Maëlan

Zeste de savoir

Comprendre les encodages

12 août 2019

Table des matières

Vos jolies lettres accentuées cèdent la place à d'affreux Ã© et ?

Webmestre, vous recopiez au début de vos pages HTML une ligne

```
:::html
<meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1"/>
```

sans comprendre?

Vous avez déjà croisé des mots comme *charset*, *encodage*, *ASCII*, *UTF-8*, *ISO-8859*, *latin-1*… et vous vous demandez ce que sont ces bestioles?

Vous êtes curieux de savoir comment un ordinateur code du texte?

Ce cours est pour vous. On va tout expliquer en douceur.

FIGURE 0.1. – Martine Ã©crit en UTF-8

1. La théorie : le texte en informatique

Un ordinateur ne peut stocker que des nombres, ou plus précisément des 0 et des 1 (des «bits») qu'on regroupe pour former des nombres en binaire.

1.1. Concepts

Comment fait-on alors pour écrire du texte? Tout simplement, on associe à chaque caractère (une lettre, un signe de ponctuation, une espace…) un nombre. Un texte est alors une séquence de tels nombres, ce qu'on appelle une «chaîne de caractères».

Par exemple, on peut décider ceci :

Caractère	`A`	`B`	`C`	…	`Z`	`0`	`1`	`2`	…	`9`	`.`	`,`	`:`
Nombre associé	0	1	2	…	25	26	27	28	…	35	36	37	38

TABLE 1.2. – Exemple de correspondance nombre–caractère

Avec cet exemple, le texte `MARTINE ECRIT EN UTF-8.` serait codé comme suit :

```
:::text
 M  A  R  T  I  N  E     E  C  R  I  T     E  N     U  T  F  -  8
       .
12 00 17 19 08 13 04 42 04 02 17 08 19 42 04 13 42 20 19 15 41 38
    36
```

On a donné un nombre unique pour chaque lettre de l'alphabet (de `A` à `Z`), pour les dix chiffres (de `0` à `9`) et pour les signes de ponctuation, sans oublier l'espace.
Cet exemple montre comment les informaticiens inventent une façon de coder du texte en mémoire.

1. D'abord, on décide de l'ensemble des **caractères** dont on a besoin. On appelle cet ensemble un **répertoire (ou jeu) de caractères** (*character set* en anglais, abrégé *charset*).
2. Ensuite, on assigne à chaque caractère un numéro unique appelé **code**. Le résultat est appelé **jeu de caractères codés** et se résume à un tableau de correspondance comme

ci-dessus.

3. Enfin, il faut déterminer l'**encodage** (*encoding*), c'est-à-dire la façon de transcrire du texte grâce aux codes des caractères qui le composent.

Évidemment, l'encodage le plus simple consiste à écrire directement chaque code. Le jeu et l'encodage sont alors confondus, et on parle de **page de code** (*charmap*). La distinction entre les deux peut sembler artificielle et, en fait, elle n'était pas faite jusqu'à récemment[1], le besoin ne s'en étant fait ressentir qu'avec Unicode. Mais pas de panique, on verra tout ça.

Mais qu'est-ce qu'un caractère? On pourrait les définir comme les éléments permettant de coder du texte. Ça correspond *grosso modo* aux «graphèmes», c'est-à-dire aux plus petits éléments constitutifs d'un système d'écriture : les lettres pour un système alphabétique, les idéogrammes pour un système idéographique… sans oublier les divers symboles de ponctuation. Ce n'est pas toujours évident à définir. Par exemple, pour l'alphabet latin, faut-il considérer une lettre accentuée comme un seul graphème, ou comme deux? Jusqu'en 2010, l'espagnol considérait `ll` comme une seule lettre, et un lingüiste vous dirait que `ch`, `ou`, `ai`… forment chacun un seul graphème en français. Pourtant, dans les jeux actuels, ces digrammes ne possèdent pas de caractère attitré, contrairement aux ligatures1 comme , æ, ß (allemand), | (néerlandais) | (polonais)…
Un caractère n'est donc pas exactement un graphème, d'autant plus qu'on définit aussi des «caractères de contrôle» qui ne représentent pas des symboles «imprimables», mais contrôlent le codage du texte et donnent des indications aux programmes qui le lisent. Par exemple, un caractère «fin de texte».

1

Ça, c'est pour la théorie. L'exemple qu'on a inventé était simpliste. En pratique, on a deux complications principales.

1. Il faut considérer les contraintes matérielles. En effet, comme je l'ai dit, un ordinateur ne connaît que le binaire. Les bits sont regroupés par groupes de huit appelés «octets». Un octet ne peut stocker que les nombres entiers de 0 à 255 (soit $256 = 2^8$ possibilités). Si cela ne suffit pas, on peut rassembler les octets par deux, quatre ou plus pour avoir de plus grands nombres. Dans nos codages, il faudra tenir compte de ces limites, par exemple en faisant tenir tous les codes de notre jeu sur un ou deux octets.
2. Il y a beaucoup plus de caractères à gérer. Dans notre exemple, on a été négligent, on en a oublié plein :
   — Il serait utile d'ajouter des caractères de contrôle.
   — Ce serait vraiment bien d'avoir les lettres minuscules…
   — Pour écrire vraiment correctement en français, il faudrait aussi les accents, le C cédille (`ç`) l'E dans l'O (), les symboles de monnaie…
   — Enfin, n'oublions pas que l'alphabet latin est très loin d'être le seul système d'écriture du monde : les Arabes et les Chinois — entre autres — ont sans doute envie de parler dans leur langue maternelle, eux aussi. C'est là que ça devient vraiment problématique, comme nous allons voir.

1. d'où une tendance à mélanger allégrement tous les termes
1. https ://fr.wikipedia.org/wiki/Ligature_(écriture)

2. L'épopée des encodages

Pour bien cerner les problèmes causés par les encodages et la situation actuelle, le mieux reste encore de retracer leur histoire.

2.1. Au commencement était l'ASCII

L'un des premiers encodages historiques est l'ASCII, soit l'*American Standard Code for Information Interchange* (en français, le «code américain normalisé pour l'échange d'informations»). C'est une norme étasunienne, inventée en 1961, qui avait pour but d'organiser le bazar informatique à l'échelle nationale. Ce n'est donc pas le premier codage de l'Histoire, mais il s'est imposé au point d'éclipser les précédents.[2]
[1]

Le terme « ASCII » est souvent employé incorrectement pour désigner des pages de code qui étendent l'ASCII ou qui en dérivent.
L'ASCII proprement dit utilise **sept bits** (et non huit!) et dispose donc de 128 (2^7) caractères uniquement, numérotés de 0 à 127. En effet, à l'époque, il était encore courant de regrouper les bits par sept et non par huit, ou alors de réserver le huitième bit pour vérifier l'intégrité des données (bit de parité).
Sans plus attendre, voici la table de l'ASCII. Pour trouver le code d'un caractère, il faut mettre bout à bout le chiffre *hexadécimal* de sa ligne et celui de sa colonne. Ainsi le caractère `Z` a pour code hexadécimal 0x5A (soit 90 en décimal).

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-
0-	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	*HT*	*LF*	*VT*	*F*
1-	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS
2-	*SP*	!	"	#	$	%	&	'	(	)	*	+	,
3-	0	1	2	3	4	5	6	7	8	9	:	;	<
4-	@	A	B	C	D	E	F	G	H	I	J	K	L
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\
6-	`	a	b	c	d	e	f	g	h	i	j	k	l
7-	p	q	r	s	t	u	v	w	x	y	z	{	|

TABLE 2.2. – Table des caractères ASCII

Un exemple (les valeurs sont en hexadécimal) :

2. Parmi les dinosaures dont il subsiste encore des fossiles aujourd'hui, on peut citer EBCDIC1.
1. https ://fr.wikipedia.org/wiki/EBCDIC

```
:::text
texte :         "  A     v  a  i  n  c  r  e     s  a  n  s     p
    e  r  i  l  .  .  .  "
codage ASCII :  22 41 20 76 61 69 6E 63 72 65 20 73 41 6E 73 41 70
    65 72 69 6C 2E 2E 2E 22
```

Regardons un peu ce qu'il y a dans l'ASCII :

- les vingt-six lettres de l'alphabet latin, en majuscules (0x41–0x5A) et en minuscules (0x61–0x7A), ainsi que les chiffres de 0 à 9;
- divers signes de ponctuation, et d'autres symboles tels que les crochets, les accolades, l'arrobase…;
- des caractères blancs, c'est-à-dire l'espace mais aussi d'autres tels que le retour à la ligne (eh oui, c'est aussi un caractère). Ils sont marqués comme *ça*, en voici la liste :
  - *SP* (0x20) : espace (*space*);
  - *HT* (0x09) : tabulation horizontale (*horizontal tab*), le `'\t'` des programmeurs;
  - *VT* (0x0B) : tabulation verticale (*vertical tab*), le `'\v'` des programmeurs;
  - *LF* (0x0A) : nouvelle ligne (*line feed*), le `'\n'` des programmeurs;
  - *CR* (0x0D) : retour chariot (*carriage return*), le `'\r'` des programmeurs; marque la fin d'une ligne;
  - *FF* (0x0C) : nouvelle page (*form feed*), le `'\f'` des programmeurs;
- des caractères de contrôle non imprimables (0x00–0x1F, et 0x7F), marqués comme ça; en voici quelques-uns :
  - NUL (0x00) : caractère nul (*null*), le `'\0'` des programmeurs; marque la fin d'une chaîne de caractères;
  - des caractères servant à la communication entre programmes, périphériques ou machines;
  - des caractères correspondant à des actions, comme BS (*backspace*, retour arrière), ESC (*escape*, échappement), CAN (*cancel*, annulation) ou BEL (*bell*, le «bip!»);
  - d'autres encore.

Notons que *LF* et *CR* remplissent des rôles proches et sont tous deux utilisés pour marquer les nouvelles lignes :

- sous Linux et Mac OS X, on utilise *LF*;
- sous Mac avant Mac OS X, on utilise *CR*;
- enfin, sous Windows, on utilise la séquence *CR* *LF*.

Un joyeux bazar! En fait, les raisons sont historiques. Jadis, les ordinateurs n'avaient pas d'écran; au lieu de s'afficher sur un écran, le texte produit par les programmes était imprimé sur du papier. Il y avait donc des caractères pour donner des instructions au chariot (la tête d'écriture) de l'imprimante : BS indiquait de reculer d'un caractère, *CR* (le «retour chariot») indiquait de revenir au début de la ligne, *LF* indiquait de descendre d'une ligne… Pour produire une nouvelle ligne, il était donc nécessaire de combiner *CR* et *LF*.

Pour poursuivre la digression culturelle, les 128 caractères de l'ASCII n'ont pas été placés au hasard. Leurs codes ont été soigneusement étudiés. Par exemples :

- À l'époque reculée où a été conçu l'ASCII, on communiquait encore parfois des données à l'ordinateur *via* des cartes perforées. Chaque emplacement codait un bit : 1 s'il y avait un trou, 0 sinon. La perforation était irréversible. Lorsqu'on n'avait pas encore spécifié de caractère, on laissait tous les emplacements intacts et le caractère valait donc 0 (tous les bits à 0). Ce caractère «non spécifié» se retrouve en ASCII avec NUL, le caractère

nul, qui vaut 0. De même, lorsqu'on voulait effacer un caractère on perçait tous les emplacements, ce qui donnait 127 (tous les bits à 1); le caractère ASCII DEL (*delete*) correspond justement à cette suppression.
— Les lettres majuscules sont séparées de leurs homologues minuscules par un intervalle de 32. Cela signifie qu'il suffit de modifier un bit (le sixième) pour passer des unes aux autres.

Pour plus de détails, consultez l'article de Wikipédia[2].
[2]

Comme on le voit, l'ASCII est simple. Il ne comporte que le strict nécessaire pour l'époque, pour utiliser un ordinateur… en anglais. Évidemment, les autres pays ont voulu employer leur propre langue.

2.2. La révolte gronde

C'est pourquoi des extensions de l'ASCII sont apparues.
Certaines gardent la base ASCII et utilisent le huitième bit laissé libre afin d'avoir plus de caractères à disposition (deux fois plus, $256 = 2^8$ au total). Ainsi, les codes de 0 à 127 correspondent encore aux caractères ASCII, tandis que les codes supérieurs (de 128 à 255, c'est-à-dire ceux avec le huitième bit valant 1) servent pour les nouveaux caractères.
D'autres restent sur sept bits, et modifient carrément les 128 caractères de l'ASCII pour leur propres besoins.
On imagine la pagaille que ça a été, lorsque chaque pays s'est mis à éditer sa propre page de code. Ça fonctionnait bien tant que les documents restaient dans la zone où leur propre encodage était en usage, mais les échanges internationaux étaient sujets à problèmes. Comme un même code signifiait des caractères différents d'une page à l'autre, le récepteur ne lisait pas la même chose que l'expéditeur. Par exemple, le symbole du dollar (`$`) aux États-Unis devenait celui de la livre (`£`) au Royaume-Uni : dégâts assurés!
Il y a bien eu des tentatives de stopper la multiplication des pages de code, mais elles ont été insuffisantes. L'exemple le plus significatif en est la norme ISO 646[3] de 1972. Elle a défini une page de code sur sept bits dérivant de l'ASCII, avec des caractères fixes (principalement les lettres, les chiffres et la ponctuation principale), les autres caractères étant laissés au choix. Cette norme, qui devait permettre une certaine compatibilité et un semblant d'ordre, a donné naissance à plusieurs pages de code nationales, mais elle n'était pas adaptée aux langues non latines et ne permettait pas de représenter assez de caractères.
[3]

2.2.1. ISO 8859 : huit bits pour les langues latines

Plus tard, une norme mieux pensée a fait son apparition : la norme **ISO 8859**. Cette fois, elle utilise **huit bits donc 256 caractères au maximum**. La norme ISO 8859 comporte en fait plusieurs «parties», c'est-à-dire des pages de code indépendantes, nommées ISO 8859-n où n est le numéro de la partie. ISO 8859 a été pensée afin que les parties soient le plus largement compatibles entre elles. Ainsi, la plage inférieure (codes 0 à 127) est la base commune ASCII, et la plage supérieure (codes 128 à 255) est spécifique à chaque page de code; elle est remplie en s'arrangeant pour qu'entre deux pages de code, des caractères en commun ou qui se ressemblent

2. https ://fr.wikipedia.org/wiki/ASCII
3. https ://fr.wikipedia.org/wiki/ISO_646

occupent le même code.
Cette norme a principalement servi aux langues latines d'Europe pour mettre au point une page de code commune. À elles seules, elles utilisent en tout dix parties d'ISO 8859, parfois appelées «latin-1», «latin-2», etc.; la première est la principale et les suivantes en sont des évolutions visant à améliorer la prise en charge de certaines langues. En voici deux à connaître :

— **ISO 8859-1 («latin-1» ou «Occidental»)** est une page de code très courante dans les pays latins et sur la Toile. C'est l'encodage utilisé initialement par les distributions Linux, mais elles migrent progressivement vers l'UTF-8 qu'on verra plus tard. Les systèmes Windows utilisent également un jeu dérivé, comme on le verra bientôt. Il a l'avantage de permettre d'écrire *grosso modo* toutes les langues latines, et ceci avec un octet par caractère seulement.

— **ISO 8859-15 («latin-9» ou «Occidental (euro)»)**, datant de 1998, introduit le signe de l'euro () et complète le support de quelques langues dont le français (avec) en abandonnant des symboles peu usités (dont le mystérieux ¤ signifiant «monnaie»). Il est néanmoins moins utilisé que son grand frère ci-dessus.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-
0- 7-	ASCII ***(sauf les caractères de contrôle, inutilisés)***												
8- 9-	*non utilisé*												
A-	NBSP	¡	¢	£	¤/	¥	¦/	§	¨/	©	ª	«	¬
B-	°	±	²	³	´/	µ	¶	·	¸/	¹	º	»	¼/
C-	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì
D-	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü
E-	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì
F-	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü

Table des caractères ISO 8859-1 (latin-1) et ISO 8859-15 (latin-9,

i

Le caractère NBSP (0xA0) est l'espace insécable (*non-breaking space*), c'est-à-dire une espace qui, contrairement à l'espace habituelle, ne «sépare» pas les mots. Ça veut dire que si on écrit «Bonjour!» avec une espace insécable et que ça se retrouve à la fin d'une ligne d'affichage, «Bonjour» ne sera pas séparé du point d'exclamation (ils iront ensemble au début de la ligne suivante) contrairement à une espace simple. Ce caractère est surtout utilisé avec les signes de ponctuation (avant «?», «!», «;», «:» et entre les guillemets "«" et "»"). Ce sont les espaces grises bizarres dans LibreOffice.

À nouveau un exemple, avec cette fois des caractères accentués, des guillemets et des espaces

insécables (symbolisées par pour qu'on les voit), conformément aux règles typographiques :

```
:::text
texte :          «  À   v a i n c r e   s a n s
    p é r i l . . .  »
codage latin-1 :  AB A0 C0 20 76 61 69 6E 63 72 65 20 73 41 6E 73
    41 70 E9 72 69 6C 2E 2E 2E A0 BB
```

Les codes de 0x00 à 0x1F et 0x7F (les caractères de contrôle ASCII) et de 0x80 à 0x9F sont laissés inutilisés par la norme 8859. Pour les communications Internet, l'IANA[4] a créé la norme ISO-8859 (attention, c'est le tiret qui change tout!), qui ajoute des caractères de contrôle à ces emplacements.

Windows s'est aussi basé sur latin-1 pour mettre au point son nouveau jeu de caractères occidental dans les années 1990. **Windows-1252**[5] (ou «CP1252», parfois dit «ANSI» à titre officieux) est maintenant le jeu utilisé dans tous les systèmes Windows occidentaux, et remplace les anciennes pages de code (la page de code 437[6] pour les États-Unis, et la 850[7] pour l'Europe). Il étend latin-1 avec des caractères supplémentaires dans les codes libres (de 0x80 à 0x9F).

Écrire des «caractères spéciaux» sous Windows

Sous Windows, on peut insérer n'importe quel caractère de Windows-1252 (donc de latin-1 et d'ASCII) avec la combinaison Alt + *code*[8] où *code* est le code décimal du caractère, précédé du chiffre zéro (sinon, le code est lu selon la vieille page de code du système — 850 ou 437). Les exemples les plus utiles pour le français :

- Alt + 0, 1, 9, 2 → À
- Alt + 0, 1, 9, 9 → Ç
- Alt + 0, 2, 0, 1 → É
- Alt + 0, 1, 6, 0 (ou Alt + 2, 5, 5) → espace insécable
- Alt + 0, 1, 7, 1 (ou Alt + 1, 7, 4) → «
- Alt + 0, 1, 8, 7 (ou Alt + 1, 7, 5) → »

Écrire des «caractères spéciaux» sous Linux

Sous Linux, le clavier est plus complet et la touche AltGr (ou AltGr + Maj) combinée avec les autres donne accès à de nombreux caractères. Et si cela ne suffit pas, la touche Compose[9] donne accès à encore plus. Je vous laisse chercher!

À partir de maintenant, plus d'excuse pour massacrer le français! D'ailleurs, si vous êtes intéressé, je conseille la lecture de ce cours sur la typographie[10].

4. https ://fr.wikipedia.org/wiki/IANA
5. https ://fr.wikipedia.org/wiki/Windows-1252
6. https ://fr.wikipedia.org/wiki/CP437
7. https ://fr.wikipedia.org/wiki/CP850
8. https ://fr.wikipedia.org/wiki/Alt_codes

9
10

ISO 8859 a aussi été utilisé pour l'alphabet cyrillique (ISO 8859-5), l'arabe (ISO 8859-6), le grec (-7), l'hébreu (-8) et même le thaï (-11).
Il existe au total seize parties et il n'y en aura pas plus. En effet, on privilégie désormais le développement de l'Unicode.

2.2.2. ISO 2022 : du multi-octet pour les langues asiatiques

Pendant ce temps en Asie…

Les langues latines s'en sont relativement bien tirées. Elles ont réussi à ne pas dépasser la limite fatidique de l'octet, ce qui reste quand même le plus pratique à la fois pour les traitements et pour la consommation de mémoire. Mais les langues asiatiques — japonais, coréen et chinois — disposent de bien trop de caractères pour que tout tienne sur huit bits. Les encodages mis au point en Asie de l'Est étaient donc **multi-octet**. Certains utilisaient deux octets par caractère, ce qui offre 65536 (2^{16}) codes différents.
Comme pour les langues latines, une norme a été mise au point pour les organiser, on l'appelle **ISO 2022**. Elle permet de jongler entre plusieurs jeux à l'aide de séquences d'échappement; celles-ci indiquent le jeu utilisé dans ce qui suit.
On peut ainsi employer plusieurs jeux différents dans une même chaîne de caractères. En fait, la norme distingue une plage de codes pour les octets 0 à 127, une autre pour les octets 128 à 255, et permet d'assigner un jeu à chaque plage de façon indépendante. Ainsi, on peut même employer deux jeux simultanément (par exemples ASCII et kanjis japonais, ou sinogrammes et latin-1)! Ou alors, en n'utilisant que la première plage, on peut faire passer du texte dans un vieux système à sept bits.
Les différents jeux sont totalement indépendants ; ils n'ont même pas à faire la même taille! Certains requièrent un octet par caractère, d'autres deux.
En théorie, ISO 2022 est une norme universelle puisqu'elle permet de passer d'un encodage arbitraire à un autre. Elle contient d'ailleurs ISO 8859. Toutefois, elle a plusieurs défauts, dont sa complexité et sa nature d'encodage *à état* : non seulement les caractères n'ont pas une taille fixe, mais en plus il faut absolument lire le texte depuis le début pour interpréter les octets!
En pratique, les occidentaux n'ont donc pas adopté ISO 2022, qui n'a servi que pour le chinois (ISO 2022-CN), le coréen (ISO 2022-KR) et le japonais (ISO 2022-JP). Ces variantes restent encore répandues, surtout la japonaise, mais l'Unicode gagne du terrain.

2.3. Unicode, la solution ultime

Avec des normes comme ISO 8859 et ISO 2022, le problème d'interropérabilité était à moitié résolu. On avait certes normalisé les communications entre langues voisines, mais le système conservait des limites. Comment transmettre un texte (envoyer de l'arabe à un français p · ex ·) si le jeu du destinataire ne contient pas tous les caractères nécessaires? Comment écrire un texte avec des caractères de plusieurs alphabets (arabe et russe p · ex ·), s'il n'existe pas de jeu qui les contienne simultanément? ISO 2022 constituait une réponse à ces problèmes, mais on a vu qu'elle était loin d'être idéale.

9. https ://fr.wikipedia.org/wiki/Touche_compose
10. http ://www.siteduzero.com/tutoriel-3-454279-l-orthotypographie-bien-ecrire-pour-bien-etre-lu.html

La solution s'impose d'elle-même : créer un jeu de caractères universel, qui contienne tous les alphabets! Cette idée toute simple a donné naissance à deux monuments : la norme ISO 10646 et Unicode.

FIGURE 2.1. – logo d'Unicode

Unicode est une norme développée par le consortium Unicode[11] depuis 1990 (version 8.0 en juin 2015). Elle repose sur le **jeu universel de caractères** ou JUC (en anglais, *UCS* pour *Universal Character Set*), défini dans la norme parallèle **ISO 10646**.

2.3.1. Le jeu universel de caractères

L'objectif de ce répertoire est d'accueillir tous les caractères existants de toutes les langues du monde, actuelles ou passées. Un travail titanesque! Concrètement, c'est un bête jeu de caractères, sauf que celui-ci offre pas moins de $17 \times 2^{16} = 1114112$ codes. À l'origine, il prévoyait même $2^{31} = 2147483648$ codes, mais il a vite été restreint : c'est déjà bien suffisant.

> Contenu masqué n°1

En juin 2015, environ 11% des codes sont déjà attribués à des caractères — en grande partie des idéogrammes asiatiques. Il reste encore de la place à revendre. Et pourtant, dans ces 11%, on a tout mis ou presque : les caractères de tous les anciens jeux, tous les alphabets modernes, pléthore de symboles… Le JUC n'est peut-être pas la solution définitive et éternelle, mais on peut dormir sur nos deux oreilles pour les décennies à venir.
Par souci de compatibilité, le JUC reprend latin-1 (et donc l'ASCII) pour ses 256 premiers caractères.
Le jeu n'est pas définitif : de nouveaux caractères sont régulièrement créés et assignés à des codes encore libres. Cependant, les caractères déjà en place ne bougent plus.

Au fait, **U+*code*** est une notation officielle pour désigner le caractère de code hexadécimal donné. Par exemples, U+0041 est la lettre `A` et U+23CF est le symbole ⏏.

Écrire des caractères Unicode sous Linux
Sous Linux, dans certains cas (selon la méthode d'entrée utilisée), le raccourci Ctrl+Maj+U+*code* insère directement le caractère Unicode spécifié.

On peut voir Unicode comme une surcouche d'ISO 10646. ISO 10646 liste les caractères du jeu en leur assignant un nom et un code. Unicode leur ajoute des attributs et des relations. Unicode

11. http ://www.unicode.org
12. https ://fr.wikipedia.org/wiki/Table_des_caractères_Unicode/U1D200

décrit aussi des algorithmes de traitement notamment pour les codages équivalents, les sens d'écriture, l'ordre alphabétique et les encodages pour transcrire le JUC.

2.3.2. Graphèmes *vs* points de code

J'avais évoqué en introduction la nuance entre graphème et caractère (qu'on appelle aussi **point de code** pour éviter les ambigüités). Cette distinction importe en Unicode, à cause d'un mécanisme appelé **composition**.
Pour écrire le graphème `é` (e accent aigu), je peux utiliser le caractère «précomposé» U+00E9 (hérité de latin-1), mais aussi la séquence U+0065, U+0301. U+0065 est le caractère de base (`e`, la lettre e) et U+0301 est une diacritique (, l'accent aigu) ; les deux se composent pour former un seul graphème. Ce sont deux codages possibles du *même* graphème, ils doivent donc être considérés comme équivalents.

i

En pratique, la composition est peu utilisée pour les alphabets latins car on a hérité des caractères précomposés des jeux précédents ; et, de fait, les systèmes de rendu les gèrent assez mal. Testons le navigateur :

```
1 é (U+00E9)
2 é  (U+0065, U+0301)
```

Chez moi, Firefox 45 affiche l'accent de la deuxième version trop à droite.
En revanche, une application intéressante de la composition, et qui montre qu'elle n'est pas restreinte aux diacritiques, est le hangeul[13], l'alphabet coréen. Comme les alphabets européens, les unités de base sont des lettres, mais au lieu d'être écrites une à une, elles sont regroupées par syllabes. Chaque syllabe forme un graphème composé de deux à six lettres. Par exemple, 한 est la syllabe *han*, composée des trois lettres ㅎ (*h*), ㅏ (*a*) et ㄴ (*n*). Unicode inclut les lettres du hangeul (dont 51 sont encore d'actualité et de nombreuses autres optionnelles), avec pour chacune une version isolée[14] et plusieurs versions composables[15] (selon la position dans la syllabe : initiale, médiane, finale). Cependant, Unicode inclut également les syllabes précomposées[16] (utiles), évidemment beaucoup plus nombreuses (11172).

```
1 (forme précomposée)
2 (forme décomposée)
```

Le hangeul est un fantasme de lingüistes, je vous laisse en apprendre plus sur l'article Wikipédia!

13. https ://fr.wikipedia.org/wiki/Hangeul
14. https ://en.wikipedia.org/wiki/Hangul_Compatibility_Jamo
15. https ://en.wikipedia.org/wiki/Hangul_Jamo_(Unicode_block)
16. https ://en.wikipedia.org/wiki/Hangul_Syllables

Ces particularités d'Unicode ont plusieurs conséquences :

1. Le mécanisme de composition implique que, même si les points de code sont encodés avec une taille fixe — ce qui n'est pas le cas, comme on verra —, les graphèmes ont, eux, une taille variable!
2. L'existence de caractères précomposés implique que le codage des graphèmes n'est pas unique. Unicode définit les équivalences entre codages, ainsi que des algorithmes pour calculer une forme normale.

Une application soigneuse doit prendre en compte tout ceci. Toutefois, je ne développerai pas plus sur ce sujet, cet article se focalisant surtout sur les encodages.

2.3.3. Un jeu, des encodages

Comme je l'ai dit, Unicode définit plusieurs encodages du JUC.

? Des encodages? Mais, je croyais qu'Unicode *était* un encodage?

Unicode est essentiellement un **jeu de caractères** (un ensemble de caractères auxquels on attribue à chacun un point de code unique) et non un **encodage** (façon de représenter ce point de code en mémoire). C'est ici que la distinction prend tout son sens. Auparavant, les deux se confondaient puisque tous les jeux de caractères étaient associés à un encodage simple : vu que leurs codes tenaient sur un ou deux octets, on se contentait de les écrire tels quels en mémoire.
Or, les points de codes d'Unicode nécessitent beaucoup plus qu'un octet : il leur en faudrait quatre[3]! Cela voudrait dire que si l'on continuait à faire comme avant, on utiliserait quatre fois plus de mémoire qu'avec nos pages de code sur un octet, comme latin-1.
Ça ferait beaucoup de mémoire utilisée, voire gaspillée, puisqu'il y aurait plein d'octets nuls :

Codes	Encodage en UTF-32	Caractères disponibles dans cet intervalle
jusqu'à U+00FF (2^8-1)	**`00000000 00000000 00000000 bbbbbbbb`(latin-1)**	**langues occidentales**
jusqu'à U+FFFF (2^{16}-1)	`00000000 00000000 bbbbbbbb` la plupart des alphabets actuels (BMP17)
jusqu'à U+10FFFF (2^{21}-1)	`00000000 000bbbbb bbbbbbbb` tous les caractères

TABLE 2.5. – `b` symbolisent les bits occupés)
Consommation mémoire en UTF-32
Comme on le voit, il y aurait toujours au moins un octet nul, très souvent deux. Le cas extrême est celui d'un Occidental qui gaspille trois octets par caractère.
Cet encodage existe tout de même, il est nommé **UTF-32**, mais est rarement employé — sauf en interne par quelques programmes, car il reste plus facile à traiter que ses confrères que nous allons voir.

3. Des petits malins me diront que trois octets suffiraient, mais en informatique on préfère les puissances de 2.

2.3.4. UTF-16

Pour faire des économies, on a donc mis au point des encodages plus futés. Tout d'abord, il y a l'**UTF-16** (*UCS transformation format*, seize bits). Celui-ci a pour unité de base un doublet, c'est-à-dire deux octets (d'où le « 16 » dans son nom, comme le nombre de bits de l'unité de base). Les points de code inférieurs à $2^{16} = 65536$ sont représentés tels quels sur deux octets, et les codes supérieurs sont représentés sur quatre octets *via* une transformation mathématique simple[4].

Codes	Encodage en UTF-16	Caractères disponibles dans cet intervalle
jusqu'à U+FFFF (2^{16}-1)	**`bbbbbbbb bbbbbbbb`**	**la quasi-totalité des alphabets actuels (BMP17)**
jusqu'à U+10FFFF (2^{21}-1)	`bbbbbbbb bbbbbbbb bbbbbbbb bbbbbbbb`	tous les caractères

TABLE 2.7. – `b` symbolisent les bits occupés)

Consommation mémoire en UTF-16

Les caractères de cet encodage ne font donc pas tous la même taille, ce qui complique un peu les traitements : si on stocke une chaîne UTF-16 dans un tableau de codes de deux octets, le n^e caractère ne se trouve pas forcément dans la n^e case. Ceci dit, étant donné que l'ensemble des points de codes inférieurs à 2^{16} (le «BMP17», pour *Basic Multilingual Plane*) contient déjà tous les alphabets actuels et symboles courants, il peut être suffisant de s'y restreindre (ce que fait Windows), et dans ce cas tous les caractères font bien deux octets.
17

De plus, l'UTF-16 (et l'UTF-32) fait surgir une difficulté technique supplémentaire : le boutisme18 (*endianness*). Ce terme mystique désigne l'ordre des octets d'un nombre multi-octet. Il en existe deux variantes principales : le **gros boutisme** (*big-endian, BE*) et le **petit boutisme** (*little-endian, LE*). Le boutisme dépend des machines. Or, en UTF-16, on lit les données par groupes de deux octets, donc le boutisme influe sur le résultat : si l'on lit du texte UTF-16 envoyé par quelqu'un utilisant un boutisme différent du sien, les deux octets de chaque caractère seront échangés, et c'est l'erreur assurée!
18

Pour pallier à cela, on utilise parfois un caractère spécial appelé BOM (*byte order mark*, marque d'ordonnancement des octets). l'IANA a aussi créé deux noms pour préciser le boutisme employé sur Internet : `UTF-16BE` et `UTF-16LE`.

Reprenons le texte d'exemple de tout à l'heure et encodons-le cette fois en UTF-16. En passant, remplaçons les trois caractères « point » (`.`) par le nouveau caractère « points de suspension » () qui est fait pour ça. Pour bien faire, ajoutons la BOM facultative :

```
1 :::text
```

4. C'est d'ailleurs le codage UTF-16 qui a fixé la limite inhabituelle de 17×2^{16} points de code (la limite précédente, 2^{31}, vient du codage UTF-8).

17. https ://en.wikipedia.org/wiki/Plane_(Unicode)#Basic_Multilingual_Plane

18. https ://fr.wikipedia.org/wiki/Boutisme

```
texte :          [BOM] «____ ▒____ À____       v____ a____ i____
    n____ c____ r____ e____       s____ a____ n____ s____
    p____ é____ r____ i____ l____ …____ ▒____ »____
codage UTF-16 :  FE.FF 00.AB 00.A0 00.C0 00.20 00.76 00.61 00.69
    00.6E 00.63 00.72 00.65 00.20 00.73 00.41 00.6E 00.73 00.41
    00.70 00.E9 00.72 00.69 00.6C 20.26 00.A0 00.BB
```

Ici, la BOM indique que le texte est gros-boutiste (l'octet de poids fort est en premier). En petit-boutiste, on aurait ça :

```
:::text
texte :          [BOM] «____ ▒____ À____       v____ a____ i____
    n____ c____ r____ e____       s____ a____ n____ s____
    p____ é____ r____ i____ l____ …____ ▒____ »____
codage UTF-16 :  FF.FE AB.00 A0.00 C0.00 20.00 76.00 61.00 69.00
    6E.00 63.00 72.00 65.00 20.00 73.00 41.00 6E.00 73.00 41.00
    70.00 E9.00 72.00 69.00 6C.00 26.20 A0.00 BB.00
```

Si la chaîne contient une BOM, il suffit donc de lire ses deux premiers octets pour connaître son boutisme.
Remarquons que ce texte s'encode en UTF-16 exactement comme en latin-1, avec des valeurs sur deux octets au lieu d'un (donc un octet nul sur deux). Ici, on occupe donc deux fois plus de mémoire.

2.3.5. UTF-8

On a aussi inventé l'**UTF-8**. La taille des caractères codés est encore plus variable, et l'économie de mémoire plus grande. Comme son nom l'indique, l'unité de base est l'octet. Il code les premiers caractères (ceux de l'ASCII) sur un octet, les suivants sur deux, trois et jusqu'à quatre octets.

Codes	Encodage en UTF-8	Caractères disponibles dans cet intervalle
jusqu'à U+007F (2^7-1)	**`0bbbbbbb`**	**latin de base (ASCII)**
jusqu'à U+07FF (2^{11}-1)	`110bbbbb 10bbbbbb`	alphabets d'Europe et du Moyen-Orient [5]
jusqu'à U+FFFF (2^{16}-1)	`1110bbbb 10bbbbbb 10bbbbbb`	la quasi-totalité des alphabets actuels (BMP17)
jusqu'à U+10FFFF (2^{21}-1)	`11110bbb 10bbbbbb 10bbbbbb 10bbbbbb`	tous les caractères

TABLE 2.9. – `b` symbolisent les bits occupés)

Consommation mémoire en UTF-8
[19]

On voit que l'UTF-8 est particulièrement intéressant pour les langues à alphabet latin[8], qui bénéficient en plus de la compatibilité ASCII. Pour les alphabets grec, arabe ou cyrillique, UTF-8 ou UTF-16, c'est kif-kif ; on peut préférer UTF-16 pour sa plus grande constance, et l'économie qu'on réalise quand même sur quelques caractères moins fréquents. Pour tous les autres alphabets (dont les langues asiatiques), l'UTF-16 est préférable.
En plus d'être compatible avec l'ASCII, l'encodage UTF-8 a été conçu afin que certains algorithmes de traitement (comparaison lexicographique par exemple) soient réutilisables sans modification. Ainsi, de vieux programmes fonctionneront encore si on leur passe du texte en UTF-8 : magique!
Enfin, contrairement à l'UTF-16, on n'a pas de problème de boutisme puisqu'on lit les données octet par octet.
UTF-8 a de nombreux avantages (économie de mémoire, compatibilité, boutisme, résistance aux erreurs). Un défaut aussi : la grande variabilité de taille des caractères rend les traitements plus compliqués et nuit à leurs performances. Ses points forts ont cependant suffi à lui assurer le succès : il est l'encodage le plus répandu aujourd'hui. Il est utilisé pour les documents et les communications, et les systèmes d'exploitation (Mac et Linux) s'y mettent aussi.
Pour ne pas changer, voici notre texte d'exemple encodé en UTF-8 :

```
:::text
texte :         «____ ▒____ À____    v  a  i  n  c  r  e     s  a
    n  s     p  é____ r  i  l  …_______ ▒____ »____
codage UTF-8 :  C2.AB C2.A0 C3.80 20 76 61 69 6E 63 72 65 20 73 41
    6E 73 41 70 C3.A9 72 69 6C E2.80.A6 C2.A0 C2.BB
```

Constat flagrant, la consommation de mémoire est singulièrement réduite par rapport à l'UTF-16! On peut aussi observer la compatibilité ASCII, et les caractères non-ASCII s'étendant sur plusieurs octets (ici symbolisés par `__`). Tous les caractères de latin-1 utilisent au plus deux octets, mais le caractère en prend trois.

2.4. Et aujourd'hui ?

Notre périple au fil des âges nous a ramené au temps présent. On a vu les encodages historiques, mais lesquels sont utilisés aujourd'hui, et dans quels contextes?
Avant l'apparition d'Unicode, les systèmes d'exploitation fonctionnaient avec un système de pages de code régionales, c'est-à-dire qu'on utilisait des variantes d'encodage sur huit bits en

5. alphabets[6] latin, API, grec, copte, cyrillique, arménien, hébreu, syriaque, arabe, thâna (utilisé aux Maldives), n'ko (utilisé en Afrique de l'Ouest) ; la carte se trouve ici19.
6. Le saviez-vous? à l'exception peut-être du n'ko (création récente vaguement inspirée de l'arabe), tous ces alphabets[7] sont les descendants encore en vie de l'alphabet phénicien, voire de son évolution en alphabet araméen. :-)
7. En fait, il faudrait dire «systèmes d'écriture» (l'arabe au moins n'étant pas un «alphabet»). Vous me pardonnez?
19. https://fr.wikipedia.org/wiki/Table_des_caractères_Unicode_(0000-FFFF)#Alphabets,_abjads,_abugidas_et_syllab

8. qui, d'après Wikipédia , représenteraient tout de même 39% de la population mondiale et 84% des connexions à Internet

fonction du pays. Depuis, ils ont migré vers Unicode en interne (y compris Windows, depuis XP, même si ça ne se répercute par forcément pour l'utilisateur).

— Sous Windows, **UTF-16**[9] (*little endian*) est maintenant utilisé en interne et UTF-8 est également supporté, mais Unicode cohabite encore avec l'ancienne famille de pages de code. Ces dernières, incorrectement appelées «ANSI» et proches des ISO 8859, sont encore fréquemment employées. Je vous rappelle que celle pour nous autres Occidentaux est **Windows-1252** et dérive de latin-1. Cette famille avait elle-même remplacé l'antique famille dite «OEM» (pages de code CP437 et CP850 en Occident), dont l'usage perdure pourtant pour la console.

— Sous Mac OS, l'encodage en vigueur en Occident était nommé MacRoman[20], mais depuis Mac OS X on utilise **UTF-8**.

— Les distributions GNU/Linux utilisaient latin-1 par défaut, mais elles ont maintenant (presque?) toutes migré à **UTF-8**.

[20]

Changer la page de code sous Windows

Sous Windows, on peut changer la page de code en vigueur depuis le Panneau de Configuration, «Options régionales et linguistiques», onglet «Avancé». En fait, c'est la *locale*[21] du système qu'on règle ainsi, ce pourquoi c'est présenté comme le choix de la langue par défaut des programmes.

Il est également possible de changer temporairement la page de code d'une console Windows grâce à la commande «CHCP»; elle prend en argument le n° de la page à utiliser (`CHCP 1252` pour Windows-1252); sans argument, elle affiche la page actuelle de la console (probablement 437 ou 850).

[21]

Changer la *locale* sous Linux

Sous un système unixoïde, c'est la *locale*[22] qui détermine l'encodage en vigueur. On peut donc le changer à la volée *via* des variables d'environnement, à condition que la *locale* désirée soit installée. La vôtre est probablement `fr_FR.UTF-8` (votre ordinateur vous parle en français de France avec l'encodage UTF-8).

Le système de *locales* sert aussi (surtout!) à adapter les programmes aux conventions propres à chaque langue et région :

```
:::console
$  date
Sun Aug 26 13:37:42 CET 2012
$  LC_ALL=fr_FR.UTF-8  date
dimanche 26 août 2012, 13:37:42 (UTC+0100)
```

[22]

Il est bien sûr possible, avec les bons outils, d'encoder un fichier de n'importe quelle façon, sous

9. ou plutôt sa restriction à deux octets, comme dit précédemment.
20. https ://fr.wikipedia.org/wiki/MacRoman
21. https ://fr.wikipedia.org/wiki/Paramètres_régionaux
22. https ://wiki.archlinux.org/index.php/Locale

n'importe quel système d'exploitation. L'encodage d'un OS est simplement celui qu'il utilise en interne, donc aussi celui que les programmes ont tendance à utiliser, donc aussi celui des documents d'un utilisateur .
De plus, UTF-8 est maintenant l'encodage le plus utilisé sur la Toile, ce qui semble logique puisqu'il s'agit d'un réseau international mettant en contact toutes les langues. L'autre principal encodage sur Internet reste latin-1.
L'ASCII seul n'est plus employé, mais tous les encodages répandus conservent la base ASCII par compatibilité. Les trois principaux, cités ci-dessus, se basent même tous sur latin-1 (même si l'UTF-8 n'est pas compatible avec ce dernier puisque les codes supérieurs à 127 sont encodés par deux octets).

i

Si Unicode tend à remplacer progressivement tout le reste, il reste un endroit où survivent des encodages plus exotiques : l'informatique embarquée dans les petits appareils (lave-linges, calculettes, affichage des arrêts dans un bus…). En effet, ceux-ci n'ont parfois besoin que d'un jeu très restreint, et n'échangent pas de données avec un réseau.
Dans le même ordre d'idées, il existe une application qu'on utilise tous les jours et qui n'utilise pas Unicode : les SMS! Ils emploient un encodage nommé GSM 03.3823, un ensemble de pages de code sur sept bits vaguement compatibles avec l'ASCII. Toutefois, on peut utiliser UTF-16 à la place, ce que fait évidemment le trio Chine-Japon-Corée. Les téléphones actuels utilisent GSM 03.38 par défaut et basculent automatiquement en UTF-16 si jamais on écrit un caractère non supporté autrement. Un SMS étant limité à 140 octets, GSM 03.38 permet jusqu'à 160 caractères, alors qu'UTF-8 n'en permet que 70. On a beau s'entendre répéter que de nos jours la mémoire ne coûte rien et qu'on peut se permettre l'UTF-16, voilà un cas où l'on préférerait nettement faire des économies!

23

Que retenir?

— Le principe du codage du texte en informatique : jeux de caractères et encodages.
— Qu'un caractère **n'est pas** un octet. Ni même un nombre fixe d'octets. Que les programmeurs se le disent!
— Les principaux encodages : ASCII, latin-1, UTF-8, pour n'en donner que trois.
— Leurs usages actuels.

Il va de soi que je vous recommande d'utiliser l'UTF-8 exclusivement, partout. Ou éventuellement l'UTF-16 si vous utilisez un alphabet non latin.

Contenu masqué

Contenu masqué n°1

Devant une telle quantité, les polices d'écriture (qui n'ont rien à voir avec les encodages, il s'agit ici d'**affichage** des caractères et non de leur **codage** en mémoire) peinent à suivre. C'est ce

23. https ://en.wikipedia.org/wiki/GSM_03.38

qui explique les petits symboles du type □ qu'on peut voir sur certaines pages (en regardant bien, vous lirez le code du caractère dans le carré). Généralement, les auteurs de polices se contentent de créer les glyphes des parties qui les intéressent, et parfois de copier les glyphes de référence pour le reste; si on a vraiment besoin de rédiger une partition suivant la notation musicale grecque ancienne12 avec le JUC, on utilisera une police spécialisée… Retourner au texte.

3. En pratique : jongler avec les encodages

Maintenant qu'on a vu tout ça, il nous reste à pratiquer un peu!
Lorsqu'on consulte un document, que ce soit un fichier texte (code source par exemple) ou une page web, il faut la lire avec le bon encodage. Sinon, les valeurs seront mal interprétées. Par exemple, si on encode ce texte en UTF-8 :

> l'événement du siècle

et qu'on le lit en latin-1, on verra ceci :

> l'Ã©vÃ©nement du siÃ¨cle

Splendide, non? Vous avez déjà dû croiser ce genre d'erreurs… Pour l'expliquer, souvenons-nous qu'UTF-8 encode certains caractères, dont les lettres accentuées, sur deux octets; ici la lettre `é` donne les octets 0xC3 et 0xA9. Or, latin-1 encode tous ses caractères sur un octet. Les octets 0xC3 et 0xA9 sont donc interprétés séparément, et donnent les caractères `Ã` et `©`, respectivement. Notons que tous les autres caractères sont lus correctement, car ils appartiennent à la base commune ASCII. On voit maintenant l'intérêt de cette compatibilité : même avec un mauvais encodage, le texte reste globalement lisible.
Réciproquement, si ce texte était encodé en latin-1 et qu'on tentait de le lire en UTF-8, on aurait sans doute quelque chose comme :

> l'vnement du sicle

car la séquence d'octets 0xE9.0x67, qui code `év` en latin-1, est invalide en UTF-8.
Ces deux cas de figure sont faciles à reconnaître, et représentent une grande partie des problèmes qui surviennent en pratique. Cependant, le diagnostic peut être plus difficile quand il s'agit de deux encodages dans lesquels tous les caractères font la même taille (le fameux dollar `$` devenu livre `£`).
Déterminer l'encodage d'un fichier est donc crucial, et compliqué par la diversité des encodages existants. Or, les renseignements associés à un fichier (sa date de création par exemple) n'indiquent rien sur son encodage. On doit donc tenter de le deviner. Les programmes (navigateur web, éditeur de texte…) emploient des algorithmes qui analysent le contenu du fichier. Ces algorithmes sont efficaces la plupart du temps, mais peuvent échouer[1]. Une détection incorrecte explique l'affichage bizarre de certains documents.
[1]

Un moyen plus simple serait d'inclure cette indication directement dans le contenu du fichier, au tout début pour diminuer les risques de perturbation. On utilise pour cela la base de compatibilité ASCII. On verra une application de cette idée avec les pages HTML.

1. https ://en.wikipedia.org/wiki/Bush_hid_the_facts

3.1. Lire & écrire avec le bon encodage

Commençons par apprendre à régler manuellement l'encodage si jamais la détection automatique échoue.

3.1.1. Lire une page web

La plupart du temps, lire une page web ne pose aucun souci car son encodage est précisé dans son code source. Il arrive toutefois que ce ne soit pas fait, ou mal fait, et que le navigateur échoue à le deviner. Par exemple :

FIGURE 3.1. – Exemple de page lue avec un mauvais encodage

C'est moche. C'est désagréable à lire. Heureusement, on peut y remédier manuellement. Tout navigateur qui se respecte permet de jongler entre les encodages. Pour Firefox, le menu est caché sous «Affichage» :

FIGURE 3.2. – Menu des encodages dans Firefox

Pour l'instant, le navigateur est en «détection automatique», ce qui a conduit à l'utilisation incorrecte d'ISO-8859-1 (latin-1). On peut en changer. Ici, la page est probablement en UTF-8 (ça ressemble à l'erreur vue en introduction), donc on essaie cet encodage. On choisit l'option correspondante dans le menu, et…

FIGURE 3.3. – Tadaam!

La page s'affiche correctement. C'est du beau boulot.

3.1.2. Éditer un fichier

On va maintenant voir comment gérer l'encodage de nos fichiers. Tout éditeur de texte digne de ce nom permet de le faire de façon précise… mais commençons par le Bloc-Notes de Windows. Ouvrons le Bloc-Notes, tapons un peu de texte avec des accents, puis enregistrons.

FIGURE 3.4. – Boîte de dialogue d'enregistrement du Bloc-Notes

Le Bloc-Notes permet au premier enregistrement de choisir l'encodage du fichier. Le choix est assez limité cependant : «ANSI» (Windows-1252), «Unicode» (UTF-16LE), «Unicode *big endian*» (UTF-16BE), ou UTF-8. Comme exercice, vous pouvez vous amuser à vérifier que les tailles des fichiers affichées sur ma capture sont correctes, sachant que mon texte comporte quinze caractères et que le Bloc-Notes ajoute automatiquement une BOM pour tous les encodages Unicode (qui fait deux octets en UTF-16 et trois octets en UTF-8).
Le Bloc-Notes est vraiment très limité. Ainsi on ne peut pas choisir le latin-9 par exemple. De plus, rien n'indique l'encodage du fichier sur lequel on travaille, et on ne peut pas en changer après coup.
Si vous programmez, vous utilisez certainement un éditeur plus avancé. Comme les navigateurs web, la plupart incluent un menu pour passer d'un encodage à un autre. Toujours sous Windows, voici l'exemple de Notepad++[2] (ici j'ai rouvert les fichiers que je viens de créer avec le Bloc-Notes) :

FIGURE 3.5. – Le menu des encodages dans Notepad++

Première remarque, Notepad++ détecte automatiquement l'encodage et l'indique dans la barre de statut (en bas, encadré en bleu). C'est déjà mieux. En passant, remarquons la mention «Dos\Windows». Elle indique le style utilisé pour les fins de ligne. On avait vu qu'il existait plusieurs conventions, selon les OS. Ici, notre fichier utilise le style Windows, c'est-à-dire CR LF.
Ensuite, le menu «Encodage» permet de changer en direct l'encodage utilisé. Comme dans les navigateurs web, les options «Encoder en *xxx*» changent l'interprétation des octets existants; en plus, elles déterminent le codage des caractères nouvellement insérés. Pour modifier l'encodage d'un fichier, il ne faut pas cliquer sur «Encoder en *xxx*», car cela n'adapte pas le contenu existant; pour ça, il faut faire «Convertir en *xxx*».
Enfin, on a quand même plus de choix que dans le Bloc-Notes!
Après cet aperçu, faites un tour dans la configuration de votre éditeur. Il y a certainement des options qui nous intéressent.

2. http ://notepad-plus-plus.org/fr/

FIGURE 3.6. – Fenêtre de configuration de Notepad++

Ici, j'ai encadré la partie intéressante en vert. On peut choisir l'encodage (et le format des fins de ligne) qui sera utilisé par défaut lors de la création d'un nouveau fichier.

3.1.2.1. Avec ou sans BOM ?

Remarquons qu'il y a deux encodages UTF-8. L'une porte la mention «(sans BOM)», ce qui signifie que l'autre est un «UTF-8 avec BOM». On a déjà parlé de la BOM, ce caractère Unicode spécial placé tout au début d'un fichier pour en indiquer le boutisme. Cette technique est utilisée pour l'UTF-16 et l'UTF-32. En revanche, elle est inutile en UTF-8 puisqu'on n'a pas de problème de boutisme. Pire, elle peut rendre des fichiers invalides pour certains programmes. C'est par exemple le cas des pages web, comme on verra plus tard. Pourtant, certains éditeurs dont le fameux Bloc-Notes la rajoutent automatiquement même en UTF-8, car ça les aide à détecter l'encodage du fichier. C'est une pratique déconseillée. Dans votre éditeur favori, choisissez toujours la version sans BOM si vous avez le choix.
Ici, nettoyons notre fichier de cette hérésie avec Notepad++. Pour ça, on fait simplement «Convertir en UTF-8 (sans BOM)» et on enregistre. Dans les paramètres, on choisit aussi l'UTF-8 sans BOM par défaut.

3.1.3. Convertir un fichier

3

4

On peut aussi convertir un fichier sans passer par un éditeur. C'est par exemple la fonction d'`iconv`[3], programme en ligne de commande disponible sur les unixoïdes (il a donné son nom à l'API standard[4], intégrée à la *glibc*, qui fait la même chose). Il s'utilise comme ça :

```
:::console
$  iconv [-f DEPUIS] [-t VERS]
```

les formats `DEPUIS` et `VERS` pouvant être omis pour utiliser la *locale* actuelle. Par exemple : <–COMMENT : : :console $ echo ”déjà” | iconv -t latin1 | hexdump -C 00000000 64 e9 6a e0 0a |d.j..| COMMENT–>

```
:::console
$  echo "déjà"  | iconv -t utf16  | hexdump -C
00000000  ff fe 64 00 e9 00 6a 00  e0 00 0a 00
    |..d...j.....|
```

On observe la BOM (U+FEFF) qui nous dit que l'encodage est petit-boutiste, et les octets nuls insérés *après* chaque octet de latin-1.

Attention, pour modifier un fichier en place, il ne faut surtout pas faire :

3. http ://man7.org/linux/man-pages/man1/iconv.1.html
4. http ://man7.org/linux/man-pages/man3/iconv.3.html

```
:::console
$  commande  < fichier > fichier
```

car cela effacerait le contenu du fichier… Il faut passer par un fichier temporaire. De toute façon, il est plus prudent de vérifier le résultat avant d'écraser le fichier.

Il existe aussi le programme `recode`[5], qui s'utilise de façon similaire :

```
:::console
$  recode [DEPUIS][..VERS]
```

Sa spécificité est qu'il ne gère pas seulement les encodages textuels (le sujet de cet article), mais plus généralement divers types de codages (ce que le logiciel nomme «surfaces»), qui peuvent se superposer. Cela inclut les formats de fin de ligne (LF, CR ou CRLF) et les «encodages de transfert» (Base64[7], Quoted-Printable[6]…) utilisés notamment par les courriels.

Par exemple, pour convertir le texte en UTF-16, coder les octets obtenus avec Quoted-Printable, puis appliquer le format de fin de ligne CR-LF au tout :
<–COMMENT : : :console $ echo ”déjà” | recode ..latin1/QP/CRLF | hexdump -C 00000000 64 3d 45 39 6a 3d 45 30 0d 0a |d=E9j=E0..| COMMENT–>

```
:::console
$  echo "déjà"  | recode ..utf16/QP/CRLF  | hexdump -C
00000000  3d 46 45 3d 46 46 3d 30  30 64 3d 30 30 3d 45 39
    |=FE=FF=00d=00=E9|
00000010  3d 30 30 6a 3d 30 30 3d  45 30 3d 30 30 0d 0a
    |=00j=00=E0=00..|
```

Observons que chaque octet qui ne correspond pas à un caractère affichable de l'ASCII est codé par `=XX` (Quoted-Printable), et que la fin de ligne est codée par la séquence 0x0D, 0x0A (CR-LF).
`recode` a aussi une foncton très pratique pour examiner du texte!

```
:::console
$  echo "déjà, "  | recode ..dump
UCS2   Mné   Description

0064   d     lettre minuscule latine d
00E9   e'    lettre minuscule latine e accent aigu
```

5. http ://linux.die.net/man/1/recode
6. https ://fr.wikipedia.org/wiki/Quoted-Printable
7. https ://fr.wikipedia.org/wiki/Base64

```
006A   j     lettre minuscule latine j
00E0   a!    lettre minuscule latine a accent grave
002C   ,     virgule
0020   SP    espace
1112         hangûl tch'ôsong hiûh
1161         hangûl djoungsong a
11AB         hangûl djôngsong niûn
000A   LF    interligne (lf)
```

3.1.4. Corriger un encodage mixte

Il arrive qu'un fichier mélange plusieurs encodages, comme l'UTF-8 et le latin-1. Ce peut être le cas d'un fichier texte récupéré sur Internet, ou de la base de données d'un site web. Démonstration :

```
:::console
$  cat test
ligne encodée en UTF-8
ligne encodée en latin-1
```

Normalement, ma console qui est en UTF-8 devrait plutôt afficher :

```
:::console
$  cat test
ligne encodée en UTF-8
ligne encode en latin-1
```

mais elle utilise un mode spécial pour afficher quand même les caractères encodés en latin-1 : lorsqu'une séquence d'octets n'est pas de l'UTF-8 valide, elle est relue comme du latin-1. Cette astuce est utilisée par de nombreux logiciels, dont la plupart des clients IRC[10]. On peut la considérer comme pratique, ou commme nuisible parce qu'elle masque les erreurs.
Bref. On peut vérifier que le fichier mélange effectivement les deux encodages :

```
:::console
$  cat test  | hexdump -C
00000000  6c 69 67 6e 65 20 65 6e  63 6f 64 c3 a9 65 20 65  |ligne
    encod..e e|
00000010  6e 20 55 54 46 2d 38 0a  6c 69 67 6e 65 20 65 6e  |n
    UTF-8.ligne en|
00000020  63 6f 64 e9 65 20 65 6e  20 6c 61 74 69 6e 2d 31  |cod.e
    en latin-1|
00000030  0a                                                |.|
```

C'est bien une erreur, qui fait planter des programmes plus stricts :

10. En effet, IRC n'offre aucun moyen de préciser l'encodage des messages, alors qu'en pratique il met en contact des gens qui en utilisent de toutes les sortes…

```
:::console
$  cat test  | iconv -f utf8
ligne encodée en UTF-8
ligne encod
iconv: séquence d'échappement non permise à la position 35
$  cat test  | recode utf8..
ligne encodée en UTF-8
ligne encod
recode: Entrée invalide dans « UTF-8..CHAR »
```

`iconv` peut ignorer les erreurs, mais ce n'est pas idéal :

```
:::console
$  cat test  | iconv -f utf8 -t //IGNORE
ligne encodée en UTF-8
ligne encode en latin-1
```

Il faut corriger un tel fichier. Il ne semble pas y avoir de programme répandu pour ça, mais ce n'est pas difficile à coder. Vous pouvez par exemple jeter un œil à ce script[8] en Perl ou à celui-ci[9] en OCaml.

[8]

[9]

```
:::console
$  cat test  | fix-mixed-utf8  | hexdump -C
00000000  6c 69 67 6e 65 20 65 6e  63 6f 64 c3 a9 65 20 65  |ligne
    encod..e e|
00000010  6e 20 55 54 46 2d 38 0a  6c 69 67 6e 65 20 65 6e  |n
    UTF-8.ligne en|
00000020  63 6f 64 c3 a9 65 20 65  6e 20 6c 61 74 69 6e 2d  |cod..e
    en latin-|
00000030  31 0a                                             |1.|
```

Ou alors, ce script interactif (suggéré par Taurre[10]) qui demande l'encodage d'origine de chaque ligne et convertit tout en UTF-8 :

```
:::sh
#!/bin/sh
last=utf8
while read line ; do
        echo "$line" | od -c -w256 >&2
        read -p "Quel encodage ? [$last] " code </dev/tty >&2
        last="${code:-$last}"
```

8. https ://gist.github.com/chansen/1522213
9. https ://gist.github.com/Maelan/e3f25ec0a6832ba1e0eb9584c3c58eff

```
        echo "$line" | iconv -f "$last" -t utf8
done
```

Exemple d'utilisation :

```
:::console
$  ./convert-mixed.sh < test > test-corrigé
0000000   l   i   g   n   e       e   n   c   o   d 303 251   e
    e   n       U   T   F   -   8  \n
0000030
Quel encodage ? [utf8]
0000000   l   i   g   n   e       e   n   c   o   d 351   e       e
    n       l   a   t   i   n   -   1  \n
0000031
Quel encodage ? [utf8] latin1
```

10

3.2. Déclarer l'encodage

Comme évoqué en introduction, plutôt que de se reposer sur une détection automatique, certains types de documents permettent — voire requièrent — de préciser leur encodage directement dans le texte du fichier. On va voir l'exemple des pages HTML et des documents LaTeX.

3.2.1. HTTP, HTML & XML

> *i* Au passage, notons ce tutoriel11 pour migrer son site web de latin-1 vers UTF-8.

11

Pour le web, les serveurs HTTP peuvent indiquer l'encodage avec un champ d'en-tête :

```
:::text
Content-Type: text/html; charset=‹ENCODAGE›
```

D'ailleurs, cet en-tête fournit aussi le type MIME du document (page HTML, fichier CSS, image PNG…).
Toutefois, cette technique nécessite un serveur (ce qu'on n'a pas pour consulter un fichier local) et empêche de fournir avec le même serveur des fichiers avec des encodages différents (cas d'un serveur mutualisé).
À la place, pour les pages HTML, on peut renseigner l'encodage… directement dans le fichier HTML. Ça semble bizarre puisqu'en théorie, on ne peut pas encore lire le fichier. Mais le socle ASCII vient à notre secours : quel que soit l'encodage, si on n'utilise que les caractères de l'ASCII, on pourra lire sans problème. On utilise une balise `<meta http-equiv />` (dans

10. https ://zestedesavoir.com/membres/voir/Taurre
11. https ://openclassrooms.com/courses/passer-du-latin1-a-l-unicode

`<head/>`) qui est l'équivalent de l'en-tête HTTP.

```
:::html
<meta http-equiv="Content-Type" content="text/html;
    charset=‹ENCODAGE›" />
```

En HTML5, cette balise a été simplifiée en :

```
:::html
<meta charset="‹ENCODAGE›"/>
```

Elle doit se trouver au tout début de `<head/>` afin de ne pas perturber la détection, et parce que le navigateur relit le fichier depuis le début dès qu'il l'a rencontrée. Seul le strict nécessaire doit précéder cette balise, avec uniquement des caractères ASCII (donc pas de BOM en UTF-8!).

Mauvais (il y a des caractères non-ASCII dans le commentaire) :

```
:::html
<html>
    <head>
        <!-- ligne nécessaire pour spécifier l'encodage : -->
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8" />
…
```

Bon :

```
:::html
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8" />
…
```

On peut trouver ici[12] les noms d'encodage autorisés (insensibles à la casse). Les plus intéressants pour nous sont `ISO-8859-1` (alias `latin-1`) et `UTF-8`. Facile!

Dans les langages XML en général (dont HTML), on peut également renseigner l'encodage dans le prologue XML, une sorte de balise spéciale optionnelle qui doit être placée sur la toute première ligne du fichier.

12. http ://www.iana.org/assignments/character-sets

```
:::xml
<?xml version="1.0" encoding="‹ENCODAGE›" ?>
```

i

À propos, XML (donc HTML) permet d'insérer un caractère Unicode arbitraire, même s'il n'existe pas dans l'encodage du code source : pour insérer le point de code U+CODE, la syntaxe est `&#xCODE;` (`&…;` est la syntaxe des entités XML, `#` signifie «nombre» et `x` signifie «hexadécimal»).
Un mécanisme similaire existe dans de nombreux langages, par exemples `\CODE` en CSS ou `\uCODE` en C.

3.2.2. LaTeX

L'encodage est très important avec LaTeX. Comme en HTML, il faut déclarer avec quel encodage est enregistré le fichier en le précisant en paramètre du paquet `inputenc` :

```
:::latex
\usepackage[‹ENCODAGE›]{inputenc}
```

(`latin1` ou `utf8` pour les plus courants). Les fichiers éventuellement inclus avec la commande `\input` doivent être enregistrés avec le même encodage, mais il n'y a pas besoin de le déclarer à nouveau.

3.3. Programmer

Enfin, lorsqu'on crée un programme, il faut bien sûr faire attention à l'encodage du texte qu'on manipule!

3.3.1. Le langage C

Le langage C est bas niveau. En C, on manipule directement les octets, avec le type `char`. En particulier, **un `char` est un octet, *pas* un caractère**. C'est le type `wchar_t` qui permet de stocker un caractère arbitraire[12].
La gestion des encodages en C de façon portable est excessivement compliquée si l'on se contente de la norme. En effet, le C est un vieux langage, créé dans les années 1970 lorsqu'on n'utilisait encore que des pages de code sur un *byte*. Tout était plus simple. Avec le développement d'encodages plus complexes, il a fallu adapter la norme… et comme d'habitude, le comité a décidé de ne pas normaliser grand-chose (pour ne pas trop contraindre les implémenteurs et pour garder la bibliothèque standard minimaliste).
Voici une synthèse des types disponibles et des encodages associés; évidemment, la norme laisse ces derniers non spécifiés, donc dans ce tableau figurent les valeurs usuelles.

Type des chaînes	Taille d'un caractère	Jeu de caractère & encodage	Version

`char[]`	**nombre variable de `char`** [13] **(1 `char` = 1 octet)**	**déterminé par la «*locale*»15 (donc dynamique)** — **Windows : Windows-1252 (taille fixe)** — **Linux : UTF-8 (taille variable)**	**C89**
`wchar_t[]`	exactement un `wchar_t` [14] (= 1, 2 ou 4 octets, selon le système)	dépend du système — Windows : UTF-16[11] — Linux : UTF-32	C99
`int16_t[]`	exactement un `char16_t` (= 16 bits)	dépend du système, prévu pour UTF-16	C11
`int32_t[]`	exactement un `char32_t` (= 32 bits)	dépend du système, prévu pour UTF-32	C11

TABLE 3.2. – Types et encodages en C

Ces encodages sont ceux utilisés par les fonctions de la bibliothèque standard, telles les fonctions de conversion16, mais bien sûr on fait ce qu'on veut de nos octets.

13

14

15

16

17

La console de Windows crée une difficulté supplémentaire puisqu'au lieu de la nouvelle page de code 1252, elle utilise toujours l'ancienne (850 ou 437) par défaut. Un logiciel qui respecte la *locale* écrira en Windows-1252 et ses accents s'afficheront mal dans la console.

11. Eh oui, la norme demande que les caractères soient de taille fixe, mais Windows a choisi UTF-16 restreint au BMP, comme on l'a déjà vu quand on a présenté UTF-16. Ceci permet un gain de mémoire par rapport à UTF-32.
12. La norme appelle «jeu étendu» (*extended character set*) l'ensemble de caractères supporté par le système.
13. La norme parle d'«encodage multi-octet» (*multibyte encoding*)14.
14. La norme parle de «caractères larges» (*wide characters*)13.
13. http ://www.sensi.org/~alec/man/man_h/wchar.html
14. http ://www.unix.com/man-page/FreeBSD/3/multibyte/
15. http ://pwet.fr/man/linux/conventions/locale
16. http ://pwet.fr/man/linux/fonctions_bibliotheques/mbtowc
17. http ://pwet.fr/man/linux/fonctions_bibliotheques/mbsinit

> Pour y remédier, demander à l'utilisateur de faire `CHCP 1252` ou, mieux, ajuster soi-même l'encodage de la console avec les fonctions `setConsoleOutputCP` et `GetACP` définies dans `<windows.h>`.

Bref, pour un résultat portable et sans prise de tête, il est avisé d'utiliser une bibliothèque tierce. L'implémentation de référence est la bibliothèque ICU[18], extrêmement complète. Également, j'ai déjà évoqué l'API iconv[4] intégrée à POSIX, pour les conversions d'encodages.
[18]
[19]

Je ne m'étendrai pas plus sur le sujet. Si vous voulez pousser plus loin, j'ai donné des mots-clés pour guider vos recherches. Pour plus de détail sur la norme C à ce sujet, lisez donc ceci[20], c'est un bon moyen de se rendre compte à quel point c'est l'enfer…
[20]

3.3.2. Autres langages

Les langages de plus haut niveau, ou tout simplement plus récents, peuvent mieux prendre en charge Unicode et les conversions d'encodages.
Par exemple, en Java, la classe `String`[21] utilise l'UTF-16. De plus, des fonctionnalités supplémentaires pour contrôler plus finement l'Unicode sont fournies par le *package* `Java.text`[22].
[21]
[22]

De même, Python permet de gérer Unicode et les encodages[23] facilement. De plus, Unicode est devenu le défaut en Python 3.

- La classe `unicode` (renommée `str` en Python 3…) est un texte Unicode décodé. C'est ce qu'il faut utiliser partout en interne, Unicode étant universel.
- La classe `str` (renommée `byte` en Python 3) est un texte encodé suivant un certain encodage. Il ne faut l'utiliser que pour les entrées et sorties.

En Python 2, `u"déjà"` est un objet de type `unicode` tandis que `"déjà"` est un objet de type `str`. Les méthodes `.encode(…)` et `.decode(…)` permettent de passer de l'un à l'autre; elles prennent en paramètre l'encodage du texte `str`.
[23]

En OCaml, la maigre bibliothèque standard ne prend pas en charge Unicode (jusqu'à récemment, la norme du langage utilisait latin-1), mais de nombreuses bibliothèques tierces le font, comme Camomile[24] (intégrée à Batteries[25]) ou Uutf[26].
[24]
[25]
[26]

J'espère que ce tutoriel aura aidé certains à comprendre ce qui se passe et à ne plus avoir peur

18. https ://fr.wikipedia.org/wiki/International_Components_for_Unicode
19. http ://man7.org/linux/man-pages/man3/iconv.3.html
20. https ://pdp.microjoe.org/forums/sujet/358/c-caracteres-unicode ?page=1#p6937
21. http ://docs.oracle.com/javase/6/docs/api/java/lang/String.html
22. http ://docs.oracle.com/javase/6/docs/api/java/text/package-summary.html
23. http ://sametmax.com/lencoding-en-python-une-bonne-fois-pour-toute/
24. https ://github.com/yoriyuki/Camomile
25. https ://github.com/ocaml-batteries-team/batteries-included
26. http ://erratique.ch/software/uutf

des «caractères spéciaux»…
Il n'y a rien de sorcier et tout programmeur (voire tout informaticien) devrait connaître ça pour ne pas faire n'importe quoi.

4. Liens

Quelques autres documents sur le sujet :

— «Ce que tout programmeur doit savoir»[1] (et sa version anglaise[2]) : un article de *Joel on Software* destiné à sensibiliser les programmeurs ;

— «Introduction aux jeux de caractères»[3] : un cours de Steve Frécinaux sur le site Openweb ;

— `charsets(7)`[4] : une page du manuel de Linux qui décrit les encodages couramment utilisés avant leur remplacement progressif par Unicode.

Ensuite, Wikipédia est très bien fournie sur le thème (ne pas hésiter à lire les articles anglais qui sont souvent plus complets). On peut par exemple consulter cet article général[5], celui-ci[10] pour apprendre comment fonctionne UTF-8, ou ce comparatif des encodages d'Unicode[6] (en anglais). Elle contient les tables de codes de tous les jeux utiles, par exemples : ASCII[2], CP850[7], latin-1[9], et même le JUC[11] qui y est plutôt bien organisé.

On peut aussi trouver les tables du JUC ici[12], avec des glyphes informatifs et les noms officiels des caractères en français (car la norme ISO 10646 est publiée conjointement en anglais et en français). Toutefois, cette page n'a plus été mise à jour depuis 2007 (Unicode version 5.0); la dernière version peut être trouvée en anglais là[13] (mais les ajouts sont peu susceptibles de vous intéresser).

Enfin, un outil pratique pour calculer l'UTF-8[14].

1
2
3
4
5
6
7
8
9
10
11
12
13

1. http ://french.joelonsoftware.com/Articles/Unicode.html
2. http ://joelonsoftware.com/Articles/Unicode.html
3. http ://openweb.eu.org/articles/jeux_caracteres
4. http ://man7.org/linux/man-pages/man7/charsets.7.html
5. https ://fr.wikipedia.org/wiki/Codage_des_caractères
6. https ://en.wikipedia.org/wiki/Comparison_of_Unicode_encodings
7. https ://fr.wikipedia.org/wiki/ASCII
8. https ://fr.wikipedia.org/wiki/CP850
9. https ://fr.wikipedia.org/wiki/latin-1
10. https ://fr.wikipedia.org/wiki/UTF-8
11. https ://fr.wikipedia.org/wiki/Table_des_caractères_Unicode
12. http ://www.unicode.org/fr/charts/
13. http ://www.unicode.org/charts/

14

14. http ://www.ltg.ed.ac.uk/~richard/utf-8.cgi

Liste des abréviations

API Alphabet Phonétique International. 16, 23, 31

ASCII American Standard Code for Information Interchange. 1, 2, 5–11, 15, 16, 18, 20, 24, 27, 28, 33

BMP Basic Multilingual Plane. 13–15, 30

BOM Byte Order Mark. 14, 15, 22, 23, 28

CHCP CHange CodePage. 17

EBCDIC Extended Binary Coded Decimal Interchange Code. 5

JUC Jeu Universel de Caractères. 11–13, 19, 33

NBSP Non-Breaking SPace. 8