4. Number Representations

Domain of Types `int`, `float` and `double` Mixed Expressions and Conversion; Holes in the Domain;Floating Point Number Systems; IEEE Standard; Limits of Floating Point Arithmetics; Floating Point Guidelines

Binary Number Representations

Binary representation ("Bits" from $\{0,1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

corresponds to the number $b_n \cdot 2^n + \cdots + b_1 \cdot 2 + b_0$

Example: 101011 corresponds to 43.

Least Significant Bit (LSB)

Most Significant Bit (MSB)

Binary Numbers: Numbers of the Computer?

Truth: Computers calculate using binary numbers.


NEUE ZÜRCHER ZEITUNG

TECHNIK

Das programmgesteuerte Rechengerät an der Eidgenössischen Technischen Hochschule in Zürich

Merkmale des Gerätes

Das duale Zahlsystem

Binary Numbers: Numbers of the Computer?

Stereotype: computers are talking 0/1 gibberish

Computing Tricks

- Estimate the orders of magnitude of powers of two.[3]:

$2^{10} = 1024 = 1\text{Ki} \approx 10^3$.
$2^{20} = 1\text{Mi} \approx 10^6$,
$2^{30} = 1\text{Gi} \approx 10^9$,
$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi} \approx 4 \cdot 10^9$.
$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}$.

[3] Decimal vs. binary units: MB - Megabyte vs. MiB - Megabibyte (etc.)
kilo (K, Ki) – mega (M, Mi) – giga (G, Gi) – tera(T, Ti) – peta(P, Pi) – exa (E, Ei)

Domain of Type `int`

```
public class Main {
    public static void main(String[] args) {
        Out.print("Minimum int value is ");
        Out.println(Integer.MIN_VALUE);
        Out.print("Maximum int value is ");
        Out.println(Integer.MAX_VALUE);
    }
}
```

```
Minimum int value is -2147483648.
Maximum int value is 2147483647.
```

Where do these numbers come from?

Domain of the Type `int`

Representation with 32 bits. Domain comprises the 2^{32} integers:

$$\{-2^{31}, -2^{31}+1, \ldots, -1, 0, 1, \ldots, 2^{31}-2, 2^{31}-1\}$$

Where does this partitioning come from?

Computing with Binary Numbers (4 digits)

Simple Addition

2	0010
$+3$	$+0011$
5	0101

Simple Subtraction

5	0101
-3	-0011
2	0010

Computing with Binary Numbers (4 digits)

Addition with Overflow

7	0111
$+9$	$+1001$
16	$(1)0000$

Negative Numbers?

5	0101
$+(-5)$	$????$
0	$(1)0000$

Computing with Binary Numbers (4 digits)

Simpler -1

1	0001
$+(-1)$	1111
0	$(1)0000$

Utilize this:

3	0011
$+?$	$+????$
-1	1111

Computing with Binary Numbers (4 digits)

Invert!

3	0011
$+(-4)$	$+1100$
-1	$1111 \widehat{=} 2^B - 1$

a	a
$+(-a-1)$	$+\bar{a}$
-1	$1111 \widehat{=} 2^B - 1$

Computing with Binary Numbers (4 digits)

- Negation: inversion and addition of 1

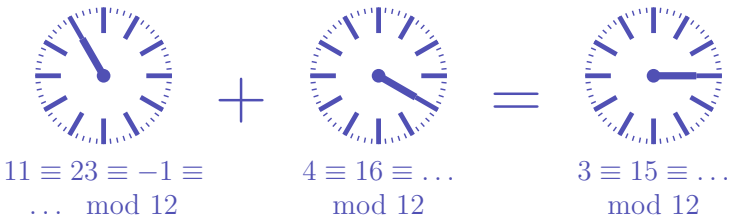
$$-a \quad \widehat{=} \quad \bar{a} + 1$$

- Wrap around semantics (calculating modulo 2^B

$$-a \quad \widehat{=} \quad 2^B - a$$

Why this works

Modulo arithmetics: Compute on a circle[4]



[4] The arithmetics also work with decimal numbers (and for multiplication).

Negative Numbers (3 Digits)

	a	$-a$	
0	000	000	0
1	001	**111**	-1
2	010	**110**	-2
3	011	**101**	-3
4	**100**	**100**	-4
5	**101**		
6	**110**		
7	**111**		

The most significant bit decides about the sign.

Two's Complement

- Negation by bitwise negation and addition of 1

$$-2 = -[0010] = [1101] + [0001] = [1110]$$

  - Arithmetics of addition and subtraction *identical* to unsigned arithmetics

$$3 - 2 = 3 + (-2) = [0011] + [1110] = [0001]$$

  - Intuitive "wrap-around" conversion of negative numbers.

$$-n \to 2^B - n$$

  - Domain: $-2^{B-1} \ldots 2^{B-1} - 1$

Over- and Underflow

- Arithmetic operations (`+`,`-`,`*`) can lead to numbers outside the valid domain.
- Results can be incorrect!

`power8`: $15^8 = -1732076671$

`power20`: $3^{20} = -808182895$

- There is *no error message!*

“Proper Calculation”

```
public class Main {

    public static void main(String[] args) {
        Out.print("Celsius: ");
        int celsius = In.readInt();
        int fahrenheit = 9 * celsius / 5 + 32;
        Out.print(celsius + " degrees Celsius are ");
        Out.println(fahrenheit + " degrees Fahrenheit");
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        Out.print("Celsius: ");
        float celsius = In.readInt();
```


Types `float` and `double`

- are the fundamental types for floating point numbers
- approximate the field of real numbers $(\mathbb{R}, +, \times)$ from mathematics
- have a great domain, sufficient for many applications (`double` provides more places than `float`)
- are fast on many computers

Fixpoint numbers

- fixed number of integer places (e.g. 7)
- fixed number of decimal places (e.g. 3)

`0.0824 = 0000000.082` ← third place truncated

Nachteile

- Domain is getting *even* smaller than for integers.
- If a number can be represented depends on the position of the comma.

Floating point Numbers

- fixed number of significant places (e.g. 10)
- plus position of the comma

$$\texttt{82.4} = 824 \cdot 10^{-1}$$

$$\texttt{0.0824} = 824 \cdot 10^{-4}$$

- Zahl ist $\mathit{Mantissa} \times 10^{\mathit{Exponent}}$

Domain

Integer Types:

- Over- and Underflow relatively frequent, but ...
- the domain is contiguous (no “holes”): $\mathbb{Z}$ is “discrete”.

Floating point types:

- Overflow and Underflow seldom, but ...
- there are holes: $\mathbb{R}$ is “continuous”.

Holes in the Domain

```
public class Main {
    public static void main(String[] args) {
    Out.print("First number =? ");
    float n1 = In.readFloat();

    Out.print("Second number =? ");
    float n2 = In.readFloat();

    Out.print("Their difference =? ");
    float d = In.readFloat();

    Out.print("computed difference − input difference = ");
    Out.println(n1−n2−d);
    }
}
```

input 1.1

input 0.1

output 2.2351742E-8

What is going on here?

Floating Point Number Systems

A Floating Point Number System is defined by the four natural numbers:

- $\beta \geq 2$, the Basis,
- $p \geq 1$, the precision (number of places),
- $e_{\min}$, the smallest possible exponent,
- $e_{\max}$, the largest possible exponent.

Notation:

$$F(\beta, p, e_{\min}, e_{\max})$$

Floating Point Number Systems

$F(\beta, p, e_{\min}, e_{\max})$ comprises the numbers

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$

represented with Basis β:

$$\pm\, d_0 \bullet d_1 \ldots d_{p-1} \times \beta^e,$$

Floating Point Number Systems

Example

- $\beta = 10$

Representations of the decimal number 0.1

$$1.0 \cdot 10^{-1}, \quad 0.1 \cdot 10^{0}, \quad 0.01 \cdot 10^{1}, \quad \ldots$$

Normalized Representation

Normalized Number:

$$\pm\, d_0 \bullet d_1 \ldots d_{p-1} \times \beta^e, \qquad d_0 \neq 0$$

Bemerkung 1

The normalized representation is unique and therefore prefered.

Remark 2

The number 0 (and all numbers smaller than $\beta^{e_{\min}}$) have no normalized representation (we will deal with this later)!

Set of Normalized Numbers

$$F^*(\beta, p, e_{\min}, e_{\max})$$

Normalized Representation

Example $F^*(2, 3, -2, 2)$ (only positive numbers)

$d_0 \bullet d_1 d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Binary and Decimal Systems

- Internally the computer computes with $\beta = 2$ (binary system)
- Literals and inputs have $\beta = 10$ (decimal system)
- Inputs have to be converted!

Conversion Decimal $\rightarrow$ Binary

Angenommen, $0 < x < 2$.

- Hence: $x' = b_{-1}\bullet b_{-2}b_{-3}b_{-4}\ldots = 2 \cdot (x - b_0)$
- Step 1 (for x): Compute b_0:

$$b_0 = \begin{cases} 1, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

- Step 2 (for x): Compute $b_{-1}, b_{-2}, \ldots$:

  Go to step 1 (for $x' = 2 \cdot (x - b_0)$)

Binary representation of 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2
1.2	$b_{-5} = \mathbf{1}$	0.2	0.4

$\Rightarrow 1.0\overline{0011}$, periodic, *not* finite

Binary Number Representations of 1.1 and 0.1

- are not finite, there are errors when converting into a (finite) binary floating point system.
- `1.1f` and `0.1f` do not equal 1.1 and 0.1, but slightly inaccurate approximation of these numbers.

$$\begin{aligned} \texttt{1.1} &= \underline{1.1000000000000000}888178\ldots \\ \texttt{1.1f} &= \underline{1.1000000}238418\ldots \end{aligned}$$

Computing with Floating Point Numbers

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rl} & 1.111 \cdot 2^{-2} \\ + & 1.011 \cdot 2^{-1} \\ \hline & = 1.001 \cdot 2^{0} \end{array}$$

1. adjust exponents by denormalizing of one number 2. binary addition of the mantissa 3. renormalize 4. round to p significant places, if necessary

The IEEE Standard 754

- defines floating point number systems and their rounding behavior
- is used nearly everywhere
- Single precision (`float`) numbers:

  $F^*(2, 24, -126, 127)$ plus $0, \infty, \ldots$
- Double precision (`double`) numbers:

  $F^*(2, 53, -1022, 1023)$ plus $0, \infty, \ldots$
- All arithmetic operations round the *exact* result to the next representable number

32-bit Representation of a Floating Point Number

$\pm$ Exponent Mantisse

$\pm$ $2^{-126}, \ldots, 2^{127}$ $1.00000000000000000000000$ $\ldots$ $1.11111111111111111111111$

$0, \infty, \ldots$

The IEEE Standard 754

Why

$$F^*(2, 24, -126, 127)?$$

- 1 sign bit
- 23 bit for the mantissa (leading bit is 1 and is not stored)
- 8 bit for the exponent (256 possible values)(254 possible exponents, 2 special values: 0, ∞,...)

$\Rightarrow$ 32 bit overal.

The IEEE Standard 754

Why

$$F^*(2, 53, -1022, 1023)?$$

- 1 sign bit
- 52 bit for the mantissa (leading bit is 1 and is not stored)
- 11 bit for the exponent (2046 possible exponents, 2 special values: 0, ∞,...)

$\Rightarrow$ 64 bit overal.

Floating Point Rules — Rule 1

Rule 1

Do not test rounded floating point numbers for equality.

```
for (float i = 0.1; i != 1.0; i += 0.1){
    Out.println(i);
}
```

endless loop because i never becomes exactly 1

Floating Point Rules — Rule 2

Rule 2

Do not add two numbers providing very different orders of magnitude!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & +1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \\ & \text{“=”}\ 1.000 \cdot 2^5 \quad \text{(Rounding on 4 places)} \end{aligned}$$

Addition of 1 does not provide any effect!

Floating Point Guidelines — Rule 3

Rule 4

Do not subtract two numbers with a very similar value.

Cancellation problems (without further explanations).

5. Logical Values

Boolean Functions; the Type `boolean`; logical and relational operators; shortcut evaluation

Our Goal

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

Behavior depends on the value of a Boolean expression

Boolean Values in Mathematics

Boolean expressions can take on one of two values:

$$F \text{ or } T$$

- F corresponds to *"wrong"*
- T corresponds to *"true"*

The Type `boolean` in Java

- represents *logical values*
- Literals `false` and `true`
- Domain $\{$*false*, *true*$\}$

```
boolean b = true; // Variable with value true
```

Relational Operators

`a < b` (smaller than)
`a >= b` (greater than)
`a == b` (equals)
`a != b` (unequal)

number type $\times$ number type $\rightarrow$ `boolean`

Boolean Functions in Mathematics

- Boolean function

$$f : \{F, T\}^2 \rightarrow \{F, T\}$$

- F corresponds to “false”.
- T corresponds to “true”.

$\mathrm{AND}(x, y)$ $x \wedge y$

- “logical and”

$$f : \{F, T\}^2 \rightarrow \{F, T\}$$

- F corresponds to “false”.
- T corresponds to “true”.

x	y	$\mathrm{AND}(x, y)$
F	F	F
F	T	F
T	F	F
T	T	T

Logischer Operator &&

`a && b` (logical and)

`boolean` $\times$ `boolean` $\rightarrow$ `boolean`

```
int n = -1;
int p = 3;
boolean b = (n < 0) && (0 < p); // b = true
```

$\mathrm{OR}(x, y)$ $x \vee y$

- "logical or"

$$f : \{F, T\}^2 \to \{F, T\}$$

- F corresponds to "false".
- T corresponds to "true".

x	y	$\mathrm{OR}(x, y)$
F	F	F
F	T	T
T	F	T
T	T	T

Logical Operator ||

`a || b` (logical or)

`boolean` $\times$ `boolean` $\to$ `boolean`

```
int n = 1;
int p = 0;
boolean b = (n < 0) || (0 < p); // b = false
```


$\mathrm{NOT}(x)$ $\neg x$

- "logical not"

$$f : \{F, T\} \to \{F, T\}$$

- F corresponds to "false".
- T corresponds to "true".

x	$\mathrm{NOT}(x)$
F	T
T	F

Logical Operator !

`!b` (logical not)

`boolean` $\to$ `boolean`

```
int n = 1;
boolean b = !(n < 0); // b = true
```

Precedences

```
!b && a
   ⇕
(!b) && a

a && b || c && d
   ⇕
(a && b) || (c && d)

a || b   &&   c || d
   ⇕
a || (b && c) || d
```


Precedences

The unary logical operator `!`
provides a stronger binding than
binary arithmetic operators. These
bind stronger than
relational operators,
and these bind stronger than
binary logical operators.

```
7 + x < y && y != 3 * z || ! b
7 + x < y && y != 3 * z || (!b)
```


DeMorgan Rules

- `!(a && b) == (!a || !b)`
- `!(a || b) == (!a && !b)`

`!` (rich *and* beautiful) == (poor *or* ugly)

Application: either ... or (XOR)

`(x || y)`	`&& !(x && y)`	x or y, and not both
`(x || y)`	`&& (!x || !y)`	x or y, and one of them not
`!(!x && !y)`	`&& !(x && y)`	not none and not both
`!(!x && !y || x && y)`		not: both or none

Shortcut Evaluation

- Logical operators `&&` and `||` evaluate the *left operand first*.
- If the result is then known, the right operand will *not be* evaluated.

```
x != 0 && z / x > y
```

$\Rightarrow$ No division by 0