

# Architecture et Système

Stefan Schwoon

Cours L3, 2024/2025, ENS Paris-Saclay

# Problème de codage

---



# Codage et traitement des caractères

---

Importance : échange de données, comparaisons, enjeux socio-culturel

Complications:

beaucoup de normes différents et partiellement compatibles ;

ignorance ou indifférence des programmeurs/utilisateurs/institutions

difficile à debugger

Sujet connexe : fonctionnement de base d'un terminal

# Normes pour représentation de texte

---

Dans un premier temps, un standard d'encodage affecte des caractères à des entiers, appelés *points de code*. P.ex.  $A \hat{=} 65$ ,  $\$ \hat{=} 36$  etc

Les standards les plus importants :

**ASCII** : 7 bit

**Latin-1** (= ISO-8859-1) : 8 bit

**Unicode** : un million de caractères, évolue encore

Solution technique pour représenter les points de code:

un octet par caractère pour ASCII/Latin-1, plus compliqué pour Unicode !

# Tableau ASCII (de Wikipédia)

---

PDF : <a href="#">en</a> <a href="#">[archive]</a> <a href="#">v</a> · <a href="#">d</a> · <a href="#">m</a>	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
<b>U+0000</b>	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
<b>U+0010</b>	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
<b>U+0020</b>	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
<b>U+0030</b>	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
<b>U+0040</b>	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
<b>U+0050</b>	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
<b>U+0060</b>	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
<b>U+0070</b>	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

# Codes à 7 ou 8 bits

---

**ISO-646** in 1963/1972, version américaine : ASCII

codes de 7 bits, caractères de contrôle avec codes  $< 32$  ;

suffit pour programmer et pour écrire du texte anglais, mais pas plus.

Codes de 8 bit :

**Codepage 437** (et d'autres), dans les IBM PC (1981) ; permet de dessiner des tableaux en mode texte.

**ISO-8859** (à partir de 1985) : diverses variants, les plus importantes étant **ISO-8859-1** (= Latin-1) pour les langages ouest-européens, et **ISO-8859-15** (= Latin-9) qui contient tous le caractères français

**Windows-1252** (arrivée avec MS Windows) : remplit quelques positions inutilisés de Latin-1

# Unicode/ISO 10646

---

(à partir de 1991) effort pour définir un codage *universelle*, c'est à dire pour pouvoir représenter tous les langages du monde.

Unicode permet 1 million de caractères différents (dont 144,000 utilisés).

On affecte un point de code hexadécimal à chaque caractère :

p.ex. **A** = U+0041, **é** = U+00E9, etc

Les points de code d'Unicode sont compatible en arrière avec Latin-1.

(Mais pas avec Windows-1252.)

**Usage actuel** : plus de 80% des sites Internet en Unicode, 10% en Latin-1

# Représentation des points de code

---

ASCII/Latin-1/Windows : facile, un caractère égale un octet

Unicode : plusieurs standards différents

UTF-32 (BE/LE): 4 octets par caractère

UTF-16 (BE/LE): 2 ou 4 octets par caractère

UTF-7: codage vers des codets 'imprimables' en ASCII

UTF-8: codage à longueur variable, le plus utilisé



# Découpage UTF-8

---

Codes représentés par un ou plusieurs octets.

Pour les ASCII: un seul octet entre 0..127.

Pour les autres : 2 à 4 octets entre 128..255. Exemple :

Les codes entre 128 (= 0x80) et 2047 (= 0x7FF) prennent deux octets.

Le code pour “é” égale 233, en binaire sur 11 positions: 00011101001

Le premier octet est composé de 110 suivi par les cinq premiers bits du code, le deuxième de 10 suivi par les six autres bits.

Ça donne 11000011 10101001 = 0xC3 0xA9.

# Les terminaux

---

**Terminal** : à l'origine, les *terminaux* (ou *consoles*) sont des simples machines branchées sur un ordinateur central.



Un terminal est équipé d'un clavier et d'un écran capable d'afficher du texte (mais pas du graphisme).

# Fonctionnement d'un terminal

---

Un terminal fonctionne d'une manière très simple :

Échange des caractères avec l'ordinateur central

Affichage des caractères saisis sur clavier  
+ transfert vers l'ordinateur

Affichage des données venues de l'ordinateur

Distinction entre caractères dits **imprimables** ou de **contrôle** (retour chariot CR, saut de ligne LF, tabulation TAB, appel BEL, retour arrière BS, ...)

# Émulation de terminal

---

Les applications dites 'terminal' (ou similaire) dans un Unix/MacOS d'aujourd'hui simulent des terminaux historiques.

**Démonstration** (keydump.c)

**Mode echo** : choisit si le terminal affiche les caractères venus du clavier ou pas

**Mode canonique** : choisit si les touches sont dispos tout de suit ou seulement après une nouvelle ligne

Voir aussi : `stty`

# Approfondissement : gestion des touches

---

Mais comment le terminal apprend-t-il que l'utilisateur a tapé un A ?

Auprès du clavier, toute touche possède un numéro. Disons que la touche A possède le code 38.

Lorsqu'on tape sur une touche, le clavier signale une **interruption** auprès du processeur central. Le processeur va alors interrompre son travail actuel et sauter vers un soi-distant *gestionnaire d'interruption*.

Le système d'exploitation fait pointer ce gestionnaire vers son propre code qui reçoit alors le code 38.

---

Le système fait appel à un **pilote** qui traduit le code de touche vers un **symbole**. Dans notre cas, un pilote bien configuré traduit le code 38 en un symbole `a` ou `À` etc en fonction des *modificateurs* (Shift, Ctrl, Alt) en vigueur.

Cas spécial de Linux/X11:

Le gestionnaire génère un **évènement** auprès de l'**application de premier plan** qui contient le nom de touche et le symbole (voir `xev`, `xmodmap`).

Si cette application est un terminal, il traduit le symbole en fonction de l'encodage en vigueur et met le résultat à la disposition d'un processus.

# Terminal et encodage

---

**Demonstration :** Utiliser `keydump` avec encodage UTF-8 et ISO-8859-1 (Latin1) et avec caractères accentués.

L'encodage en vigueur guide l'interprétation (par le terminal) des données qui lui sont envoyées.

Il est essentiel que le terminal et ses applications sont d'accord sur l'encodage !

# Séquences de contrôle dans le terminal

---

Certains caractères **non-imprimables** ont une interprétation spéciale dans le terminal :

des contrôles simples (nouvelle ligne etc)

des séquences commençant par 'escape' (27), p.ex.:

`ESC[31m` : afficher texte en rouge

`ESC[42m` : arrière-plan en vert

`ESC[0m` : arrière-plan en noir

`ESC[2J` : effacer l'écran

`ESC[10;20H` : mettre le cursor sur position (10,20)

Le caractère ESC correspond à `\e` en C. Dans le shell, `echo -e "\e..."`

En C, la bibliothèque `ncurses` permet de gérer ces séquences.



# Parenthèse : nouvelle ligne

---

Pour des raisons historiques, il existe deux standards pour représenter une nouvelle ligne :

**format Unix** : un seul octet (10 décimal) entre deux lignes

**format DOS**: nouvelle ligne représentée par CR+LF = octets 13+10

La plupart des éditeurs de texte sait traiter les deux formats.

Conversion explicite : `dos2unix`, `unix2dos`

# Traitement de text sous Unix

---

La plupart des outils (`grep` etc) ignorent les codages ;  
ils traitent des séquences d'octets.

Des éditeurs de texte (`vi`, `emacs`) ou navigateurs (`firefox`) encodent selon les options en vigueur (ou selon leur propre 'intelligence').

`file` essaie à déterminer le codage des fichiers texte ;  
(sinon, `hexdump -C` s'avère utile...)

`iconv` sait convertir entre une grande variété d'encodages.

# Faire connaître le codage d'un document

---

En HTML ou XML, dans les entêtes :

HTML : `<meta http-equiv="content-type"  
content="text/html; charset=ISO-8859-1"/>`

depuis HTML5 : `<meta charset="ISO-8859-1">`

XML: `<?xml version="1.0" encoding="UTF-8"?>`

Pour des fichiers texte : pas de standard, il faut spécifier autrement !

# Unicode : formes composées et décomposées

---

Problème spécifique dans Unicode : un même caractère peut être codé sous plusieurs formes !

P.ex., é = U+00E9 (forme composée) ou U+0065 U+0301 (forme décomposée)

En fait, la partie U+03xx de Unicode contient des diacritiques qui se combinent avec le caractère précédent.

Une application conforme aux standards Unicode est censée traiter ces séquences identiquement !

# Les équivalences et formes normales

---

Unicode connaît deux formes d'équivalence entre (séquences de) caractères :

**Canonical equivalence** (équivalence forte) – deux chaînes fortement équivalentes sont à traiter identiquement dans tous les contextes

**Compatibility equivalence** (équivalence faible) – équivalence entre caractères qui ont la même apparence mais une sémantique différente selon le contexte (p.ex. 2 normal et <sup>2</sup> superscript)

Formes normales canoniques : NFC/NFD (composé/décomposé)

Formes normales de compatibilité : NFKC/NFKD (composé/décomposé)

L'utilitaire `uconv` sait convertir un texte aux formes normales.

# Forme de codage

---

Les codets sont typiquement sous la forme d'une valeur 0..255.

Normalement, on représente un codet dans sa forme binaire dans un octet.

Mais parfois, on souhaite une représentation "imprimable" :

P.ex., les protocoles Internet (mail, web) ne permettent que des caractères imprimables dans les entêtes, dans les adresses, ...

Les formes imprimables résistent mieux aux erreurs car tout codage habituel est compatible avec ASCII.

# Exemples de forme de codage non-trivial

---

Codage 'pourcent' dans les adresses web, p.ex. :

`https://fr.wikipedia.org/wiki/Caract%C3%A8re`

Codage 'quoted-printable', p.ex. dans les entêtes mail :

Subject: `=?utf-8?Q?T=C3=A9st?='`

(donne 'Tést' comme sujet du mail)

Codage 'base64', pour le même contenu :

Subject: `=?utf-8?B?VMOpC3Q=?='`

Dans base64, on se donne 64 symboles (A-Za-z0-9+/) qui code des séquences de 6 bits ; quatre symboles font trois codets.