



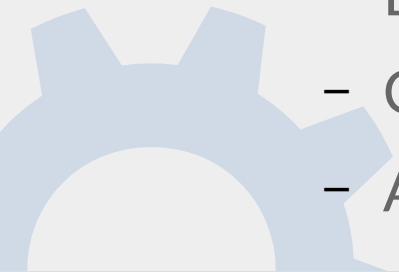
# Algorithmique

Denis Elsig

Ingénieur HES en systèmes de gestion  
Master MSE en ingénierie

# Présentation

- denis.elsig1@heig-vd.ch
- Ingénieur + Master MSE
- Consultant indépendant
  - Architecture
  - Développement logiciel
  - Mise en service et maintenance
  - Planification
  - Budget
  - Conseil stratégique
  - Audit



# Objectifs du cours

- Utiliser des schémas algorithmiques fondamentaux et des structures de données
- Envisager plusieurs algorithmiques différents pour répondre au même problème
- Faire preuve de rigueur dans le raisonnement
- Analyser un problème et le décomposer sous forme d'algorithme grâce à une démarche structurée
- Comprendre, écrire et évaluer un algorithme
- Manipuler des structures de données
- Porter un jugement sur la qualité de l'algorithme réalisé



# Organisation

- 80 périodes de cours, incluant des exercices en classe
- 90 heures de travail personnel
- Guy-Michel Renard



# Évaluation

- Contrôle continu : 60 %
  - Au minimum 2 tests écrits
- Examen d'unité : 40 %
- Moyens auxiliaires : aucun



# Sommaire

- Introduction
- Concepts de programmation
- Variables et concepts de base
- Structures de contrôle
  - Conditions et boucles
- Données structurées
  - Vecteurs, structures, types énumérés
- Sous-algorithmes



# Introduction

- Qu'est-ce qu'un algorithme ?



# Qu'est-ce qu'un algorithme?





# Qu'est-ce qu'un algorithme?

- Définition :

« Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations. »



# Qu'est-ce qu'un algorithme?

- Exemples :
  - la recette en « art culinaire »
  - le mode d'emploi s'il s'agit de mettre en route un appareil
  - les instructions de montage d'un meuble IKEA®
  - la partition qui permet d'exécuter un morceau de musique
  - le plan de l'architecte ou du contremaître
  - le schéma de l'électronicien qui réalise un appareil
  - ...

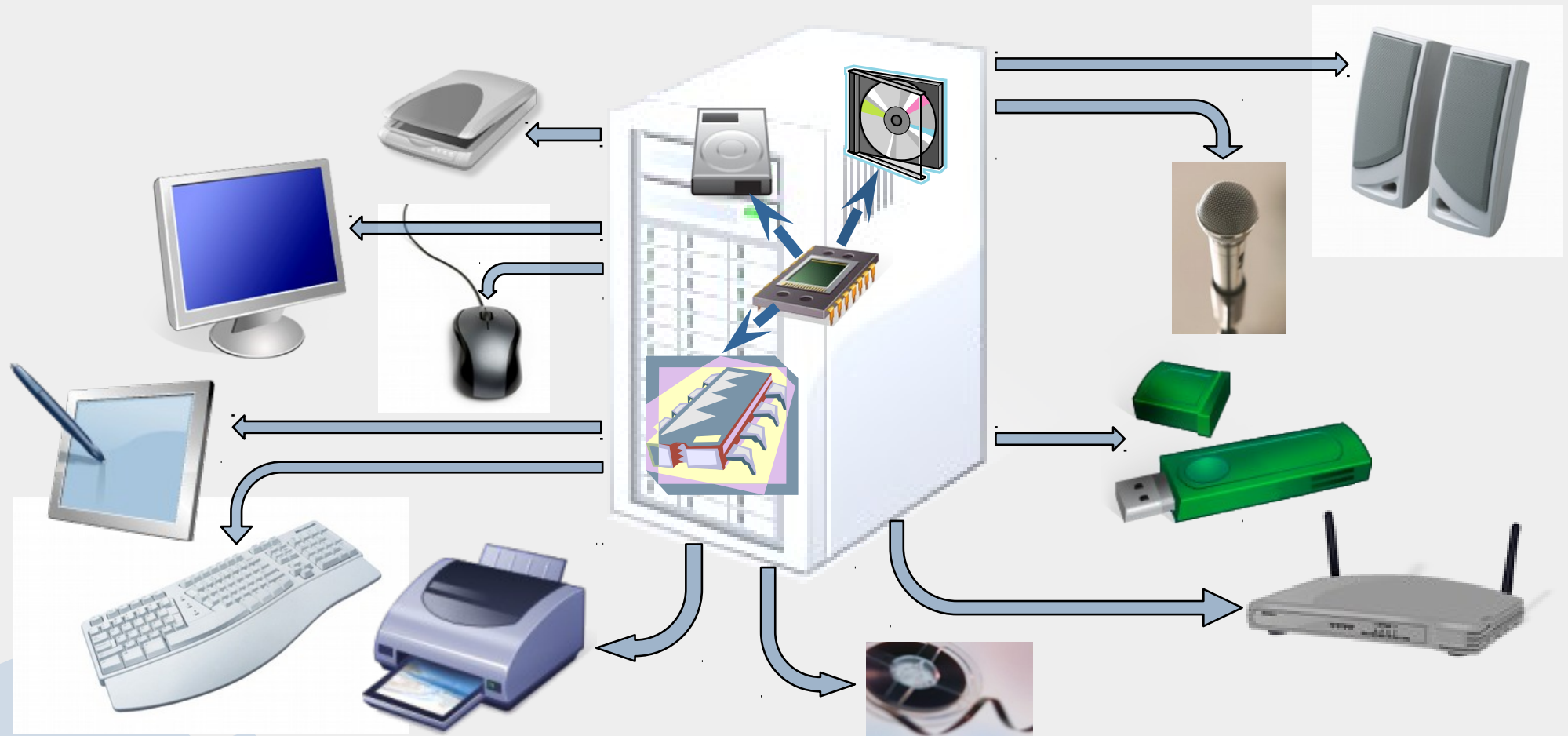


# Pourquoi étudier l'algorithmique?

- Utilisation de l'algorithmique avant d'aborder la programmation en divers langages informatiques
  - Afin de construire des algorithmes corrects et robustes
    - **correct** : un algorithme doit effectuer rigoureusement ce que l'on attend de lui, dans tous les cas de figure
    - **robuste** : il doit pouvoir être adapté à des situations non prévues lors de sa conception, être facilement mis à jour (*maintenance*)
- Description des algorithmes dans un **pseudo-langage**
  - Indépendant des spécificités des langages
  - Permet de s'attaquer à la résolution du problème et non à l'implémentation



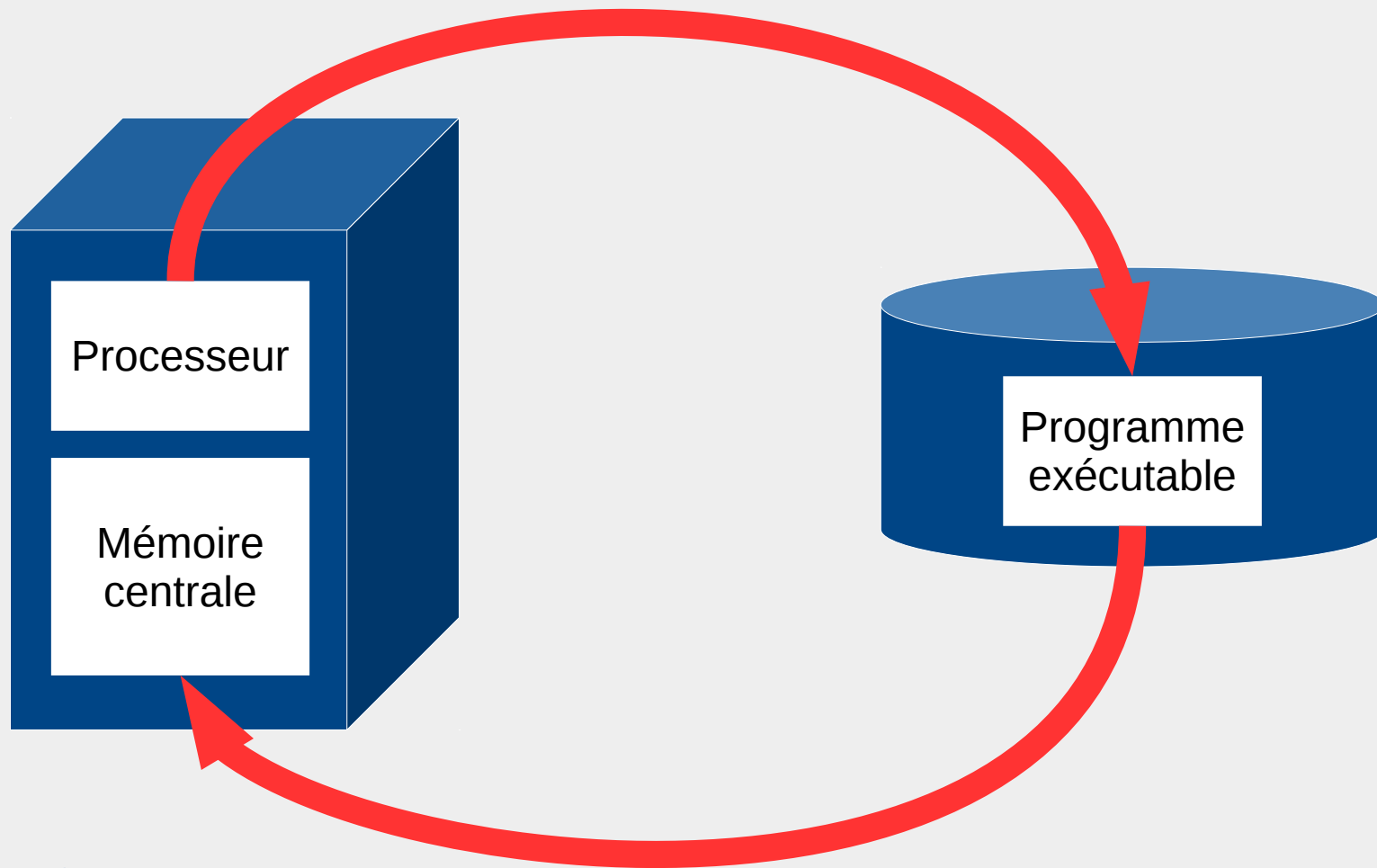
# L'ordinateur - le matériel



# L'ordinateur – les logiciels

Fonctionnalité	Description - exemples
<b>Système d'exploitation (OS)</b> pilotes du matériel gestion de la <b>mémoire centrale</b> gestion du <b>système de fichiers</b> <b>lancement</b> des applications <b>communication</b> avec l'extérieur	<i>applications de contrôle de chaque élément du <b>matériel</b> réservation, protection, sauvegarde de zones mémoires organisation de la mémoire de masse en répertoires et fichiers exécution d'applications et de « services » ± simultanément gestion du réseau, communication avec d'autres ordinateurs</i>
<b>Outils système</b> des milliers d' <b>applications</b> diverses	<b>Exemples</b> : <i>gestion des utilisateurs, explorateur, administration, sauvegardes, sécurité, test du matériel, maintenance...</i>
<b>Applications « utilisateur »</b> SGBDR : système de gestion de <b>bases de données</b> relationnelles  édition de texte (et graphique) <b>compilateur/interpréteur</b> - « linker »	<i>permet de gérer efficacement de grandes quantités de données, ainsi que leurs interrelations; ajout, modification, suppression ...  création de documents, écriture d'algorithmes ... transforment le programme source en applications exécutables</i>
<b>Applications « métier »</b> ...	<i>des <b>centaines de milliers</b> d'applications diverses permettent de réaliser les multiples tâches que l'on confie aujourd'hui à l'ordinateur</i>

# Exécution d'une application



# Exécution des applications

- Il existe 2 sortes d'applications :
  - Les applications compilées
  - Les applications interprétées



# Exécution des applications compilées

Les programmes écrits en langage de programmation sont des **fichiers sources**

- Pas directement exécutables par l'ordinateur
  - Ce ne sont que des fichiers textes
- Phase de traduction : compilation
- Réalisée par un compilateur





# Exécution des applications compilées

**compiler** = traduire le *code informatique* en une suite d'instructions  
*primitives* propres au *processeur*

(fichier) **source**  $\Rightarrow$  **compilateur**  $\Rightarrow$  (fichier) **objet**

**Linker** = consolider plusieurs fichier objets pour en faire un exécutable

(fichiers) **objet1, .. Objetn**  $\Rightarrow$  **linker**  $\Rightarrow$  (fichier) **exécutable**

L'application peut alors être exécutée par l'OS  
comme toute autre programme.



# Exécution des applications interprétées

**interpréter** = pour chaque ligne du *code informatique*, la **traduire** en *primitives* propres au *processeur*, puis **exécuter** immédiatement ces primitives et passer à la ligne suivante

(fichier) **source**  $\Rightarrow$  **interpréteur**  $\Rightarrow$  **exécution ligne par ligne**

Un interpréteur est propre à un langage de programmation



# Erreurs de programmation

- 2 types
- Les erreurs de **syntaxe**
  - un *langage de programmation* est constitué d'*instructions*, et chaque *instruction* doit respecter une syntaxe bien précise
  - une erreur de construction ou d'utilisation d'une instruction est appelée erreur de syntaxe
  - le compilateur (ou l'interpréteur) est en mesure de la détecter

erreur de syntaxe  $\Rightarrow$  message du compilateur



# Erreurs de programmation

- Les erreurs de **programmation**, dites de **logique**
  - parfois, l'algorithme codé ne réalise pas ce que l'on attend
  - *Le programme est pourtant compilé ou interprété*
  - *les résultats obtenus ne sont pas ceux que l'on attendait (tests)*
  - seul le programmeur peut détecter ... et **corriger** ce type d'erreur

erreur de logique  $\Leftrightarrow$  **algorithme(s) erroné(s)**



# Variables et concepts de base

- Structure de base d'un algorithme
- Déclaration de variables et de constantes
- Variables et concepts de base
- Affectation d'une valeur à une variable
- La notion d'expression
- Entrées / sorties : saisie et affichage des données



# Structure de base d'un algorithme

- Résolution d'un problème
  - Description du *problème*
  - Détail des *données* disponibles en **entrées**
  - Détail des *données* à fournir en **sortie**
  - Description du **traitement** à effectuer sur ces *données*



# Structure de base d'un algorithme

- Structure d'un algorithme :
  - **l'en-tête** : série de commentaires, définis et ordonnés selon certaines conventions
  - **les déclarations** : décrivent les données et la nature des données dont l'algorithme a besoin (constantes et variables)
  - **les instructions** : permettent de faire réaliser (exécuter) quelque chose à l'algorithme: des calculs, des saisies, des affichages...



# Algorithme – structure schématique

**En-tête**  
de l'algorithme  
(commentaires)

// : description formelle de  
l'algorithme, informations

**Déclarations**

**CONST** : déclaration de *constante*

**TYPE** : déclaration de *type*

**VAR** : déclaration de *variable*

**Corps** de  
l'algorithme  
(traitements)

*instruction* du langage

FIN



# Algorithme – L'en-tête

- En-tête : texte décrivant l'algorithme
  - **Nom** de l'algorithme, son **auteur** et la **date** de sa conception
  - **Objectif** visé
  - **Evolution** de l'algorithme au gré des **modifications**
  - Explication des **données extérieures**
    - Informations nécessaires à son exécution : **données d'entrée**
    - Informations issues de son exécution : **données de sortie**



# Algorithme – L'en-tête (exemple)

```
// NOM : algo_0, exemple calcul de la racine carree
// AUTEUR : Bernard Collet, HEIG-VD fee
// DATE : 18 juillet 2009
// OBJECTIF : calcul de la racine carrée
// MODIFICATIONS : -----
// ENTREES : a (valeur dont on calcule la racine)
// SORTIES : xi_1 (valeur finale des approximations)
```

} en-tête

```
// Debut du programme principal
// =====
VAR a : REEL
VAR xi, xi_1 : REEL
VAR nb : ENTIER
```

```
// Saisie de la valeur d'entree a, nombre dont on
//      veut calculer la racine carree
```

```
REPETER
    ENTREE("Racine carree de quel nombre ( $\geq 0$ ) ? ")
    LIRE(a)
JUSQU'À a  $\geq 0$ 
```



# Algorithme – Déclarations et corps

- Déclarations : décrivent la nature des données dont l'algorithme a besoin
  - Constantes par **CONST** ...
  - Variables par **VAR** ... : en principe, on rencontre au moins les variables stockant les *données d'entrée et de sortie* de l'algorithme
- Corps de l'algorithme
  - *Instructions* (pseudo-langage) permettant de réaliser les traitements



# Algorithme – Déclarations et corps (exemple)

```
// NOM : algo_0, exemple calcul de la racine carree
// AUTEUR : Bernard Collet, HEIG-VD fee
// DATE : 18 juillet 2009
// OBJECTIF : calcul de la racine carrée
// MODIFICATIONS : -----
// ENTREES : a (valeur dont on calcule la racine)
// SORTIES : xi_1 (valeur finale des approximations)
```

```
// Debut du programme principal
// =====
VAR a : REEL
VAR xi, xi_1 : REEL
VAR nb : ENTIER
```

} Déclarations

```
// Saisie de la valeur d'entree a, nombre dont on
// veut calculer la racine carree
```

```
REPETER
  ENTREE("Racine carree de quel nombre ( $\geq 0$ ) ? ")
  LIRE(a)
JUSQU'À a  $\geq 0$ 
```

} Corps



# Algorithme – Les commentaires

- Qu'est-ce qu'un commentaire ?
  - du texte inséré dans l'algorithme
  - il sert au lecteur **humain** comme aide à la compréhension
- Pourquoi des commentaires dans un algorithme ?
  - ils rendent l'algorithme compréhensible au lecteur
  - ils permettent de mettre en évidence sa structure
- Syntaxe d'un commentaire
  - dès le double caractère **//**, tout ce qui suit jusqu'à la fin de la ligne est un commentaire
  - un commentaire peut suivre une *déclaration* ou une *instruction*
  - une *ligne entière*, débutant par **//** constitue un commentaire



# Déclaration de variables

- Qu'est-ce qu'une **variable** ?
  - il s'agit d'une *zone en mémoire* pouvant contenir une valeur
  - on donne un *nom* à cette zone, pour l'identifier clairement
  - la *valeur* contenue peut être de différentes natures (**types**)
- Avant son utilisation, une **variable** doit être déclarée
- Syntaxe : déclaration d'une (ou plusieurs) variable(s)

**VAR** *nomDeLaVariable[, nomDAutreVariable]... : TYPE*

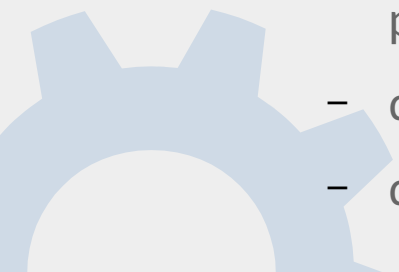
# Nommage des variables

- La notion d'**identificateur**
  - Chaque **variable** dispose d'un nom la distinguant des autres variables de l'algorithme
  - mais d'autres éléments également disposeront d'un nom
  - le « nom » de ces éléments est appelé **identificateur**
- Conventions quant à l'identificateur
  - il doit commencer par une *lettre* : ageEnfant, x3, v\_a...
  - qui peut être suivie d'autres *lettres*, de *chiffres* et de *\_*
  - règles *facultatives*, mais dont le respect aide à la lecture des algorithmes
    - **variables** : *en minuscule*
    - **Constantes** : *en majuscules*



# Types des variables

- Le **type** précise la nature de la *valeur* contenue dans une **variable**
- Types **simples** fournis par notre *pseudo-langage*
  - ENTIER : valeurs numériques entières (dénombrables, ensemble  $\mathbb{Z}$ )
  - REEL : valeurs numériques réelles (non dénombrables, ensemble  $\mathbb{R}$ )
  - CARACTERE : valeur *caractère*
  - CHAINE : chaîne de caractères de taille variable
  - BOOLEEN : valeurs VRAI ou FAUX, souvent issues d'un test
- Opérateurs autorisés par chaque type de données
- Autres types de données
  - des **tableaux** (à 1 ou plusieurs dimensions) d'éléments d'un certain type : permet de disposer de *vecteurs* ou de *matrices* de valeurs
  - des types **énumérés** dont on fournit le nom des valeurs possibles
  - des types **structurés** dont on déclare la structure  $\pm$  complexe





# Déclaration de constantes

- Qu'est-ce qu'une **constante**?
  - une *valeur* (numérique ou d'un autre type) introduite dans l'algorithme selon certaines conventions : -27, ' a ', 9.128
  - on peut **déclarer** des *constantes* et les munir d'un *identificateur*, qui sera utilisé à la place de la *valeur* correspondante
- Avant son usage, une **constante** doit être déclarée
- Syntaxe : déclaration d'une constante

**CONST** *NOM\_CONSTANTE* **C'EST** *valeur*



# Conventions de notation

- ENTIER :            -12                            4                            5
- REEL :                4.5        12.78
- CARACTERE :        'a'        ' '                            'Z'
- CHAINE :            "bonjour"        "Bonjour"
- BOOLEEN :            VRAI                            FAUX



# Algorithme: exemple complet

```
// NOM : algo_0, exemple calcul de la racine carree
// AUTEUR : Bernard Collet, HEIG-VD fee
// DATE : 18 juillet 2009
// OBJECTIF : algorithme code selon les regles
// MODIFICATIONS : -----
// ENTREES : a (valeur dont on calcule la racine)
// SORTIES : xi_1 (valeur finale des approximations)

CONST MAX_ITER C'EST 25          // maximum d'iterations
CONST PRECISION C'EST 1.0E-8    // precision souhaitee

// newton(arg, valActuelle) : retourne l'approximation
// ===== suivante de la racine
//                               carree de <arg>,
//                               <valActuelle> depart
//
// Application de l'algorithme de Newton
// (methode connue des Babyloniens vers 1700 av JC)
FONCTION newton(arg, valActuelle : REEL) : REEL
    RETOURNER ((valActuelle + arg/valActuelle) / 2)
FINFONCTION

// Debut du programme principal
// =====
VAR a : REEL
VAR xi, xi_1 : REEL
VAR nb : ENTIER
```

```
// Saisie de la valeur d'entree a, nombre dont on
// veut calculer la racine carree
REPETER
    ECRIRE("Racine carree de quel nombre (>= 0) ? ")
    LIRE(a)
JUSQU'A a >= 0


// Initialisation des variables necessaires aux
// approximations successives
nb <- 0          // limite en cas de non convergence
xi_1 <- 1.0      // approximation initiale

// Boucles d'approximations successives
REPETER
    xi <- xi_1
    ECRIRE(xi)
    nb <- nb + 1
    SI nb MOD 6 <> 0
    ALORS
        ECRIRE(" - ") // afficher un -
    SINON
        ECRIRE()      // afficher un saut de ligne
    FINSI
    xi_1 <- newton(a, xi)
JUSQU'A nb >= MAX_ITER OU abs(xi_1 - xi) <= PRECISION

// Affichage du resultat
ECRIRE("Approximation : ", xi_1)
ECRIRE(" et par sqrt() : ", racine(a))
ECRIRE(" en ", nb, " iteration(s).")
```

# Les instructions

- On distingue **trois** sortes d'instruction
  - l'affectation d'une *valeur* à une *variable* :
    - `nb <- nb + 1`
  - Les **appels** de *procédure* :
    - `ECRIRE("Racine: ", xi_1)`
  - les **structures de contrôle**
    - construites selon certaines règles
    - Conditions, boucles



```
SI NON estPositif
ALORS nb <- -nb
FINSI
```

```
POUR i DE 1 A N FAIRE
  ECRIRE(i, " au carré: ", i*i)
FAIT
```

# L'instruction d'affectation

- Opération permettant d'attribuer une *valeur* à une *variable*
  - la *variable* **doit** avoir été déclarée au préalable
  - la *valeur* est en général fournie par l'évaluation d'une **expression**
  - le **type** de la *valeur* doit être le même que celui de la *variable*  
Exception : une *valeur* ENTIER peut être affectée à une *variable* REEL
- Syntaxe d'une affectation :

*NomVariable* ← *expression*



# Notion d'expression

## ■ Qu'est-ce qu'une expression ?

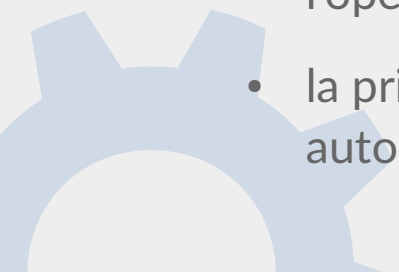
- il s'agit de la généralisation informatique de l'expression arithmétique
- à l'exécution, chaque *expression* est évaluée et retourne une valeur
- elle est constituée d'un mélange d'**opérandes** et d'**opérateurs**, combinés selon certaines règles de *priorité* et de *regroupement*

## ■ Exemple d'une expression un peu complexe

**$x - 1.5 / \sin(2 * a) + -7.3 * y < (25 + z) \text{ MOD } i \text{ ET NON ok}$**

## ■ Quelques remarques basées sur l'exemple

- $\sin(2 * a)$  constitue un seul *opérande*, la valeur *retournée* par  $\sin(...)$  (l'expression  $2 * a$  est évaluée avant l'appel de la « fonction »  $\sin$ )
- le quotient  $1.5 / \sin$  est évalué avant sa soustraction de  $x$  : on dit que l'opérateur  $/$  a une priorité supérieure à celle de  $-$
- la priorité de l'opérateur  $\text{MOD}$  est supérieure à celle de  $+$ , mais les parenthèses autour de  $(25 + z)$  assurent que l'addition sera effectuée avant le calcul du modulo



# La concaténation

## Assemblage de valeurs et variables de type CHAINE

Il arrive que l'on désire "fabriquer" une *chaîne de caractère* (type CHAINE) à partir de diverses valeurs textuelles **et** numériques

- on peut le faire par une expression composée d'opérandes représentant toutes ces *valeurs* qui devront constituer le texte de la *chaîne*
- tous ces éléments de *texte* sont ensuite juxtaposés pour constituer la *chaîne* finale: on appelle **concaténation** cette dernière opération

L'opérateur + est utilisé pour cela : appliqué sur des opérandes de type CHAINE ou CARACTERE, il n'agit plus d'un opérateur d'**addition**, mais de **concaténation**

Exemple :

```
msg <- "résultat:" ;   sp <- ' '  
txt <- "Mon " + msg + sp + "suffisant"
```

txt vaudra : **"Mon résultat: suffisant"**



# Les opérandes d'une expression

Pour évaluer une expression, on effectue des *opérations* sur des opérandes

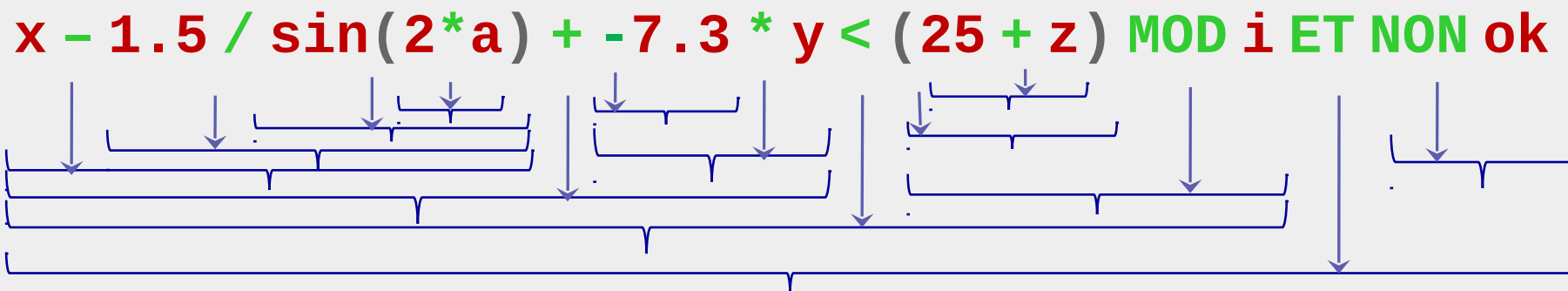


Illustration de la **priorité** et du mode de **groupement** associés aux opérateurs



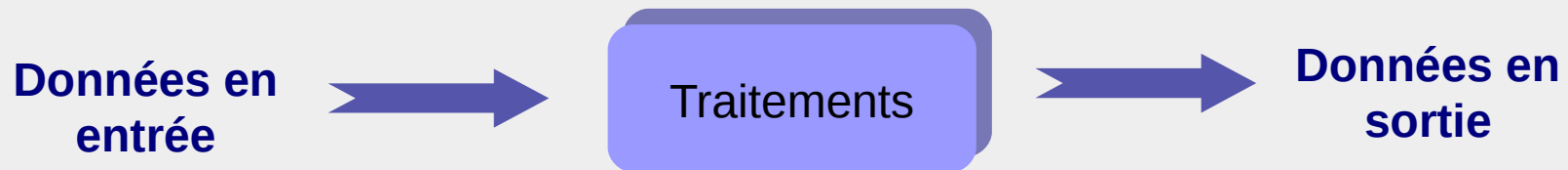
# Priorité des opérateurs

Type d'opération	Niveau de priorité	Opérateurs	TYPE des opérandes	Groupement
<i>sous-expression</i>	1	( )	type simple	Gauche à Droite
<i>exponentiation</i>	2	^	numériques	Droite à Gauche
<i>unaire (monadique)</i>	3	— NON	numérique BOOLEEN	Droite à Gauche
<i>multiplicatif</i>	4	* / MOD	numériques ENTIERs	Gauche à Droite
<i>additif</i>	5	+ —	numériques	Gauche à Droite
<i>relationnel</i>	6	< <= > >= = <>	types simples, deux de même type	Gauche à Droite
<i>ET logique</i>	7	ET	BOOLEENs	Gauche à Droite
<i>OU logique</i>	8	OU	BOOLEENs	Gauche à Droite

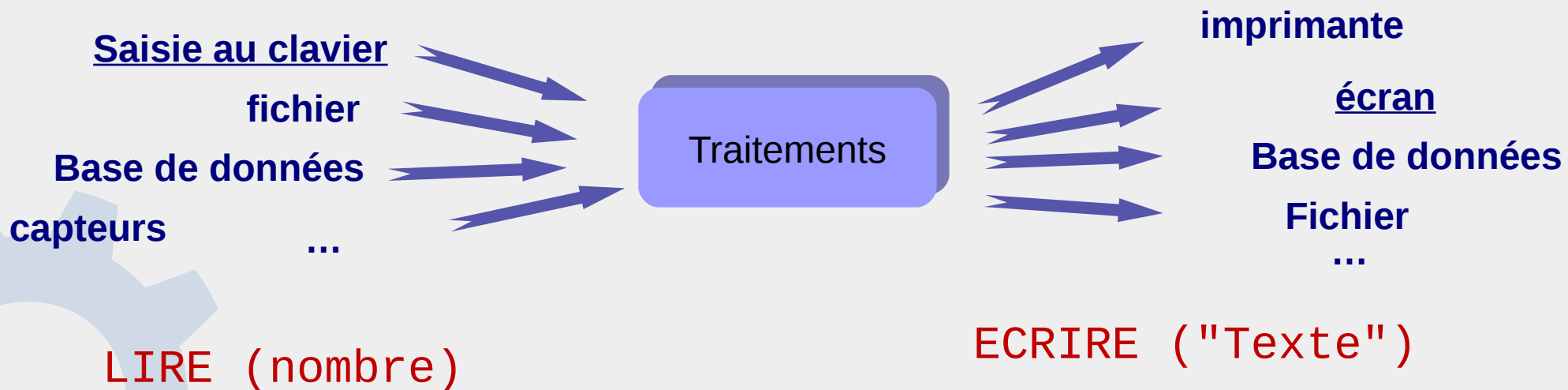
# Entrées / sorties (IO)

Le concept d'entrées/sorties

But d'un algorithme : réaliser des *traitements* sur des *données* pour produire d'autres *données*



Différents types d'E/S



# Sortie d'informations par affichage

## ECRIRE (...)

affiche un texte (expression de type CHAINE)

**ECRIRE("Texte à afficher") ; ECRIRE(message)**

...mais aussi des nombres ou le résultat d'expressions *numériques*

**ECRIRE(23) ; ECRIRE(nombre) ; ECRIRE(3 + 2.1 \* -x)**

...et même des expressions de type BOOLEEN (afficheront VRAI ou FAUX selon la valeur de l'expression fournie)

**ECRIRE(FAUX) ; ECRIRE(continuer) ; ECRIRE(x <= 5)**

L'instruction ECRIRE( ) permet de combiner ces types en affichant successivement plusieurs valeurs séparées par une *virgule*

**ECRIRE( $expr_1$ [,  $expr_n$ ]...)**



# Entrée d'informations par saisie

LIRE(...)

LIRE(*var*) saisit une valeur et la place dans la variable *var* en mémoire

**LIRE(message) ; LIRE(ageMere) ; LIRE(continuer)**

- si la variable *var* est
  - de type CARACTERE : un seul caractère est saisi
  - de type CHAINE : une suite de caractères est saisie
  - de type ENTIER ou REEL : un nombre est saisi
  - de type BOOLEAN : la valeur VRAI ou la valeur FAUX est saisie et convertie en booléen

**LIRE(*nomVariable*)**



# Exercice en langage naturel

- Écrire en langage naturel la procédure complète du changement d'une roue d'un véhicule.
  - Doit être compréhensible par tout le monde
  - Doit être exhaustif
  - Doit être « fool proof »
- Discussion ouverte sur les solutions proposées



# Structures de contrôle

- Structures conditionnelles
  - Structure de choix conditionnel alternatif
  - Structure de choix multiple
- Structures de répétition
  - Structure de répétition déterminée
  - Structure de répétition *pré*-conditionnelle
  - Structure de répétition *post*-conditionnelle



# Qu'est-ce qu'une structure de contrôle?

- Ce sont des **instructions de base** (du pseudo-langage ou d'un langage de programmation)
  - Elle permettent d'exécuter ou non des bouts de code en fonction d'une hypothèse ou d'un test logique (conditions).
  - Elle permettent de répéter le même traitement sur un ensemble de données (répétitions).



# Structure conditionnelle alternative

- Elle exécute **sous condition** un groupe d'instructions
  - Un *groupe d'instructions* est une suite d'instructions exécutées l'une après l'autre
    - elles peuvent se trouver sur la même ligne, séparées par des ;
    - ou se trouver sur des lignes différentes, sans autre séparateur
    - ... ou toute combinaison de ces deux manières de faire
- la *condition* est exprimée par l'évaluation d'une *expression booléenne*






# Exemples de structures conditionnelles alternatives

1

```
SI nb = 0
ALORS
    ECRIRE ("le nombre est nul")
FINSI
```

2

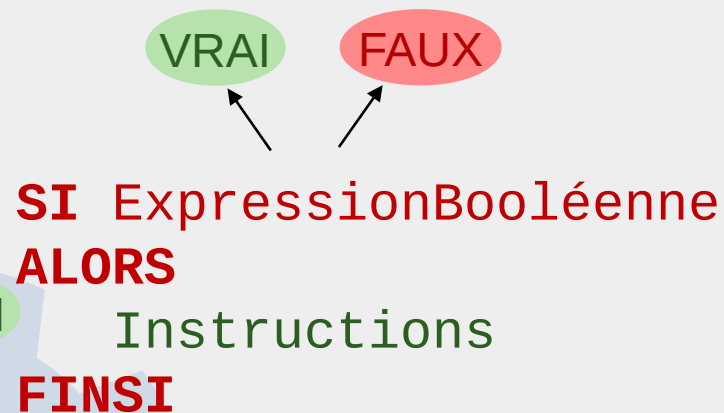
```
SI nb = 0
ALORS
    ECRIRE ("le nombre est nul")
SINON
    ECRIRE ("le nombre est différent de zéro")
FINSI
```



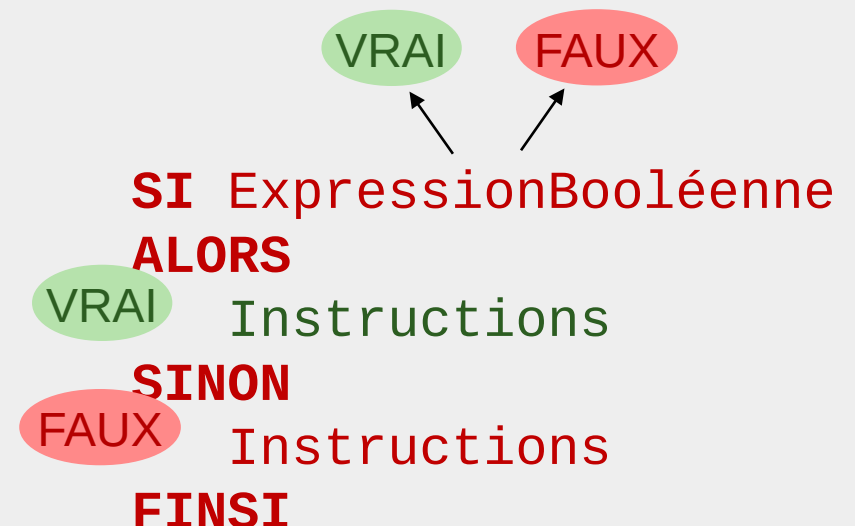
# Syntaxe et exécution de la structure SI...ALORS

```
SI exprBooleenne  
ALORS groupeInstr_VRAI  
[SINON groupeInstr_FAUX]  
FINSI
```

## Forme simple



## Forme complète



# Autres exemples

```
SI nb > 0
ALORS estPositif ← VRAI
FINSI
```

```
SI NON estPositif
ALORS nb ← -nb
FINSI
```

```
SI estNul
ALORS etat ← "nul"
SINON etat ← "non nul"
FINSI
```

## Forme imbriquée

```
SI nb = 0
ALORS
    estNul ← VRAI
    estPair ← VRAI ; estMultiple3 ← VRAI
SINON
    estNul ← FAUX
    estPair ← nb MOD 2 = 0
    SI nb MOD 3 <> 0
    ALORS estMultiple3 ← FAUX
    SINON estMultiple3 ← VRAI
    FINSI
FINSI
```

} car c'est **aussi**  
une instruction



# Expressions booléennes

- Utilisation des tables de vérité

ET	Vrai	Faux
Vrai		
Faux		

OU	Vrai	Faux
Vrai		
Faux		

NON	
Vrai	
Faux	



# Expressions booléennes complexes

Soit à résoudre le petit problème suivant :

- On aimerait mettre en évidence le fait qu'une valeur `val` est **non nulle**, mais aussi qu'elle est **paire** et **positive** ou qu'elle est **multiple de cinq** et **négative**

1

```
SI val <> 0
ALORS
  SI val > 0
  ALORS
    SI val MOD 2 = 0
    ALORS ECRIRE("La valeur satisfait les critères")
    FINSI
  SINON
    SI -val MOD 5 = 0
    ALORS ECRIRE("La valeur satisfait les critères")
    FINSI
  FINSI
FINSI
```

2

```
SI val <> 0 ET ((val > 0 ET val MOD 2 = 0) OU (val < 0 ET -val MOD 5 = 0))
ALORS ECRIRE("La valeur satisfait les critères")
FINSI
```

# Structure de choix multiple

- Elle exécute, **selon la valeur** d'une expression, un groupe d'instructions parmi plusieurs groupes fournis
  - dans le cas où aucun groupe n'a été prévu pour une certaine valeur, un « groupe par défaut » **peut** être fourni qui sera alors exécuté
  - l'expression doit être d'un type simple, (ENTIER, CARACTERE, CHAINE, BOOLEEN ou *énumération*)



# Structure de choix multiple (exemple)

```
CONST MIN c'est 1
```

```
VAR nb : ENTIER
```

```
VAR resultat : REEL
```

```
LIRE (nb)
```

```
CHOIX nb
```

```
    CAS MIN : ECRIRE("Il s'agit de la valeur minimale ", MIN)
```

```
    CAS 2 : ECRIRE("Et là le nombre DEUX")
```

```
    CAS 4 : ECRIRE("Vous avez saisi la valeur QUATRE")
```

```
    CAS 8 : ECRIRE("Encore une puissance entière de DEUX : ", nb)
```

```
    SINON : resultat ← nb * nb
```

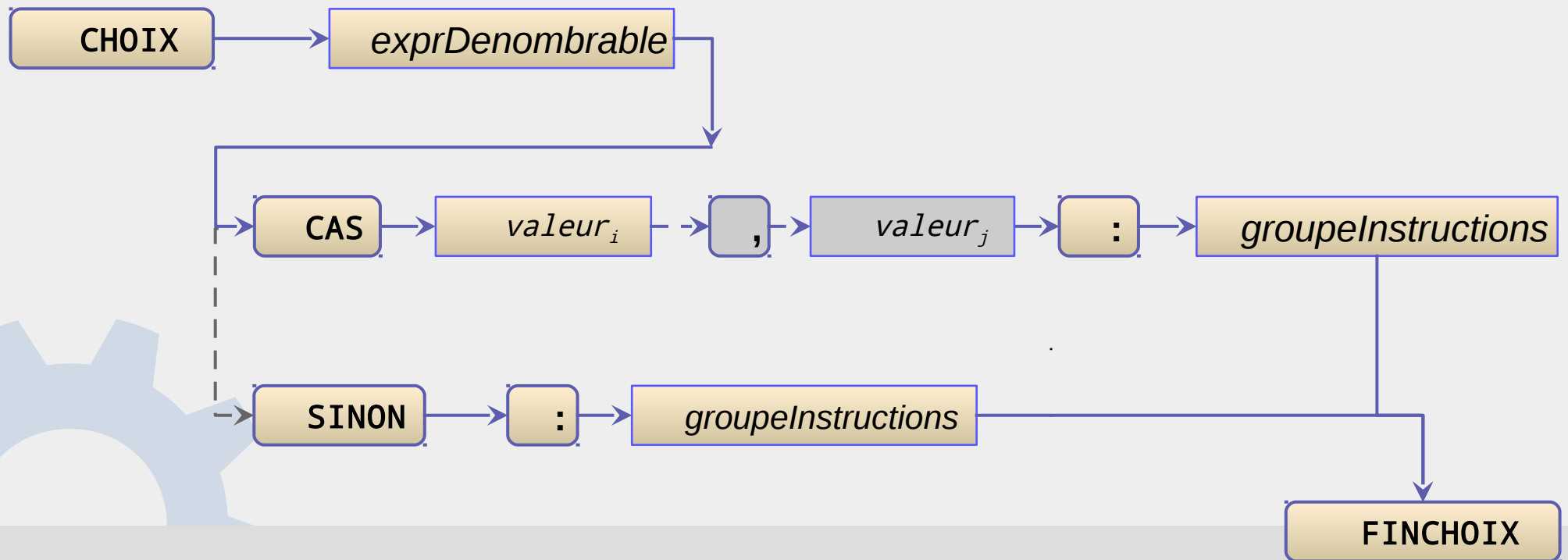
```
        ECRIRE("Le carré du nombre saisi ", nb, " vaut ", resultat)
```

```
FINCHOIX
```



# Syntaxe de la structure CHOIX

```
CHOIX exprDenombrable  
  CAS  $val_{11}[, val_{1m}]... : groupeInstr_1$   
  [CAS  $val_{21}[, val_{2m}]... : groupeInstr_2$ ]  
  ...  
  [SINON :  $groupeInstr_{defaut}$ ]  
FINCHOIX
```





# Les boucles

- *Boucles : structure de répétition*
  - permettent de répéter un *groupe d'instructions*
- **trois types de boucles**
  - Boucle déterminée
    - **POUR ... ALLANT DE ... A ... [A PAS DE ...] FAIRE ... FAIT**
  - Boucles indéterminées
    - **TANTQUE ... FAIRE ... FAIT**
    - **REPETER ... JUSQU'À ...**



# Boucle déterminée POUR

- Le nombre de tours de boucles est prédéterminé
  - une *variable*, appelée **compteur** (de type *dénombrable* ENTIER, CARACTERE, BOOLEAN ou *énumération*) parcourt une plage de *valeurs* entre deux *limites* fournies au départ
  - le ***pas*** (incrément ou décrément), par défaut 1, peut également être spécifié; on aura décroissance de la variable s'il est négatif



# Boucle déterminée POUR (exemple)

```
VAR i, nb : ENTIER
```

```
VAR resultat : CHAINE
```

```
nb <- 4
```

```
POUR i ALLANT DE 1 A nb A PAS DE 1 FAIRE
```

```
    // clause PAS facultative, car egale a 1
```

```
    ECRIRE ("Bonjour")
```

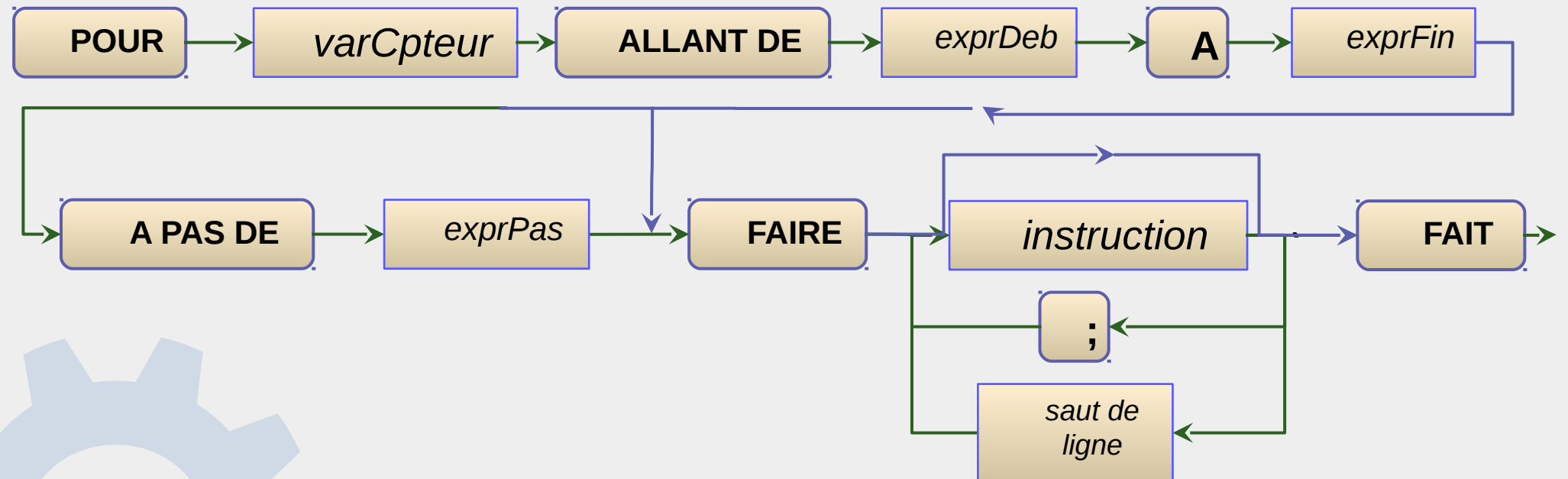
```
FAIT
```

```
ECRIRE (resultat)
```



# Syntaxe de la boucle POUR

POUR *varCpteur* ALLANT DE *exprDeb* A *exprFin* [A PAS DE *exprPas*] FAIRE  
*groupeARepeteter*  
FAIT



# Fonctionnement de la boucle POUR

- 1) les *expressions* `exprDeb`, `exprFin`, et `exprPas` sont évaluées
- 2) `exprDeb` est affectée à la variable `varCpteur`, qui doit avoir un type compatible
- 3) si la valeur de `varCpteur` est  $\leq$  `exprFin`, le *groupe d'instructions* est alors exécuté; sinon on quitte la structure de contrôle
- 4) `varCpteur` est incrémenté de `exprPas` (automatiquement par la boucle) puis on retourne ensuite au point 3 ci-dessus



# Quelques règles

- Le compteur de boucle (`varCpteur` ) ne peut être modifié dans la boucle
- Les valeurs limites (`exprDeb` et `exprFin`) ne peuvent être modifiées dans la boucle
- Le compteur de boucle (`varCpteur` ) s'incrémente automatiquement (ne pas le gérer)
- Si `exprDeb` est plus grand que `exprFin` et que le pas est positif, la boucle ne s'exécute pas
- Et inversement...



# Exemple 1 de boucle POUR

```
VAR i, nb : ENTIER
nb <- 4
POUR i ALLANT DE 1 A nb A PAS DE 1 FAIRE
    // clause A PAS DE facultative, car égale a 1
    ECRIRE ("Bonjour")
FAIT
```

Exercice 1 : écrire un algorithme qui écrit les nombre de 1 à 10  
Exercice 2 : écrire un algorithme qui écrit les lettres de a à z



## Exemple 2 de boucle POUR

```
VAR resultat, i, nb : ENTIER  
resultat <- 0  
nb <- 4
```

```
POUR i ALLANT DE 1 A nb FAIRE  
    //clause A PAS DE facultative, car egale a 1  
    resultat <- resultat + i  
FAIT
```

```
ECRIRE (resultat)
```

