INTRODUCTION

- Un ordinateur est une machine dénuée d'intelligence
- Un ordinateur est capable d'effectuer un grand nombre de tâches plus rapidement que le cerveau humain
- Il exécute sans erreurs les instructions qu'on lui a données
- S'il y a des erreurs, elles sont humaines dans la conception des programmes utilisés par l'ordinateur

PROGRAMMATION

- Un programme est un enchaînement d'instructions rédigées dans un fichier « texte »
- Un programme est écrit dans un langage de programmation
- L'écriture est parfois assistée : auto-complétion, vérification de la syntaxe du fichier …
- Il est exécuté par un ordinateur afin de traiter les données d'un problème et apporter des solutions et des résultats à l'utilisateur

DONNÉES ET TRAITEMENTS

- Une donnée est une valeur stockée
- Elle est variable ou constante
- Elle a un type
- Un traitement utilise les données et peut les transformer à l'aide d'instructions

ALGORITHMIQUE

Un algorithme est un enchaînement d'instructions ayant pour but de résoudre un problème

On parle aussi de :

- langage algorithmique
- pseudo-langage
- pseudo-code

EXEMPLE D'ALGORITHME

```
ALGORITHME NomAlgorithme

    /* Possibilité de déclarer des constantes */
    CONSTANTES constante1 <- 10 : entier
                       constante2 <- "bonjour" : chaîne

    /* Possibilité de déclarer des variables */
    VARIABLES    variable1, variable2 : réels
                 variable3 : entier
                      variable4 : chaîne

DEBUT
    Instruction1
    Instruction2
…
FIN
```

Déclaration des variables et constantes

Corps de l'algorithme

VARIABLES

Une variable possède :

- une **valeur** contenue dans une case mémoire
- un **identificateur** = un nom unique par lequel on peut accéder à son contenu
- un **type** qui définit la taille de la place occupée en mémoire
- il ne faut pas confondre la **variable** et son **contenu**
- une variable est un **contenant** (case ou boîte)
- le contenu d'une variable est **une valeur** numérique (entier, réel), alphanumérique, date …

VARIABLES

- Une variable est donc une « boite » dans laquelle on stocke une valeur (un nombre, un mot…)
- Cette variable peut « varier » (changer) si l'utilisateur en modifie la valeur par exemple
- Une variable dont la valeur ne change pas au cours de l'exécution du programme est appelée constante

DÉCLARATION DE VARIABLES

Déclarer une variable, c'est :

- réserver une place en mémoire
- attribuer l'identificateur à cette place mémoire

La déclaration indique :

- l'identificateur
- le type

- Exemple : `VARIABLE maVariable : entier`

TYPES DE VARIABLES

Type caractère

- lettres, chiffres, ponctuation, opérations, caractères spéciaux …
- exemples : ‘a’ ‘+’ ‘.’

Type chaîne de caractère

- suite de caractères
- exemples : « bonjour » « nom.prenom@gmail.com »

TYPES DE VARIABLES

Type **entier**

- les nombres entiers
- exemples : 2 23 63

Type **réel**

- les nombres réels (décimaux)
- exemples : 3,14 37,7

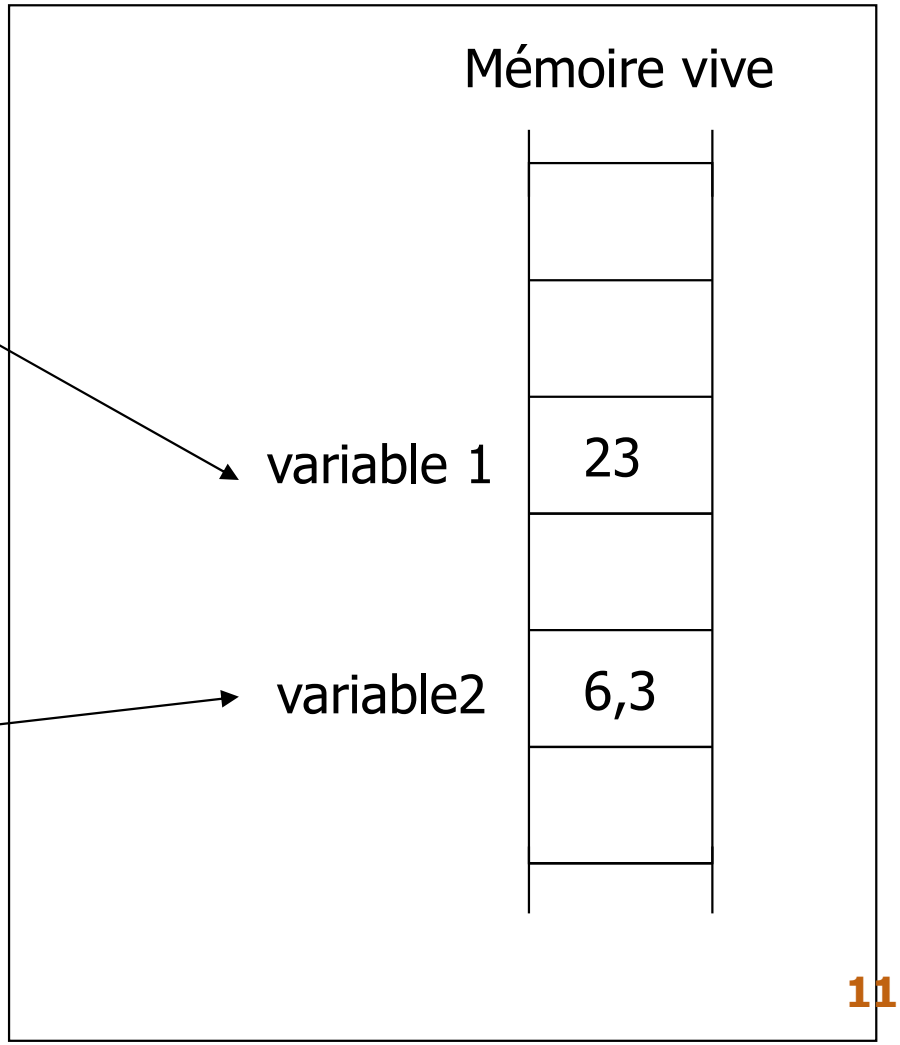
Type **booléen**

- deux valeurs possibles
- soit VRAI (1, TRUE) soit FAUX (0, FALSE)

EXEMPLES DE VARIABLES

- Identificateur : variable I
- Type : entier
- Valeur : 23

- Identificateur : variable2
- Type : réel
- Valeur : 6,3



OPÉRATEURS

- Les opérations possibles sur les variables dépendent des types des variables concernées

- On ne peux pas multiplier des mots…

ENTIERS

opérations possibles	symbole ou mot clé
addition	+
soustraction	-
multiplication	*
division	/
division entière	DIV
modulo	MOD
comparaisons	<, ≤, >, ≥, =, ≠

RÉELS

opérations possibles	symbole ou mot clé
addition	+
soustraction	-
multiplication	*
division	/
comparaisons	<, ≤, >, ≥, =, ≠

CARACTÈRES

opérations possibles	symbole ou mot clé
comparaisons	<, ≤, >, ≥, =, ≠ (exemple ‘a’ < ‘z’)

CHAÎNES DE CARACTÈRES

opérations possibles	symbole ou mot clé
concaténation	&
longueur	longueur (*chaîne*)
extraction	extraction (*sous-ch, ch*)

BOOLÉENS

opérations possibles	symbole ou mot clé
comparaison	=, ≠
négation	NON
conjonction	ET
disjonction	OU

INSTRUCTIONS

Une instruction est un ordre que l'on donne à exécuter par l'ordinateur

L'exécution d'un programme consiste à :

- échanger des informations en mémoire
- faire des calculs
- afficher des résultats

INFORMATIONS

Les informations manipulées par les instructions peuvent prendre plusieurs formes:

- des variables
- des constantes
- des valeurs littérales ("bonjour", 45, VRAI)
- des expressions complexes : combinaisons de variables, constantes et valeurs littérales avec des opérateurs

  (2 * r * 3,14)

LES INSTRUCTIONS ÉLÉMENTAIRES

- L'affectation permet de donner une nouvelle valeur à une variable
- L'instruction Saisir permet à l'ordinateur de récupérer ce que l'utilisateur tape au clavier
- L'instruction Afficher permet à l'ordinateur d'afficher sur l'écran ce qu'on souhaite : le résultat d'un calcul, un texte, une ou plusieurs variables …

L' AFFECTATION

Syntaxe :

- Variable ← Valeur
- Variable ← Constante
- Variable1 ← Variable2
- Variable ← expression complexe (calcul par exemple)

EXEMPLES

- X ← Y
- X ← 25
- X ← 3,3
- C ← ‘a’
- maChaine ← «bonjour»
- B ← VRAI
- X ← 25 + Y + 3

EXEMPLES

- Algo : x ← y différent de y ← x
- Maths : x = y équivaut à y = x

- Algo : x + 12 ← y c'est impossible
- Maths : x + 12 = y ceci a un sens mathématique

- Algo : x ← x + 7 a un sens
- Maths : x = x + 7 c'est impossible

LA SAISIE

Syntaxe :

- Saisir (variable)
- Cela permet à un utilisateur de fournir des données au programme
- Cela stocke une valeur entrée au clavier à une variable
- Tant que l'utilisateur ne saisit rien au clavier, le déroulement du programme est stoppé

Exemples

- Saisir (x)
- Saisir (x); Saisir(y)

EXEMPLES

Syntaxe :

- Afficher (variable)

Cela permet d'afficher à l'écran des informations pour l'utilisateur

- Afficher (x)
- Afficher («bonjour»)
- Afficher (x, y, z)
- Afficher (x + y)
- Afficher («le résultat de x + y est : », x + y)

On peut afficher plusieurs éléments à la suite grâce à la virgule.

AVANTAGES DE L'AFFICHAGE

L'affichage permet de :

- fournir un résultat
- de guider l'utilisateur en lui donnant des instructions sur ce qu'il doit faire ou saisir
- de débuguer

EXEMPLE COMPLET

```
ALGORITHME bonjour
  CONSTANTES bj <- «Bonjour» : chaîne
  VARIABLES varNom, varPrénom, ch : chaîne

DEBUT
  Afficher («Quel est votre nom ?»)
  Saisir (varNom)
  Afficher («Quel est votre prénom ?»)
  Saisir (varPrénom)
  ch <- varPrénom & varNom
  Afficher (bj, ch)
FIN
```

LES INSTRUCTIONS

- L'ordre des instructions est primordial
- Le processeur exécute les instructions dans l'ordre dans lequel elles apparaissent dans le programme
- L'exécution est séquentielle : une fois que le programme a fini une instruction, il passe à la suivante
- Tant qu'une instruction n'est pas terminée, il attend avant de continuer (exemple : Saisir)

INSTRUCTIONS CONDITIONNELLES ET BOUCLES

Il peut-être nécessaire pour résoudre un problème

- de n'exécuter les instructions que sous certaines conditions
- de recommencer plusieurs fois les mêmes instructions

Il existe des instructions particulières appelées structures de contrôle qui le permettent :

- instructions conditionnelles : exécuter certaines instructions uniquement sous certaines conditions
- instructions répétitives (boucles) : répéter des instructions un certain nombre de fois (sous certaines conditions)

INSTRUCTIONS CONDITIONNELLES

Elles permettent d'exécuter des instructions différentes en fonction de certaines conditions.

Une condition est évaluée et elle est :

- soit vraie
- soit fausse

Selon le résultat, les instructions à réaliser ne sont pas les mêmes.

Il y a 3 types d'instructions conditionnelles :

- instruction conditionnelle au sens strict : Si … Alors
- instruction alternative : Si … Alors … Sinon
- instruction conditionnelle multiple : Selon … Faire

INSTRUCTION CONDITIONNELLE STRICTE

- Elle permet d'exécuter une ou plusieurs instructions si une condition est respectée, et ne rien faire si la condition est fausse

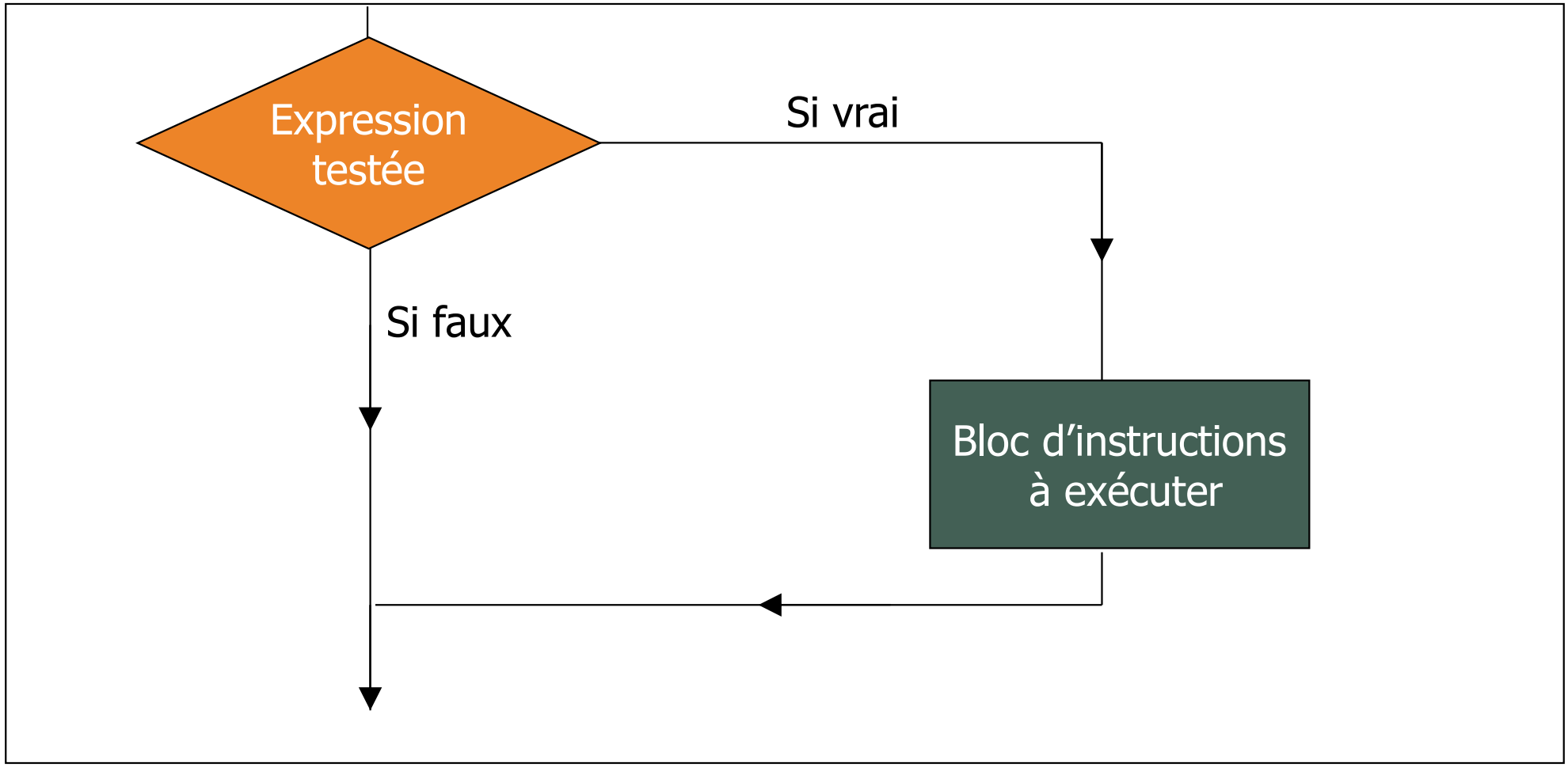
- Une instruction :

```
Si <condition> Alors
    <instruction>
Fin Si
```

- Un bloc d'instructions :

```
Si <condition> Alors
    <instruction1>
    <instruction2>
    <instruction3>
Fin si
```

EXÉCUTION


Expression testée
Si vrai
Si faux
Bloc d'instructions à exécuter

EXEMPLE

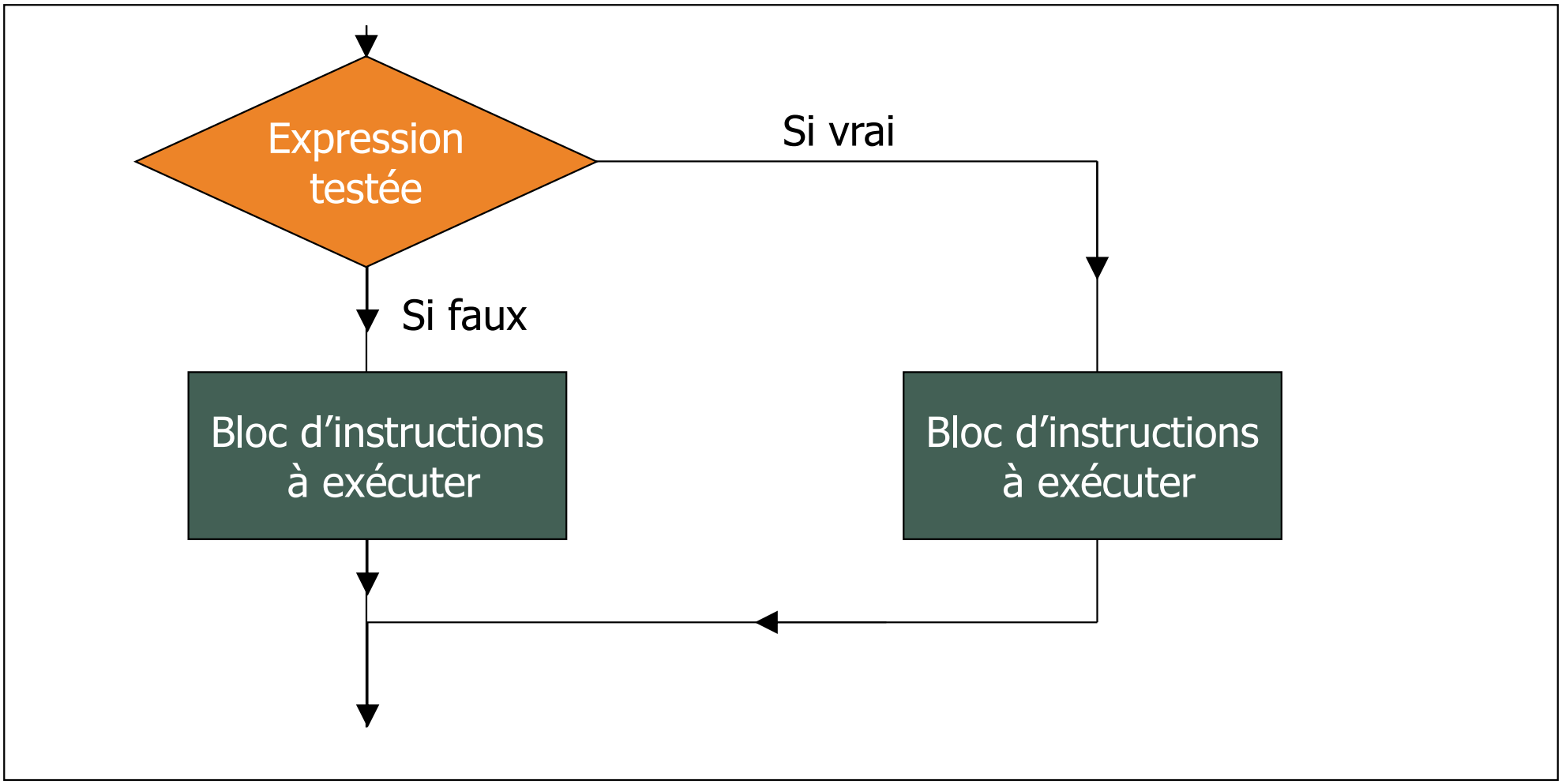
```
ALGORITHME voter
    CONSTANTE majorite ← 18 : entier
    VARIABLE age : entier

DEBUT
    Afficher («Quel est votre âge ?»)
    Saisir (age)
    Si age >= majorité Alors
        Afficher («Vous pouvez voter car vous êtes majeur depuis : », age – 18, « ans »)
    Fin si
FIN
```

INSTRUCTION ALTERNATIVE

- Elle permet de choisir entre deux actions, suivant une condition.
- L'instruction alternative va permettre d'effectuer des choix.

```
Si <condition> Alors
        <instruction1>
        <instruction2>
    Sinon
        <instruction3>
        <instruction4>
Fin si
```

EXÉCUTION


Expression testée
Si vrai
Si faux
Bloc d'instructions à exécuter
Bloc d'instructions à exécuter

EXEMPLE

```
ALGORITHME voter
   CONSTANTE majorite ← 18 : entier
   VARIABLE age : entier

DEBUT
   Afficher («Quel est votre âge ?»)
   Saisir (age)
   Si age ≥ majorité Alors
            (Afficher «Vous pouvez voter car vous êtes majeur depuis : », age – 18, « ans»)
      Sinon
            (Afficher «Vous devez attendre : », 18 - age , « ans avant de voter »)
   Fin si
FIN
```

EXPRESSION CONDITIONNELLE

Une expression conditionnelle (ou expression booléenne) est :

- soit VRAIE
- soit FAUSSE

Il y a plusieurs types :

- condition simple
- condition complexe
- variable booléenne

CONDITION SIMPLE

- Une condition simple est une comparaison de deux expressions de même type

- Les symboles de comparaison sont :
  - $<$
  - $>$
  - $=$
  - $\leq$
  - $\geq$
  - $\neq$

EXEMPLES

Si c = ‘a’

    Alors (Afficher « le caractère est a »)

Fin Si

Si r = 3,3 * x

    Alors (Afficher « l’expression est vraie »)

Fin Si

Si (x – 3 + y) * a ≤ z – 2 + b / 3

    Alors (Afficher « l’expression est vraie »)

Fin Si

CONDITION COMPLEXE

- Une condition complexe est une comparaison formée de plusieurs conditions simples ou variables booléennes reliées entre elles par les opérateurs logiques

- Les opérateurs logiques sont :
  - ET
  - OU
  - NON

EXEMPLES

Si (c1 = ‘a’) ET (c2 = ‘a’)

Alors (Afficher « Les caractères sont a »)

Fin Si

Si (r = 3,3 * x) OU (r = 3,3 * y)

Alors (Afficher « Une expression est vraie »)

Fin Si

Si (((x – 3) * a) ET (z + b / 3)) OU c < 2

Alors (Afficher « Tout est vrai »)

Fin Si

INSTRUCTIONS CONDITIONNELLES IMBRIQUÉES

```
Saisir (x)
Si (x < 0)
|    Alors Afficher (« x est négatif »)
|    Sinon  Si (x < 10)
|                |          Alors Afficher (« x est une unité »)
|                |          Sinon Si (x < 20)
|                |                    |                Alors (Afficher « x est une dizaine »)
|                |                    |                Sinon (Afficher « x ≥ 20 »)
|                |          Fin si
|               Fin si
Fin si
```

INSTRUCTION CONDITIONNELLE MULTIPLE

Elle permet de choisir les instructions à effectuer en fonction de la valeur (ou de l'intervalle de valeur) d'une variable ou d'une expression.

Elle permet de remplacer une succession d'instructions

Si … Alors

SYNTAXE

Selon expression Faire

    valeur 1 : bloc d'instructions 1

    valeur 2 : bloc d'instructions 2

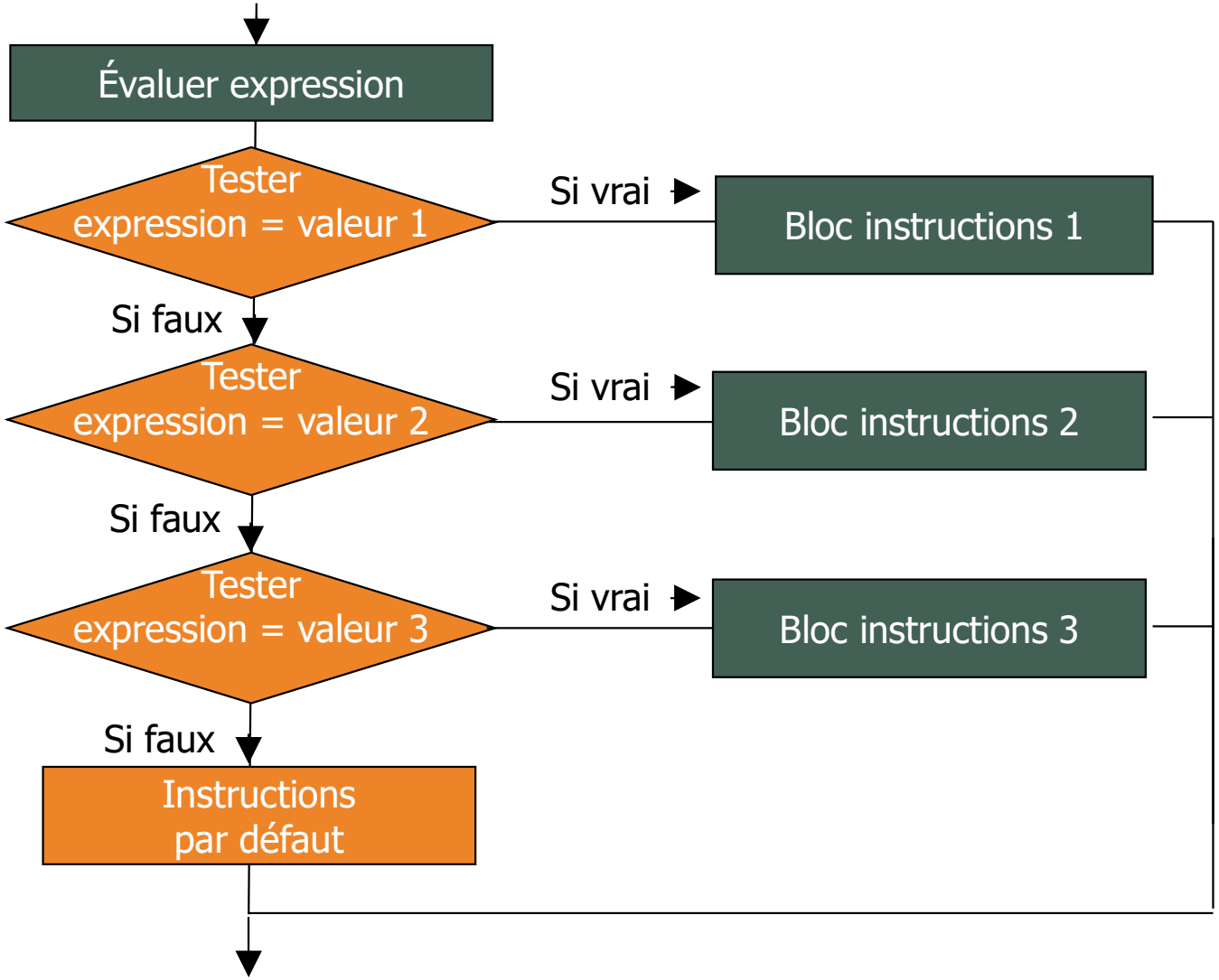
    valeur 3 : bloc d'instructions 3

    …

    [Sinon instructions par défaut]

Fin selon

EXÉCUTION


Évaluer expression
Tester expression = valeur 1
Si vrai
Bloc instructions 1
Si faux
Tester expression = valeur 2
Si vrai
Bloc instructions 2
Si faux
Tester expression = valeur 3
Si vrai
Bloc instructions 3
Si faux
Instructions par défaut

EXEMPLE

Saisir (mois)

Selon mois Faire

1 : Afficher (« Janvier »)

2 : Afficher (« Février »)

3 : Afficher (« Mars »)

4 : Afficher (« Avril »)

…

11: Afficher (« Novembre »)

12: Afficher (« Décembre »)

Sinon (Afficher "Un numéro de mois doit être compris entre 1 et 12 »)

Fin selon

INSTRUCTIONS RÉPÉTITIVES (BOUCLES)

Les boucles permettent de répéter une instruction (ou un bloc d'instructions) autant de fois qu'il est nécessaire :

- soit tant qu'une condition est vraie
- soit un nombre déterminé de fois

Il y a 3 formes de boucles :

- Tant que … Faire
- Pour
- Répéter … Jusqu'à

POURQUOI UTILISER DES BOUCLES ?

Exercice simple : je veux compter jusqu'à 5 et afficher chaque chiffre

Algorithme compter

Variables :

Début

    Afficher (« 1 »)

    Afficher (« 2 »)

    Afficher (« 3 »)

    Afficher (« 4 »)

    Afficher (« 5 »)

Fin

POURQUOI UTILISER DES BOUCLES ?

```
Algorithme compter avec tant que
Variables : i : entier
Début
    i ← 1
    tant que i <= 5 faire
        Afficher (i)
        i ← i + 1
    Fin tant que
Fin
```

POURQUOI UTILISER DES BOUCLES ?

```
Algorithme compter avec pour
Variables : i : entier
Début
    Pour i allant de 1 à 5 par pas de 1 faire
        Afficher (i)
    Fin pour
Fin
```

FORME 1
TANT QUE … FAIRE

Cette forme permet d'exécuter un bloc d'instructions

tant qu'une condition est remplie (satisfaite)

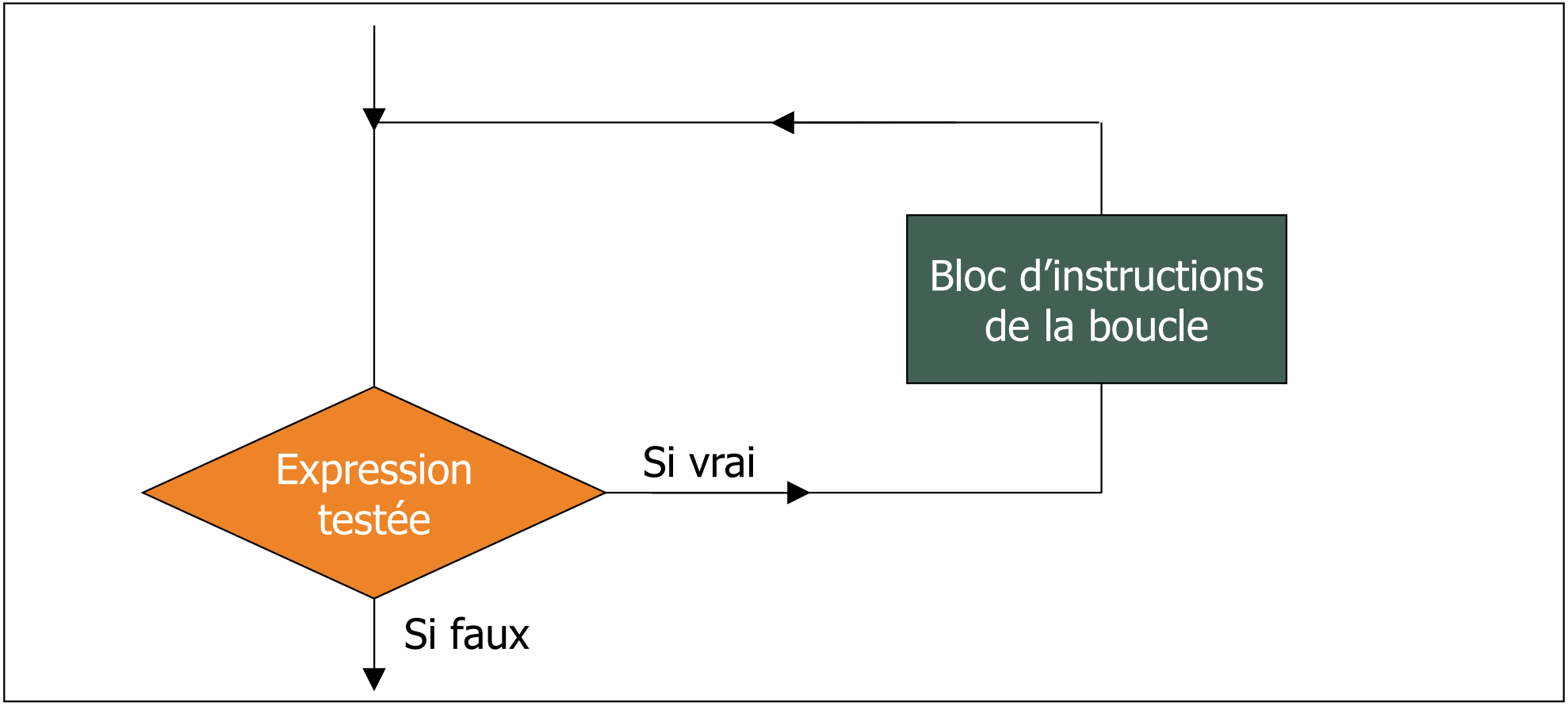
Syntaxe :

Tant que <condition> **Faire**

<instruction>

Fin Tant que

EXÉCUTION


Bloc d'instructions de la boucle
Expression testée
Si vrai
Si faux

FORME 1
EXEMPLE

```
/* gestion d'un budget */
ALGORITHME  forme1

VARIABLES budget, depense : entier

DEBUT
    Afficher (« Quel est le budget de départ ? »)
    Saisir (budget)
    Tant que budget > 0 Faire
        Afficher (« Vous ne pouvez pas dépenser plus de », budget)
        Afficher (« Combien voulez vous dépenser ? »)
        Saisir (depense)
        budget ← budget – depense
    Fin Tant que
    Afficher (« Le budget a été dépassé de: », budget * (-1))
FIN
```

FORME 2
BOUCLE POUR

Cette forme permet de répéter un bloc d'instructions un certain nombre de fois

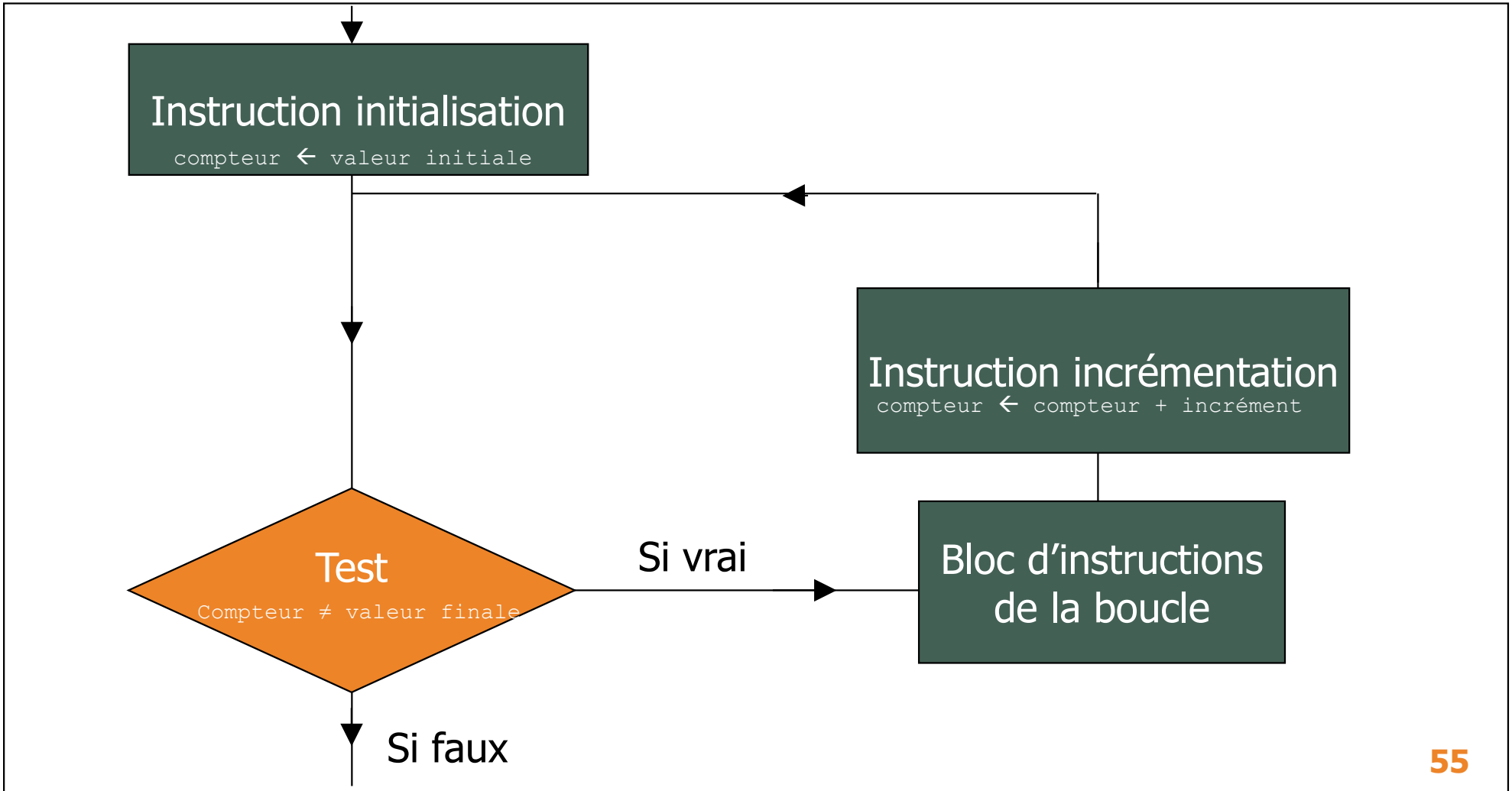
Syntaxe :

Pour <compteur> **allant de** <valeur initiale> **à** <valeur finale> [**par pas de** <incrément>] **Faire**

    <bloc instructions>

Fin Pour

EXÉCUTION


Instruction initialisation
compteur ← valeur initiale
Instruction incrémentation
compteur ← compteur + incrément
Test
Compteur ≠ valeur finale
Si vrai
Bloc d'instructions de la boucle
Si faux

FORME 2
EXEMPLE

```
/* multiplication par 3 de 10 entiers allant de 1 à 10 */

Algorithme forme2
Variable x : entier
Début
    Pour x allant de 1 à 10 par pas de 1 Faire
        Afficher (x, " * 3 = ", x * 3)
    Fin Pour
Fin
```

FORME 3
BOUCLE RÉPÉTER

Cette forme permet de répéter un bloc d'instructions jusqu'à ce qu'une condition soit satisfaite

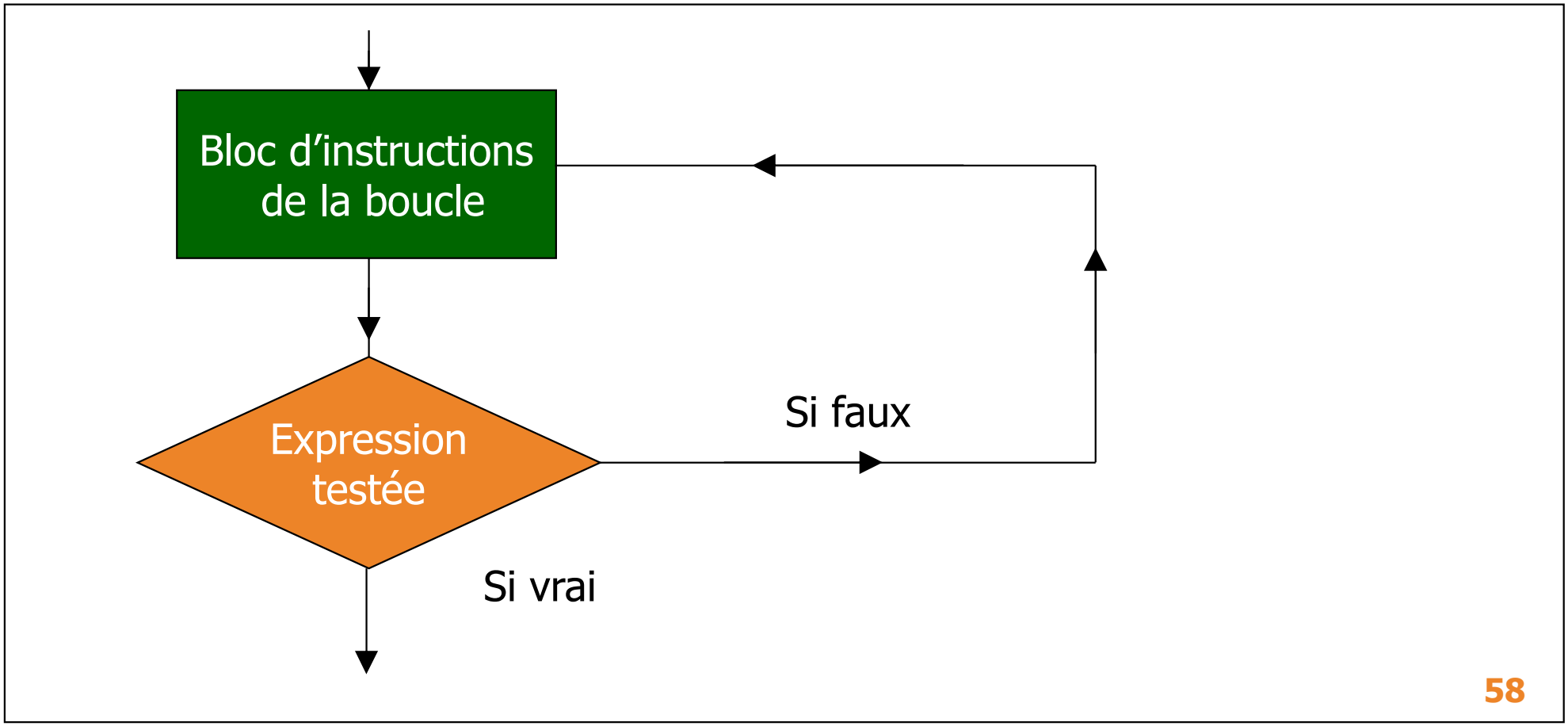
Syntaxe :

Répéter

<*instructions*>

Jusqu'à <condition>

EXÉCUTION


Bloc d'instructions de la boucle
Expression testée
Si faux
Si vrai

FORME 3
EXEMPLE I

```
Algorithme carré
Variables  x, res : entier
           rep : caractère

Début
    Répéter
        Afficher (« Entrez un entier : »)
        Saisir (x)
        Afficher (x, « au carré = », x * x)
        Afficher (« Voulez-vous continuer ? (O/N)»)
        Saisir (rep)
    Jusqu'à (rep = 'N')
    Afficher (« Merci, au revoir »)
Fin
```

FORME 3
EXEMPLE 2

```
Algorithme exposant

Variables x, n, res : entier
          rep : caractère

Début
    Afficher (« Veuillez saisir un entier : »)
    Saisir (x)
    res ← x
    n ← 1
    Répéter
        res ← res * res
        n ← n + 1
        Afficher (x, « puissance », n, « = », res)
        Afficher (« Voulez-vous continuer ? (O/N)»)
        Saisir (rep)
    Jusqu'à (rep = 'N')
    Afficher (« merci, au revoir »)
Fin
```

LE TABLEAU

- Un tableau est une structure contenant plusieurs éléments du même type
- Un tableau est identifié par :
  - un nom
  - une taille
  - le type des éléments qu'il contient
- **TAB[10] tableau d'entiers** : c'est un tableau nommé TAB composé de 10 éléments de type entier
- Un tableau de 10 éléments est plus simple à manipuler que 10 variables séparées. On pourra le parcourir, le remplir, le manipuler **à l'aide de boucles**

LE TABLEAU

- Chaque élément du tableau correspond à un **indice**
- Indice = Rang = Numéro de case

Dans le tableau TAB[10] tableau d'entiers, la première case du tableau a pour indice 0, la seconde a pour indice 1, … et la dernière a pour indice 9

Soit le tableau d'entiers suivant :

TAB[10] = [17, 23, 63, 15, 32, 33, 15, 11, 99, 60]

La première case du tableau a pour indice 0 et son contenu est 17

Dans ce cas on dit que TAB[0] est égal à 17

De même, TAB[1] vaut 23, TAB[2] vaut 63 et TAB[9] vaut 60

BONNES PRATIQUES

- Prendre le **temps de l'analyse** du problème posé
- Adopter une **convention de nommage** pour l'équipe de développement : les variables, les fonctions, les algorithmes
- Bien choisir les **types** et **structures** de données : par exemple utiliser les tableaux à bon escient
- Réfléchir « **modulaire** » : selon la complexité de votre algorithme il est utile de le diviser en **sous-problèmes** plus petits et plus simple à résoudre séparément et faire usage de **fonctions** pour limiter la redondance d'instructions

BONNES PRATIQUES

- **Optimiser** son algorithme pour diminuer :
  - les temps d'exécution
  - l'usage de la mémoire
  - le volume du code pour en faciliter la maintenabilité
- Veiller à la bonne **lisibilité** du code = « **code clair** » :
  - Indentation pertinente
  - Commentaires, documentation

BONNES PRATIQUES

- **Tests :**
  - unitaires
  - fonctionnels
  - d'intégration
  - bout en bout (end to end)
  - automatisés
  - en volume
  - de non-régression

BONNES PRATIQUES : DRY

DRY : éviter les répétitions

" Don't Repeat Yourself "

- On veut éviter la répétition (la redondance) de code
- Il faut donc factoriser notre code en favorisant l'usage de codes internes ou externes **appelables** (fonctions, modules, classes, bibliothèques …)
- On améliore ainsi la lisibilité et la maintenabilité du code
- On évite les modifications à répétition

BONNES PRATIQUES : YAGNI

YAGNI : une approche pragmatique du code

" You Ain't Gonna Need It " → " Tu n'en auras pas besoin "

- On fait ce qui est **strictement nécessaire aujourd'hui**, et pas sur ce qui pourrait l'être dans le futur
- On n'ajoute pas de fonctionnalités **superflues** ou sorties de l'imaginaire parfois trop fertile du développeur
- On ne se substitue pas au client
- On évite donc de rendre le code plus complexe, plus difficile à maintenir, potentiellement plus sensible aux bugs

BONNES PRATIQUES : KISS

KISS : la recherche de la simplicité

" Keep It Simple, Stupid " ou " Keep It Short and Simple "

- Objectif : une simplicité maximale de la conception et de l'écriture des algorithmes et du code
- Les solutions les plus simples sont souvent les meilleures
- Les solutions les plus simples sont souvent les plus solides
- Elles sont plus simples à comprendre, maintenir et faire évoluer