Types et valeurs de base – Encodage

Lycée Monte-Cristo Allauch – 1ère NSI

1 Encodage des entiers naturels (ℕ)

1.1 Le décimal (Base 10)

Vocabulaire

- "Déci" → 10 termes possibles : de 0 à 9
- Un chiffre : un terme du langage
- Base utilisée par les humains

Notation Si X est un nombre décimal, on le note : X_{10} ou X

1.2 Le binaire (Base 2)

Vocabulaire

- "Bi" → 2 termes possibles : 0 et 1
- Un "bit" (de l'anglais Binary Digit) : un terme du langage

Notation Si X est un nombre binaire, on le note : X_2

Exemple $11100101_2 = 229_{10}$

1.3 L'hexadécimal (Base 16)

Vocabulaire

- "Hexadéci" → 16 termes possibles : de 0 à 9 et de A à F
- Un terme du langage peut être traduit en une suite de 4 bits

Notation Si X est un nombre hexadécimal, on le note : X_{16}

Exemple $E5_{16} = 11100101_2 = 229_{10}$

Hexadécimal	Décimal
A	10
B	11
C	12
D	13
E	14
F	15

1.4 Le n-aire (Base n)

Représentation générale en base n Pour passer de la base décimale à une base n, il existe 2 méthodes :

- Décomposer le nombre décimal en sommes de puissances de n.
- Remonter les restes des divisions euclidiennes successives du nombre par n.

Exemples

$$\begin{aligned} 229_{10} &= 1\times2^7+1\times2^6+1\times2^5+0\times2^4+0\times2^3+1\times2^2+0\times2^1+1\times2^0 \\ &= \quad 1 \qquad 1 \qquad 1 \qquad 0 \qquad 0 \qquad 1 \qquad 0 \qquad 1 \\ &= 11100101_2 \end{aligned}$$

$$\begin{aligned} 229_{10} &= 14\times16^1+5\times16^0 \\ &= \quad 14 \qquad\quad 5 \\ &= \quad E \qquad\quad 5 \\ &= E5_{16} \end{aligned}$$

229	16	
5	14	16
	14	0

$$\begin{aligned} 389_{10} &= 19\times20^1+9\times20^0 \\ &= \quad 19 \qquad\quad 9 \\ &= \quad J \qquad\quad 9 \\ &= J9_{20} \end{aligned}$$

389	20	
9	19	20
	19	0

229	2	
1	114	2
0	57	2
1	28	2
0	14	2
0	7	2
1	3	2
1	1	2
1	0	

Notation Si X est un nombre n-aire, on le note : X_n

Autres exemples de bases

- Base 3 = Trinaire
- Base 8 = Octal
- Base 9 = Nonaire
- Base 12 = Duodécimal
- Base 20 = Vigésimal
- Base 60 = Sexagésimal
- Base 150 = Indienne
- Base d'or (puissances du nombre d'or $\frac{1+\sqrt{5}}{2}$)
- Base de Fibonacci (termes de la suite de Fibonacci)
- Base factorielle (termes des factorielles)
- ...

2 Encodage des entiers relatifs (ℤ)

Représentation

- Définir la longueur de la représentation en binaire
- Bit de poids fort (à l'extrême gauche) = 0 si ≥ 0, 1 si < 0
- Le reste des bits représente l'entier
- Entier positif : entier naturel (voir partie 1)
- Entier négatif : principe du complément à deux

Intervalles de valeurs encodables

- Sur 8 bits (1 octet) : $[-128; 127]$
- Sur 16 bits (2 octets) : $[-32768; 32767]$
- Sur 32 bits (4 octets) : $[-2147483648; 2147483647]$
- Sur 64 bits (8 octets) : $\left[-9,223372037\times10^{18}; 9,223372037\times10^{18}-1\right]$

Le complément à deux

1. Convertir la valeur absolue de l'entier en binaire
2. Inverser les bits
3. Ajouter 1 au résultat

Exemple : -65 sur 8 bits

1. $|-65_{10}| = 65_{10} = 01000001_2$
2. $01000001_2 \rightarrow 10111110_2$
3. $10111110_2 + 1 = 10111111_2$ $\rightarrow -65_{10} = 10111111_2$

3 Encodage des nombres réels ($\mathbb{R}$)

C'est comme pour les entiers !

— Somme de puissances de 10 : $652,375 = 6 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$

— Somme de puissances de 2 : $110,101_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$

3.1 Conversion de binaire vers décimal

$$\begin{aligned} 110,101_2 &= 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 4 + 2 + 0 + 0,5 + 0 + 0,125 \\ &= 6,625_{10} \end{aligned}$$

3.2 Conversion de décimal vers binaire

3.2.1 Première méthode (virgule fixe)

Exemple 1 $65,54_{10} = 2^6 + 2^0 + 0,54_{10}$
$= 1000001_2 + 0,54_{10}$

$$\left.\begin{aligned} 0,54 \times 2 &= \mathbf{1},08 \\ 0,08 \times 2 &= \mathbf{0},16 \\ 0,16 \times 2 &= \mathbf{0},32 \\ 0,32 \times 2 &= \mathbf{0},64 \\ 0,64 \times 2 &= \mathbf{1},28 \\ 0,28 \times 2 &= \mathbf{0},56 \\ 0,56 \times 2 &= \mathbf{1},12 \\ 0,12 \times 2 &= \mathbf{0},24 \\ 0,24 \times 2 &= \mathbf{0},48 \\ &\dots \end{aligned}\right\} 65,545_{10} = 1000001,100010100...{}_2$$

Exemple 2 $65,375_{10} = 2^6 + 2^0 + 0,375_{10}$
$= 1000001_2 + 0,375_{10}$

$$\left.\begin{aligned} 0,375 \times 2 &= \mathbf{0},75 \\ 0,75 \times 2 &= \mathbf{1},5 \\ 0,5 \times 2 &= \mathbf{1} \end{aligned}\right\} 65,375_{10} = 1000001,011_2$$

Différence entre premier et deuxième exemple

— Exemple 1 : suite de bits après la virgule infinie

— Exemple 2 : suite de bits après la virgule finie

→ Pourquoi ? Parce que la partie décimale du deuxième exemple est une somme de puissances de 2 !

Conclusion partielle Ainsi, un nombre à développement décimal fini en base 10 ne l'est pas forcément en base 2 (par contre à l'inverse, un nombre à développement décimal fini en base 2 l'est forcément en base 10). De plus, on ne parle pas ici des nombres à développement décimal infini en base 10...
Un ordinateur ne peut donc pas représenter n'importe quel nombre réel. Ce qui importe, c'est la précision avec laquelle on souhaite représenter le nombre réel.

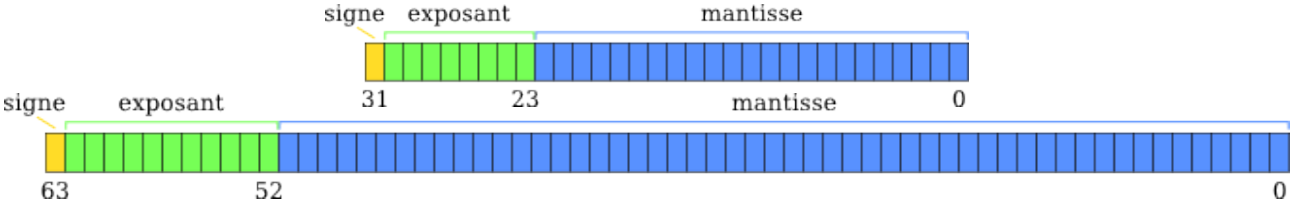
3.2.2 Deuxième méthode (virgule flottante – norme IEEE 754)

La norme IEEE 754 définit la façon de coder un nombre réel. Cette norme propose de coder le nombre sur 32/64 bits et définit 3 composantes :

— Le **signe** est représenté par le bit de poids fort

— L'**exposant** positif est codé sur les 8/11 bits suivants le bit de signe

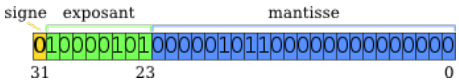
— La **mantisse** (bits après la virgule) sur les 23/52 bits restants



Codage de 65,375 selon la norme IEEE 754 sur 32 bits

1. Conversion en binaire (grâce à la première méthode) : $65,375_{10} = 1000001,011_2$
2. Mise sous la forme 1,**partie décimale** : $1000001,011 = 1,\mathbf{000001011} \times 2^6$
3. La partie décimale (mantisse) sur 23 bits est donc : **00000101100000000000000**
4. Décalage de l'exposant : $6 + 127 = 133_{10} = 10000101_2$

Pourquoi décaler l'exposant ? Il est possible que l'exposant soit négatif, pour représenter des nombres du type $0,0015542_{10}$. Il faut donc pouvoir représenter des exposants négatifs. La solution trouvée est d'effectuer un décalage de l'exposant de $2^{nb\ de\ bits\ de\ l'exposant-1} - 1$. Ainsi, pour 32 bits, le décalage est de $2^{8-1} - 1 = 127$ et pour 64 bits, le décalage est de $2^{11-1} - 1 = 1023$.



En hexadécimal, cela donne 4282C000

Règles de codage

— L'exposant 00000000 est interdit

— L'exposant 11111111 est réservé pour signaler des erreurs, et est appelé NaN (Not a Number)

4 Encodage des valeurs booléennes

Faux/False ⇒ 0
Vrai/True ⇒ 1

5 Encodage du texte

5.1 La représentation des caractères

Des normes définissent comment représenter les caractères, selon les caractères à représenter :
— Unicode — ASCII — UTF-8 — Une multitude d'autres...

Pourquoi plusieurs normes ? → Problèmes de différences des alphabets entre les langues Rien qu'en anglais et en français, on n'utilise pas les mêmes caractères. Ceci se voit encore plus si on compare à des langues comme l'arabe, le chinois, le japonais, le coréen, etc... qui utilisent des caractères totalement différents.

Règle importante Les normes d'encodage de caractères respectent la **casse**, c'est-à-dire différencient les minuscules et les majuscules (a ≠ A).

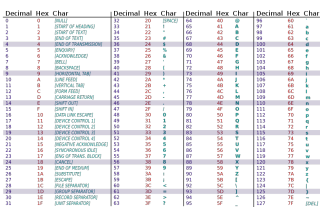
5.2 ASCII (American Standard Code for Information Interchange)

5.2.1 ASCII basique (début des années 1960)

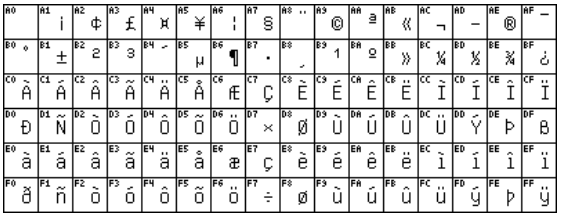
Norme américaine d'encodage des caractères :
— Ne sait représenter que les caractères américains, donc par exemple le caractère "è" n'est pas représenté
— Unicité des correspondances (Un code ↔ un caractère)
— Sur 7 bits → 128 caractères représentés

5.2.2 ASCII étendu (fin des années 1960)

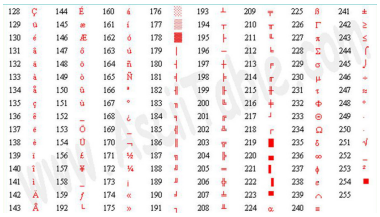
Arrivée de l'informatique en Europe. Plusieurs extensions de ASCII :
— Ajout d'un 8ème bit → 2 fois plus de caractères
— Normalisation de l'ISO/CEI 8859-1 au niveau européen, aussi appelée Latin-1 ou Europe Occidentale



(a) Table ASCII basique



(b) ISO 8859-1



(c) Table ASCII étendue de IBM

5.3 Unicode (début des années 1990)

Problème Beaucoup de tables de conversion à travers le monde

Solution Unification au niveau mondial grâce à une norme commune appelée **Unicode** [1], aujourd'hui utilisée pour les claviers d'ordinateur :
— Identifiant unique pour chaque caractère existant, appelé **Point de code**
— Les 128 premiers points de code sont compatibles avec ASCII
— Il existe 1 114 112 points de code, représentés sur l'intervalle $[00000000_{16}; 0010FFFF_{16}]$ ($2^{20} = 1048576$)
— Les 65 536 premiers points de code (2 octets) représentent le BMP (Basic Multilingual Plane), c'est-à-dire les caractères les plus communément utilisés dans le monde

Exemple 'A' a pour point de code 00000041_{16}

1. `https://www.unicode.org/charts/fr/`

5.3.1 Unicode Transformation Format (UTF-32) – Longueur 32 bits / 4 octets

— Un point de code = une représentation en UTF-32
— Un octet inutilisé (à l'heure actuelle !)
— Utile si on désire une longueur fixe ou un accès unique des caractères

Exemples de codes UTF-32
— "A" a pour code 00000041_{16}
— "Ω" a pour code $000003A9_{16}$
— "文" a pour code 00006587_{16}
— "𐌼" a pour code 00010384_{16}

5.3.2 UTF-16 – Longueur 16 bits / 2 octets

— Représentation des points de code de l'intervalle $[0000_{16}; FFFF_{16}]$
— Les caractères de l'intervalle $[010000_{16}; 10FFFF_{16}]$ sont représentés par des paires de code de 16 bits
→ Longueur variable de 1 ou 2 codes
— Optimisé pour BMP (65536 premiers points de code)

Exemples de codes UTF-16
— "A" a pour code 0041_{16}
— "Ω" a pour code $03A9_{16}$
— "文" a pour code 6587_{16}
— "𐌼" a pour code $D800\ DF84_{16}$

5.3.3 UTF-8 – Longueur 8 bits / 1 octet

— Représentation des points de code de l'intervalle $[00_{16}; FF_{16}]$
— Les caractères de l'intervalle $[000100_{16}; 10FFFF_{16}]$ sont représentés par 2 à 4 codes de 8 bits
→ Longueur variable de 1 à 4 codes
— Optimisé pour ASCII et la majorité des langages
— Est représenté sur 1 octet (unité la plus utilisée !)

Exemples de codes UTF-8
— "A" a pour code 41_{16}
— "Ω" a pour code $CE\ A9_{16}$
— "文" a pour code $E6\ 96\ 87_{16}$
— "𐌼" a pour code $F0\ 90\ 8E\ 84_{16}$

Pour UTF-16 et UTF-8, comment savoir si le code est en plusieurs parties ? Étant donné qu'il y a parfois besoin de plusieurs codes pour représenter un caractère (en raison de la longueur variable de ces deux formats), des préfixes sont réservés pour préciser que le caractère représenté nécessite plusieurs codes.
Les 2 tableaux suivants représentent les correspondances entre points de code et valeurs UTF-16 et UTF-8 :

Point de code	Valeur en UTF-16
xxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxx
000yyyyyxxxxxxxxxxxxxxxx	110110zzzzxxxxxx 110111xxxxxxxxxx

zzzz = yyyyy - 1

Les caractères nécessitant 2 codes UTF-16 seront donc compris dans l'intervalle $[D800_{16}; DFFF_{16}]$

Point de code	Valeur en UTF-8
00000000 0xxxxxxx	0xxxxxxx
00000yyy yyxxxxxx	110yyyyy 10xxxxxx
zzzzyyyy yyxxxxxx	1110zzzz 10yyyyyy 10xxxxxx
000uuuuu zzzzyyyy yyxxxxxx	11110uuu 10uuzzzz 10yyyyyy 10xxxxxx

5.4 Autres représentations

Liste non exhaustive des autres représentations
— Windows 1252 (Microsoft en Europe)
— EBCDIC (IBM pour cartes perforées)
— ISCII (Inde)
— VISCII (Vietnam)
— ...

6 Les types de données

Mais alors comment sait-on si c'est un entier naturel, un entier relatif, un réel, une suite de booléens ou un texte ? Si on vous donne une suite de 32 bits, comment savoir quelle valeur est représentée ?
⇒ C'est le principe du typage des données !