PF1 — Principes de Fonctionnement des machines binaires

Jean-Baptiste Yunès
Jean.Baptiste.Yunes@univ-paris-diderot.fr

Version 1.12

Problème :

que fait un langage comme Java avec différents types dans une même expression ?

différents cas à considérer…

L’affectation d’une variable avec un littéral (une constante écrite avec des chiffres dans un code source)

si la valeur peut-être représentée dans le type de la variable, ok

sinon erreur à la compilation!

```
byte b = 12;
System.out.println(b);

b = 255;
System.out.println(b);
```

En général, les valeurs entières d'un type plus petit que `int` sont promues comme valeurs entières `int`

Lorsque nécessaire les `int` sont promus comme `long`

L'opération est appelée **promotion** (widening)

Les promotions sont implicites (car non dangereuses)

```
byte b = 12;
System.out.println(b);

short s = b;
System.out.println(s);

int i = s;
System.out.println(i);

long l = i;
System.out.println(l);
```

La **promotion** (widening) consiste simplement en l'extension du signe

i.e. on complète à gauche avec le signe de la valeur à compléter

rappel : les positifs ont une *infinité* de 0 à gauche, les négatifs une *infinité* de 1 à gauche…

Pour l'inverse, c'est-à-dire la réduction d'un grand entier dans un plus petit, l'opération doit être explicite (en Java)

Le nom de l'opération utilisée est le cast ou la conversion ou encore **coercition**

coercition : action de contraindre par contrainte physique (ou psychologique)

La solution est de convertir explicitement si nécessaire (réduction entière, narrowing)

```
int v = 2293759;
short vv = (short)v;
System.out.println(v);
System.out.println(vv);
```

de la sorte la valeur produite est tronquée dans son écriture, on ne retient que les bits de poids faibles.

Les choses changent aussi lorsqu'on effectue des opérations dans des expressions

ex : `(a+b)*(c+d)`, comment ça marche ?

Exemples :

```
short a,b,c; a=0; b=0;
c = a+b;
```

problème car `a+b` est évaluée dans le type `int`, a et b sont promus comme `int` et le calcul effectué ce qui produit un `int`. Mais on ne peut ranger a-priori sans danger un `int` dans un `short`…

il faut le demander explicitement, de la sorte on est encouragé à réfléchir à ce que l'on fait…

```
short a,b,c; a=0; b=0;
c = (short)(a+b);
```

Pour les opérateurs binaires (c'est-à-dire avec deux opérandes) :

c'est toujours le plus grand type qui l'emporte

attention : le plus petit type pour calculer est le type `int`

on a `byte<short<int<long<float<double`

et accessoirement `char<int`

`byte+byte` ⟼ converti en `int+int` ⟼ `int`

`short*int` ⟼ converti en `int*int` ⟼ `int`

Un peu de syntaxe Java

Les littéraux d'entiers :

littéral entier en base 10, suite de chiffres décimaux (éventuellement préfixés du signe + ou du signe –) mais ne commençant pas par le chiffre `0`

```
+123, 1993, -56
```

Les littéraux d'entiers :

littéral entier en base 8, suite de chiffres octaux (`0`—`7`), préfixés par `0` et le tout éventuellement préfixé du signe `+` ou du signe `-`

`+0123`, `01773`, `-056`

Les littéraux d'entiers :

littéral entier en base 16, suite de chiffres hexadécimaux (`0`—`9A-F`), préfixés par `0x` et le tout éventuellement préfixé du signe `+` ou du signe `-`

`+0x123`, `0x1773`, `-0xBEEF`, `0xDEAD`

Les versions modernes de Java :

littéral entier en base 2, suite de chiffres binaires (`01`), préfixés par `0b` et le tout éventuellement préfixé du signe + ou du signe -

```
+0b111, 0b00101
```

Pour aider à la lecture des littéraux un peu longs on peut utiliser le caractère _ (soulignement) qui n'est utile qu'à la décoration et permet de grouper des chiffres

par exemple `int i = 123_456_789;` permet de faire apparaître les milliers, millions, etc.

Les littéraux d'entiers :

pour les entiers `long` les littéraux ont même la même écriture additionnée du suffixe `l` ou `L`

`0xBAD_F00D_FEE1_DEADL` est un `long`

`-0711` aussi

Quelques opérateurs Java…

Attention si beaucoup d’opérateurs Java utilisent la « syntaxe mathématique », il s’agit d’opérateurs qui opèrent sur des représentations particulières

Sémantique de l’arithmétique modulaire

L’addition : +

Attention! Il y a **deux opérations d’addition différentes** sur les entiers en Java!

l’addition agit différemment sur des `int` ou des `long`!

la taille de la représentation n’est pas la même!

les calculs sont modulaires mais pas avec le même modulo!

Pour la soustraction – et la multiplication *

idem…

Et la division ?

Les valeurs sur lesquelles on agit sont des entiers, il s'agit donc d'une **division entière**

Rappel : une division entière produit un quotient et un reste

en Java le calcul du quotient se note `/`

en Java le calcul du reste se note `%`

Attention : la division entière des langages de programmation n'est pas toujours la division Euclidienne (pour les nombres négatifs)

Rappel (division Euclidienne)

pour les entiers a,b la division Euclidienne leur associe l'unique couple q,r (resp. quotient et reste) tels que

$a=qb+r$, avec $0 \leq r < b$

exemple : $121 = 9 \times 13 + 4$

$a = 121, b = 13, q = 9, r = 4$

pour les relatifs a,b, il faut y ajouter une condition supplémentaire (arbitraire mais le choix n'est pas mathématiquement neutre).

Habituellement on définit q,r (resp. quotient et reste) tels que
$a=qb+r$, avec $0 \leq r < |b|$ (reste positif)

exemple : (-121/13) : -121 = -10x13 + 9

exemple : (121/-13) : 121 = -9x-13 + 4

exemple : (-121/-13) : -121 = 10x-13 + 9

un autre choix consiste à définir q,r tels que a=qb+r avec q le quotient de la division Euclidienne de |a| par |b|, (quotient des valeurs absolues)

exemple : (-121/13) : -121 = -9x13 + -4

exemple : (121/-13) : 121 = -9x-13 + +4

exemple : (-121/-13) : -121 = 9x-13 + -4

Selon les langages les choix diffèrent :

langages Python, outil Excel (avec ENT,MOD pas avec QUOTIENT!)

-121/13 = -10x13 + 9, quotient -10 reste 9

outil bc, langages Java, C, OCaml

-121/13 = -9*13 + -4, quotient -9 reste -4

On reviendra sur ce point, mais le langage Java possède aussi des **opérateurs de comparaison**

plus-petit-que, plus-petit-ou-egal, etc

`<`, `<=`, `==`, `!=`, `>`, `>=`

Attention! Ces opérateurs aussi sont différents selon les types des arguments (`int`, `long`, `char`)

Opérations particulières

En base 2 à quoi correspondent la multiplication par 2 et la division par 2 ?

par extension pour 2^n ?

la multiplication par 10 en base 10

1834 * 1<u>0</u> = **1834**<u>0</u>

la multiplication par 100 en base 10

1834 * 1<u>00</u> = **1834**<u>00</u>

la multiplication par 2

1834 * 2 = 3668

en binaire ($2_{10} = 10_2$)
11100101010*10=**111001010100**

1834 * 16 = 29344

en binaire ($16_{10} = 10000_2$)
11100101010*10000=**11100101010**0000

la division (entière)

1834 / 2 = 917

en binaire

11100101010/10=**1110010101**

1834 / 16 = 114

en binaire

11100101010/10000=**1110010**

Les opérations correspondantes sur les mots sont appelées décalage vers la gauche et vers la droite
En java elles se notent à l'aide des opérateurs <<, >>> et >>
leur argument est le nombre de décalages unitaires (donc la puissance de 2 utilisée comme multiplicateur ou diviseur)

```
int v = 1834;
System.out.println(v<<1); // 2 =2^1
System.out.println(v<<4); // 16=2^4
System.out.println(v>>1); // 2 =2^1
System.out.println(v>>4); // 16=2^4
```

```
for (int i=0; i<32; i++)
  System.out.println(1<<i);
```

Comment cela fonctionne t-il avec les signes ?

```
for (int i=0; i<32; i++)
  System.out.println(-1<<i);
```

Comment cela fonctionne t-il avec les signes ?

```
for (int i=0; i<32; i++)
  System.out.println(-1>>i);
```

```
for (int i=0; i<32; i++)
  System.out.println(-1>>>i);
```

>> utilise l'extension de signe

>>> ne fait pas d'extension du signe

Bits, Octets, Mémoire…

Les ordinateurs sont équipés de dispositifs fournissant une mémoire (de quoi stocker et relire des bits)

L'unité de mesure de la mémoire est **l'octet** (byte) : 8 bits

C'est l'octet car c'est historiquement la plus petite quantité d'information manipulable directement…

i.e. dans une machine on ne peut pas lire ou écrire un seul bit à la fois (pour des raisons d'efficacité)

toujours des paquets de 8 bits (octets) : 8, 16, 32, 64, 128 bits…

par conséquent le stockage d’une donnée sur 32 bits (un entier du langage Java par exemple) occupe 4 cases consécutives de la mémoire

autrement dit on peut considérer la mémoire comme un espace d’écriture des nombres en base 256…

Un problème :

nous écrivons de gauche à droite

mais de nombreux autres peuples écrivent de droite à gauche, de haut en bas, en boustrophédon (keskeussé ?)

Y a t-il un équivalent en machine ?

Oui!

adresse mémoire	0	1	…	a	a+1	a+2	a+3	…
chiffres base 256				c0	c1	c2	c3	

petit indien - petit boutiste - petit boutien - little endian - least significant byte first (LSB)

adresse mémoire	0	1	…	a	a+1	a+2	a+3	…
chiffres base 256				c3	c2	c1	c0	

grand indien - grand boutiste - grand boutien - big endian - most significant byte first (MSB)

Intel x386 x386-64, petit-boutiste

Sparc, grand-boutiste

Power-PC, comme vous voulez! (bi-endian)

JVM (Java Virtual Machine), grand-boutiste

NBO (Network Byte Ordering), grand-boutiste

Le problème existe aussi pour la transmission des chiffres binaires en série (serial) :

bit numbering - bit endianness

Ce niveau n’est jamais accessible au logiciel

mais est crucial dans une transmission en série…

il existe même des choses exotiques mélangeant les deux!

middle-endian (vraiment exotique)

rechercher « NUXI problem »

Attention! Si la mémoire est accessible par octets seulement (ou paquets d'octets) en interne les opérations sont réalisées avec des registres de taille fixe

Par exemple sur une machine dite 32-bits en général cela signifie que lesdits registres sont de taille 32-bits

Sur une machine 64-bits…

Attention! L'ordre des bits à l'intérieur des octets est sans signification car on ne peut y accéder directement!

Que faire avec les réels ?

ok pour les entiers, mais les réels ?

la situation est bien pire, puisqu'on ne dispose que d'un ensemble fini de mots pour représenter un ensemble non dénombrable!

on ne codera donc, au mieux, qu'un sous-ensemble fini dénombrable des réels… un sous-ensemble de densité nulle!

le choix du sous-ensemble…

Il existe deux normes très communément utilisées pour la représentation des flottants (abréviation pour nombres en virgule flottante)

IEEE 754

simple précision (sur 4 octets)

double précision (sur 8 octets)

la norme définit aussi certaines valeurs spéciales utiles : zéro, dénormalisés, infinis, indéterminés, etc

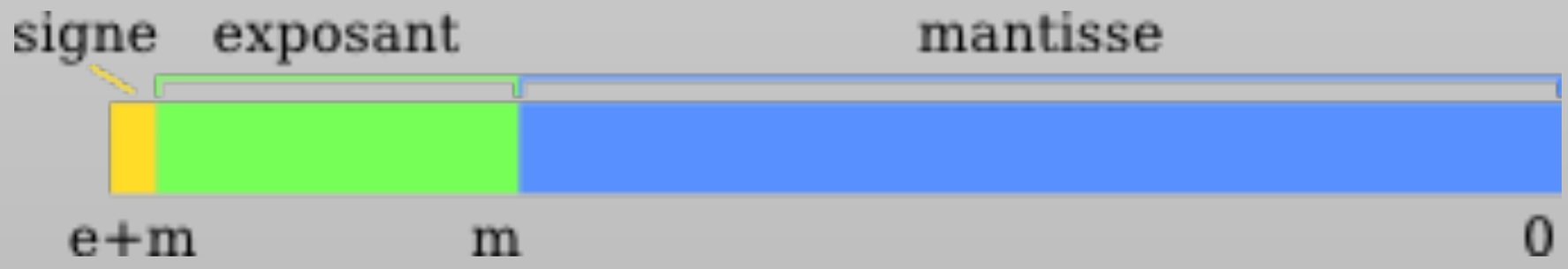
les nombres normalisés

la règle générale est d'utiliser

un bit de signe (bit de poids fort), qui code le signe s = +1 ou -1, 0 code s=+1 et 1 code s=-1

un exposant (N_e bits qui suivent), e est combiné avec un excès (aussi appelé décalage, $d=2^{Ne-1}-1$). Simple N_e=8, double N_e=11 (resp. d=127, d=1023)

une mantisse (les $N-N_e-1$ bits restants), m. Simple sur 23 bits, double sur 52 bits



valeur = $s \times (1+m) \times 2^{e-d}$

tout nombre binaire a son chiffre de poids le plus fort égal à 1 (sauf le nombre 0, cas particulier - voir plus tard)

par conséquent le 1 de tête peut être omis… (optimisation)

soit le nombre 5,34375 sa représentation en base 2 est finie (sa partie décimale est 1/4 + 1/16 + 1/32) : 101,01011

soit le nombre 5,34375 sa représentation en base 2 est finie (sa partie décimale est 1/4 + 1/16 + 1/32) : $(101,01011)_2$

on sait donc que 5,34375 $= (1,0101011 \times 10^{10})_2$

on a donc s = 0, m = 0101011 et e = 10+1111111 = 10000001

en simple précision 5,34375 s'écrit
01000000101010110000000000000000

soit quoi en hexadécimal ?

en utilisant `Float.intBitsToFloat()` vérifier le résultat…

Pièges de la notation en virgule flottante

des nombres ne sont pas représentables! Même des « triviaux »

0,3

il faut les approximer…

la distance entre deux flottants successifs n’est pas constante, certains calculs « triviaux » sont faux…

Est-ce que 0,3-0,2 est égal à 0,2-0,1 ?

0,3; 0,6; 1,2; 0,4; 0,8; 1,6; 1,2; 0,4…

$0,0(1001)^{\omega}$, normalisé: $1,(0011)^{\omega} \times 10^{-10}$, l'exposant flottant sera donc 01111111-10 soit 01111101

le flottant est 00111111010011001100110011001100**10** (arrondi) 0x3E99999A

0,2; 0,4; 0,8; 1,6; 1,2; 0,4…

$0,(0011)^{\omega}$, normalisé: $1,(1001)^{\omega} \times 10^{-11}$, l'exposant 01111111-11 donc 01111100

le flottant est 00111111001001100110011001100110**1** (arrondi) 0x3E4CCCCD

0,1; 0,2; 0,4; 0,8; 1,6; 1,2; 0;4…

$0,0(0011)^{\omega}$, normalisé $1,(1001)^{\omega} \times 10^{-100}$, l'exposant 01111111-100 donc 01111011

le flottant est 00111110111001100110011001100110**1** (arrondi) 0x3DCCCCCD

```
  0011111010011001100110011001110010
- 0011111000100110011001100110011101

    0,0100110011001100110011001101000
  - 0,0011001100110011001100110011101

    0,0001100110011001100110011001111

  0011110111001100110011001100111110
```

0,3-0,2

0x3DCCCCCE

```
  00111111001001100110011001100110011001101
- 00111110111001100110011001100110011001101

    0,00110011001100110011001100110011010
  - 0,00011001100110011001100110011001101

    0,00011001100110011001100110011001101
```

```
  00111110111001100110011001100110011001101
```

0,2-0,1

0x3DCCCCCD

Rappel 0,3-0,2 s’écrit 0x3DCCCCCE

```
public class Essai{
  public static void main(String argv[]) {
    int n = 0x12345678;
    float f = n;
    int m = (int)f;
    System.out.println(n);
    System.out.println(f);
    System.out.println(m);
  }
}
```

Pourquoi ?

```
% java Essai
305419896
3.05419904E8
305419904
%
```

Il existe d'autres valeurs flottantes

le zéro (tous les bits sont égaux à 0, sauf le signe, -0 et +0)

l'infini (bits de l'exposant à 1, mantisse à 0 et signe de l'infini)

NaN (erreur type 1/0), bits de l'exposant à 1, mantisse ≠0

dénormalisés (pour les toutes petites valeurs autour de 0), bits de l'exposant à 0, la mantisse ≠0, la valeur est $0,m \times 2^{-126}$

Tout ceci pour affirmer :

faire très attention avec les calculs flottants

ils sont la source de très nombreuses erreurs!

au mieux faire ses calculs à ε près

déterminer ε est un problème (difficile) en soi (c'est le domaine de l'**analyse numérique**)

Analyse numérique : méthodes de résolution numérique de problèmes d'analyse. Calcul de propagation d'erreurs.

```
float r = 0;
while (r!=20) {
   System.out.println(r);
   r = r+0.5f;
}
```

Tester…

Remplacer 0.5 par 1, tester

Remplacer 0.5 par 1/7, tester

Remplacer 0.5 par 1/5 tester

Pourquoi ?