

IFT1015 Programmation 1

Expressions et variables numériques

Marc Feeley

(avec ajouts de Aaron Courville et Pascal Vincent)

Syntaxe

- **Syntaxe d'un langage** : forme textuelle que peuvent prendre les programmes valides
- Tout comme pour les langages naturels (français, anglais, ...), la syntaxe est normalement définie par une **grammaire**
- **Grammaire** : ensemble de règles pour former des programmes valides syntaxiquement à partir de **fragments** de programme valides

Expressions

- Tous les langages de programmation offrent la possibilité de faire des **calculs numériques**
- Ces calculs s'expriment par des **expressions**
- Exemple en JS :
 - 5
 - $2+3*5$
 - $(2+3)*5$

Expressions

- Toute expression a une **valeur**, qui est le résultat du calcul exprimé par l'expression
- En JS, 25 est la valeur de l'expression **$(2+3) * 5$**
- Dans presque tous les langages, un **nombre décimal non-négatif** est une expression simple (une *constante littérale*), dont la valeur est le nombre en question
- En JS, 123 est la valeur de l'expression **123**

Expressions

- Des expressions plus complexes sont bâties à l'aide d'**opérateurs** et d'expressions plus simples (les **opérandes**)
- Les opérateurs de base en JS :
 - **+** addition
 - **-** soustraction
 - ***** multiplication
 - **/** division

Opérateurs binaires

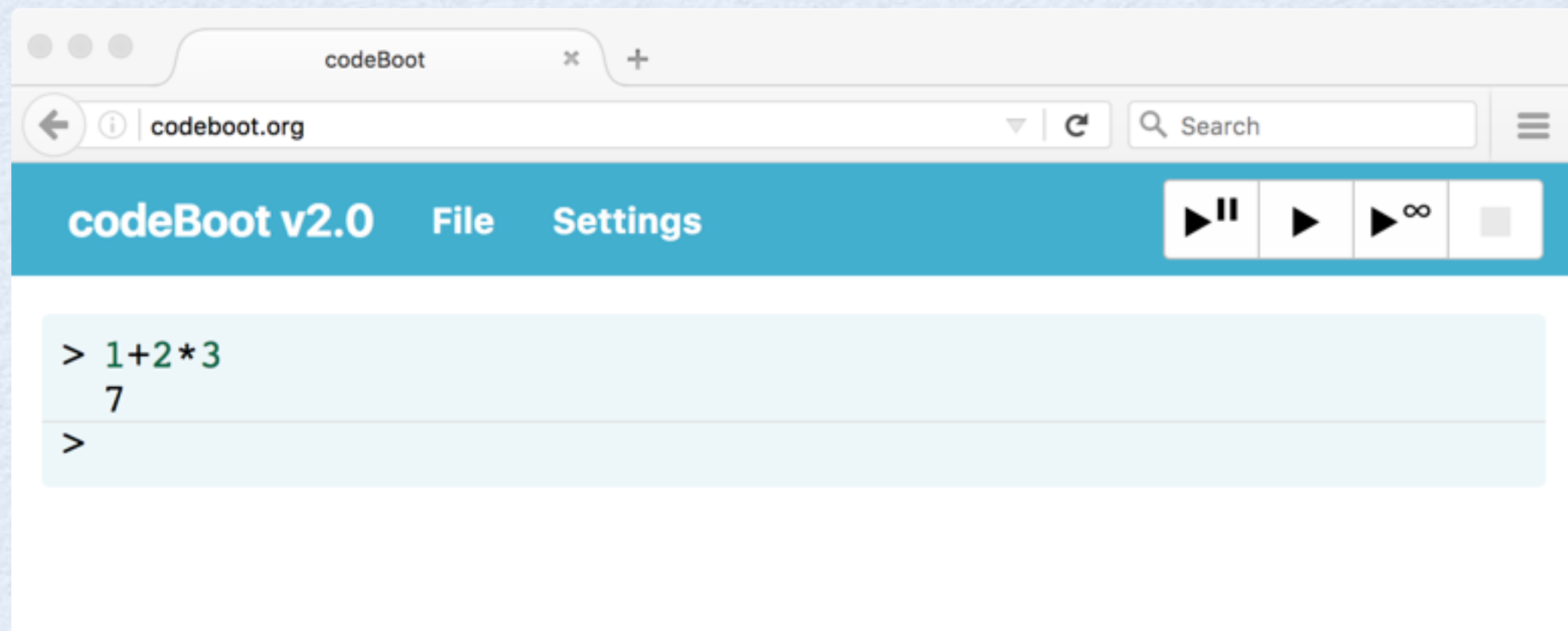
- “Binaire” pour 2 opérandes
- Syntaxe : $\langle expression \rangle \langle op \rangle \langle expression \rangle$
- Exemple :
 $3 + 7$
valeur : 10
- Exemple :
 $2 * 5 - 1$
valeur : 9
- Exemple :
 $2 * 5 - 3 * 3$
valeur : 1

Opérateurs unaires




- Les opérateurs de signe (+ et -) peuvent être utilisés comme préfixe d'une expression
- Syntaxe : $\langle \text{signe} \rangle \langle \text{expression} \rangle$
- Exemple :
 $- \quad 5$
valeur : -5
- Exemple :
 $+ \quad 13$
valeur : 13
- Exemple :
 $- \quad - \quad 5$
valeur : 5

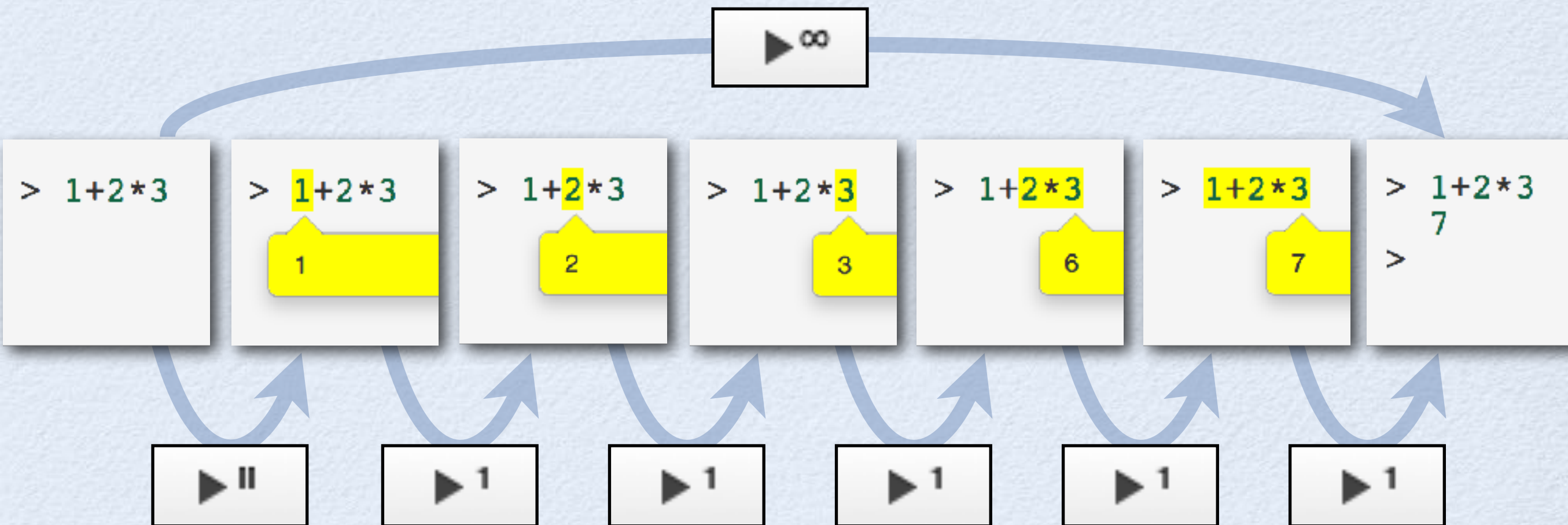
codeBoot

- Pour développer des programmes JS nous allons utiliser l'environnement **codeBoot**
- codeBoot est un interprète de JS conçu à l'UdeM qui est **interactif** et **pédagogique**



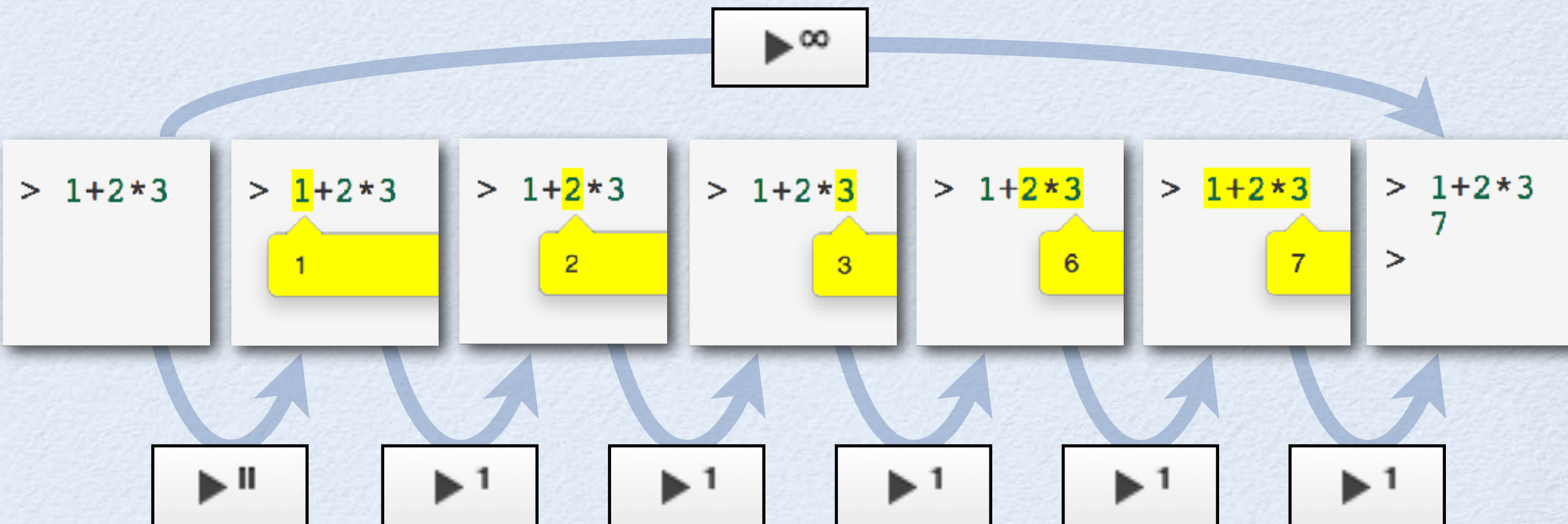
Exécution pas-à-pas

- L'exécution peut se faire d'un seul coup (avec la touche **enter** ou le bouton ) , ou bien chaque étape de l'exécution (**pas**) peut être obtenu en cliquant sur le bouton  puis le bouton 



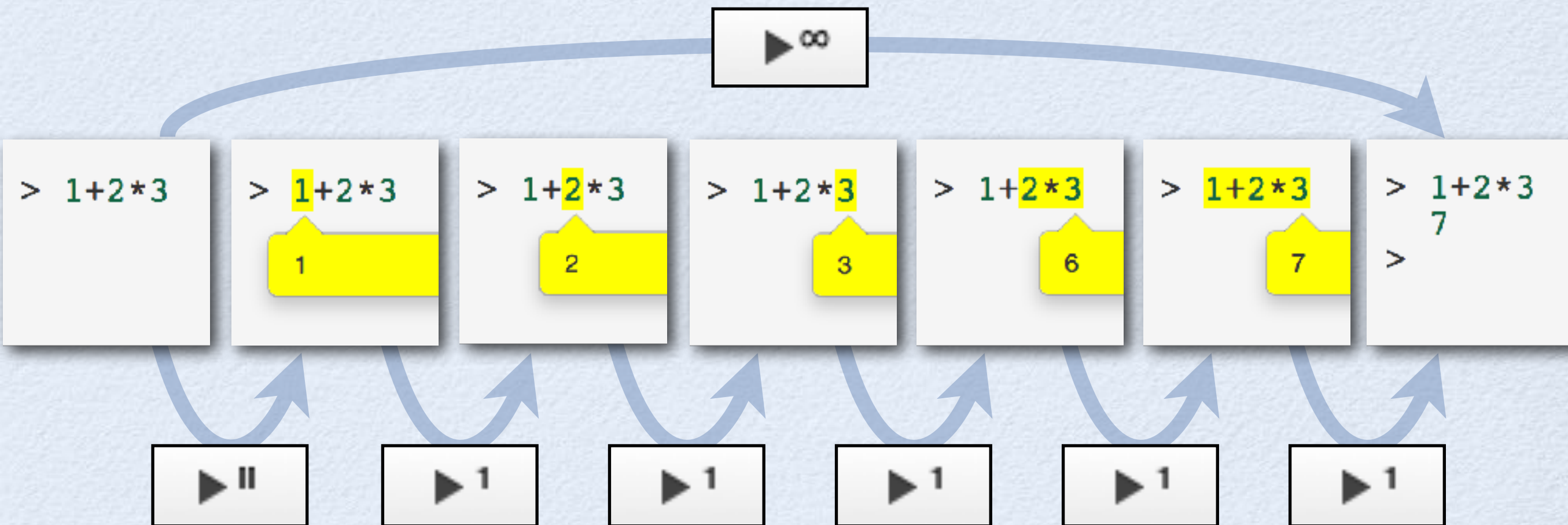
Exécution pas-à-pas

- À chaque pas codeBoot affiche en jaune l'expression qui vient juste d'être calculée, et, dans une boîte jaune en dessous ou au dessus, sa valeur



Ordre d'exécution

- Généralement, l'exécution est de **gauche à droite**
- Comment expliquer que la multiplication se fait avant l'addition?



Préséance des opérateurs

- Chaque opérateur a un **niveau de préséance**
 - $+, -$: les opér. *binaires* additifs (niveau 1)
 - $*, /$: les opér. *binaires* multiplicatifs (niveau 2)
- Pour déterminer comment les sous-expressions se regroupent, il faut regrouper les sous-expressions aux côtés des opérateurs de niveau 2 **avant** de le faire pour les opérateurs de niveau 1
- Pour forcer un groupement spécifique, on peut se servir de **parenthèses**
- $1+2*3$ est égal à $1+(2*3)$ mais pas à $(1+2)*3$

Préséance des opérateurs

- Les opérateurs *unaires* $+$ et $-$
 - -5 a la valeur -5
 - $+5$ a la valeur 5
- Ces opérateurs sont de niveau 3
 - $-+-5$ est égal à $-(+(-5))$
 - $-8*-5-3$ est égal à $((-8)*(-5))-3$
- JS a plusieurs autres opérateurs et un total de 17 niveaux de préséance!

Associativité des opérateurs

- Pour des opérateurs de **même niveau** de préséance, pour déterminer comment les sous-expressions se regroupent, il faut tenir compte de l'**associativité** des opérateurs
- Les opérateurs $+, -, *, /$ sont **associatifs à gauche**
- $1-2+3$ est égal à $(1-2)+3$ mais pas à $1-(2+3)$
- $1-2-3-4-5-6-7-8-9$ est égal à $(((((1-2)-3)-4)-5)-6)-7)-8)-9$

Parenthèses redondantes

- Certaines paires de parenthèses dans une expression peuvent être **redondantes** (c'est-à-dire qu'on obtient le même regroupement lorsqu'on les retirent)
- Exemple : $(8 * 9) / (7 - 1)$ est égal à $8 * 9 / (7 - 1)$
- Si ça aide à comprendre la logique du calcul, il est bon de garder des parenthèses redondantes
- Exemple : $(-8) + (-5)$ au lieu de $-8 + -5$

Erreurs de syntaxe

- L'interprète fait l'**analyse syntaxique** du code avant de l'exécuter
- Si le code ne correspond pas à la grammaire de JS un **message d'erreur** sera affiché

```
> 1+2*3
```

```
7
```

```
> 1+2x3
```

```
syntax error -- unexpected token
```

- L'éditeur fait le **balancement de parenthèses** automatiquement pour aider le programmeur

Balancement de parenthèses

> 1+((2+3) * (4+5

> 1+((2+3) * (4+5)

balancées

> 1+((2+3) * (4+5))

balancées

> 1+((2+3) * (4+5)))

pas
balancée

Les nombres

Notation positionnelle

- Dans cette notation un nombre est **encodé** par une séquence de symboles (chiffres)
- Si la base est k , il y a k symboles distincts pour représenter les valeurs $0, 1, 2, \dots, k-1$
- Par exemple en base $k=10$: **0,1,2,3,4,5,6,7,8,9**
- Chaque chiffre de la séquence a un *poids* qui est k fois plus grand que le *poids* du chiffre à sa droite

Notation positionnelle

- Donc le *poids* des chiffres dépend de la position dans la séquence et le *poids* progresse suivant les puissances de k
- Par ex. avec la base $k=10$, **2087** a la valeur

1000	100	10	1
2	0	8	7

$$= 2 \times 1000 + 0 \times 100 + 8 \times 10 + 7 \times 1 = 2087$$

Notation positionnelle

- Donc le *poids* des chiffres dépend de la position dans la séquence et le *poids* progresse suivant les puissances de k
- Par ex. avec la base $k=10$, **2087** a la valeur

$$\begin{array}{cccc} 10^3 & 10^2 & 10^1 & 10^0 \\ \boxed{2} & \boxed{0} & \boxed{8} & \boxed{7} \end{array} = 2 \times 10^3 + 0 \times 10^2 + 8 \times 10^1 + 7 \times 10^0 = 2087$$

Encodage des nombres

- Il y a 3 façons principales de stocker les nombres en mémoire à l'aide d'un groupe de bits :
 - Encodage **binaire non signé**
 - Encodage **complément à 2**
 - Encodage **point flottant** (norme IEEE 754)
- L'encodage se fait avec un **nombre fixe de bits** ($n=32, 64, \dots$)

Encodage binaire non signé

- Notation positionnelle avec base 2 pour le stockage des **nombre entiers ≥ 0**
- $0, 1, 2, 3, 4, 5, \dots 2^n - 1$ avec n bits
- Les bits ont un index de 0 à $n-1$ (l'index 0 est le plus à droite)
- Le *poids* du bit i est 2^i (c'est-à-dire sa contribution si ce bit est égal à 1)

Encodage binaire non signé

- Exemples avec 4 bits (i.e. $n=4$) :

2^3	2^2	2^1	2^0
0	1	0	1

$$= 2^2 + 2^0 = 4 + 1 = 5$$

2^3	2^2	2^1	2^0
1	0	0	0

$$= 2^3 = 8$$

2^3	2^2	2^1	2^0
1	1	1	1

$$= 2^3 + 2^2 + 2^1 + 2^0 = 15$$

Encodage binaire non signé

- Plus petit nombre avec 4 bits :

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} \end{array} = 0$$

- Plus grand nombre avec 4 bits :

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} \end{array} = 2^3 + 2^2 + 2^1 + 2^0 = 15 = 2^4 - 1$$

Encodage complément à 2

- Pour le stockage des **nombre**s entiers avec signe positif ou négatif
- $-2^{n-1} \dots, -3, -2, -1, 0, 1, 2, 3, \dots 2^{n-1}-1$ avec n bits
- Les bits ont un index de 0 à $n-1$ (l'index 0 est le plus à droite)
- Le poids du bit i est 2^i , sauf le bit $n-1$ qui a un poids de -2^{n-1}

Encodage complément à 2

- Exemples avec 4 bits (i.e. $n=4$) :

-2^3	2^2	2^1	2^0
0	1	0	1

$$= 2^2 + 2^0 = 4 + 1 = 5$$

-2^3	2^2	2^1	2^0
1	0	0	0

$$= -2^3 = -8$$

-2^3	2^2	2^1	2^0
1	1	1	1

$$= -2^3 + 2^2 + 2^1 + 2^0 = -1$$

Encodage complément à 2

- Exemples avec 4 bits (i.e. $n=4$) :

-2^3	2^2	2^1	2^0
0	1	0	1

$$= 2^2 + 2^0 = 4 + 1 = 5$$

-2^3	2^2	2^1	2^0
1	0	0	0

$$= -2^3 = -8$$

-2^3	2^2	2^1	2^0
1	1	1	1

$$= -2^3 + 2^2 + 2^1 + 2^0 = -1$$

Encodage complément à 2

- Plus petit nombre avec 4 bits :

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} \end{array} = -2^3 = -8$$

- Plus grand nombre avec 4 bits :

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} \end{array} = 2^2 + 2^1 + 2^0 = 7 = 2^3 - 1$$

Les nombres JS

- La syntaxe des nombres permet de préciser des **décimales** et une **puissance de 10**
- Voici quelques exemples :
 - **1.25** $= 1.25$
 - **42e3** $= 42 \times 10^3 = 42000$
 - **.2e-1** $= 0.2 \times 10^{-1} = 0.2 \div 10^1 = 0.02$
 - **1.030E+10** $= 1.03 \times 10^{10} = 10300000000$

Les nombres JS

- La syntaxe des nombres permet de préciser des **décimales** et une **puissance de 10**
- Voici quelques exemples :

notation scientifique

- **42e3**

$$= 42 \times 10^3 = 42000$$

- **.2e-1**

$$= 0.2 \times 10^{-1} = 0.2 \div 10^1 = 0.02$$

- **1.030E+10**

$$= 1.03 \times 10^{10} = 10300000000$$

Les nombres JS

- L'affichage d'un nombre qui n'a pas de partie fractionnaire ne contient **pas de décimales**
- Pour les nombres $\geq 10^{21}$, l'affichage se fait avec la **notation scientifique**

- Example :

$$> \frac{3.25 - 1.25}{2}$$

> 10000000000000000
10000000000000000

```
> 1000000000000000000000  
1e+21
```


Les nombres JS

- On peut également exprimer des nombres entier, sans partie fractionnaire, en base 16 (**hexadécimal**), à l'aide du préfixe **0x**
- Cela revient à la notation **binaire** où les groupes de 4 bits sont encodés par **0...9, A(=10)...F(=15)**
- Par exemple :

$$\underline{\mathbf{0xA7}} = \overbrace{1010}^{\mathbf{A}} \overbrace{0111}^{\mathbf{7}}_2 = 167$$

Encodage point flottant

- Pour le stockage des **nombre**s réels
- 3 groupes de bits (signe, exposant, fraction)
- Avec $n=64$ (précision **double**) :

s	e	f
1 bit	11 bits	52 bits

$$\text{valeur} = (-1)^s \times (1 + f \times 2^{-52}) \times 2^{e-1023}, \quad 1 \leq e \leq 2046$$

$$\text{valeur} = (-1)^s \times f \times 2^{-1074}, \quad e = 0$$

Encodage point flottant

- Nombre positif le plus grand $\cong 1.8 \times 10^{308}$

[illegible]

$$s=0 \quad e=2046 \quad f=11111111 \dots 11111111_2$$

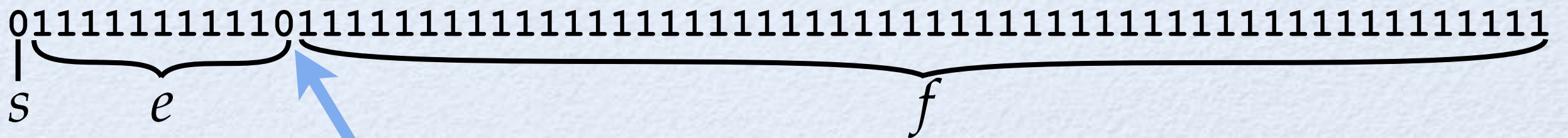
- Nombre positif le plus petit $\cong 5 \times 10^{-324}$

[illegible]

$$s=0 \quad e=0 \quad f=000000000\dots000000001_2$$

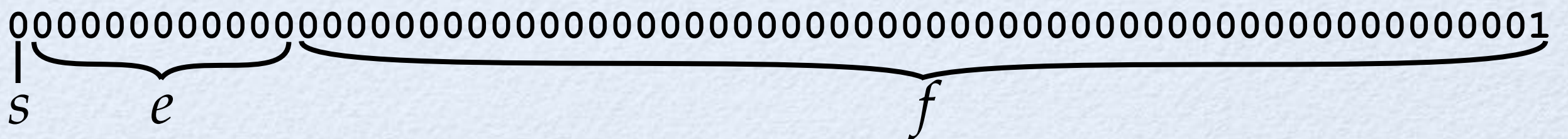
Encodage point flottant

- Nombre positif le plus grand $\cong 1.8 \times 10^{308}$


$$S=0$$

Pourquoi 0? La norme IEEE 754 réserve
e = 111111111111 pour représenter

- Nombre pos $\pm\infty$ (lorsque $f = 0$) et NaN (lorsque $f \neq 0$)


$$s=0 \quad e=0 \quad f=000000000\dots000000001_2$$

Erreurs d'arrondi

- Valeur exacte de l'encodage de $0.1 =$
0.1000000000000000000055511151231257827021181583404541015625
- Valeur exacte de l'encodage de $0.2 =$
0.200000000000000000011102230246251565404236316680908203125
- Valeur exacte de l'encodage de $0.3 =$
0.299999999999999999988897769753748434595763683319091796875
- Valeur exacte de l'encodage de $0.1+0.2 =$
0.3000000000000000000444089209850062616169452667236328125
- **Attention :**

> 0.1 + 0.2
0.30000000000000000004

Des valeurs spéciales

- Zéro négatif
- \pm Infini
- Not-a-Number (NaN)

```
> 1/0  
Infinity
```

```
> 1/(-0)  
-Infinity
```

```
> 0/0  
NaN
```


Précision limitée

- JS stocke tous les nombres, **incluant les entiers**, avec l'encodage point flottant
- Donc, **100** et **100.0** et **1e2** et **0.1e+3** représentent **le même nombre**
- Vu le nombre limité de bits pour f , les grands entiers ne sont pas représentés exactement (c'est le nombre encodable le plus proche) :

> 999

999

> 9999999999999999

1000000000000000000

Utilisation des nombres

- Il est conseillé d'utiliser les nombres non-entiers pour les **calculs scientifiques** seulement car une petite erreur de calcul est tolérable
- Pour les **calculs monétaires**, il est mieux de s'en tenir aux entiers qui sont représentés exactement (par exemple, calculer en nombre de cents plutôt qu'en nombre de dollars)

Abstraction

Un texte lourd

- Examinons le texte suivant :
 - «Le fils de Rose-Anne Monna et de Legrand Feeley qui réside à Montréal a acheté un télescope au troisième mois de 2012. Le fils de Rose-Anne Monna et de Legrand Feeley qui réside à Montréal a observé la quatrième planète en orbite autour du Soleil.»
- Ce texte est plutôt lourd... Que peut-on faire pour l'alléger?

Abstraire en nommant

- Utilisons des **noms propres** pour abstraire :
 - «**Marc** a acheté un télescope en **Mars** 2012.
Marc a observé **Mars**.»
- Le texte est beaucoup plus court, agréable à lire et compréhensible
- Les noms prennent le sens de leur définition (p.ex. Marc = «Le fils de Rose-Anne Monna et de Legrand Feeley qui réside à Montréal»)

Abstraire en nommant

- En programmation, les noms sont des **identificateurs**, et on en donne la définition dans une **déclaration**
- Lorsqu'on **réfère** à un identificateur, c'est une déclaration spécifique à laquelle on fait référence
- Les ambiguïtés possibles, comme pour **Mars**, sont réglées par le **contexte de la référence** c'est à dire où et comment la référence est faite (p.ex. grâce aux **règles de portée**)

Syntaxe des identificateurs

- En JS, les identificateurs sont des symboles composés de lettres (majuscules / minuscules), des chiffres (0–9), et les caractères \$ et _
- Les chiffres sont interdits au début d'un ident.
- La casse (majuscule / minuscule) est significative
- Exemples : **n** **x2** **prix** **Point** **été**
 temp_max **tempMax** **\$\$** **Δ**
- Incorrect : **3amis** **temp-max**

Syntaxe des identificateurs

- Certains identificateurs ne peuvent être déclarés par le programmeur car ils sont **réservés** par la grammaire :

<code>break</code>	<code>do</code>	<code>if</code>	<code>package</code>	<code>throw</code>
<code>case</code>	<code>else</code>	<code>implements</code>	<code>private</code>	<code>true</code>
<code>catch</code>	<code>enum</code>	<code>import</code>	<code>protected</code>	<code>try</code>
<code>class</code>	<code>export</code>	<code>in</code>	<code>public</code>	<code>typeof</code>
<code>const</code>	<code>extends</code>	<code>instanceof</code>	<code>return</code>	<code>var</code>
<code>continue</code>	<code>false</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>debugger</code>	<code>finally</code>	<code>let</code>	<code>super</code>	<code>while</code>
<code>default</code>	<code>for</code>	<code>new</code>	<code>switch</code>	<code>with</code>
<code>delete</code>	<code>function</code>	<code>null</code>	<code>this</code>	<code>yield</code>

- Pour des raisons de portabilité, il est mieux de s'en tenir aux caractères ASCII (pas d'accents, pas de lettres Grècques, symboles Unicode, ...)

Syntaxe des identificateurs

- Pour les identificateurs composés de plusieurs mots, nous favoriserons la notation **CamelCase** qui est populaire
- Les mots sont collés les uns aux autres, avec la première lettre de chaque mot en majuscule, à l'exception du premier mot qui est seulement en majuscule si c'est l'identificateur d'un constructeur (nous verrons plus tard):
 - **kilogrammesParLivre**
 - **DateGregorienne**

Syntaxe des identificateurs

- Un bon identificateur clarifie ce à quoi il réfère (il évite les ambiguïtés)
- Si la déclaration peut être référée de partout dans un gros programme, il est mieux d'utiliser un identificateur le plus descriptif possible :
 - **temperatureCongelationHydrogene**
 - **tempCongHydrogene**
- Si la portée est locale, il est mieux d'utiliser un identificateur court pour alléger le code :
 - **tch**
 - **t**

Déclaration de variable

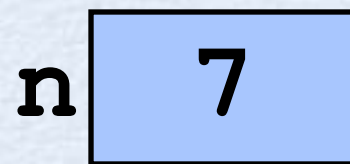
- En JS, on peut nommer une valeur à l'aide d'une **déclaration de variable**
- Syntaxe : **var** *⟨identificateur⟩* = *⟨expression⟩*
- *⟨identificateur⟩* est le nom de la variable **créée**
- La valeur de *⟨expression⟩* est **liée** à la variable
- Une référence subséquente à *⟨identificateur⟩* s'évaluera à la valeur liée à la variable

Déclaration de variable

- Exemple :

```
> var n = 1+2*3  
> n*n  
49
```

- Visualisation de la variable **n** :



La variable
correspond à une
cellule mémoire

Déclaration de variable

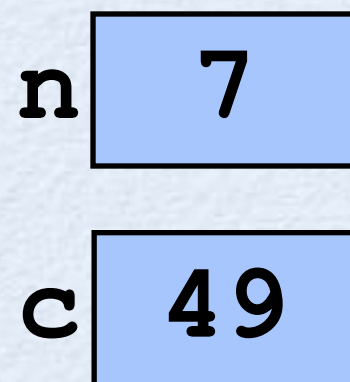
- Une déclaration n'est **pas une expression**
- Une déclaration n'a donc **pas de valeur**
- On dit plutôt que la déclaration de variable a un **effet** (celui de créer une variable et la lier)
- C'est pour son effet que la déclaration est exécutée
- La boucle d'interaction (Read-Eval-Print-Loop = **REPL**) de la console n'affiche pas de valeur

Déclaration de variable

- Exemple :

```
> var n = 1+2*3
> var c = n*n
> c
49
```

- Visualisation des variables :



Chaque variable correspond à une cellule mémoire

Déclaration de variable

- Exemple :

```
> var n = 1+2*3, c = n*n
> c
49
```
- Syntaxe : **var** $\langle id. \rangle_1 = \langle exp. \rangle_1, \langle id. \rangle_2 = \langle exp. \rangle_2, \dots$
- Cette syntaxe est utile lorsqu'il y a plusieurs variables à déclarer
- Une expression peut référer aux variables précédentes

Déclaration de var. : exemple

- Problème : calcul de la circonférence et de la surface d'un cercle de rayon 5
- Rappel : circonférence = $2\pi r$ surface = πr^2

```
> 2*3.141592653589793*5  
31.41592653589793  
> 3.141592653598793*5*5  
78.53981633996982
```


Déclaration de var. : exemple

- Solution avec variables :

```
> var r = 5
> var pi = 3.141592653589793
> 2*pi*r
31.41592653589793
> pi*r*r
78.53981633974483
```

r	5
pi	3.141592653589793

Déclaration de var. : exemple

- Cette solution est :

sans variables:

```
> 2*3.141592653589793*5
31.41592653589793
> 3.141592653598793*5*5
78.53981633996982
```

avec variables:

```
> var r = 5
> var pi = 3.141592653589793
> 2*pi*r
31.41592653589793
> pi*r*r
78.53981633974483
```

- Plus lisible
- Plus facile à comprendre
- Plus facile à maintenir (changer le rayon ou la précision de π)
- Correcte (la première solution a un bogue à cause de la duplication de la constante π)
- Principe : éviter la duplication de code

Calculs numériques

Calculs numériques

- Outre $+$, $-$, $*$ et $/$, JS offre plusieurs opérations sur les nombres
- Les opérateurs **bit-à-bit** (\sim , $\&$, $|$, \wedge , \ll , \gg , \ggg)
- L'opérateur **modulo** ($\%$)
- Les **fonctions mathématiques** (racine carrée, sinus, cosinus, puissance, ...)

Opérateurs bit-à-bit

- JS possède des opérateurs binaires et unaires pour faire des calculs sur l'encodage complément à 2 des entiers 32 bits
- Ces opérateurs ont une correspondance directe avec les **opérations bit-à-bit** (bitwise) de la machine :
 - \sim (complément), $\&$ (et), $|$ (ou), \wedge (ou-exclusif)
 - \ll et \gg (décalages à gauche et à droite)
 - \ggg (décalage à droite, encodage non signé)

Opérateurs bit-à-bit

- Pour l'opérateur unaire \sim (complément):
 - il y aura un 1 dans l'encodage de la valeur résultante si et seulement si il y a un 0 à la position correspondante de l'encodage de l'opérande

0	1	1	0
---	---	---	---

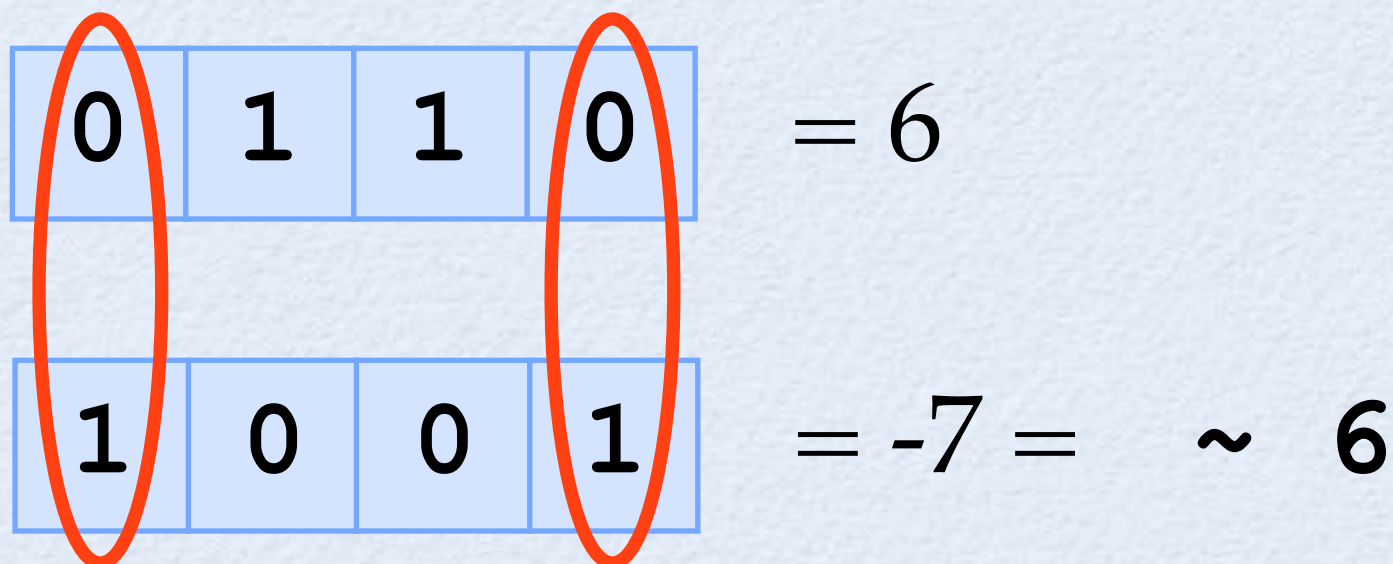
 = 6

1	0	0	1
---	---	---	---

 = -7 = \sim 6

Opérateurs bit-à-bit

- Pour l'opérateur unaire \sim (complément):
 - il y aura un 1 dans l'encodage de la valeur résultante si et seulement si il y a un 0 à la position correspondante de l'encodage de l'opérande



Opérateurs bit-à-bit

- Pour l'opérateur binaire **&** il y aura un 1 dans l'encodage de la valeur résultante si et seulement si il y a un 1 à la position correspondante des encodages des 2 opérandes

0	1	0	1
---	---	---	---

 = 5

0	1	1	0
---	---	---	---

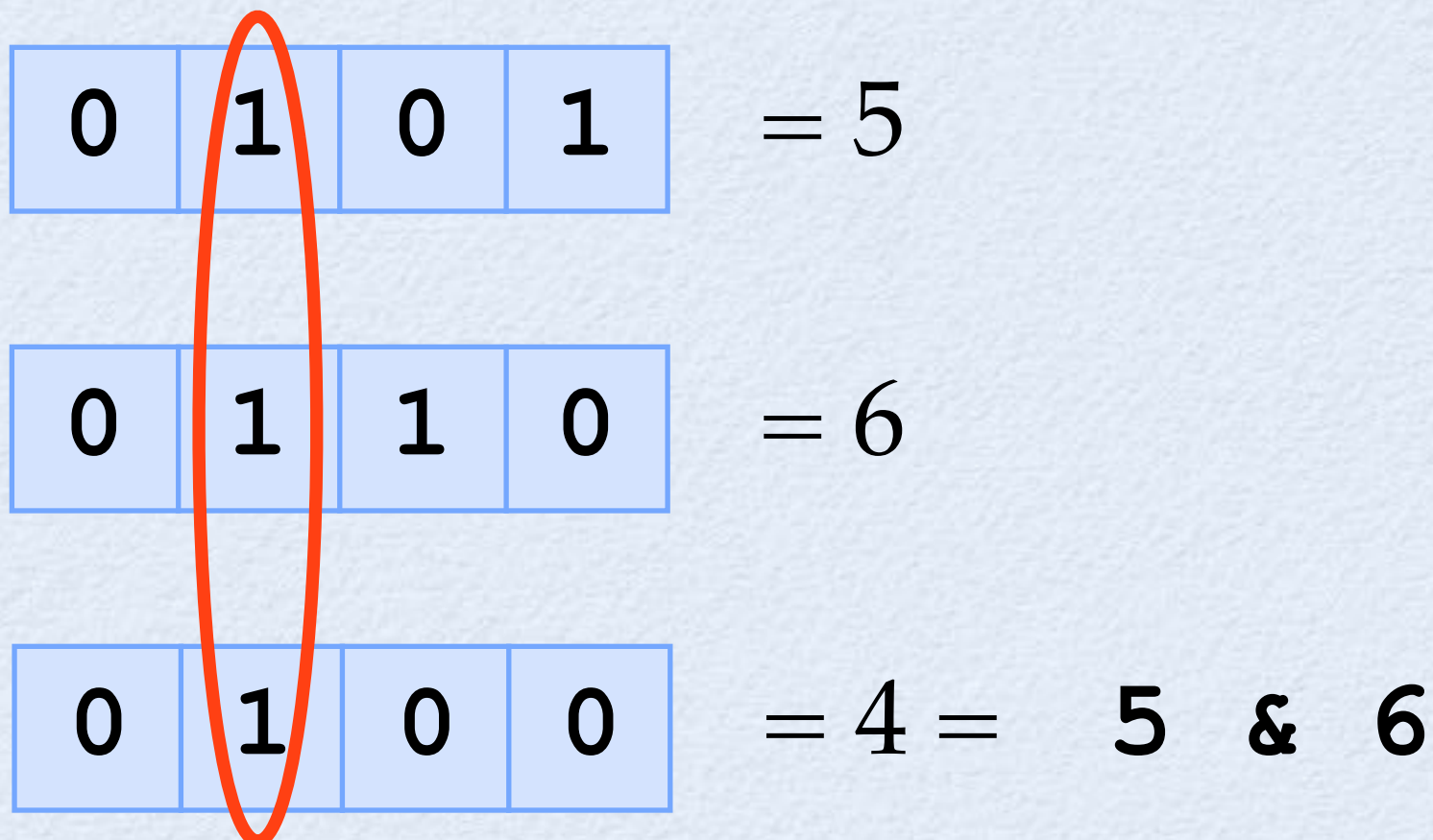
 = 6

0	1	0	0
---	---	---	---

 = 4 = 5 & 6

Opérateurs bit-à-bit

- Pour l'opérateur binaire **&** il y aura un 1 dans l'encodage de la valeur résultante si et seulement si il y a un 1 à la position correspondante des encodages des 2 opérandes



Opérateurs bit-à-bit

- Pour l'opérateur binaire $|$ il y aura un 0 dans l'encodage de la valeur résultante si et seulement si il y a un 0 à la position correspondante des encodages des 2 opérandes

0	1	0	1
---	---	---	---

 = 5

0	1	1	0
---	---	---	---

 = 6

0	1	1	1
---	---	---	---

 = 7 = 5 | 6

Opérateurs bit-à-bit

- Pour l'opérateur binaire $|$ il y aura un 0 dans l'encodage de la valeur résultante si et seulement si il y a un 0 à la position correspondante des encodages des 2 opérandes

0	1	0	1	= 5
0	1	1	0	= 6
0	1	1	1	= 7 = 5 6

Opérateurs bit-à-bit

- Pour l'opérateur binaire \wedge il y aura un 0 dans l'encodage de la valeur résultante si et seulement si il y a la même valeur à la position correspondante des encodages des 2 opérandes

0	1	0	1
---	---	---	---

 = 5

0	1	1	0
---	---	---	---

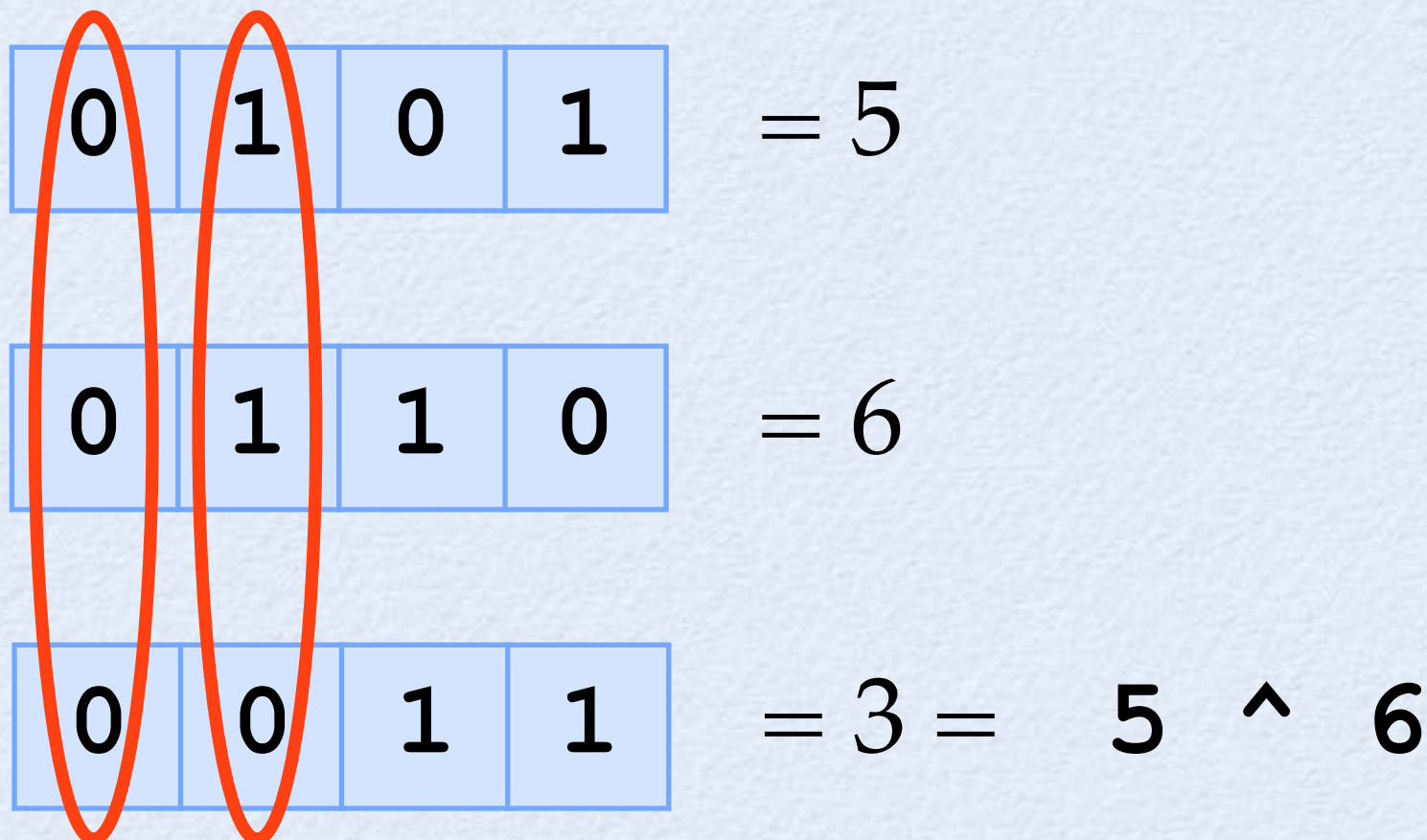
 = 6

0	0	1	1
---	---	---	---

 = 3 = 5 \wedge 6

Opérateurs bit-à-bit

- Pour l'opérateur binaire \wedge il y aura un 0 dans l'encodage de la valeur résultante si et seulement si il y a la même valeur à la position correspondante des encodages des 2 opérandes



Opérateurs bit-à-bit

- Pour l'opérateur binaire \ll , l'encodage de l'opérande de gauche se fait décaler vers la gauche d'un nombre de bits égal à l'opérande de droite (bits entrants = 0)

1	1	...	1	1	0	0	1
---	---	-----	---	---	---	---	---

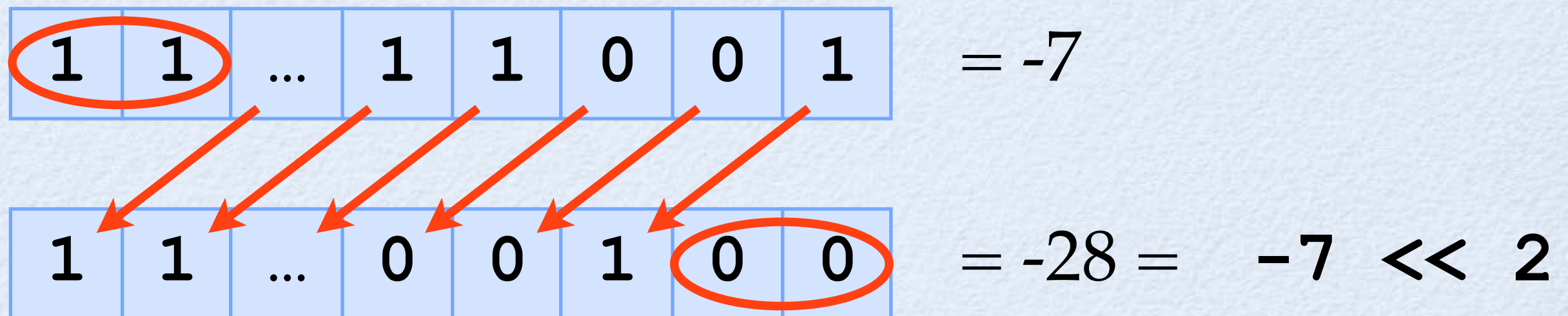
 $= -7$

1	1	...	0	0	1	0	0
---	---	-----	---	---	---	---	---

 $= -28 = -7 \ll 2$

Opérateurs bit-à-bit

- Pour l'opérateur binaire \ll , l'encodage de l'opérande de gauche se fait décaler vers la gauche d'un nombre de bits égal à l'opérande de droite (bits entrants = 0)



Opérateurs bit-à-bit

- Pour l'opérateur binaire \gg , l'encodage de l'opérande de gauche se fait décaler vers la droite d'un nombre de bits égal à l'opérande de droite (bits entrants = même que signe)

1	1	...	1	1	0	0	1
---	---	-----	---	---	---	---	---

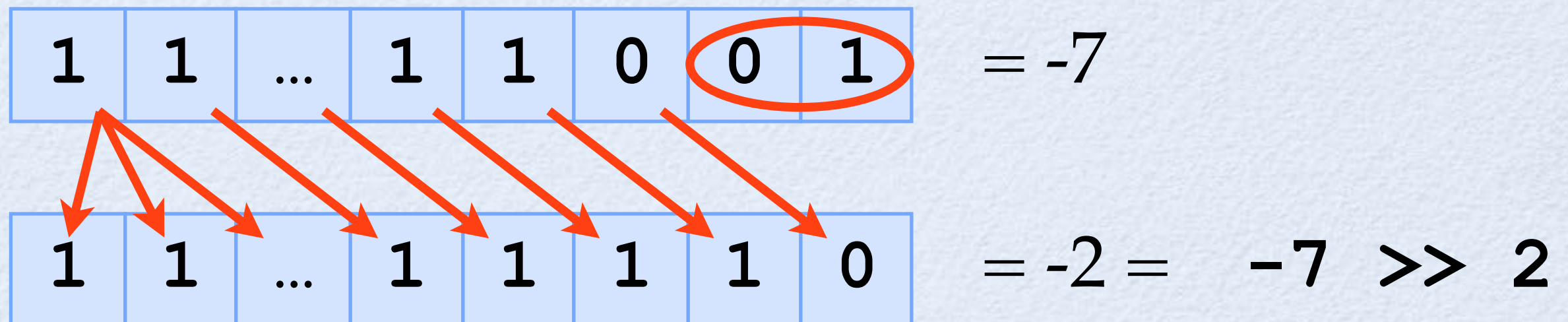
 $= -7$

1	1	...	1	1	1	1	0
---	---	-----	---	---	---	---	---

 $= -2 = -7 \gg 2$

Opérateurs bit-à-bit

- Pour l'opérateur binaire \gg , l'encodage de l'opérande de gauche se fait décaler vers la droite d'un nombre de bits égal à l'opérande de droite (bits entrants = même que signe)



Opérateurs bit-à-bit

- Pour l'opérateur binaire \ggg , l'encodage de l'opérande de gauche se fait décaler vers la droite d'un nombre de bits égal à l'opérande de droite (bits entrants = 0)

1	1	...	1	1	0	0	1
---	---	-----	---	---	---	---	---

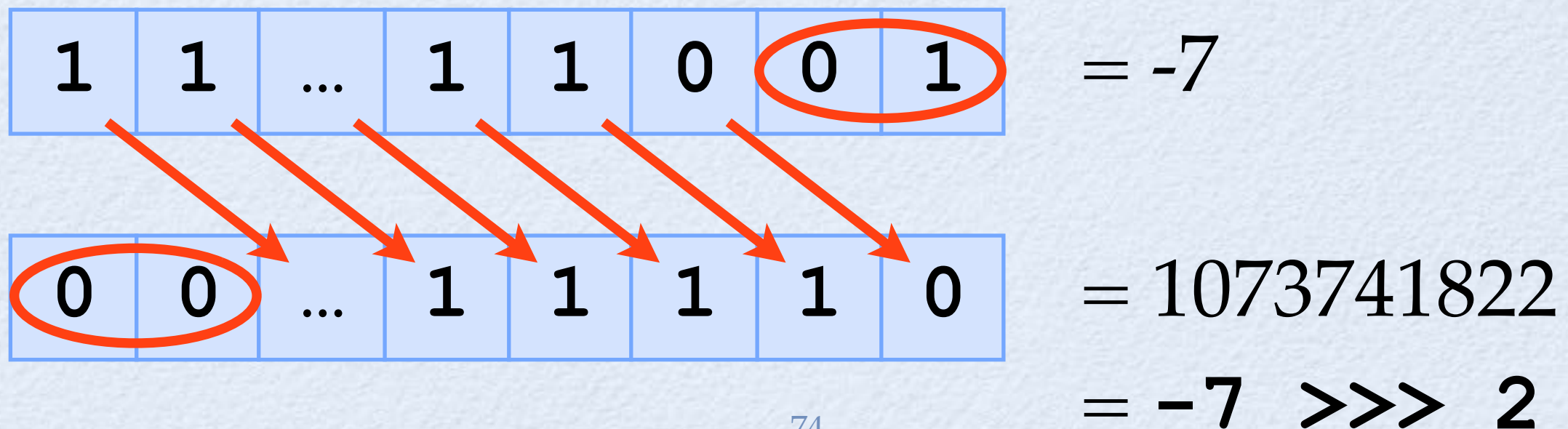
 = -7

0	0	...	1	1	1	1	0
---	---	-----	---	---	---	---	---

 = 1073741822
= -7 \ggg 2

Opérateurs bit-à-bit

- Pour l'opérateur binaire \ggg , l'encodage de l'opérande de gauche se fait décaler vers la droite d'un nombre de bits égal à l'opérande de droite (bits entrants = 0)



Fonctions mathématiques

- JS possède des **fonctions prédéfinies** qui correspondent à des fonctions mathématiques bien connues
- On accède à ces fonctions par l'objet **Math**
- Par exemple **Math.sqrt(9)** calcule $\sqrt{9}$
- Pour l'instant on considèrera simplement que **Math.sqrt** est un identificateur (comme si le caractère `< . >` pouvait faire partie d'un ident.)

Fonctions mathématiques

- **Math.abs** (x) : valeur absolue de x
- **Math.sin** (x) : sinus de x
- **Math.cos** (x) : cosinus de x
- **Math.tan** (x) : tangente de x
- **Math.asin** (x) : arc sinus de x
- **Math.acos** (x) : arc cosinus de x
- **Math.atan** (x) : arc tangente de x
- **Math.atan2** (y, x) : arc tangente de y/x
- **Math.exp** (x) : e à la puissance x
- **Math.log** (x) : logarithme de x en base e

Fonctions mathématiques

- **Math.ceil**(x) : plus petit nb. entier $\geq x$
- **Math.floor**(x) : plus grand nb. entier $\leq x$
- **Math.round**(x) : nb. entier plus proche de x
- **Math.min**(x, y, \dots) : minimum de x, y, \dots
- **Math.max**(x, y, \dots) : maximum de x, y, \dots
- **Math.pow**(x, y) : x à la puissance y
- **Math.sqrt**(x) : racine carrée de x
- **Math.random**() : nombre aléatoire ≥ 0 et < 1

Fonctions mathématiques

```
> 4*Math.atan(1)
3.141592653589793
> Math.pow(2,5)
32
> Math.floor(13.75)
13
> Math.floor(6*Math.random()+1)
2
> Math.floor(6*Math.random()+1)
6
> Math.floor(6*Math.random()+1)
1
```


Opérateur modulo

- L'opérateur binaire `%` calcule le **modulo** (ou **reste après division**)
- Par exemple `7.25 % 2 = 1.25` car

$$3 \times 2 + 1.25 = 7.25$$

- Pour deux nombres positifs x et y on a que :

$$x \% y = x - y * \text{Math.floor}(x/y)$$

Opérateur modulo

>	0	%	3
>	0		
>	1	%	3
>	1		
>	2	%	3
>	2		
>	3	%	3
>	0		
>	4	%	3
>	1		
>	5	%	3
>	2		
>	6	%	3
	0		