fprintf(fderr, "There are more errors silently"
                        " being slogged through...\n");
                quelled = 1;
        }
        if ( errors >= MAX_PRINTED_ERRORS ) return;
#endif
        va_start(argp, s);
        vfprintf(fderr, s, argp);
        va_end(argp);
}


/* UTF-8 decoder action codes */
typedef enum {
        NOPROBLEM=0,
        DISCARD,  /* discard all read bytes */
        CONV_EACH,    /* for each read byte latin-1 => UTF-8 */
        PRINT_DECODED       /* print value derived by attempted
                        decoding of the sequence */
} utf8error_t;

#define SHORT_SEQ  CONV_EACH
#define LONG_SEQ   PRINT_DECODED

/* these arrays indexed by number of bytes in the UTF-8
sequence*/
static const u32 UTF8_ATLEAST[] =
                {       0,
                        0,
                  0x00000080,
                  0x00000800,
                  0x00010000,
                  0x00200000,
                  0x04000000 };

static const unsigned char UTF8_HIBITS[] =
        { '\x00',
          '\x00',
          '\xC0',
          '\xE0',
          '\xF0',
          '\xF8',
          '\xFC' };

static inline unsigned char utf8hiword_datamask(short int
nbytes) {
        return 0x7f & ((~(UTF8_HIBITS[nbytes]))>>1);
}

static inline short int utf8len(u32 c)
{
        short int bytes;
        for(bytes=6; bytes; bytes--)
                if( c >= UTF8_ATLEAST[bytes] ) {
                        return bytes;
                }
        return 1;
}

/* given the first byte of a UTF-8 sequence,
 * how long do we expect the sequence to be?
 * 0 means an illegal character */
static inline short int utf8encodedlen(char c)
{
        short int bytes = 6;
        char bytemask = UTF8_HIBITS[6];
        if ((c & '\xFE') == '\xFE' )  return 1;     /* Byte-Order Mark
*/
        if ((c < 0xC0) && (c >= 0x80)) return 0; /* yikes! */
```

```
        while ( (bytemask & c) != bytemask ) {
                bytes--;
                bytemask <<= 1;
        }
        return bytes;
}

/* encode a unicode 32-bit value into a buffer large enough to
 *      contain the encoded value + a null byte
 * to be safe, pass a 7-byte buffer
 */
void utf8encode(u32 c, char *buffer) {
        int bytes;
        u32 hibits;

        bytes  = utf8len(c);
        hibits = UTF8_HIBITS[bytes];
        buffer[bytes] = 0;
        while ( --bytes ) {
                buffer[bytes] = 0x80 | (0x3F & c);
                c >>= 6;
        }
        buffer[0] = hibits | c;
}

int utf8lint(FILE * fdin, FILE * fdout, FILE * fderr)
{
        char read_buffer[8];
        char reencode_buffer[8];

        /* file-global variables */
        int lines=0, chars=0;  /* error reporting counters */
        int non_ascii=0;     /* have we seen non 7-bit ASCII
bytes? */
        int errors=0;        /* how many errors during translation?
*/

        /* single character decoding variables */
        u32 actual_char;   /* what is the decoded 31-bit charac-
ter? */
        short int pos;     /* how many characters of this multi-byte
                              seq have we read? */

        int i, c;
        utf8error_t error_type;

        while( (c=getc(fdin)) != EOF ) {
                pos = 1;
                error_type = NOPROBLEM;
#define ERROR_OUT(code) do { error_type = code; goto
error_out; } while (0)

                if( c < 0x80 ) {
                        /* good old 7-bit ASCII */
                        if( c == '\n' ) { lines++; chars = 0; }
                        else if ( c < 0x20 && c != '\t' && c != '\f' && c
!= '\r') {
                                utf8log( 0, fderr, "Warning: control char-
acter 0x%02x at "
                                        "line %d, position %d\n", c, lines,
chars );
                        }
                        fputc(c, fdout);
                        chars++;
                        continue;
                } else {
                        short int mblen;
                        non_ascii = 1;
                        read_buffer[0] = c;
                        /* decode a multibyte sequence */
                        if ( ( mblen = utf8encodedlen(c) ) == 0 ) {
                                /* whoops... not a valid first char of a
multibyte */
                                actual_char = c;
                                ERROR_OUT(SHORT_SEQ);
                        } else {
                                actual_char =
(utf8hiword_datamask(mblen) & c);
                                while(pos < mblen) {
                                        c = getc(fdin);
                                        if ( c == EOF )
ERROR_OUT(SHORT_SEQ);
                                        if ( ( c & 0xC0) != 0x80 ) {
                                                ungetc( c, fdin );
                                                ERROR_OUT(SHORT_SEQ);
                                        }
                                        read_buffer[pos++] = c;
                                        actual_char = ((actual_char<<6) | (
c & 0x3F ));
                                }
                                /* check to see if the encoding is smallest-pos-
sible */
                                if ( actual_char < UTF8_ATLEAST[mblen]
) ERROR_OUT(LONG_SEQ);
                        }
                }
```

```
            if ( error_type == NOPROBLEM ) {
                /* successful verification of a UTF-8 sequence,
print it */
                read_buffer[pos]=0;
                fprintf(fdout, read_buffer);
                chars++;
                continue;
            }


error_out: /* error handling code */
                errors++;
                utf8log(errors, fderr, "Invalid byte sequence:");
                for( i = 0; i < pos; i++ )
                    utf8log(errors, fderr, " 0x%02x",
read_buffer[i]);

                switch(error_type) {
                case PRINT_DECODED:
                    utf8log(errors, fderr, " translating to
U+%04x", actual_char);
                    utf8encode(actual_char, reencode_buffer);
                    fprintf(fdout, reencode_buffer);
                    chars++;
                    break;
                case CONV_EACH:
                    utf8log(errors, fderr, " translating each
byte (assuming ISO Latin-1)");
                    for( i = 0; i < pos; i++ ) {
                        utf8encode(read_buffer[i], reen-
code_buffer);
                        fprintf(fdout, reencode_buffer);
                        chars++;
                    }
                    break;
                case DISCARD:
                default:
                    utf8log(errors, fderr, " discarding it all");
                    break;
                };
                utf8log(errors, fderr, " at line %d, char %d\n",
lines, chars);
                if ( c == EOF )
                    utf8log(errors, fderr, "\t(+ premature
EOF)\n");
        }

        /* file summary information */
        if ( !errors )
            utf8log(0, fderr, "Legitimate %s file\n", non_ascii ?
"UTF-8" : "7-bit ASCII");
        else
            utf8log(0, fderr, "Errors encountered during transla-
tion\n");

        return errors;
}

int main(int argc, char *argv[]) {
        FILE *foo;
        FILE *fdin  = stdin;
        FILE *fdout = stdout;
        FILE *fderr = stderr;

        if (argc > 1) {
            if ((foo = fopen(argv[1],"r")) != NULL)
                fdin  = foo;
            else { fprintf(stderr, "error opening file %s, abort-
ing...\n", argv[1]); return -1; }
        }

        if (argc > 2) {
            if ((foo = fopen(argv[2],"w")) != NULL)
                fdout = foo;
            else { fprintf(stderr, "error opening file %s, abort-
ing...\n", argv[2]); return -1; }
        }

        if (argc > 3) {
            if ((foo = fopen(argv[3],"w")) != NULL)
                fderr = foo;
            else { fprintf(stderr, "error opening file %s, abort-
ing...\n", argv[3]); return -1; }
        }

        return utf8lint(fdin, fdout, fderr);
}
```