

IFT1015 Programmation 1

Programmes complets
Énoncés, affectations, boucles

Marc Feeley

(avec ajouts de Aaron Courville et Pascal Vincent)

Programmes complets

Éditeurs de code

- La **console** de codeBoot est idéale pour le codage de programmes d'une ligne, pour expérimenter et pour le **débogage interactif**
- Pour des programmes qu'on cherche à exécuter plus d'une fois ou qui sont plus longs il est utile de composer le code avec un **éditeur de code**
- L'éditeur de code peut être un logiciel indépendant (p.ex. **emacs, vi, notepad, textedit, atom**) ou bien il est intégré à un environnement de développement, comme c'est le cas avec codeBoot

Éditeur de code de codeBoot

- codeBoot offre un éditeur de code intégré



Éditeur de code de codeBoot

- codeBoot offre un éditeur de code intégré



Quand le curseur est dans un éditeur les boutons de contrôle lancent l'exécution du code s'y trouvant

Syntaxe des programmes

- Un programme JS est une suite d'énoncés qui seront exécutés **séquentiellement**
- Un énoncé est un fragment de programme qui est exécuté pour son **effet**
- Plusieurs des énoncés JS se terminent **optionnellement** par un « ; » (il peut être omis dans certains contextes)
- Pour améliorer la compréhension du code et la compatibilité syntaxique avec d'autres langages, c'est une bonne habitude de **toujours inclure les « ; »** même s'ils sont optionnels

Syntaxe des programmes

- Les énoncés vus à date :

<expression>;

Énoncé expression

var *<identificateur> = <expression>;*

Déclaration de var.

if (*<expression>*) *<énoncé>*

Énoncé conditionnel

if (*<expression>*) *<énoncé>* **else** *<énoncé>*

Énoncé conditionnel

- Il ne faut pas oublier qu'une *<expression>* peut être une constante littérale, une variable, une expression avec opérateur (+, -, ...), ou un appel de fonction, tel que **print** ("allo") ou **Math.max**(1+2, 15)

Syntaxe des programmes

- D'autres énoncés fréquemment rencontrés :

for (⟨exp.⟩ ; ⟨exp.⟩ ; ⟨exp.⟩) ⟨énoncé⟩

Boucle «for»

while (⟨expression⟩) ⟨énoncé⟩

Boucle «tant-que»

do ⟨énoncé⟩ **while** (⟨expression⟩) ;

Boucle «répéter»

return ⟨expression⟩ ;

Retour de fonction

switch (⟨exp.⟩) { ⟨cas⟩ }

Selection de cas

{ ⟨énoncé⟩ ⟨énoncé⟩ ... }

Bloc

;

Énoncé vide

- Si on examine en détail la grammaire complète de JS, on peut voir qu'un énoncé se termine toujours par un « ; » (optionnel en fin de ligne) ou une « } »

Commentaires

- Les programmes ont aussi des **commentaires** :
 - // texte explicatif* *une seule ligne*
 - /* texte explicatif */* *lignes multiples*
- Peuvent se placer à n'importe quel endroit où on peut retrouver des espaces blancs, et ils ont le **même effet que des espaces blancs**
- Ils sont utiles pour la **communication entre programmeurs** : pour garder des notes sur l'utilité du code, son fonctionnement, ses auteurs, sa date de création, sa licence d'utilisation, la présence de bugs connus, les trucs de codage,...

Qualité des commentaires

- Les commentaires sont destinés à être **lus par d'autres programmeurs**
- Il faut donner des informations **utiles** à la maintenance du logiciel (donc à la compréhension du code)
- C'est particulièrement important de signaler les choses qui **ne sont pas immédiatement évidentes en regardant le code**
- Des commentaires en trop grand nombre ou trop détaillés peuvent **nuire** à la compréhension autant que d'avoir trop peu de commentaires

Coder avec style

- On évite d'avoir des lignes de code de plus de 80 caractères pour que le code soit facile à visualiser quel que soit l'éditeur de code utilisé
- On utilise des lignes blanches et une indentation uniforme du code pour que sa structure soit claire
- On place normalement les commentaires courts sur la même ligne que le code visé, à sa droite
- Les commentaires plus longs (≥ 2 lignes) sont placés avant le code visé

Coder avec style... exemples

```
// On appelle prompt pour lire un nombre.  
var x = +prompt("montant?");  
// On teste si la variable est < 0.  
if (x < 0) alert("montant incorrect");
```

Mauvais... car le code est trop compacté, les identificateurs sont peu significatifs et les commentaires n'ajoutent rien pour améliorer la compréhension

```
// Lire le montant à déposer du client.  
var depot = +prompt("montant?");  
  
// Valider le montant entré.  
  
if (depot < 0) {  
    alert("montant incorrect");  
}
```

Beaucoup mieux... en particulier le choix d'identificateur et les commentaires aident à comprendre le code

Affectation

Affectation

- L'**affectation** (**assignment**) est une opération qui change la valeur contenue dans une cellule mémoire, comme celle associée à une **variable**
- Syntaxe simplifiée : $\langle \textit{identificateur} \rangle = \langle \textit{expression} \rangle$
- La valeur de $\langle \textit{expression} \rangle$ vient remplacer la valeur dans la cellule associée à $\langle \textit{identificateur} \rangle$
- Il ne faut pas confondre $x=y$ et $x==y$

Affectation

- Exemple :

```
> var n = 9;  
> var x = 1;
```

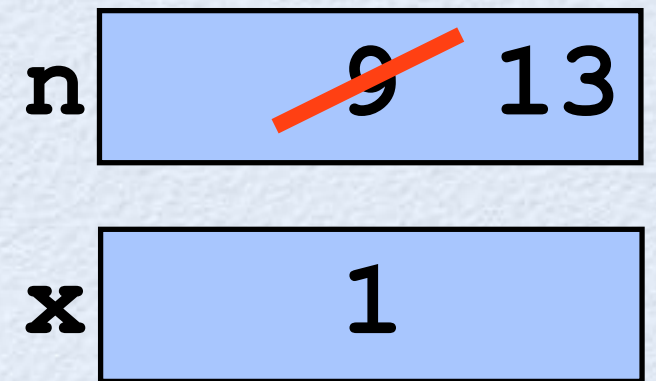
n	9
x	1

Deux déclarations de
variables (pas des
affectations)

Affectation

- Exemple :

```
> var n = 9;  
> var x = 1;  
> n = x+12;  
13
```

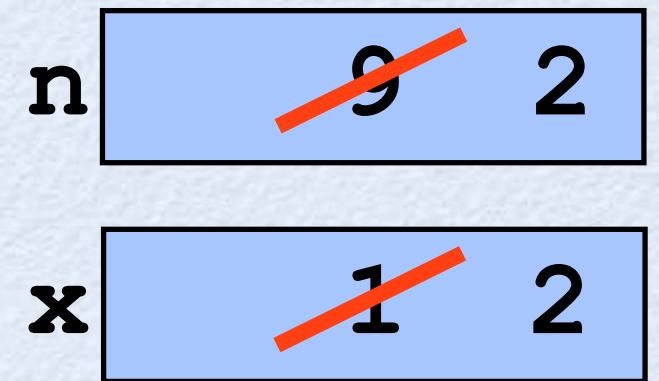


Noter que l'affectation est une **expression** (avec effet) dont la valeur est le nouveau contenu de la cellule

Affectation

- Exemple :

```
> var n = 9;  
> var x = 1;  
> n = x = 2;  
2
```



L'affectation est associative à droite, donc **n=x=2** est équivalent à **n=(x=2)**

Affectation

- Exemple : qu'affiche ce programme?

```
var n = 9;
```

```
var x = 1;
```

```
n = x;
```

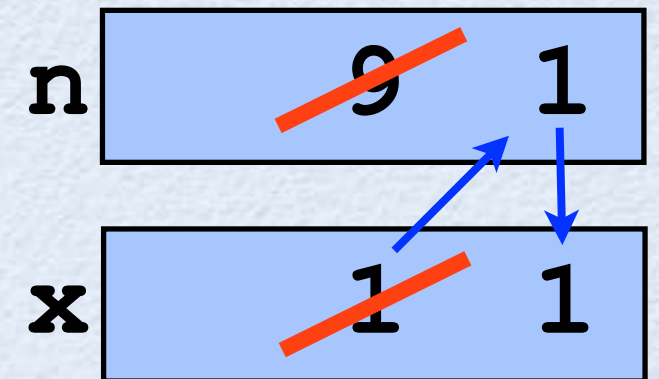
évaluation: 1
copie

```
x = n;
```

évaluation: 1
copie

```
print("x="+x) ;
```

```
print("n="+n) ;
```



Affectation

- Programme qui échange deux variables correctement

```
var n = 9;
```

```
var x = 1;
```

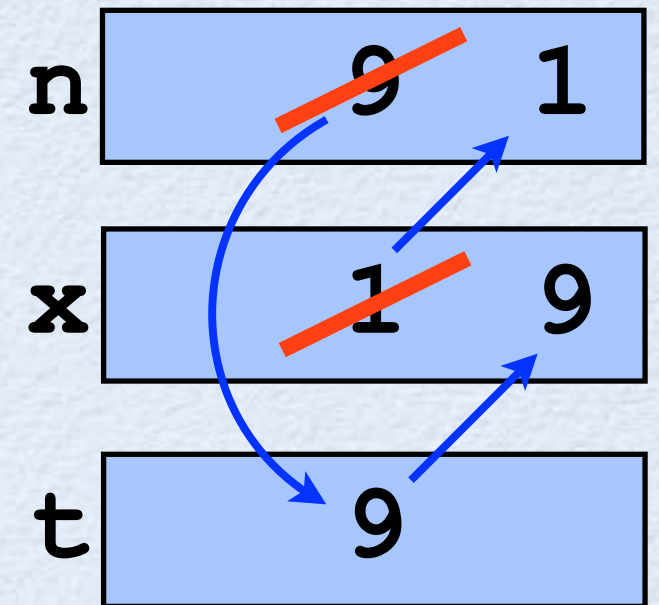
```
var t = n;
```

```
n = x;
```

```
x = t;
```

```
print("x="+x) ;
```

```
print("n="+n) ;
```



Variables indéfinies

- En JS, on peut omettre l'expression dans une **déclaration de variable**
- Syntaxe : **var** *<identificateur>*
- Dans ce cas, la cellule créée pour la variable contiendra la valeur spéciale
undefined – indéfinie

On peut ensuite se servir d'une affectation pour changer le contenu de la cellule

Variables indéfinies

- Il est préférable d'éviter de créer des variables indéfinies, car pendant une partie de l'exécution la variable est dans un état douteux

```
var prix = 10;  
var total = 1.15 * prix;
```

mieux que

```
var prix = 10;  
var total;  
total = 1.15 * prix;
```


Variables indéfinies

- Puisque c'est une source de bogues, codeBoot traite la lecture de variable indéfinie comme une **erreur**
- Exemple :

```
> var n;  
> n+1;  
    cannot read the undefined variable n  
> n = 2;  
    2  
> n+1;  
    3
```

n ~~undefined~~ **2**

Variables indéfinies

- Exemple : programme calculant la valeur absolue

```
// Fichier: abs.js

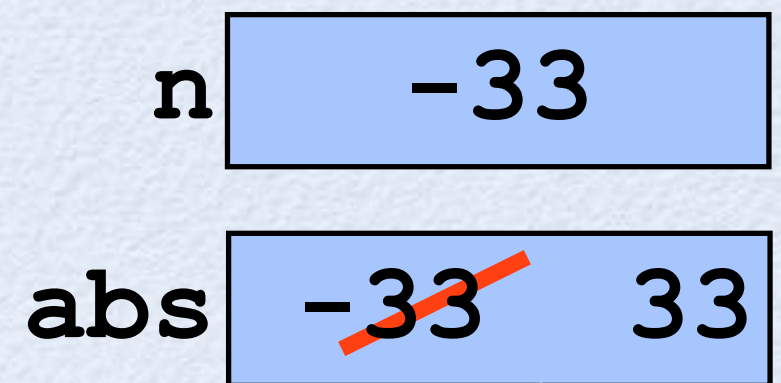
// Ce programme demande à l'utilisateur
// d'entrer un nombre et affiche à
// l'utilisateur la valeur absolue du nombre.

var n = +prompt("Nombre?");

var abs = n;

if (n < 0) // le nombre est-il négatif?
    abs = -n;

alert("La valeur absolue est " + abs);
```



La variable **abs**
n'est jamais
undefined

Variables indéfinies

- Exemple : programme calculant la valeur absolue

```
// Fichier: abs2.js

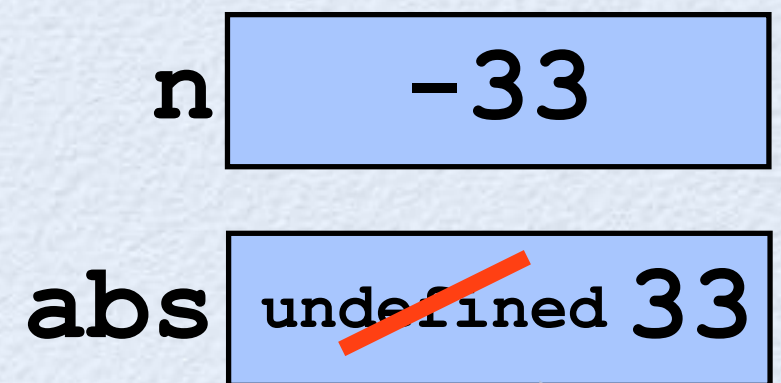
// Ce programme demande à l'utilisateur
// d'entrer un nombre et affiche à
// l'utilisateur la valeur absolue du nombre.

var n = +prompt("Nombre?");

var abs;

if (n < 0) // le nombre est-il négatif?
    abs = -n;
else
    abs = n;

alert("La valeur absolue est " + abs);
```




La variable **abs**
est **undefined**
au début

Affectation

- L'affectation est intéressante pour **accumuler** les données d'un calcul dans une ou des variables

```
> var somme = 0;  
> somme = somme + 7;  
7  
> somme = somme + 2;  
9
```

somme 

Affectation

- Exemple : calcul de la racine carrée d'un nombre par **approximation successive** (méthode de Newton), sans utiliser **Math.sqrt**

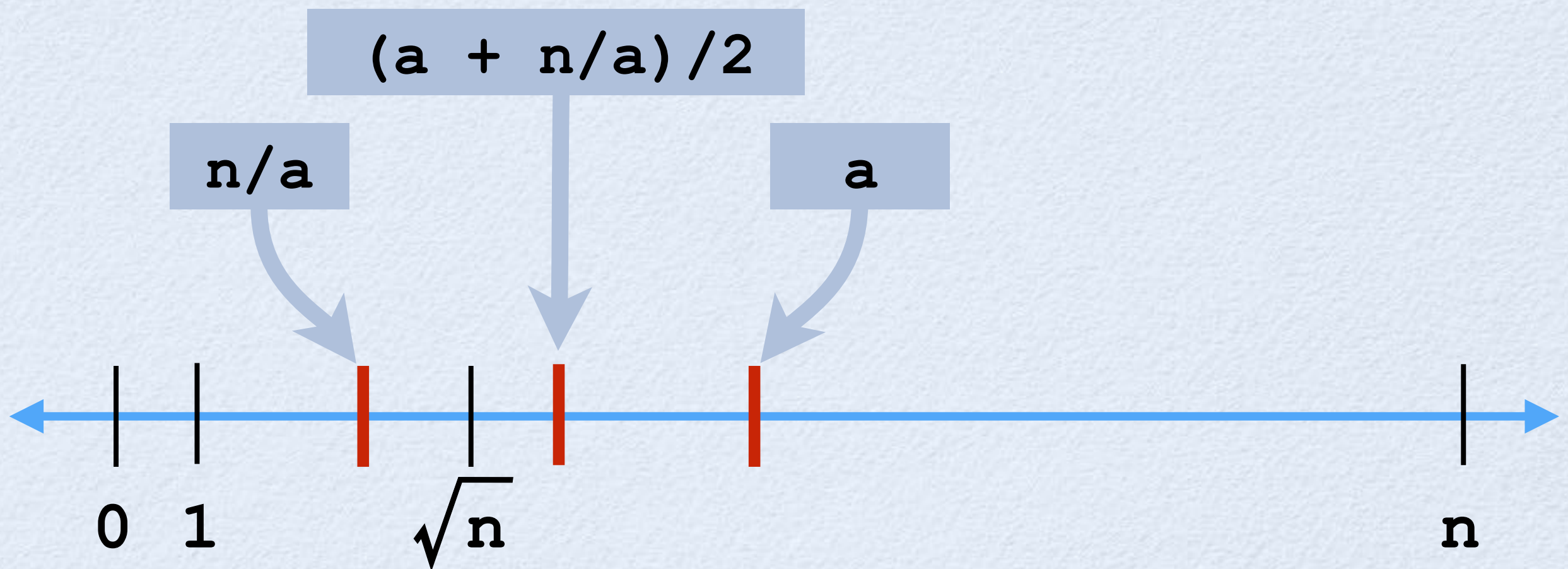
```
> var n = 16;  
> var a = n;  
> a = (a + n/a) / 2;  
8.5  
> a = (a + n/a) / 2;  
5.1911764705882355  
> a = (a + n/a) / 2;  
4.136664722546242  
> a = (a + n/a) / 2;  
4.002257524798522  
> a = (a + n/a) / 2;  
4.000000636692939  
> a = (a + n/a) / 2;  
4.00000000000000051  
> a = (a + n/a) / 2;  
4
```

a c'est
l'approximation
de la racine
carrée de **n**

constatation : **a** est
de plus en plus
près de la racine
carrée de **n**

Affectation

- La méthode de Newton raffine l'approximation à chaque nouveau calcul



Affectation

- Programme `racine16.js` :

```
// Fichier: racine16.js

// Ce programme calcule et imprime la racine carrée de 16.

var n = 16; // le nombre dont il faut calculer la racine
var a = n;   // a est l'approximation de la racine

a = (a + n/a) / 2; // calculer la prochaine approximation
a = (a + n/a) / 2; // quelques fois...
a = (a + n/a) / 2;
a = (a + n/a) / 2;
a = (a + n/a) / 2;
a = (a + n/a) / 2;
a = (a + n/a) / 2;

print(a); // affiche 4
```


Affectation

- Ça ne fonctionne pas pour des grands **n** :

```
// Fichier: racine100.js

// Ce programme calcule et imprime la racine carrée de 100.

var n = 100; // le nombre dont il faut calculer la racine
var a = n;    // a est l'approximation de la racine

a = (a + n/a) / 2; // calculer la prochaine approximation
a = (a + n/a) / 2; // quelques fois...
a = (a + n/a) / 2;
a = (a + n/a) / 2;
a = (a + n/a) / 2;
a = (a + n/a) / 2;
a = (a + n/a) / 2;

print(a); // affiche 10.000000000139897
```


Boucle while

- On aimerait **répéter** le calculer de la prochaine approximation, tant que nécessaire
- La **boucle «tant-que»** (while) permet de répéter un énoncé, **tant qu'une certaine condition est vraie**
- Syntaxe : **while** (*<expression>*) *<énoncé>*
- L'*<énoncé>* c'est le **corps de la boucle**
- Chaque exécution du corps est une **itération**
- L'*<expression>* est évaluée avant chaque itération et après la dernière itération pour vérifier la condition

Boucle while

- La condition c'est que la prochaine approximation doit être **plus proche** de la racine carrée


```
// Fichier: racine900.js

// Ce programme calcule et imprime la racine carrée de 900.

var n = 900; // le nombre dont il faut calculer la racine
var a = n;   // a est l'approximation de la racine

while (a > (a + n/a) / 2) {
    a = (a + n/a) / 2; // calculer la prochaine approx.
}

print(a); // affiche 30
```



Remarques : 1) utilisation d'un **bloc** dans le corps de boucle
2) **indentation** du corps de la boucle

Boucle while

- L'utilisation systématique d'un bloc dans le corps facilite les ajouts, par exemple pour le débogage :

```
// Fichier: racine900-pause.js

// Ce programme calcule et imprime la racine carrée de 900.

var n = 900; // le nombre dont il faut calculer la racine
var a = n;   // a est l'approximation de la racine

while (a > (a + n/a) / 2) {
    pause();
    a = (a + n/a) / 2; // calculer la prochaine approx.
}

print(a); // affiche 30
```

Dans codeBoot, la fonction **pause** force la suspension de l'exécution ce qui permet d'observer l'évolution de l'état

Débogage par «trace»

- Une autre approche de débogage consiste à insérer des énoncés qui affichent une **trace** de l'état


```
// Fichier: racine900-print.js

// Ce programme calcule et imprime la racine carrée de 900.

var n = 900; // le nombre dont il faut calculer la racine
var a = n;   // a est l'approximation de la racine

while (a > (a + n/a) / 2) {
    print("n=" + n + " a=" + a);
    a = (a + n/a) / 2; // calculer la prochaine approx.
}

print(a); // affiche 30
```



Dans les environnements qui n'ont pas **pause** on peut se servir de **print** et **alert** pour observer l'évolution de l'état

Exemple : conversion binaire

- Spécification : étant donné un nombre entier $n \geq 0$, **afficher son encodage binaire**, p.ex. pour $n=13$ on doit afficher 1101
- Conception :
 - Le bit le plus à droite de l'encodage sera 0 si et seulement si n est pair (divisible par 2)
 - Les bits à sa gauche sont l'encodage binaire du plancher de $n/2$
 - Idée : divisions **répétées** de n par 2, jusqu'à 0

Exemple : conversion binaire

```
// Fichier: encbin1.js

// Ce programme imprime l'encodage binaire de 13.

// Prototype #1

var n = 13; // le nombre à convertir

while (n > 0) { // tant qu'il reste des bits
    print("n=" + n);
    n = Math.floor(n / 2); // passer aux autres bits
}
```

La progression de **n** est intéressante

Exemple : conversion binaire

```
// Fichier: encbin2.js

// Ce programme imprime l'encodage binaire de 13.

// Prototype #2

var n = 13; // le nombre à convertir

while (n > 0) { // tant qu'il reste des bits
    print(n % 2); // afficher un bit
    n = Math.floor(n / 2); // passer aux autres bits
}
```

Ne respecte pas la spécification...

Exemple : conversion binaire

```
// Fichier: encbin3.js

// Ce programme imprime l'encodage binaire de 13.

// Prototype #3

var n = 13; // le nombre à convertir
var e = ""; // l'encodage binaire du nombre à convertir

while (n > 0) { // tant qu'il reste des bits
    e = (n % 2) + e; // accumuler un bit
    n = Math.floor(n / 2); // passer aux autres bits
}

print(e);
```


Exemple : conversion binaire

```
// Fichier: encbin4.js

// Ce programme imprime l'encodage binaire de 13.

// Version complète

var n = 13; // le nombre à convertir
var e;      // l'encodage binaire du nombre à convertir

if (n == 0) {
    e = "0"; // 0 est un cas spécial
} else {
    e = "";
    while (n > 0) { // tant qu'il reste des bits
        e = (n % 2) + e; // accumuler un bit
        n = Math.floor(n / 2); // passer aux autres bits
    }
}

print(e);
```

On dit que le cas où $n = 0$ est un cas spécial
(une exception à la règle générale)

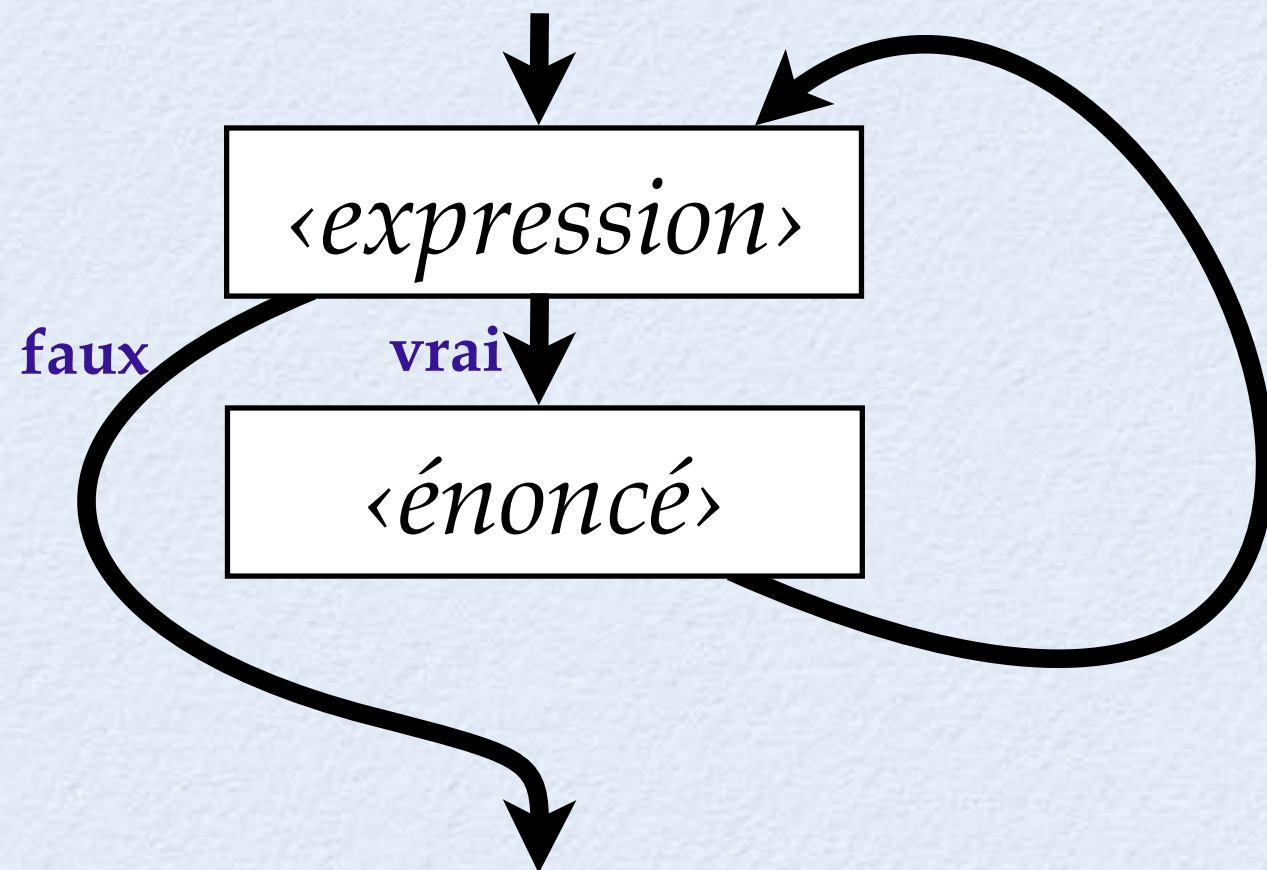
Boucle do-while

- La **boucle «répéter»** (do-while) exécute le corps de la boucle **avant** de vérifier la condition
- Syntaxe : **do** *«énoncé»* **while** (*«expression»*) ;
- Le corps de la boucle est exécuté **au moins une fois**
- Le corps de la boucle et la condition sont exécutés **le même nombre de fois**

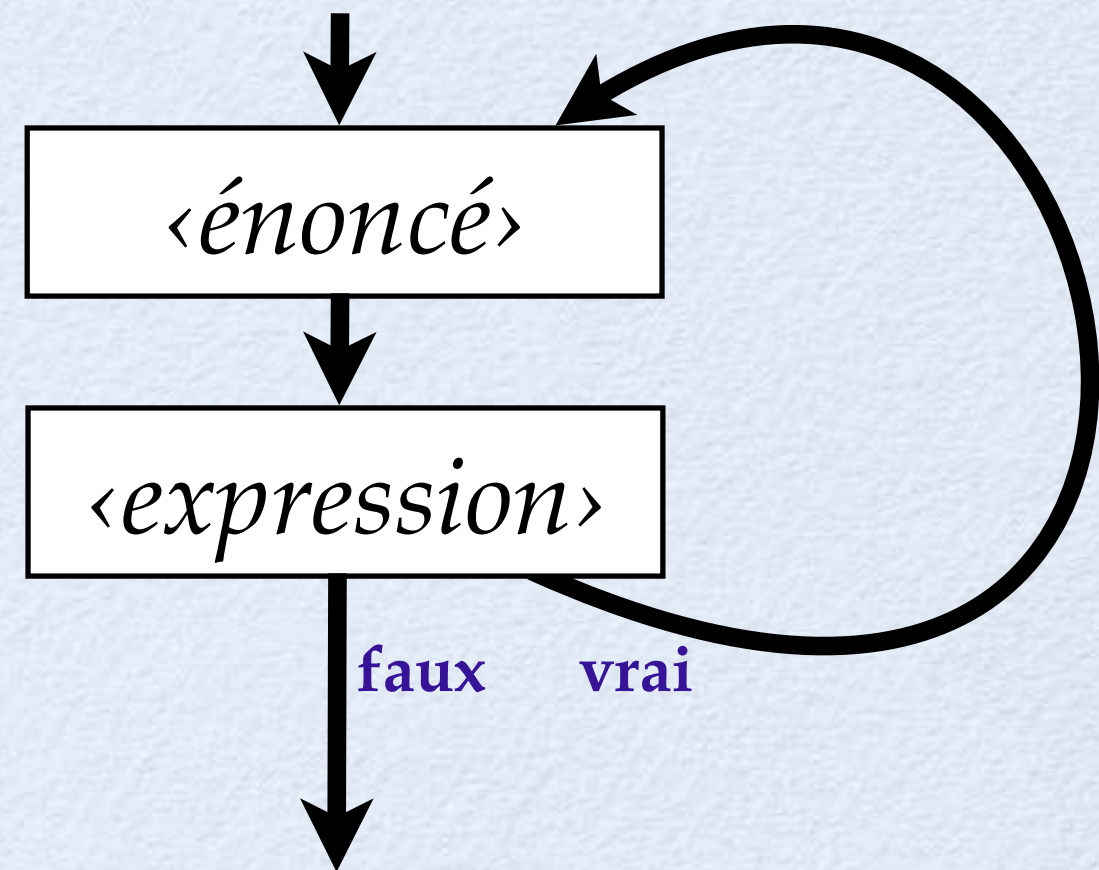
Boucles while et do-while

- Diagrammes de flux de contrôle respectifs:

while (*<expression>*) *<énoncé>*



do *<énoncé>* **while** (*<expression>*) ;



Exemple : conversion binaire

```
// Fichier: encbin5.js

// Ce programme imprime l'encodage binaire de 13.

// Version avec do-while

var n = 13; // le nombre à convertir
var e = ""; // l'encodage binaire du nombre à convertir

do {
    e = (n % 2) + e; // accumuler un bit
    n = Math.floor(n / 2); // passer aux autres bits
} while (n > 0); // continuer s'il reste des bits

print(e);
```

Dans ce code $n = 0$ n'est plus un cas spécial

Racine sans duplication

- On peut éliminer la duplication de code en ajoutant une variable **pa** “prochaine approximation”

```
// Fichier: racine900-sans-duplication.js

// Ce programme calcule et imprime la racine carrée de 900.

var n = 900; // le nombre dont il faut calculer la racine
var pa = n;  // la prochaine approximation de la racine
var a;      // l'approximation de la racine

do {
    a = pa;
    pa = (a + n/a) / 2; // calculer prochaine approximation
} while (pa < a);      // continuer si pas encore trouvé

print(a); // affiche 30
```

Raffinement successif

- L'approche de développement que nous avons utilisé est le **raffinement successif** :
 - On commence par le codage d'un **prototype**
 - On l'utilise pour **comprendre** la nature du calcul et **explorer** des solutions
 - On **teste** le code et on l'ajuste en conséquence
- Il est très rare que le codage soit bon dès la première fois!

Exemple : puissances

- Spécification : afficher les puissances de 2 jusqu'à 1024, c'est-à-dire 1, 2, 4, 8, ..., 1024
- Conception :
 - Variable **x** qui prendra les valeurs de 0 à 10
 - Chaque valeur sera l'itération d'une boucle
 - Se servir de **Math.pow(2, x)** pour calculer les puissances de 2

Exemple : puissances

```
// Fichier: puissances2.js
```

```
// Ce programme imprime les puissances de 2 entre 1 et 1024
```

```
var x = 0;
```

```
while (x <= 10) {  
    print(Math.pow(2,x)) ;  
    x = x + 1;  
}
```

```
// alternative #1
```

```
var p = 1;
```

```
var x = 0;
```

```
while (x <= 10) {  
    print(p) ;  
    p = p * 2;  
    x = x + 1;  
}
```

```
// alternative #2
```

```
var p = 1;
```

```
var x = 0;
```

```
while (x <= 10) {  
    print(p) ;  
    p *= 2;  
    x += 1;  
}
```


Affectation accumulante

- L'affectation est souvent utilisée pour changer le contenu d'une variable **en fonction de son état présent**
- Les opérateurs d'affectation accumulante **combinent une opération et une affectation**
- Les opérateurs sont :

$+=, -=, *=, /=, <<=, >>=, >>>=, \&=, ^=, |=, \%=$

$\langle id. \rangle += \langle exp. \rangle \quad \equiv \quad \langle id. \rangle = \langle id. \rangle + \langle exp. \rangle$

$\langle id. \rangle *= \langle exp. \rangle \quad \equiv \quad \langle id. \rangle = \langle id. \rangle * \langle exp. \rangle$

etc...

Note : \equiv veut dire «équivalent à»

Incrémentation/décrémentation

- Il y a des opérateurs unaires spéciaux pour faire avancer une variable par **pas** de 1 et -1 :
 - Incrémentation : $++\langle id.\rangle \equiv \langle id.\rangle += 1$
 - Décrémentation : $--\langle id.\rangle \equiv \langle id.\rangle -= 1$
- La valeur de l'expression est le contenu de la variable **après** l'incr. ou décr.

```
// alternative #4
```

```
var p = 1;  
var x = 0;  
  
while (x <= 10) {  
    print(p);  
    p *= 2;  
    ++x;  
}
```

```
// alternative #5
```

```
var p = 1;  
var x = 0;  
  
while (x <= 10) {  
    print(p);  
    p += p;  
    ++x;  
}
```


Post incr./décr.

- Il y a une version **postfixe** de ces opérateurs :
 - **Incrémentation** : $\langle id. \rangle ++ \equiv ++\langle id. \rangle - 1$
 - **Décrémentation** : $\langle id. \rangle -- \equiv --\langle id. \rangle + 1$
- La valeur de l'expression est le contenu de la variable **avant** l'incr. ou décr.

```
// alternative #7
```

```
var p = 1;
```

```
var x = 0;
```

```
while (x <= 10) {
```

```
    print(p);
```

```
    p *= 2;
```

```
    x++;
```

```
}
```


Boucle for

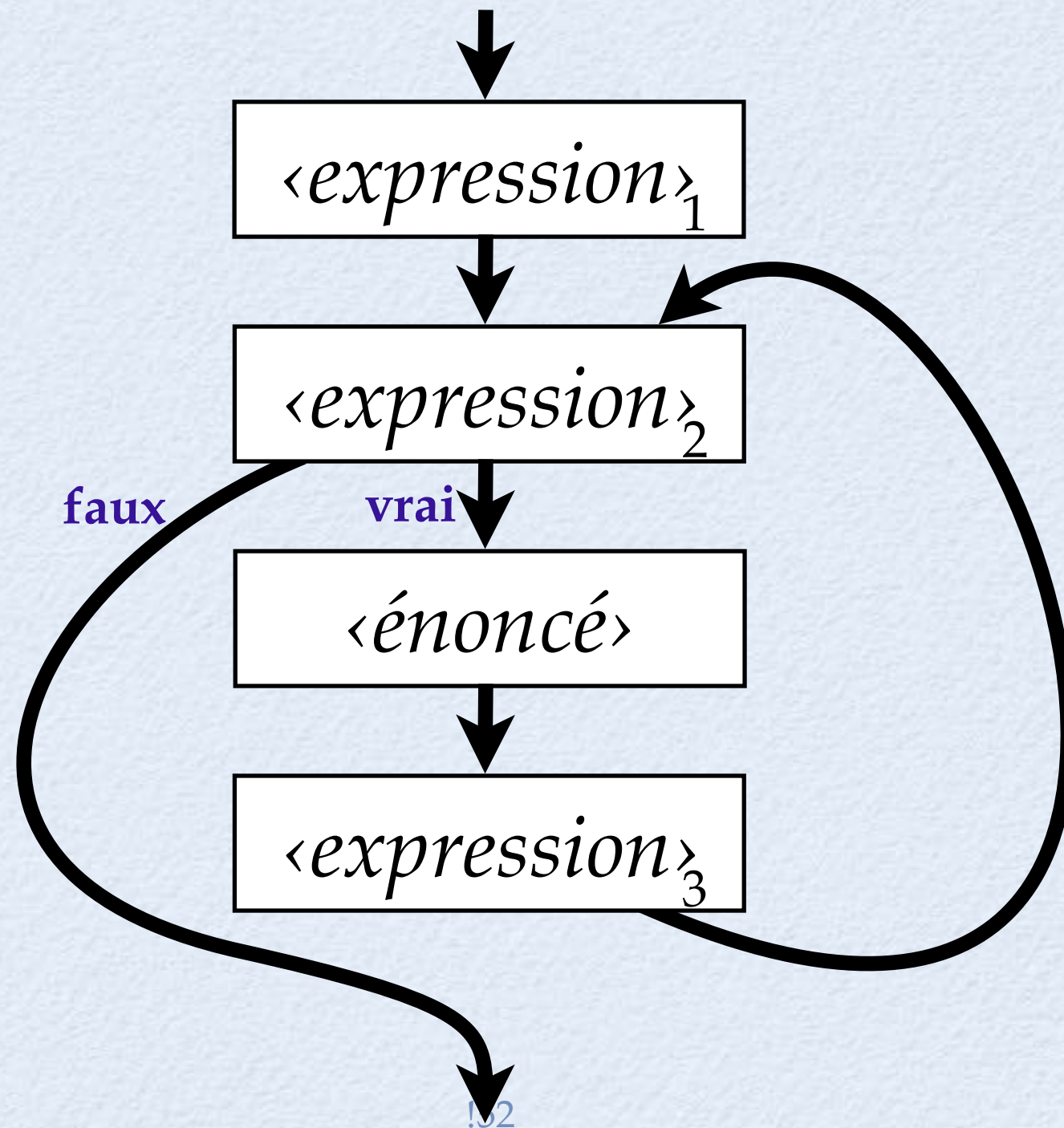
- La **boucle «for»** est utile pour faire des boucles basées sur une **variable d'itération**
- Une variable d'itération est une variable qui aura une **nouvelle valeur** à chaque itération de la boucle
- Syntaxe simplifiée :

for ($\langle exp. \rangle_1$; $\langle exp. \rangle_2$; $\langle exp. \rangle_3$) $\langle \text{énoncé} \rangle$

- Normalement $\langle exp. \rangle_1$ fait l'**initialisation** de la variable d'itération, $\langle exp. \rangle_2$ **vérifie la condition** d'itération et $\langle exp. \rangle_3$ fait **avancer** la variable d'itération à sa prochaine valeur

Boucle for

for ($\langle expression \rangle_1$; $\langle expression \rangle_2$; $\langle expression \rangle_3$) $\langle \text{énoncé} \rangle$



Boucle for

- À la place de $\langle exp. \rangle_1$ on peut mettre une **déclaration de variable**, ce qui a l'avantage d'alléger le code en déclarant la (les) variable(s) d'itération dans la boucle elle même

```
// alternative #12  
  
for (var x=0; x<=10; x++) {  
    print(1 << x);  
}
```

- Ce style est fortement conseillé car il donne du code **plus modulaire** (les déclarations de variables et leurs accès seront dans la même région de code)

Boucle for

```
// alternative #13
```

```
for (var x=0, p=1; x<=10; x++, p*=2) {  
    print(p);  
}
```

« x, y » est une expression dont la valeur est y après avoir évalué x (pour son effet)

```
// alternative #14
```

```
for (var p=1; p<=1024; p*=2) {  
    print(p);  
}
```


Boucles imbriquées

- On dit qu'une boucle est une **boucle imbriquée** lorsqu'elle se trouve dans le corps d'une autre boucle
- La boucle de niveau 1 (la **boucle principale**, ou la **boucle englobante**) est celle qui contient l'autre boucle dans son corps
- La boucle de niveau 2 (la **boucle imbriquée**, ou la **boucle interne**) est celle qui est contenue dans la boucle de niveau 1
- Il n'y a pas de limite sur le nombre de niveaux d'imbrication (mais ça peut avoir un effet négatif sur la performance du programme)

Boucles imbriquées

- Exemple : affichage visuel d'un "triangle" :

```
#  
##  
###  
####  
#####
```

Boucle principale
(de niveau 1)
ascendante

Boucle imbriquée
(de niveau 2)
descendante

```
// Fichier: triangle.js  
// Ce programme imprime un triangle.  
for (var i=1; i<=5; i++) {  
    var barre = "";  
    for (var j=i; j>=1; j--) {  
        barre += "#";  
    }  
    print(barre) ;  
}
```

Exemple : nombres premiers

- Spécification : imprimer les nombres premiers entre 2 et 100 (un nombre n est premier s'il se divise exactement, sans reste, par 1 et n seulement)
- Conception :
 - **Boucle principale** qui passe par tous les nombres entiers n entre 2 et 100
 - **Boucle imbriquée** qui essaie tous les nombres entiers d (diviseur) entre 2 et $n-1$
 - S'il existe un d pour lequel n modulo d est 0, le nombre n est certainement **pas premier**

Exemple : nombres premiers

```
// Fichier: premiers1.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #1

for (var n=2; n<=100; n++) {    // pour n entre 2 et 100

    var premier = true;        // n est premier (à date)

    for (var d=2; d<n; d++) {    // pour chaque diviseur
        if (n % d == 0) {        // as t-on trouvé un facteur?
            premier = false;      // n n'est pas premier
        }
    }

    if (premier) {                // imprimer s'il est premier
        print(n);
    }
}
```

Exemple : nombres premiers

```
// Fichier: premiers2.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #2

for (var n=2; n<=100; n++) {    // pour n entre 2 et 100

    var premier = true;        // n est premier (à date)

    for (var d=2; d<=Math.sqrt(n); d++) {
        if (n % d == 0) {      // as t-on trouvé un facteur?
            premier = false;    // n n'est pas premier
        }
    }

    if (premier) {              // imprimer s'il est premier
        print(n);
    }
}
```

pause(); // ~ 8200 steps

Exemple : nombres premiers

```
// Fichier: premiers3.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #3

for (var n=2; n<=100; n++) {    // pour n entre 2 et 100

    var premier = true;        // n est premier (à date)

    for (var d=2; premier && d<=Math.sqrt(n); d++) {
        if (n % d == 0) {      // as t-on trouvé un facteur?
            premier = false;    // n n'est pas premier
        }
    }

    if (premier) {              // imprimer s'il est premier
        print(n);
    }
}
```

pause(); // ~ 4400 steps

Exemple : nombres premiers

```
// Fichier: premiers3b.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #3b

for (var n=2; n<=100; n++) {    // pour n entre 2 et 100

    var premier = true;        // n est premier (à date)

    for (var d=2; d<=Math.sqrt(n); d++) {
        if (n % d == 0) {      // as t-on trouvé un facteur?
            premier = false;    // n n'est pas premier
            d = n;              // forcer la fin de la boucle
        }
    }

    if (premier) {              // imprimer s'il est premier
        print(n);
    }
}

pause();    // ~ 4700 steps
```

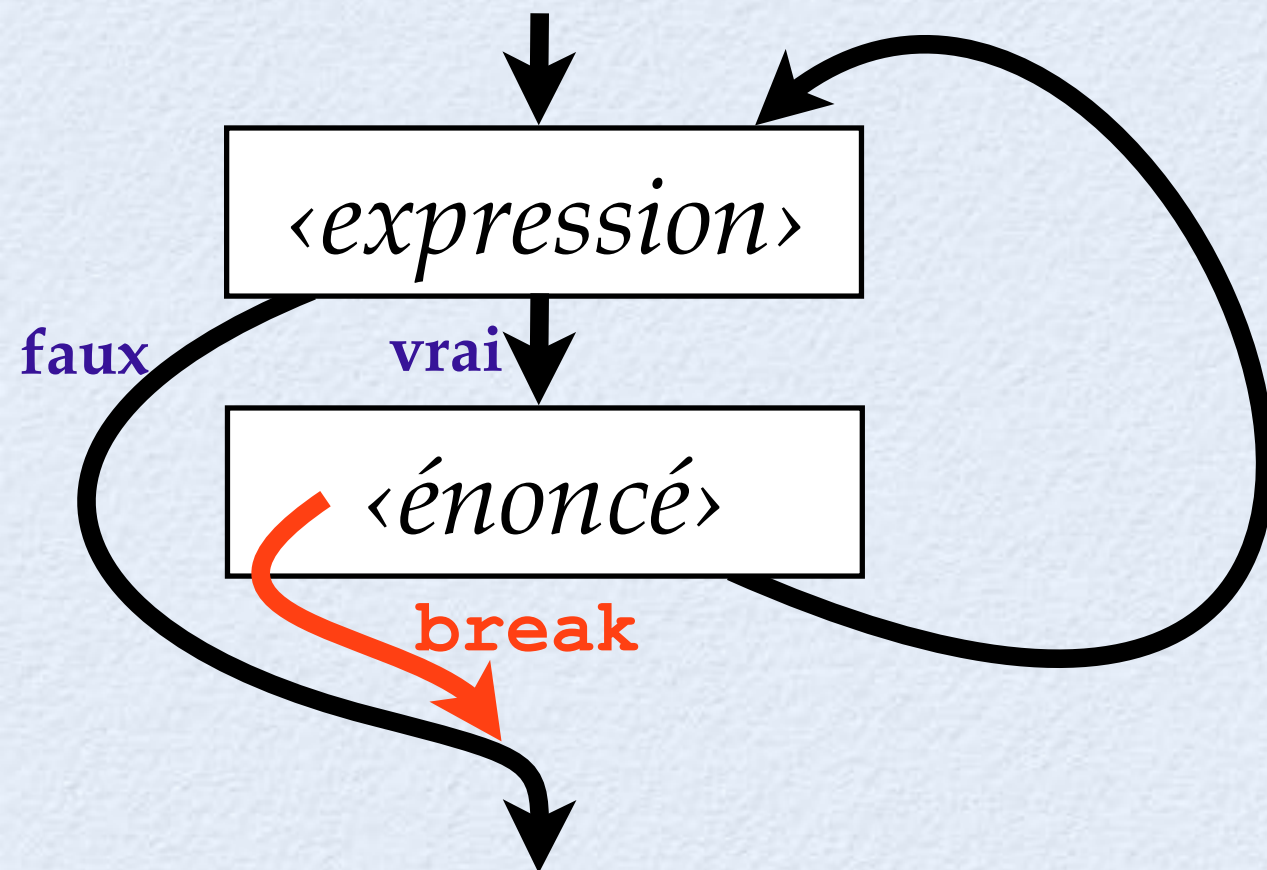

Énoncé break

- L'énoncé **break** force la fin prématurée de la boucle la plus imbriquée qui contient le **break**
- Syntaxe simplifiée : **break** ;
- Fonctionne avec toutes les formes de boucle
- L'exécution passe immédiatement au prochain énoncé qui suit la boucle

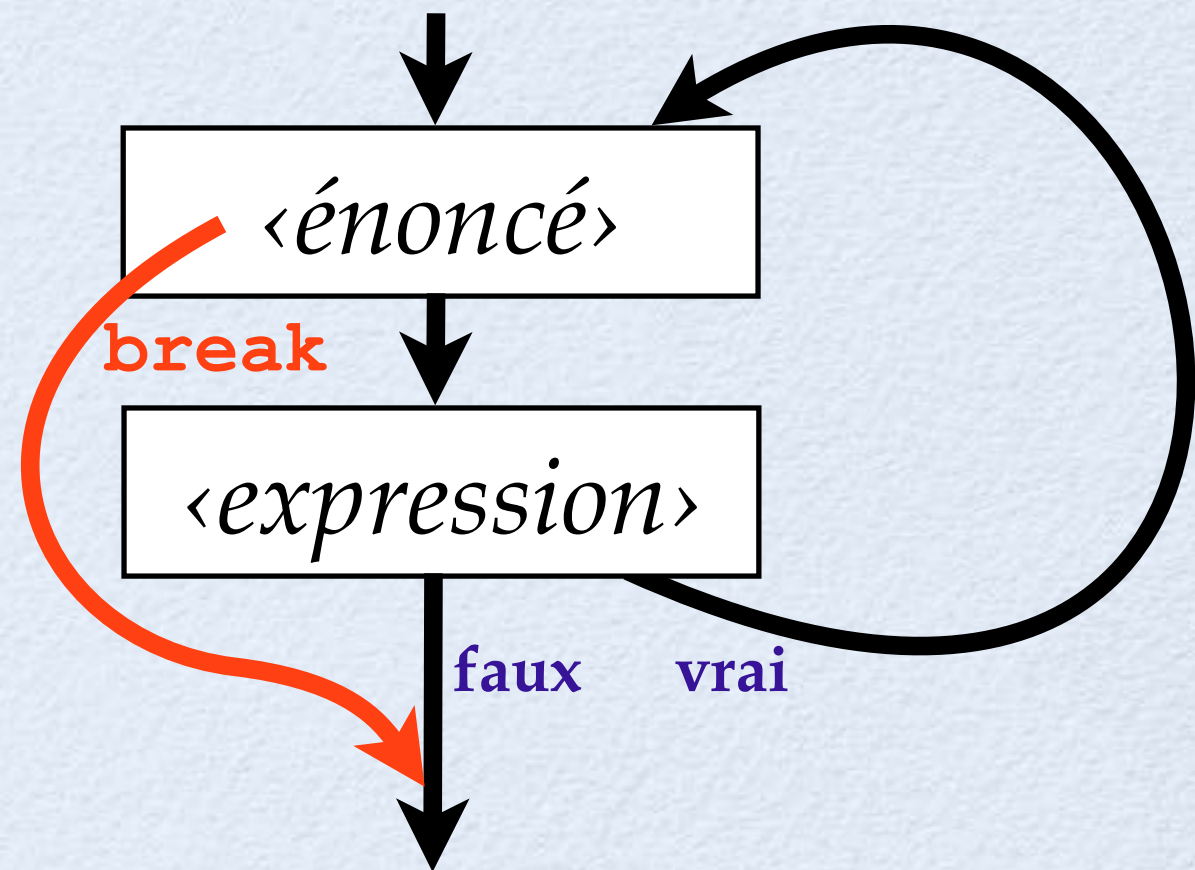
Énoncé break

- Diagrammes de flux de contrôle des boucles **while**, **do-while** et **for** en présence de **break** :

while (*<expression>*) *<énoncé>*

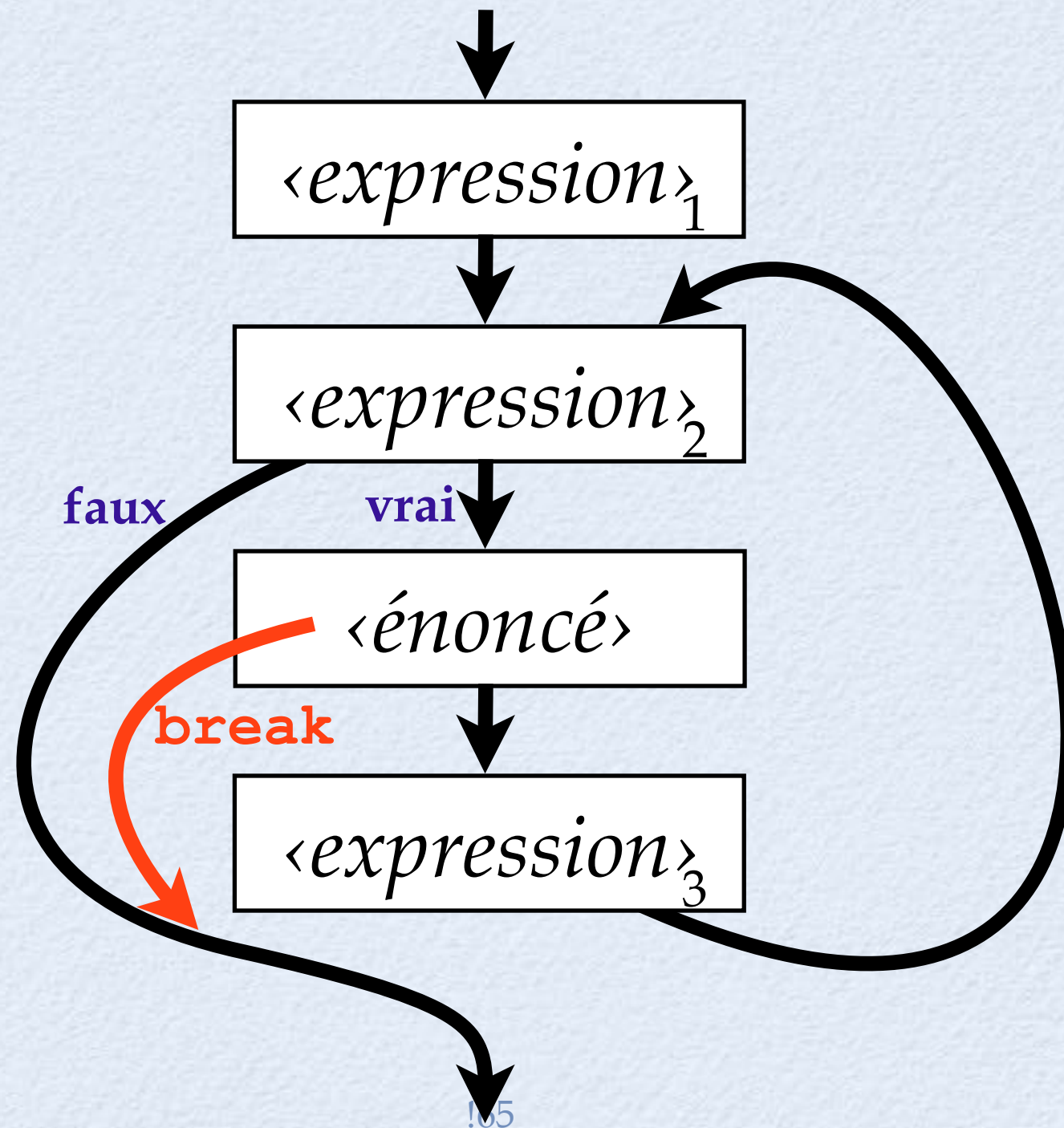


do *<énoncé>* **while** (*<expression>*) ;



Énoncé break

for ($\langle expression \rangle_1$; $\langle expression \rangle_2$; $\langle expression \rangle_3$) $\langle \text{énoncé} \rangle$



Exemple : nombres premiers

```
// Fichier: premiers4.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #4

for (var n=2; n<=100; n++) {    // pour n entre 2 et 100

    var premier = true;        // n est premier (à date)

    for (var d=2; d<=Math.sqrt(n); d++) {
        if (n % d == 0) {      // as t-on trouvé un facteur?
            premier = false;    // n n'est pas premier
            break;              // forcer la fin de la boucle
        }
    }

    if (premier) {              // imprimer s'il est premier
        print(n);
    }
}

pause();    // ~ 4100 steps
```


Exemple : nombres premiers

```
// Fichier: premiers5.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #5

for (var n=2; n<=100; n++) {    // pour n entre 2 et 100

    var max = Math.sqrt(n);    // diviseur maximal

    for (var d=2; d<=max; d++) {
        if (n % d == 0) {      // as t-on trouvé un facteur?
            max = 0;           // n n'est pas premier
            break;             // forcer la fin de la boucle
        }
    }

    if (max != 0) {            // imprimer s'il est premier
        print(n);
    }
}

pause();    // ~ 3800 steps
```

Exemple : nombres premiers

```
// Fichier: premiers6.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #6

for (var n=2; n<=100; n+=1+(n>2)) { // 2, 3, 5, 7, ... 99

    var max = Math.sqrt(n);      // diviseur maximal

    for (var d=3; d<=max; d+=2) { // d=3,5,7,...
        if (n % d == 0) {        // as t-on trouvé un facteur?
            max = 0;              // n n'est pas premier
            break;                // forcer la fin de la boucle
        }
    }

    if (max != 0) {              // imprimer s'il est premier
        print(n);
    }
}

pause(); // ~ 2000 steps
```

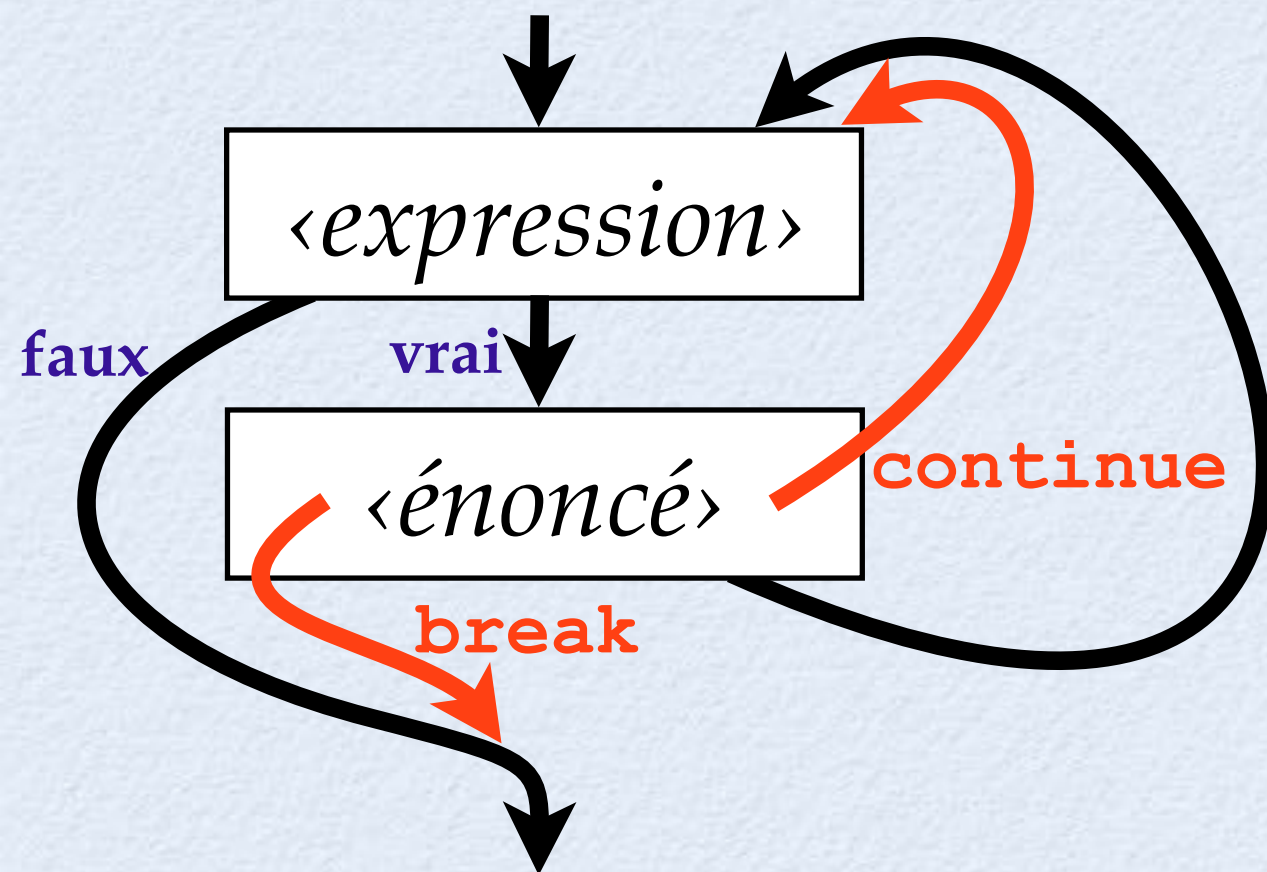

Énoncé continue

- L'énoncé **continue** force la **fin de l'itération présente** de la boucle la plus imbriquée qui contient le **continue**
- Syntaxe simplifiée : **continue** ;
- Fonctionne avec toutes les formes de boucle
- L'exécution saute immédiatement à la fin de l'itération

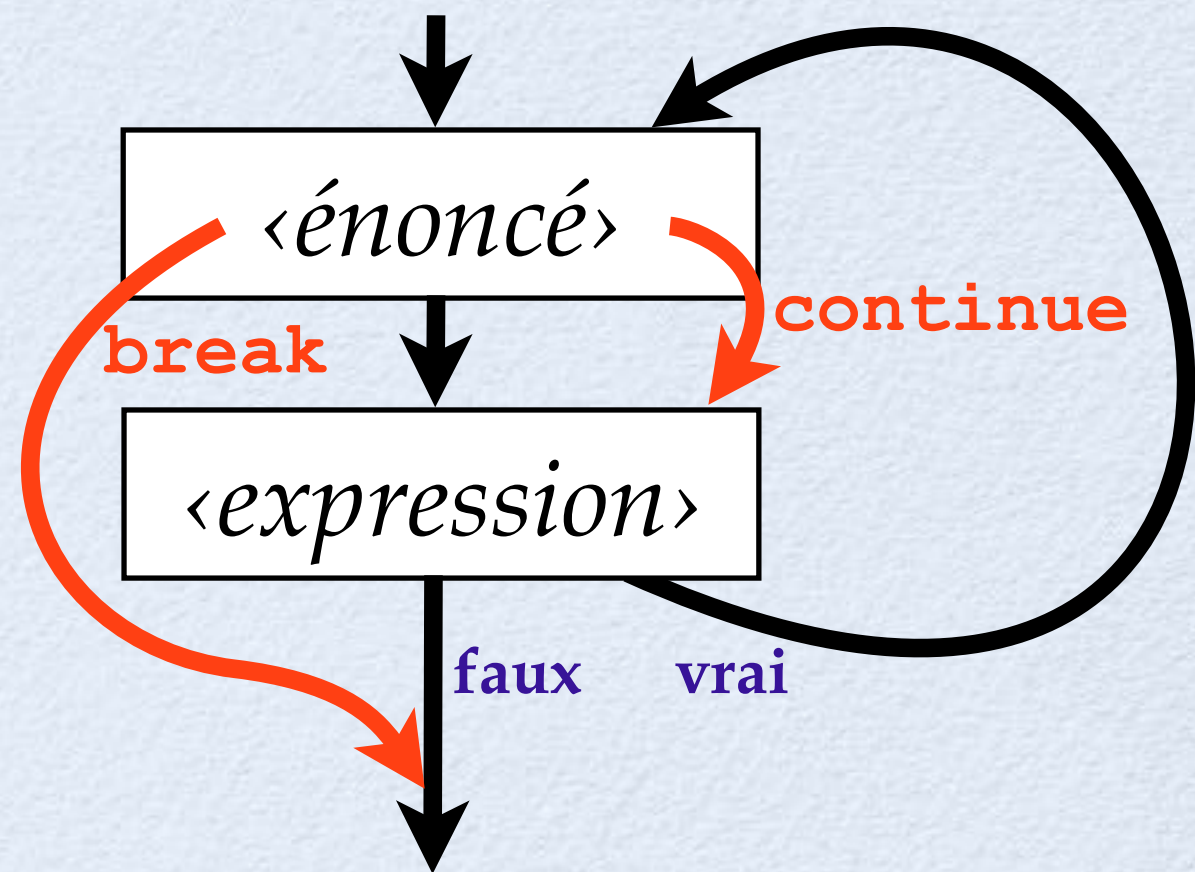
Énoncé continue

- Diagrammes de flux de contrôle des boucles **while**, **do-while** et **for** en présence de **break** et **continue** :

while (*<expression>*) *<énoncé>*

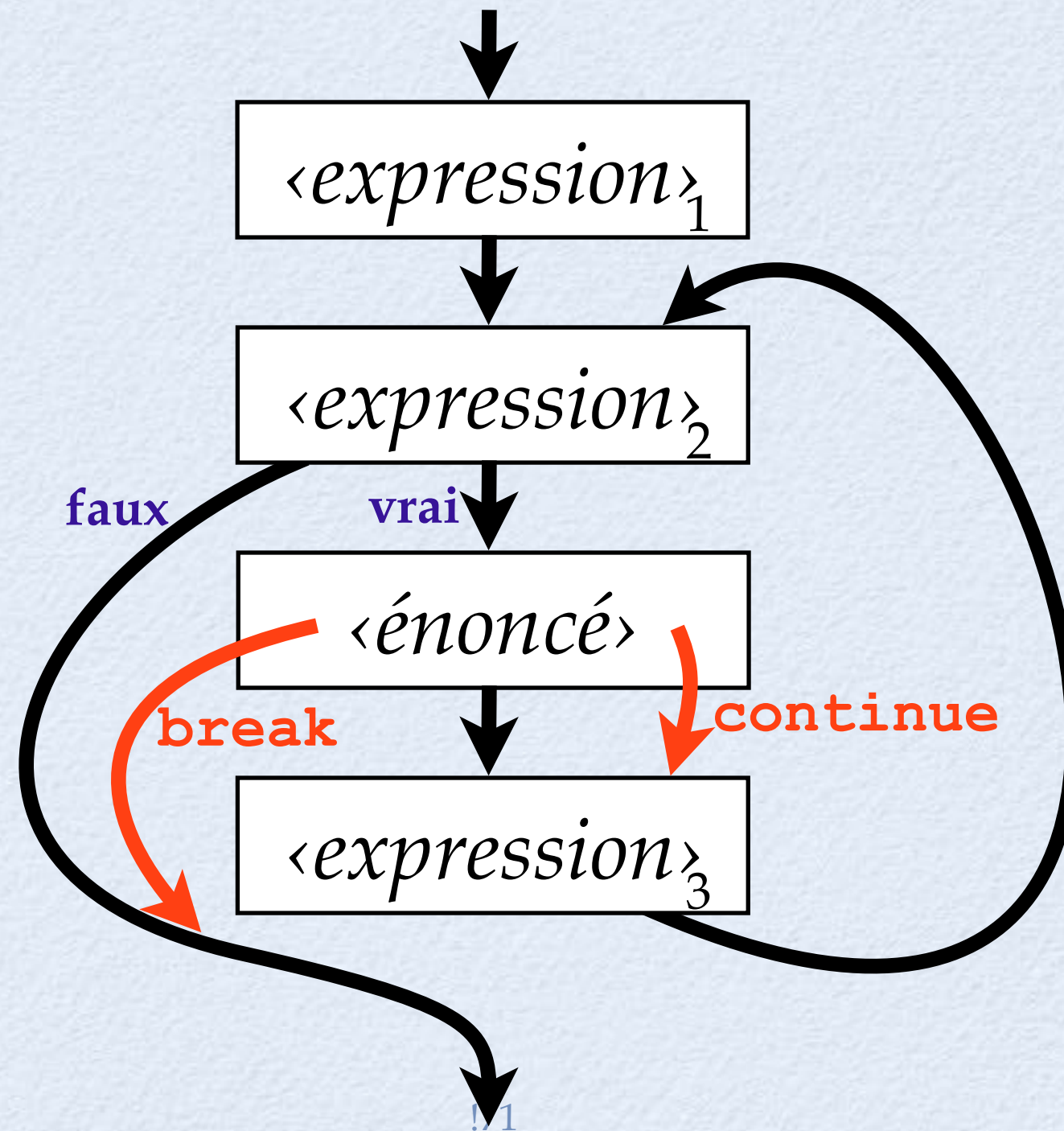


do *<énoncé>* **while** (*<expression>*) ;



Énoncé continue

for ($\langle expression \rangle_1$; $\langle expression \rangle_2$; $\langle expression \rangle_3$) $\langle \text{énoncé} \rangle$



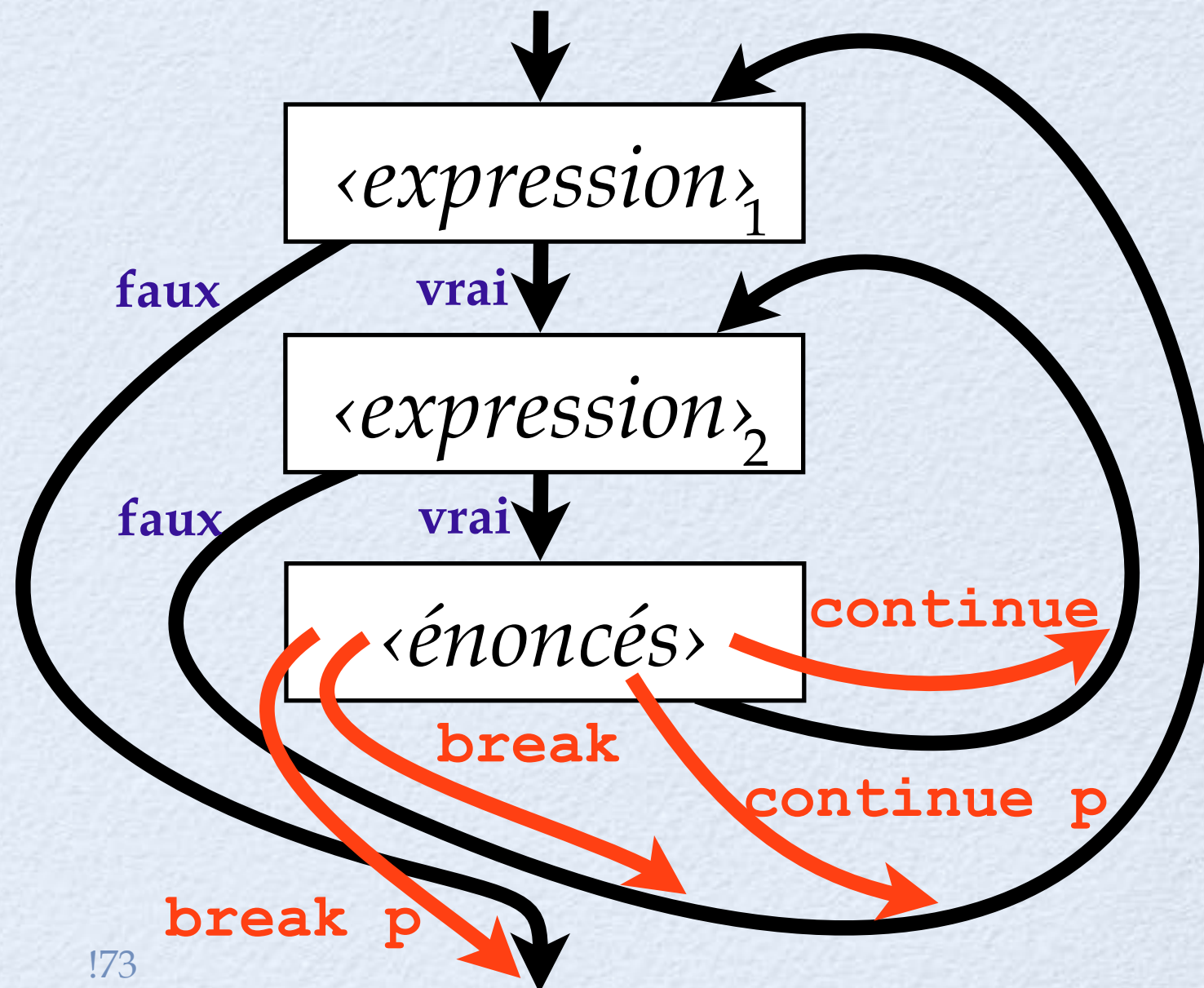
Boucles étiquetées

- Les énoncés **break** et **continue** dans des boucles visent par défaut la boucle la plus imbriquée qui les contient
- Si on vise à opérer sur une boucle plus englobante on peut assigner une **étiquette** à la boucle en question et la préciser dans le **break** et **continue**
- Syntaxe : *<id.>* : **while** (*<exp.>*) *<énoncé>*
- Syntaxe : *<id.>* : **do** *<énoncé>* **while** (*<exp.>*) ;
- Syntaxe : *<id.>* : **for** (*<exp.>* ; *<exp.>* ; *<exp.>*) *<énoncé>*
- Syntaxe : **break** *<id.>* ;
- Syntaxe : **continue** *<id.>* ;

Boucles étiquetées

- Diagramme de flux de contrôle d'une boucle **while** principale étiquetée «**p**» contenant une boucle **while** imbriquée contenant des énoncés

```
p: while (<expression1>) {  
    while (<expression2>) {  
        <énoncés>  
    }  
}
```



Exemple : nombres premiers

```
// Fichier: premiers7.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #7

candidats:
for (var n=2; n<=100; n+=1+(n>2)) { // 2, 3, 5, 7, ... 99

    var max = Math.sqrt(n);      // diviseur maximal

    for (var d=3; d<=max; d+=2) { // d=3,5,7,...
        if (n % d == 0) {        // as t-on trouvé un facteur?
            continue candidats; // oui, prochain candidat
        }
    }

    print(n);                    // n est premier
}

pause(); // ~ 1800 steps (25x174 plus vite que version #1)
```


Exemple : nombres premiers

```
// Fichier: premiers8.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #8

candidats:
for (var n=2; n<=100; n+=1+(n>2)) {           // 2, 3, 5, 7, ... 99

    if (n >= 9) {

        if (n%3==0 || n%5==0 || n%7==0) { // essayer 3, 5 et 7
            continue;
        }

        var max = Math.sqrt(n);             // diviseur maximal

        for (var d=11; d<=max; d+=2) {       // d=11,13,15,...
            if (n % d == 0) {                // as t-on trouvé un facteur?
                continue candidats;         // oui, prochain candidat
            }
        }

        print(n);                           // n est premier
    }

}

pause(); // ~ 1400 steps (32x plus vite que version #1)
```

Exemple : nombres premiers

```
// Fichier: premiers9.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #9

print(2);

candidats:
for (var n=3; n<=100; n+=2) {                                // 3, 5, 7, ... 99

    if (n >= 9) {

        if (n%3==0 || n%5==0 || n%7==0) { // essayer 3, 5 et 7
            continue;
        }

        var max = Math.sqrt(n); // diviseur maximal

        for (var d=11; d<=max; d+=2) { // d=11,13,15,...
            if (n % d == 0) { // as t-on trouvé un facteur?
                continue candidats; // oui, prochain candidat
            }
        }

        print(n); // n est premier
    }
}

pause(); // ~ 1200 steps (38x plus vite que version #1)
```


Exemple : nombres premiers

```
// Fichier: premiers10.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #10

print(2);

for (var n=3; n<=100; n+=2) { // 3, 5, 7, ... 99
    if (n < 9 || (n%3!=0 && n%5!=0 && n%7!=0)) {
        print(n);    // n est premier
    }
}

pause(); // ~ 1000 steps (45x plus vite que version #1)
```

Ce programme n'est pas facile à maintenir...
on ne peut pas simplement changer le 100 en
1000 pour imprimer les nombres premiers
entre 2 et 1000

Exemple : nombres premiers

```
// Fichier: premiers11.js

// Ce programme imprime les nombres premiers entre 2 et 100.

// Version #11

print(2);
print(3);
print(5);
print(7);
print(11);
print(13);
print(17);
print(19);
print(23);
print(29);
print(31);
print(37);
print(41);
print(43);
print(47);
print(53);
print(59);
print(61);
print(67);
print(71);
print(73);
print(79);
print(83);
print(89);
print(97);

pause(); // 77 steps (600x plus vite que version #1)
```

Ce n'est pas facile de se convaincre, ou "prouver", que ce programme imprime vraiment les nombres premiers entre 2 et 100... en as-t-on oublié un?

Exemple : nombres premiers

- Les dernières versions de ce programme illustrent que les divers critères de qualité d'un programme sont souvent en **opposition**
- Un programme conçu avec beaucoup de poids sur sa performance (*optimisé*) perd en clarté, en généralité et en maintenabilité
- Il est mieux de viser tout d'abord un programme **correct et clair**, facile à comprendre et maintenir, et de l'**optimiser seulement si sa performance laisse à désirer** et seulement dans les parties du programme qui bénéficient le plus d'une optimisation