Structures de base [bs]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 12 mai 2018

Table des matières

Java - Structures de base (Cours)

Mots-Clés Structures de base, Atomes du langage, Déclarations, Structure d'un algorithme, Interactions avec l'extérieur, Expressions algébriques, Affectation interne ■
Optionnel L'Algorithmique ■
Difficulté • ○ ○ (4 h) ■

Introduction
Ce module traite des **structures de base** : atomes du langage, variables, types et valeurs, déclarations, structure d'un programme, interactions avec l'extérieur, expressions algébriques, affectation interne, puis il détaille la trace d'exécution d'un programme. Le *Compléments* donne quelques précisions sur les variables, l'affectation et le transtypage.

Remarque
Pour certains, ce module apparaîtra comme une suite d'évidences, pour d'autres comme un foisonnement de choses nouvelles aux définitions contraignantes. Que ces derniers prennent le temps de faire les exercices. L'essentiel en algorithmique/programmation est dans la pratique.

1 Le langage

1.1 Le langage

Règles, Alphabet du langage, Éléments lexicaux

Tout **langage** de programmation possède :

- Des **règles de présentation** (aspect lexical).
- Des **règles d'écriture** (une syntaxe).
- Des **règles d'interprétation** (une sémantique).

L'**alphabet du langage** permet de construire les mots et les expressions. Comme toute phrase d'un langage courant, il est composé de mots appelés **lexèmes** ou **éléments lexicaux** (*tokens*).

Le langage Java

Langage de programmation *orienté objet* (OO) et *fortement typé*.

Alphabet du langage

C'est celui de la langue anglaise (majuscules et minuscules), les chiffres 0-9, quelques symboles spéciaux (ceux des mathématiques par exemple) et quelques symboles de ponctuation. Le codage étant en UTF8, les lettres accentuées sont autorisées dans les lexèmes.

La casse

Le langage est *case-sensitif* (sensible à la casse)[1] et les accentués sont autorisés. Ceci signifie que `cout`, `Cout` et `COUT` réfèrent trois mots différents et `coût` est licite.

1. Il fait la distinction entre minuscules/majuscules.

1.2 Les mots-clés

Mots-clés (*keywords*)
Dits aussi **mots réservés**, ils constituent le vocabulaire du langage. Ils forment l'ossature d'un programme en définissant la structure de ses données ou de ses instructions.

Propriété
Les mots-clés ont une **signification inchangeable et prédéfinie**, c.-à-d. qu'ils ne peuvent pas être redéfinis.

Liste des mots-clés
(1) C++ (2) Java

```
(1) break case catch char class const continue default do double else enum false float
    for if int long new private protected public return short static switch this throw
    true try void volatile while
(2) abstract assert boolean byte extends final finally goto implements import instanceof
    interface native package strictfp super synchronized throws transient
```

Environnements IDE
Les mots-clés y sont en gras (ou en couleurs).

1.3 Les identifiants

Identifiant

Dit aussi **identificateur**, c'est un **nom** choisi par le concepteur pour désigner les entités : variable, fonction, type, etc. Cette définition reste valable dans son domaine de validité, à condition qu'elle ne soit pas recouverte par une autre définition. Il **ne doit pas** correspondre à un mot-clé.

Identifiant

Séquence de lettres (A...Z, a...z), de chiffres (0...9), de lettres accentuées ou du caractère souligné (`_`, *underscore* en terme anglo-saxon) ou `$`, `?`, μ. Le premier caractère doit être différent d'un chiffre.

Le souligné est significatif

Ainsi `a`, `a_` et `_a` sont des identifiants différents.

1.4 Règles d'écriture d'un identifiant

Le but est de trouver un nom qui soit suffisamment **court** tout en restant **explicite** (parlant) et qui **ne prête pas à confusion**. Voici quelques règles et limitations traditionnelles. (Il est impossible de décrire ici toutes les particularités des différents langages. Reportez-vous aux règles spécifiques de votre langage.)

Identifiant

Suite de caractères alphanumériques **d'un seul tenant** (pas de caractères blancs) et **ne commençant jamais par un chiffre**. Ainsi `x1` est correct mais non `1x`.

Écriture d'un nom composé

On distingue principalement deux notations :

Notation « tiret bas » : Elle utilise le souligné entre les mots (par ex. `note_math`) et non pas le signe moins « - » qui est réservé à la soustraction dans la plupart des langages. Ainsi `note-math` serait interprété comme la soustraction des éléments `note` et `math`.

Notation « chameau » (*camelCase*) : Elle consiste à mettre une majuscule au début des mots, généralement à partir du deuxième (par ex. `noteMath` ou `valeurMaxOuMin`).

Indices et Exposants

Ils sont **proscrits** (par ex. x_1, z^6. On les notera `x1`, `z6`).

Mots-clés

Ceux des langages de programmation sont **interdits** (par ex. `for`, `if`, `return`, `while`... pour C/C++, Java, Python) et on déconseille d'utiliser les mots-clés du pseudo-code (tels que `entier`, `autre`, `pour`...).

Caractères exclus

La plupart des langages n'autorisent pas les caractères accentués (tels que à, ç, ê, ï, etc.), ni les caractères spéciaux (tels que `&`, `'`, `#`, etc.), ni les lettres des alphabets non latins (tel que Δ). Enfin certains font la distinction entre les minuscules et majuscules, d'autres non. En algorithmique, nous admettons les caractères accentués du français (par ex. `durée`, `intérêts`, etc.).

Cas du souligné

Les identifiants **commençant** par le soulignement (`_`) sont à éviter car ils sont soit réservés pour les différentes implémentations du langage (en C/C++) ou parce qu'ils ont une signification pour le langage (en Python).

Cas d'une unique lettre

Évitez la lettre `I` ou `i` et la lettre `l` qui se confondent avec le chiffre 1, ainsi que la lettre `O` ou `o` qui se confond avec le chiffre 0.

1.5 Séparateurs et espaces blancs

Espace blanc

Suite de un ou plusieurs caractères choisis parmi : espace, tabulation horizontale, tabulation verticale, changement de ligne ou changement de page.

Séparateurs

Comprend les opérateurs (`+`, `-`, `*`, `/`...) et les caractères de « ponctuation ».

Quelques séparateurs

	Symbole	Rôle
`{}`	accolades	délimitent les blocs
`()`	parenthèses	encadrent des expressions
`[]`	crochets	symboles du composant tableau
`.`	point	accès à un composant structurée
`,`	virgule	sépare les éléments d'une liste
`;`	point-virgule	sépare les instructions
`' '`	quotes	encadrent les caractères
`" "`	guillemets	encadrent les chaînes de caractères

1.6 Primitives

Bibliothèque

Ensemble de fonctionnalités ajoutées à un langage de programmation.
Chaque bibliothèque décrit un thème.

Pour les utiliser

```
import nomBiblio*; // importe tous les éléments
import nomBiblio.nomElem; // importe nomElem de nomBiblio
```

Primitives

Noms de fonctions (`abs`, `log`, `sin`...), d'opérateurs (`div`, `mod`...) ou de traitement (`afficher`, `saisir`...). Elles acceptent un ou plusieurs paramètres et jouent le même rôle syntaxique qu'un identifiant.

Appel d'une primitive

```
P(x,...); // procédure
r = F(x,...) // fonction
```

Explication

Appelle (on dit aussi **invoque**) la procédure `P` ou la fonction `F` avec les arguments `x`... Dans le cas de fonction, la valeur retournée peut être utilisée en tant que macro-expression.

2 Variables, types et valeurs

2.1 Variable, type et valeur

Variable
Élément informatique qu'un programme peut manipuler.
Décrite par :

- Un **identifiant** unique qui la désigne.
- Un **type** qui définit de quel « genre » est l'information associée.
- Une **valeur** qui doit respecter le type.

Propriété d'une variable
Une **variable conserve son nom et son type** mais peut changer de valeur.

Type d'une donnée
Appelé aussi **domaine de définition**, il spécifie la taille en mémoire occupée par la donnée, quels traitements (opérations) elle peut (et ne peut pas) subir ainsi que l'intervalle de valeurs (minimale et maximale) qu'elle peut prendre.

Utilité du type des données
Il est :

- Utile au concepteur d'algorithmes pour lui permettre de structurer ses idées.
- Très utile au programmeur car il permet de détecter des erreurs potentielles.
- Indispensable au compilateur car les différents types ne sont pas tous représentés de la même façon. La représentation d'un même type peut même varier d'un processeur à l'autre.

2.2 Types intégrés

Types intégrés (*builtins*)

Dits aussi **types de base**, **types fondamentaux** ou encore **types primitifs**, ils correspondent aux données qui peuvent être traitées directement par le langage.

Types intégrés

Domaine	Algorithmique	Équivalent Java
$\mathbb{Z}$	`Entier`	`int`
$\mathbb{R}$	`Réel`	`double`
$\mathbb{B}$	`Booléen`	`boolean`
$\mathbb{A}$	`Caractère`	`char`
$\mathbb{T}$	`Chaîne`	`String`

2.3 Littéraux

Littéral

Valeur explicite au sein d'un programme.
Il possède *un type* déterminé par sa valeur (entier, réel, caractère...).

Notation scientifique d'un réel

Expression « m e p » qui signifie $m \cdot 10^p$ (où `m` est une suite de chiffres décimaux et `p` un entier). Ainsi, $12.4e-5 \equiv 12.4 \cdot 10^{-5} \equiv 0.000124$. Elle permet d'exprimer de très petits nombres (ex : 2.5e-201) et de très grands nombres (ex : 5e156).

Point décimal

Un réel se distingue d'un entier par l'ajout d'un point (.) à la fin.
Pour la lisibilité, préférez .0 (ex. 5.0 (réel) mais 5 (entier)).

Littéraux

- `Entier` : Suite de chiffres éventuellement préfixé par un signe (+ ou −).
- `Réel` : S'écrit en notation décimale ou en notation scientifique.
- `Booléen` : Ils identifient le `Vrai` (mot-clé `true`) et le `Faux` (mot-clé `false`).
- `Caractère` : Se place entre quotes (').
- `Chaîne` : Se place entre guillemets (").

2.4 Exemples : Variables, types et valeurs

Exemples de types

Pour représenter ou décrire :

- Longueur, largeur et surface d'un rectangle : on prendra un réel.
- Nom d'une personne : une chaîne.
- Age d'une personne : un entier.
- Si un étudiant est redoublant ou pas : un booléen.
- Un mois : on préfèrera un entier donnant le numéro du mois (par ex : 3 pour le mois de mars) plutôt qu'une chaîne (par ex : `"mars"`) car les manipulations et les calculs seront plus simples.

Exemples de littéraux

(Avec leur équivalent mathématique)

Type	Équivalent Mathématique	Littéraux
`Entier`	$\mathbb{Z}$ (entiers)	`3,-5,-100,122,0`
`Réel`	$\mathbb{R}$ (réels)	`3.1425,-1.12e-10,0.0`
`Booléen`	$\mathbb{B}$ (booléens)	`false,true`
`Caractère`	$\mathbb{A}$ (alphanumériques)	`'a','1','A','&','#'`
`Chaîne`	$\mathbb{T}$ (textes)	`"c'est du texte","a",""`

3 Déclarations

3.1 Déclaration de variables

Déclaration de variables

Consiste à associer un type de données à une ou un groupe de variables. Toute variable doit impérativement avoir été déclarée avant de pouvoir figurer dans une instruction exécutable.

Déclaration de variables

```
TypeVar nomVar;
TypeVar nomVar1, nomVar2, ...;
```

Explication

Déclare des variables d'identifiants `nomVarI` (le nom) de type `TypeVar`.

3.2 Initialisation d'une variable

Initialisation d'une variable
Consiste à définir la valeur d'une variable lors de sa déclaration.

Initialisation d'une variable

```
TypeVar nomVar = valeur;
TypeVar nomVar(valeur);
```

Explication
Déclare une variable d'identifiant `nomVar` (le nom) de type `TypeVar` et l'initialise à la `valeur` (littéral ou expression) indiquée.

Déclaration v.s. Initialisation
Dans les langages typés :

- Déclaration = Création d'une variable.
- Initialisation = Association d'une valeur à la variable créée.

3.3 Définition de constante

Constante

Littéral à lequel est associé un **identifiant** (par convention, écrit en MAJUSCULES) afin de rendre plus lisible et simplifier la maintenance d'un programme. C'est donc une information pour laquelle nom, type et valeur sont figés.

Définition de constante

```java
final static TypeConst nomConst = expression; // notation impérative
final static TypeConst nomConst(expression); // notation objet
```

Explication

Définit la constante d'identifiant `nomConst` de type `TypeConst` et lui affecte une valeur (littéral ou expression) spécifiée.

Valeur immuable fixée à la déclaration

Toute tentative de modification est rejetée par tout compilateur qui signalera une erreur (ou un avertissement).

3.4 Exemples : Déclarations

Exemple : Déclarations de variables

```
int j,k,n;
double racine1,racine2;
boolean flag;
char c;
String texte;
```

Ici `j`,`k`,`n` sont déclarées comme variables entières, `racine1`,`racine2` comme variables réelles, `flag` comme variable booléenne, `c` comme variable caractère et `texte` comme variable chaîne de caractères. On aurait aussi bien pu écrire ces déclarations de la façon suivante :

```
int j;
int k;
int n;
double racine1;
double racine2;
boolean flag;
char c;
String texte;
```

Cette forme peut s'avérer pratique, notamment si l'on veut commenter les déclarations utilisées, variable par variable.

Exemple : Initialisations de variables

```
final double PI(3.14159);
final String MESSAGE("Appuyez sur une touche pour continuer");
```

`PI` désigne une constante réelle (approximation de π) et `MESSAGE` une constante chaîne de caractères du texte entre les guillemets.

4 Structure générale

4.1 Commentaire

Commentaire (narratif)
Texte qui n'est **ni lu, ni exécuté** par la machine. Il est essentiel pour rendre plus lisible et surtout plus compréhensible un programme par un être humain.

Commentaire orienté ligne

```
... // rend le reste de la ligne non-exécutable (hérité du C++)
```

Commentaire orienté bloc

```
/*
rend le code entouré non exécutable...
(hérité du C)
*/
```

4.2 Instruction composée

Instruction

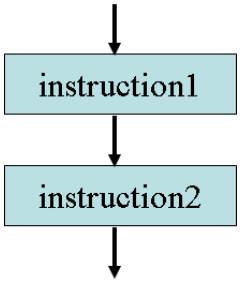
Ordre donné à l'ordinateur qui a pour effet de changer l'état de la mémoire ou le déroulement du programme ou bien de communiquer avec les unités périphériques (clavier, écran, imprimante, etc.).

Instruction composée ou Bloc

Regroupement syntaxique de 0, 1 ou plusieurs instructions (et déclarations) comme une unique instruction.

Séquentialité

Les algorithmes et programmes présentés sont exclusivement séquentiels : l'`instruction2` ne sera traité qu'une fois l'exécution de l'`instruction1` achevée.



Bloc

```
{
  instruction1;
  instruction2;
  ...
}
```

Conventions usuelles

Savoir présenter un programme, c'est montrer que l'on a compris son exécution.

- Chaque ligne comporte une seule instruction.

  Java Le point-virgule « ; » est le terminateur d'instructions.

- Les indentations sont nécessaires à sa bonne lisibilité. Ainsi :

  Java Alignez les accolades de début `{` et fin `}` de bloc l'une sous l'autre.

4.3 Structure générale

Structure générale

```
import des_trucs_utiles;
public class nomAlgo
{
  déclaration_des_objets_globaux
  déclarations_et_définitions_de_fonctions_utiles
  public static void main(String[] args)
  {
    corps_du_programme
  }
}
```

Explication

Un programme est constitué par :

- Un **en-tête** qui demande à l'interpréteur d'inclure les fichiers indiqués.
- Un **corps** lequel contient une classe (ici `nomAlgo`) qui porte le nom du fichier texte qui contient le programme source. Cette classe contient au moins la procédure particulière `main()` (« principale ») nécessaire pour produire du code machine *exécutable*.

Le programme commence son exécution sur l'accolade ouvrante de la procédure `main`, se déroule séquentiellement et se termine sur son accolade fermante.

Java : La procédure main

Quelques règles :

- Respectez la casse (m minuscule) ainsi que les parenthèses.
- Chaque programme possède une procédure `main()`.
- En l'absence de procédure `main()`, le programme ne démarre pas.

Conseil

On veillera à ce qu'un programme tienne sur une vingtaine de lignes (donc, en pratique, sur un écran de 40 x 80 caractères ou une page). Ceci implique que, si votre programme devait être plus long, il faudra le découper, comme nous le verrons plus loin.

4.4 Exemple : Le plus petit algorithme

Le plus petit programme

```java
class PGVide1 {

public static void main(String[] args)
{
}

}
```

Explication

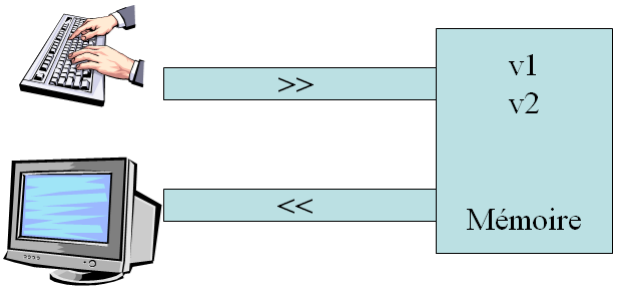
C'est le plus petit programme qui ne fait rien.

5 Interactions avec l'extérieur

5.1 Interactions avec l'extérieur

Entrées/sorties clavier/écran

Ce sont les **interactions avec l'extérieur** les plus simples et « naturelles ». Afficher de l'information ou des résultats à destination de l'utilisateur est capital pour la convivialité des programmes. Quant à la saisie, elle permet à l'utilisateur d'entrer les données qui seront stockées dans des variables en vue d'un traitement.



Pour utiliser la saisie

```
import java.util.Scanner;
```

Terminologie

La saisie consiste à affecter une valeur *externe* à une variable : on parle aussi d'**affectation externe**.

5.2 Saisie de données

Saisie de données

```
Scanner input = new Scanner(System.in);
nomVarI = input.next();       // lecture d'une chaîne
nomVarI = input.nextLine();   // lecture d'une ligne
nomVarI = input.nextInt();    // lecture d'un entier
nomVarI = input.nextDouble(); // lecture d'un réel
nomVarI = input.nextChar();   // lecture d'un caractère <=> input.next().charAt(0);
```

Explication

Ordonne à la machine de lire des valeurs `valI` depuis le clavier et de les stocker dans les variables `nomVarI` (qui doivent exister c.-à-d. déclarées).

Remarque

Par défaut, ce qui est tapé au clavier est envoyé à l'écran et temporairement placé dans un tampon pour permettre la correction d'erreurs de frappe. On peut donc se servir des touches [←] et [Suppr] pour effacer un caractère erroné ainsi que les flèches [<] et [>] pour se déplacer dans le texte.

Java : Saisie d'un réel

En France, un nombre réel s'écrit avec une virgule alors qu'aux États-Unis, on utilise le point. En utilisant la classe `Scanner` sur un système d'exploitation réglé en zone française, il faut saisir les valeurs réelles avec une virgule. Pour utiliser la notation américaine, il convient de modifier la localité (bibliothèque `java.util.Locale`) grâce à l'instruction :

```
cin.useLocale(Locale.US); // ou bien: cin.useLocale(Locale.ENGLISH)
```

L'instruction suivante permet de revenir à la saisie des valeurs réelles avec une virgule :

```
cin.useLocale(Locale.FRENCH);
```

5.3 Affichage de résultats

Affichage de résultats

```
System.out.print(expr1+expr2+...+exprN);   // SANS retour de ligne
System.out.println(expr1+expr2+...+exprN); // AVEC retour de ligne
```

Explication

Ordonne à la machine d'afficher les **valeurs** des expressions `exprI`. Par défaut, elles ne sont pas séparées par des espaces. Ajoutez le(s) délimiteur(s) si nécessaire.

Remarque

Le « ln » de `print` signifie *line-feed* (retour-à-la-ligne).

Java

Avec les versions 6 et 7, la classe `System` est dotée d'une méthode `console` qui traite correctement tous les caractères (y compris les accents).

Variables devant être initialisées

On ne peut *afficher* que des expressions dont les variables qui la composent ont été affectées préalablement.

5.4 Exemples : Communication des données/résultats

Exemple : Afficher un mot

```java
class PGAfficher1 {

public static void main(String[] args)
{
  System.out.println("Bonjour");
}

}
```

Explication

Ce programme affiche le mot « Bonjour ».

Exemple : Saisir un nombre

```java
import java.util.Scanner;

class PGSaisir1 {

public static void main(String[] args)
{
  Scanner input = new Scanner(System.in);
  double x;
  System.out.print("Tapez un nombre reel au clavier? ");
  x = input.nextDouble();
  System.out.println("==> Vous avez tape " + x + ", felicitations");
}

}
```

Explication

Ce programme permet à l'utilisateur de saisir un réel par `input.nextDouble()` et de le mémoriser dans la variable `x` (type de données `double` : long-flottant) puis de le ré-afficher par `System.out.println`.

Java : Caractères accentués

L'affichage de caractères accentués et, plus généralement, de tout caractère spécial reste problématique, surtout si le programme doit fonctionner sur des ordinateurs différents. En effet, les codes de ces caractères font partie des extensions qui diffèrent suivant les pays ou les environnements de travail. Par conséquent les caractères spéciaux et les caractères accentués ne sont pas traités de la même façon d'un environnement à l'autre. Nous vous recommandons donc d'éviter les accents dans vos programmes.

6 Expressions arithmétiques

6.1 Qu'est-ce qu'une expression ?

Expression, opérandes, opérateurs

Éventuellement accompagnés de parenthèses, une **expression** est une séquence « bien formée » (au sens de la syntaxe) d'**opérandes** (littéraux, variables ou expressions) et d'**opérateurs** destinée à l'évaluation.

Exemples

- Dans l'expression `1+2`, les valeurs 1 et 2 sont des opérandes et `+` est un opérateur.
- Dans `(a+b)*c`, l'opérande de gauche de l'opérateur `*` est `(a+b)`.
- `(2*(13-j)/(1+4))` est une expression numérique BF (bien formée).
- `(3+(5*(-2)` est non BF (parenthésage incorrect).
- `3 (5) * )-2(` est non BF (opérateur et parenthésage incorrect).

Pluralité d'un opérateur

Un opérateur :

- **unaire** (ou monadique) porte sur un opérande (ex : le `-` dans l'expression `-a`).
- **binaire** (ou dyadique) sur deux opérandes (le plus fréquent).
- **ternaire** (ou triadique) sur trois opérandes.

Un opérateur à n opérandes est dit **n-aire**.

6.2 Opérateurs arithmétiques

Opérateurs arithmétiques

Dits aussi **opérateurs algébriques**, ils agissent sur des opérandes de type numérique.

Opérateurs arithmétiques

Opérateur Mathématique	Signification	Équivalent Java
+	(unaire) valeur	`+a`
-	(unaire) opposé	`-a`
+	addition	`a + b`
-	soustraction	`a - b`
*	multiplication	`a * b`
/	division décimale	`a / b`
div	division entière	`a / b`
mod	modulo (reste de la division entière)	`a % b`

Java : Élévation à la puissance

Il est nécessaire de faire appel :

- Soit à des produits successifs pour des puissances entières pas trop grandes (par exemple, on calculera x^3 comme `x*x*x`).
- Soit à la fonction `Math.pow` de la bibliothèque standard.

Java : Que vaut a / b ?

La division s'effectue sur :

- $\mathbb{N}$: si a **et** b sont entiers (division entière)
- $\mathbb{R}$: si a **ou** b sont réels (division réelle)

Division euclidienne, cas des négatifs

Il n'y a pas unicité du quotient et du reste lorsque le dividende ou le diviseur sont négatifs. Si a et b sont deux entiers relatifs dont l'un au moins est négatif, il y a plusieurs couples (q, r) tels que $a = b \times q + r$ avec $|r| < |b|$. Par exemple, si $a = -17$ et $b = 5$ alors $(q = -3, r = -2)$ ou $(q = -4, r = 3)$ sont deux solutions possibles. Habituellement, q et r sont choisis comme le quotient et le reste de la division entière de $|a|$ et $|b|$ affectés du signe approprié (celui permettant de vérifier $a = b \times q + r$ avec $|r| < |b|$). Dans l'exemple ci-avant, la solution retenue serait $(q = -3, r = -2)$. Par contre, la norme impose que la valeur de $(a \text{ div } b) * b + a \bmod b$ soit égale à la valeur de a.

Pas d'overflow sur les entiers

Si le résultat d'une opération dépasse la capacité d'un entier, le résultat est un entier négatif ou inférieur : il n'y a pas d'erreur de **dépassement de capacité**.

Division par zéro
Tout emploi de la division devra être accompagné d'une réflexion sur la valeur du dénominateur, une division par 0 entraînant toujours l'arrêt d'un programme.

6.3 Évaluation des expressions

Évaluation d'une expression

Elle conduit à la **valeur littérale** correspondant au **résultat** de l'expression lequel s'appuie sur un *système de priorités* relatives entre les opérateurs et des *règles d'associativité*. La **priorité** (ou **préséance**) fait référence à l'ordre d'application des opérateurs et l'**associativité** peut être exprimée de droite à gauche ou de gauche à droite. Les parenthèses permettent de modifier ou de souligner la priorité.

Ordre de priorité des opérateurs arithmétiques

Comme en mathématique :

1. Les opérateurs unaires (`+`, `-`) (priorité la plus élevée)
2. L'opérateur d'exponentiation (`^`) (s'il existe)
3. Les opérateurs multiplicatifs (`*`, `/`, `div`, `mod`)
4. Les opérateurs additifs (`+`, `-`) (priorité la plus basse)

La règle d'associativité s'applique en cas d'ambiguïté entre opérateurs du même ordre de priorité.

Opérateur mod

Préférez de parenthéser vos expressions (ne serait-ce que pour la lisibilité !).
Pour des entiers :

- $a * b \mod c \equiv (a * b) \mod c$
- $a \mod b * c \equiv (a \mod b) * c$
- $a + b * c \mod d \equiv a + ((b * c) \mod d)$

Exemple : Écrivez $(a * b) \mod c$ plutôt que $a * b \mod c$.

Exponentiations

Deux exponentiations successives sont évaluées en commençant par celle de droite.
Ainsi `a^b^c` signifie a^{b^c} mais `a/b/c` représente $\frac{a}{b \cdot c}$.

6.4 Exemples : Opérateurs arithmétiques

Exemple : Cas de deux entiers

Soient `a` et `b` deux variables entières valant respectivement 10 et 3. Le tableau illustre les opérations arithmétiques qui peuvent leur être appliqués :

Opération	Expression	Valeur
addition	`a + b`	13
soustraction	`a - b`	7
multiplication	`a * b`	30
division entière	`a / b`	3
modulo	`a % b`	1

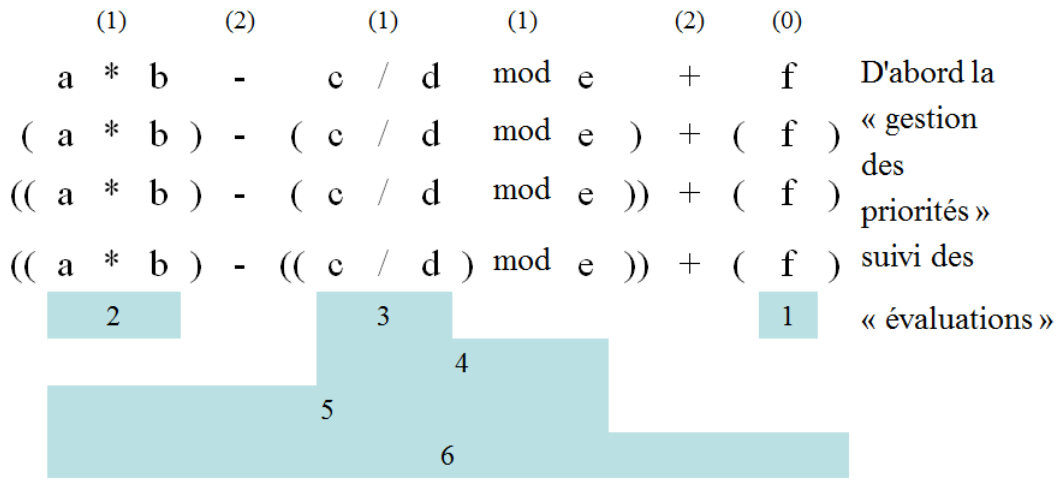
Le quotient est tronqué puisqu'il s'agit de deux opérandes entiers. De même, le reste de la division entière de ces deux nombres est un entier comme le montre la dernière ligne.

Exemple : Cas de deux réels

Soient `a` et `b` deux variables réelles valant respectivement 12.5 et 2.0. Le tableau illustre les opérations arithmétiques qui peuvent leur être appliqués :

Opération	Expression	Valeur
addition	`a + b`	14.5
soustraction	`a - b`	10.5
multiplication	`a * b`	25.0
division	`a / b`	6.25

Exemple : Priorité entre opérateurs arithmétiques



6.5 Cohérence de type numérique

Règle de promotion

Pour qu'une opération numérique binaire (+, -, *, /) puisse s'effectuer, il faut que ses deux opérandes soient du **même type** ou d'un **type compatible**. Lorsque ce n'est pas le cas, préalablement à l'opération, il y a **promotion** de l'opérande de type le plus faible vers le plus grand.

Exemple

Si le réel `r` vaut 1.0 et l'entier `j` vaut 2, l'expression `r/j` se calcule ainsi :

1. Promotion de l'entier 2 en réel 2.0
2. Évaluation du calcul de 1.0/2.0 qui donne le réel 0.5

Conseil : Écriture des littéraux réels

Écrivez explicitement les littéraux réels avec un point pour les distinguer des littéraux entiers. De plus ceci évitera les écritures du style `(1/2)*a` (avec `a` de type réel) où l'on obtient la valeur 0 au lieu du résultat `a/2`. En effet dans cette écriture, la machine réalise la division entière de 1 par 2 ce qui donne le quotient entier 0. Pour obtenir l'évaluation décimale, il faut écrire `1./2*a` (par exemple) ou simplement `a/2` (car `a` est réel).

6.6 Fonctions mathématiques

Fonctions mathématiques
Elles agissent sur des paramètres à valeurs réelles et donnent un résultat réel.

Importation implicite

```
import java.lang.Math;
```

Quelques Fonctions mathématiques

Fonctions Mathématiques	Signification	Équivalent Java
$\text{acos}(x)$	Cosinus inverse (radians)	`Math.acos(x)`
$\text{asin}(x)$	Sinus inverse (radians)	`Math.asin(x)`
$\text{atan}(x)$	Tangente inverse (radians)	`Math.atan(x)`
$\lceil x \rceil$	Réel de l'entier supérieur	`Math.ceil(x)`
$\cos(x)$	Cosinus de x (radians)	`Math.cos(x)`
e^x	Exponentielle de x (base e)	`Math.exp(x)`
$\lvert x \rvert$	Valeur Absolue de x	`Math.abs(x)`
$\lfloor x \rfloor$	Réel de l'entier inférieur	`Math.floor(x)`
$\log(x)$	Logarithme naturel (base e)	`Math.log(x)`
x^y	Puissance	`Math.pow(x,y)`
$\sin(x)$	Sinus de x (radians)	`Math.sin(x)`
$\sqrt{x}$	Racine carrée	`Math.sqrt(x)`
$\tan(x)$	Tangente de x (radians)	`Math.tan(x)`

Racine carrée
Attention de ne l'utiliser qu'avec un radicant positif.

7 Affectation interne

7.1 Affectation interne

Affectation interne
Opération qui **fixe une valeur** à une variable.

Affectation interne

```
nomVar = expression;
```

Explication
Place la valeur de l'`expression` dans la zone mémoire de la variable de nom `nomVar`. En algorithmique, le symbole <- (qui se lit « devient ») indique le sens du mouvement : de l'expression située à droite **vers** la variable à gauche.

Rappel
Toutes les variables apparaissant dans l'`expression` doivent avoir été affectés préalablement. Le contraire provoquerait un arrêt de l'algorithme.

Conversions implicites
Il est de règle que le résultat de l'expression à droite du signe d'affectation soit de même type que la variable à sa gauche.

7.2 Exemples : Affectation interne

Exemples : Affectations correctes

```
a = 3;
x = y;
delta = Math.sqrt((b*b - 4*a*c)/(2.0*a));
test = true;
```

La première instruction affecte la valeur 3 à la variable `a`, la seconde la valeur de `y` à la variable `x`, la troisième la valeur de l'expression arithmétique à la variable `delta` et la dernière la valeur logique à la variable `test`.

Exemples : Affectations incorrectes

```
somme + 1 = 3;
somme = 3n;
```

La première est invalide car `somme+1` n'est pas une variable et la deuxième car `3n` n'est ni un nom correct de variable ni une expression correcte.

7.3 Exemple : Affectations et conversions

Ce programme emploie des fonctions mathématiques. Il montre l'affectation et les conversions implicites de type dans les expressions.

Programme @[pgaffect1.java]

```java
class PGAffect1 {

public static void main(String[] args)
{
  double x, y;
  x = 6.0;
  y = x * -3;
  System.out.println("En A : y vaut " + y + " et x vaut " + x);
  x = Math.pow(x - 4,2);
  System.out.println("En B : y vaut " + y + " et x vaut " + x);
  x = Math.sin(Math.sqrt(x * -y + 9));
  System.out.println("En C : y vaut " + y + " et x vaut " + x);
}

}
```

Explication

Il déclare des variables réelles `x` et `y` et est constitué de quatre instructions d'affectation.

- La première affecte la valeur 6 à la variable `x`. A l'issue de cette instruction, `x` vaut le réel 6.0 (conversion implicite de l'entier 6 en réel).
- La deuxième est l'affectation d'une expression algébrique à une variable réelle. L'expression est d'abord reconnue comme un produit à effectuer portant sur une valeur réelle et une valeur entière. Comme les types des valeurs sont différents, la valeur -3 est d'abord convertie en 3.0 puis multipliée avec la valeur de la variable `x` (définie) qui vaut 6.0. Ainsi, la multiplication a ses deux opérandes réels, elle est effectuée et la valeur réelle -18.0 est affectée à la variable `y`.
- La troisième évalue d'abord le paramètre `x-4` de la fonction qui vaut `6.0-4=2.0` puis appelle la fonction `carré` (`pow`) qui donne la valeur 4.0 laquelle modifie la valeur de la variable `x`. L'ancienne valeur de `x`, qui était 6.0, est perdue. Le déroulement séquentiel fait naturellement « oublier » les instructions effectuées en ne conservant que les valeurs courantes des variables.
- La quatrième suit le même principe : elle évalue d'abord `x*-y+9` qui vaut `4.0*18.0+9=81.0` laquelle est transmise à la fonction `racinecarrée` (`sqrt`) qui donne 9.0, laquelle est transmise à la fonction `sin` qui donne 0.4121184852.

Résultat d'exécution

```
En A : y vaut -18.0 et x vaut 6.0
En B : y vaut -18.0 et x vaut 4.0
En C : y vaut -18.0 et x vaut 0.4121184852417566
```

8 Trace d'exécution d'un programme

8.1 Trace d'exécution

Faire la trace d'exécution d'un programme consiste à suivre pas-à-pas (instruction par instruction) sur une feuille de papier, le contenu des variables et les valeurs entrées et sorties. C'est un excellent moyen pour :

- Comprendre sa fonctionnalité.
- Tester sur quelques exemples que les opérations sont programmées correctement. Notez qu'une trace s'effectue sur un programme de syntaxe correcte.

Préparation de la trace

La trace d'exécution, exercice facile dans son principe, demande beaucoup de rigueur et d'attention. Pour réussir une trace d'exécution :

1. Prenez une feuille de papier et créez un tableau.
2. Réservez une colonne pour les valeurs en entrée. Placez-y les valeurs dans l'ordre dans lequel l'exécution de l'algorithme prévoit de les faire entrer. Cette colonne sera appelée *colonne-des-valeurs-en-entrée*.
3. Réservez une colonne pour les valeurs en sortie. Les valeurs sorties lors de l'exécution par l'instruction d'affichage seront inscrites dans l'ordre dans lequel l'exécution les fait sortir. Cette colonne sera appelée *colonne-des-valeurs-en-sortie*.
4. Réservez une colonne pour chaque variable déclarée. Intitulez-la par son nom. Nous appellerons une telle colonne, une *colonne-de-variable*.
5. Exécutez votre programme, instruction par instruction, en commençant par la première instruction.

Trace d'exécution d'une instruction saisir

La trace d'exécution d'une instruction `saisir(XXX)`, où `XXX` est le nom d'une variable, se passe en deux temps :

1. Dans la *colonne-des-valeurs-en-entrée*, barrez d'un trait horizontal la prochaine valeur à entrer. Ce faisant, les valeurs inscrites sur la *ligne-des-valeurs-en-entrée* sont biffées au fur et à mesure des exécutions des instructions `saisir`. Lors de la première exécution d'une instruction `saisir`, biffez la première valeur, lors de la deuxième exécution d'une instruction `saisir`, biffez la deuxième valeur...
2. Inscrivez sur la *colonne-de-variable* de `XXX` la valeur entrée (celle que vous venez de biffer dans la *colonne-des-valeurs-en-entrée*). Si une valeur se trouve déjà inscrite sur la colonne-de-variable de `XXX`, biffez cette valeur d'un trait horizontal. Chaque fois que le contenu d'une variable est modifié par une instruction, procédez de cette façon : biffure de l'ancienne valeur et inscription de la nouvelle valeur. Ainsi pour connaître la valeur mémorisée dans une variable, repérez la seule valeur non biffée de sa *colonne-de-variable*.

Trace d'exécution d'une instruction afficher

La trace d'exécution d'une instruction `afficher(XXX)`, où `XXX` est le nom d'une variable ou une valeur, se réduit à recopier sur la *colonne-des-valeurs-en-sortie*, la valeur mémorisée dans `XXX` ou directement la valeur. C'est-à-dire avec notre système de biffure, la seule valeur non barrée de la *colonne-de-variable* `XXX`. **Attention !** La valeur mémorisée par la variable `XXX` ne change pas, il ne faut surtout pas la biffer.

Trace d'exécution d'une instruction d'affectation

La trace d'exécution d'une instruction `XXX<--expr`, où `expr` dénote une expression, conduit à effectuer dans l'ordre :

1. L'évaluation de l'expression `expr` à droite du symbole d'affectation (`<--`). C'est le calcul en remplaçant chacune des variables par les valeurs qu'elles contiennent. Pour connaître les valeurs contenues dans les variables, reportez-vous aux *colonne-de-variable* de chacune des variables.
2. Remplacez le contenu de la variable `XXX` indiquée à gauche du symbole `<--` par la valeur trouvée au calcul en (1). Plus précisément, biffez la valeur se trouvant éventuellement sur la *colonne-de-variable* de `XXX` et inscrivez-y, la valeur trouvée.

8.2 Exemple complet de programme / pgdiffsecs

Énoncé

Calculer le nombre de secondes écoulés entre deux instants de la journée. En entrée : deux instants sous la forme heures, minutes, secondes. Supposer que le deuxième est le plus grand. En sortie : le nombre de secondes écoulées entre les deux instants. Exemple : 17, 23, 41, 21, 19, 14 comme valeur en entrée donnera après exécution 62621, 76754, 14133 comme valeur en sortie.

Algorithme

L'algorithme que nous proposons est :

- Saisir le premier instant (dans trois entiers `hr`, `mm`, `ss`).
- Calculer le nombre de secondes écoulées depuis minuit de cet instant dans `nsecs` comme suit :
  - Multiplier l'heure `hr` par le nombre de `SECONDES_PAR_HEURE`.
  - Multipliez les minutes `mm` par le nombre de `SECONDES_PAR_MINUTE`.
  - Ajoutez les secondes `ss` à la somme des résultats des deux multiplications ci-dessus.
- Afficher le résultat de l'addition `nsecs`.
- Saisir le deuxième instant (dans `hr`, `mm`, `ss`).
- Calculer le nombre de secondes écoulées depuis minuit de cet instant dans `nsecs2` comme ci-avant.
- Afficher le résultat de l'addition `nsecs2`.
- Faire la différence entre `nsecs2` et `nsecs`.
- Afficher le résultat de la soustraction.

Programme @[pgdiffsecs.java]

```java
import java.util.Scanner;

class PGDiffsecs {
final static int SECONDES_PAR_MINUTE = 60;
final static int SECONDES_PAR_HEURE = 3600;

public static void main(String[] args)
{
  Scanner input = new Scanner(System.in);
  int hr, mm, ss;
  System.out.print("hr mm ss?");
  hr = input.nextInt();
  mm = input.nextInt();
  ss = input.nextInt();
  int nsecs = hr * SECONDES_PAR_HEURE + mm * SECONDES_PAR_MINUTE + ss;
  System.out.println(nsecs);
  System.out.print("hr mm ss?");
  hr = input.nextInt();
  mm = input.nextInt();
  ss = input.nextInt();
```

```
  int nsecs2 = hr * SECONDES_PAR_HEURE + mm * SECONDES_PAR_MINUTE + ss;
  System.out.println(nsecs2);
  nsecs = nsecs2 - nsecs;
  System.out.println(nsecs);
}

}
```

Explication

Le programme est une traduction de l'algorithme. Il entre six nombres, les trois premiers nombres représentant le premier instant de la journée sous la forme heures, minutes et secondes. Les trois suivants représentent un autre instant, également sous la forme heures, minutes et secondes. Il sort trois nombres représentant respectivement le nombre de secondes écoulées entre minuit et le premier instant, le nombre de secondes écoulées entre minuit et le second instant et la différence en secondes entre ces deux instants.

Trace d'exécution

Voici la trace d'exécution avec pour valeur en entrée : 17, 23, 41, 21, 17, 14.

SECONDES_PAR_MINUTE : 60
SECONDES_PAR_HEURE : 3600
Valeurs en entrée : $\underline{17}^{M1}$ $\underline{23}^{M2}$ $\underline{41}^{M3}$ $\underline{21}^{M6}$ $\underline{17}^{M7}$ $\underline{14}^{M8}$

	hr	mm	ss	nsecs	nsecs2	Valeurs en sortie
Début	?	?	?	?	?	
M1	17					
M2		23				
M3			41			
M4				62621		
M5						62621
M6	24					
M7		17				
M8			14			
M9					76634	
M10						76634
M11				14013		
M12						14013

8.3 Erreur à l'exécution

Il peut arriver que l'exécution d'un programme ne puisse se faire jusqu'au bout. Dans un tel cas, il faut impérativement arrêter sa trace d'exécution et déclarer qu'il y a erreur à l'exécution : il est incorrect ! il faut le corriger.

Au moins cinq cas peuvent se présenter :

Cas 1 : plus de valeurs à entrer Cette erreur se rencontre lorsqu'un programme demande d'entrer une valeur dans une variable, et que toutes les valeurs de la ligne *valeurs-en-entrée* sont biffées : il n'y a plus de valeurs à entrer.

Cas 2 : toutes les valeurs n'ont pas été entrées Cette erreur se rencontre lorsqu'un programme se termine et qu'il reste une ou plusieurs valeurs de la ligne *valeurs-en-entrée* qui n'ont pas été biffées : toutes les valeurs n'ont pas été entrées.

Cas 3 : incohérence des types Cette erreur se rencontre lorsqu'un programme demande d'entrer une valeur dans une variable, et que le type de la prochaine valeur à entrer ne correspond pas au type de la variable désignée par l'instruction `saisir`. Exemple : l'instruction est `saisir(V)` avec `V` une variable numérique et la prochaine valeur à entrer est une chaîne de caractères. Il y a incohérence des types.

Ou encore il demande de faire une affectation et le type du résultat de l'évaluation de l'expression à droite du symbole `<--` n'est pas le même que celui de la variable à gauche du symbole `<--` qui doit mémoriser le résultat. Il y a aussi incohérence des types.

Notez que lorsqu'un programme est compilé avant d'être exécuté, ce type d'erreur est détecté dès la compilation avant même l'exécution.

Cas 4 : utilisation d'une variable qui ne contient pas de valeur Cette erreur se rencontre lorsqu'une instruction veut utiliser une variable (soit par l'instruction `afficher`, soit par l'instruction d'affectation) alors qu'aucune valeur ne lui a été encore affectée. Rappel : en début d'exécution aucune valeur n'est affectée aux variables. Pour qu'une variable contiennent une valeur (définie), il faut qu'une instruction `saisir` ou d'affectation y mémorise une valeur.

Cas 5 : résultat erroné A la fin de son exécution, toutes les valeurs en entrée doivent être barrées et sur la ligne réservée aux valeurs en sortie, vous devez trouver les valeurs prévues en sortie. Sinon, considérez que le résultat obtenu n'est pas le bon.

Remarque

L'intérêt de faire une ou deux traces d'exécution après avoir écrit un programme n'est pas de montrer qu'il est correct, mais simplement de relever les erreurs d'étourderie et de logique.

8.4 Méthode

Remarquez qu'une trace d'exécution est une activité dynamique ! Elle se construit au fur et à mesure de l'exécution de votre programme. Cela implique de l'exécuter pas-à-pas et de noter, au fur et à mesure, les modifications dans les différentes lignes de la trace.

Voici la méthode élémentaire d'élaboration d'un programme :

- Faites attention au problème posé. Notamment ayez à l'esprit les tenants (c.-à-d. les valeurs en entrée) et les aboutissants (les valeurs en sortie) de votre programme.
- Résolvez les exercices proposés.
- Transcrivez votre analyse en un programme.
- Soumettez votre programme écrit à une ou deux traces d'exécution.

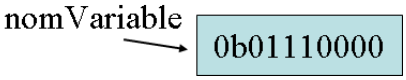
Si la trace obtenue n'est pas celle attendue, réfléchissez à nouveau et modifiez votre programme en conséquence pour que la trace d'exécution obtenue soit conforme à celle attendu.

9 Compléments

9.1 Variables et constantes

Emplacement mémoire, unicité des noms

Une variable désigne un **emplacement mémoire** qui permet de stocker une valeur. L'adresse de cet emplacement mémoire est associée au nom de la variable. Manipuler le nom de la variable c'est faire référence à son adresse en mémoire : c'est le système qui réalise cette association.



Puisqu'un programme exploite généralement plusieurs variables pour traiter les informations, chacune est désignée par un nom unique. L'**unicité des noms** de variables permet à la machine d'identifier précisément la variable à être manipulée.

Variable

Une variable peut être vue comme une boîte dans laquelle est rangée une information qui peut être récupérée à tout moment. À la différence de la boîte qui redevient vide lorsqu'on en retire son contenu, la variable stocke une « copie » des données qu'un algorithme y range (via l'affectation). Ainsi, lorsque l'algorithme récupère la valeur d'une variable, il récupère une copie du contenu de la variable.

Une variable conserve ses données (c.-à-d. son contenu) jusqu'à ce qu'une autre valeur y soit stockée, remplaçant la valeur antérieure, ou jusqu'à ce que la variable soit détruite par l'algorithme.

Initialisation par défaut

Elle consiste à donner une valeur initiale de façon automatique aux variables déclarées (par exemple 0 pour les variables numériques, `Faux` pour les booléens, etc.). Celle-ci n'existe ni en algorithmique, ni dans les langages Java ou Python. En C/C++, cette initialisation diffère selon que cette variable est locale ou globale (cf. @[Règles de visibilité] dans [Algorithmes paramétrés (2)]).

Erreurs à éviter

Voici une synthèse :

- Une variable ou une constante déclarée avec un certain type ne peut pas être déclarée une deuxième fois au même niveau.
- Il faut déclarer la variable ou la constante avant de pouvoir l'utiliser.
- Une constante doit être initialisée lors de sa déclaration.
- Il faut attribuer une valeur à la variable avant d'utiliser sa valeur.
- Lors de l'affectation d'une variable, la valeur de la partie droite doit être un type compatible de la variable.

Variables en programmation

Quelques conseils :

- Déclarez toujours une variable, même si certains langages de programmation peuvent les déclarer pour vous. Le type pris par défaut ne sera pas forcément le plus adapté.
- Déclarez les variables en début de programme si vous souhaitez un lexique clairement localisé. Sinon déclarez-les au plus près de leur première utilisation.
- Initialisez toujours les variables (par une affectation interne ou externe). Certains langages de programmation et/ou compilateurs les initialisent automatiquement à zéro, d'autres pas. En particulier, le compilateur Java émet le message d'erreur suivant dans un tel cas :

  ```
  Variable may not have been initialized
  ```

  Prenez le principe général qu'elle n'est **jamais** initialisée par défaut.

9.2 Affectation : Compléments

Affectation multiple

```
nomVar1 = nomVar2 = ... = expression
```

Explication

Effectue les affectations chaînées de la droite vers la gauche.

Notations abrégées

```
x @= y // <=> x = x @ y avec @ parmi {+,-,*,/,%}
++a // Incrémentation préfixée <=> a = a + 1 (a discret)
a++ // Incrémentation postfixée
--a // Décrémentation préfixée <=> a = a - 1
a-- // Décrémentation postfixée
```

Explication

Notations abrégées des opérateurs arithmétiques pour l'affectation.

9.3 Exemples : Compléments sur l'affectation

Exemple : Opérateurs arithmétiques pour l'affectation

Soient `a` et `b` deux variables entières de valeurs respectives 5 et 7, `f` et `g` deux variables réelles de valeurs 5.5 et -3.25. Le tableau ci-dessous illustre quelques exemples d'affectation utilisant les valeurs initiales des variables.

Expression	Expression équivalente	Valeur finale
`a += 5`	`a = a+5`	10
`f -= g`	`f = f-g`	8.75
`b *= (a-3)`	`b = b*(a-3)`	14
`f /= 3`	`f = f*3`	1.83333
`a %= (b-2)`	`a = a%(b-2)`	0

Exemple : Version préfixe et postfixe

Par exemple, si `x` a la valeur 5 :

- `y = x++` : Après évaluation, `x` aura la valeur 6 et `y` aura la valeur 5.
- `y = ++x` : Après évaluation, `x| aura la valeur 6 et \lstinline`y@ aura la valeur 6.

9.4 Transtypage

Transtypage (*casting*)
Conversion explicite du type d'une expression **vers un autre type compatible**.

Transtypage

```
(T) expr
```

Explication
Convertit le type de l'expression `expr` vers le nouveau type `T`.