

Données

En algorithmique, toute donnée est définie par

- ▶ son **nom**: désigne la donnée dans l'algorithme,
- ▶ son **type**: désigne le domaine de valeurs de la donnée, et
- ▶ sa **nature**: variable (peut changer de valeur) ou constante (ne peut pas changer de valeur).

Types Simples

Type	Domaine
booléen	$\{faux, vrai\}$
caractère	Symboles typographiques
entier	\mathbb{Z}
réel	\mathbb{R}

Opérateurs

Un opérateur est une *fonction* définie par

- ▶ son **arité**: désigne le nombre de variables d'entrée,
- ▶ sa **position**: l'opérateur peut être préfixe (devant), infixé (milieu) ou postfixé (derrière), et
- ▶ son **type**: désigne le type de ses entrées et celui de sa sortie.

Exemple: L'opérateur d'addition sur les entiers

Opérateur binaire, noté + (position infixé), dont le type est:

$+$: **entier** × **entier** → **entier**

Opérateurs sur les types simples

- ▶ **Arithmétiques (entiers)**: toutes les entrées sont des **entiers** et la sortie est un **entier**.

Nom	Symbole
addition	+
soustraction	−
multiplication	×
division entière	/
reste	mod
inversion de signe	−

- ▶ **Arithmétiques (réels)**: au moins une entrée est un **réel** et la sortie est un **réel**.

Nom	Symbole
addition	+
soustraction	−
multiplication	×
division	/
inversion de signe	−

- ▶ **Comparaisons**: les deux entrées sont des **entiers**, **caractères*** ou **réels**. La sortie est un **booléen**.

Nom	Symbole
est égal à	=
est plus petit que	<
est plus grand que	>
est plus petit ou égal à	≤
est plus grand ou égal à	≥

- ▶ **Logiques**: toutes les entrées sont des **booléens** et la sortie est un **booléen**.

Nom	Symbole
conjonction	et
disjonction	ou
négation	non

Expressions

Une expression est une *composition* d'opérations dont l'ordre est spécifié par les parenthèses. Le **type** d'une expression est donné par le type de sa valeur de sortie

Exemple: Supposons que x, y, z soient des entiers.

- ▶ $(x > 0)$ **et** $(y < 0)$ est une expression booléenne
- ▶ $(x + y)/z$ est une expression entière

Instructions

Une instruction est une *action* à accomplir par l'algorithme. Les quatre instructions de base sont la **déclaration** (mémoire), l'**assignation** (calcul), la **lecture** (entrées) et l'**écriture** (sorties).

Instruction	Spécification
Déclaration	<i>type variable</i>
Assignation	<i>variable</i> ← <i>expression</i>
Lecture	lire <i>variable</i>
Ecriture	écrire <i>expression</i>

Blocs

Un bloc est une séquence d'instructions identifiée par une barre verticale.

Exemple: permutation de valeurs

```
début
  entier a, b, temp
  lire a, b
  temp ← a
  a ← b
  b ← temp
  afficher a, b
```

fin

L'instruction de test "si alors"

Dans l'instruction **si condition alors bloc**, la condition est une expression booléenne, et le bloc n'est exécuté que si la condition est vraie.

Exemple: valeur absolue

```
début
  réel x, y
  lire x
  y ← x
  si y < 0 alors
    | y ← −y
  afficher y
```

fin

L'instruction de test "si alors sinon"

Dans **si condition alors bloc 1 sinon bloc 2**, la condition est une expression booléenne. Le bloc 1 est exécuté si la condition est vraie ; le bloc 2 est exécuté si la condition est fausse.

Exemple: racine carrée

```
début
  réel x, y
  lire x
  si x ≥ 0 alors
    | y ← sqrt(x)
  afficher y
sinon
  | afficher "Valeur indéfinie"
```

fin

L'instruction de test "suivant cas"

Dans l'instruction **suivant condition cas où v_1 bloc 1 cas où v_2 bloc 2 ...**, la condition est une expression pouvant prendre plusieurs valeurs v_1, v_2, \dots . Selon la valeur de la condition, le bloc du cas correspondant est exécuté.

Exemple: choix de menu

```
début
  entier menu
  lire menu
  suivant menu faire
    cas où 1
    | afficher "Menu enfants"
    cas où 2
    | afficher "Menu végétarien"
  autres cas
  | afficher "Menu standard"
```

fin

L'instruction de boucle "pour"

L'instruction *pour* est utilisée lorsque le nombre d'itérations est **connu à l'avance**: elle initialise un **compteur**, l'incrémente après chaque exécution du bloc d'instructions, et vérifie que le compteur ne dépasse pas la borne supérieure.

Exemple: Somme des entiers de 1 à n

```
début
  entier n, s, i
  lire n
  s ← 0
  pour i de 1 à n faire
    | s ← s + i
  afficher s
```

fin

L'instruction de boucle "tant que"

La boucle *tant que* est utilisée lorsque le nombre d'itérations **n'est pas connu à l'avance**: elle exécute le bloc d'instructions tant que la condition reste vraie.

Exemple: Somme des entrées saisies par l'utilisateur (version "tant que")

```
début
  entier n ← 1, s ← 0
  tant que n ≠ 0 faire
    afficher "Entrer un entier (0 pour arrêter) : "
    lire n
    s ← s + n
  afficher s
```

fin

L'instruction de boucle "répéter jusqu'à"

La boucle *répéter jusqu'à* est utilisée lorsque le nombre d'itérations n'est pas connu à l'avance, et qu'il faut lancer **au moins une** exécution du bloc d'instructions. Elle exécute le bloc jusqu'à ce que la condition d'arrêt devienne vraie.

Exemple: Somme des entrées saisies par l'utilisateur (version "répéter jusqu'à")

```
début
  entier n, s ← 0
  répéter
    lire n
    s ← s + n
  jusqu'à n = 0
  afficher s
```

fin

Tableaux Statiques Unidimensionnels

Un *tableau* (statique unidimensionnel) est une séquence de données du même type accessibles par leur index. Il est défini par:

- ▶ son **nom**,
- ▶ le **type** de ses éléments, et
- ▶ sa **taille** ou le nombre de ses éléments.

Le premier index d'un tableau de N éléments est **0** et le dernier index est $N - 1$.

0	1	2	3	4	5	6	7
12	14	16	09	11	10	13	17

Un tableau de 8 entiers

Les seules opérations possibles sont la **déclaration**, l'**initialisation** (déclaration avec valeurs initiales) et l'**accès** à ses éléments.

Opération	Spécification	Exemple
Déclaration	<i>type nom [taille]</i>	entier <i>tab</i> [10]
Initialisation	<i>type nom [n]</i> ← { v_1, \dots, v_n }	caractère <i>voyelles</i> [5] ← {'a','e','i','o','u','y'}
Accès	<i>nom [index]</i>	<i>voyelles</i> [<i>i</i>]

Tableaux Statiques Multidimensionnels

Un *tableau statique de dimension d* est une séquence de tableaux de dimension $d - 1$. En particulier, une *matrice* est un tableau de dimension 2. Comme pour tous les tableaux statiques, les seules opérations possibles sont la **déclaration**, l'**initialisation** et l'**accès** à ses éléments.

Opération	Spécification	Exemple
Décl.	<i>type nom [rangées][colonnes]</i>	réel <i>matrice</i> [4][4]
Init.	<i>type nom [m][n]</i> ← {{ v_{11}, \dots, v_{1n} }, ..., { v_{m1}, \dots, v_{mn} }}	réel <i>unité</i> [2][2] ← {{1, 0}, {0, 1}}
Accès	<i>nom [rangée][colonne]</i>	<i>matrice</i> [<i>i</i>][<i>j</i>]

Tableaux Dynamiques

Un *tableau dynamique (unidimensionnel)* ou *vecteur* est une séquence de données du même type ; la taille de la séquence est **variable** (elle peut changer au cours de l'exécution du programme).

En plus des opérations de tableaux statiques, les vecteurs permettent des opérations de **copie** et de **modification**.

Opération	Spécification	Exemple
Déclaration	vecteur de <i>type nom</i>	vecteur de réels <i>v</i>
Initialisation	vecteur de <i>type nom(quantité,valeur)</i>	vecteur de réels <i>v</i> (10,0)
Copie	<i>nom</i> ₁ ← <i>nom</i> ₂	<i>v</i> ← <i>w</i>
Accès aux éléments	<i>nom[index]</i>	<i>v</i> [<i>i</i>]
Accès à la taille	longueur (<i>nom</i>)	longueur (<i>v</i>)
Test du vecteur vide	vide (<i>nom</i>)	si (vide (<i>v</i>)) alors ...
Ajout d'un élément à la fin	étend	étend (<i>v</i> , 10.0)

Chaînes

Une chaîne est un tableau dynamique unidimensionnel composé de caractères ascii. En plus des opérations de vecteurs, les chaînes permettent des opérations de **comparaison lexicographique**.

Opération	Spécification	Exemple
Déclaration	chaîne <i>nom</i>	chaîne <i>c</i>
Initialisation	chaîne <i>nom</i> ← <i>constante chaîne</i>	chaîne <i>c</i> ← "Bonjour"
Copie	<i>nom</i> ₁ ← <i>nom</i> ₂	<i>c</i> ← <i>d</i>
Accès aux éléments	<i>nom[index]</i>	<i>c</i> [<i>i</i>]
Accès à la taille	longueur (<i>nom</i>)	longueur (<i>c</i>)
Test de la chaîne vide	vide (<i>nom</i>)	si (vide (<i>c</i>)) alors ...
Concaténation	+	<i>c</i> ← <i>c</i> + <i>d</i>
Comparaisons	≤, <, =, ≠, >, ≥	si (<i>c</i> ≠ <i>d</i>) alors ...

Typage

Il est possible de construire de nouveaux types à partir de types prédéfinis en utilisant le mot-clé **type**. En pseudo-code les types apparaissent avant les algorithmes.

Exemple: déclaration d'un type et d'une variable de ce type

type entier MatriceDeRotation [2][2]

```
début
  MatriceDeRotation m
  ...
```

fin

Enumérations

Une *énumération* (ou type énuméré) est un type dont le domaine de valeurs est défini par le programmeur.

Opération	Spécification	Exemple
Déclaration de type	énumération <i>Nom</i> { v_1, \dots, v_n }	énumération <i>Couleurs</i> {r,v,b}
Déclaration de variable	<i>Nom variable</i>	<i>Couleurs c</i>
Copie	<i>variable_enum</i> ← <i>donnée_enum</i>	<i>c</i> ← r
Conversion	<i>variable_enum</i> ← (<i>Nom</i>) <i>donnée.entière</i>	<i>c</i> ← (<i>Couleurs</i>)2
Comparaisons	≤, <, =, ≠, >, ≥	si (<i>c</i> = r) alors ...

Structures

Une *structure* est un type composite formé par plusieurs types groupés ensembles.

Opération	Spécification	Exemple
Décl. de type	structure <i>Nom</i> { <i>Type_1 nom_1</i> , ..., <i>Type_k nom_k</i> }	structure <i>Point</i> {réel <i>x</i> , réel <i>y</i> }
Décl. de variable	<i>Nom variable</i>	<i>Point p</i>
Copie	<i>variable</i> ₁ ← <i>variable</i> ₂	<i>p</i> ← <i>q</i>
Accès	.	<i>p.x</i>

Exemple: déclaration d'un tableau de structures

structure Point

```
réel x
réel y
```

type Point Figure [100]

```
début
  Point p
  afficher p.x // Affiche l'abscisse du point p
```

```

  Figure f
  afficher f[0].x // Affiche l'abscisse du premier point de la figure f
```

fin

Fonctions

En algorithmique, une *fonction* est définie par deux parties:

- ▶ Une **en-tête**: elle spécifie le type de la fonction, c'est à dire, le type de ses données d'entrées et le type de sa valeur de sortie.
- ▶ Un **corps**: il spécifie l'algorithme permettant de passer des données d'entrée à la valeur de sortie.

La **déclaration** d'une fonction consiste à spécifier seulement l'en-tête de la fonction. La **définition** d'une fonction consiste à spécifier à la fois l'en-tête et le corps de la fonction.

Exemple: définition de la fonction factorielle

réel fact(**entier** *n*)

```
début
  entier i
  réel f ← 1
  pour i de 1 à n faire
    f ← f * i
  retourner f
```

fin

Algorithmes de construction

Les algorithmes permettant de construire des ensembles (tableaux, chaînes, etc.) utilisent des boucles "pour": il faut construire tous les éléments de l'ensemble.

Exemple: addition de deux matrices

type réel Matrice [4][4]

Matrice addition(**Matrice** *A*, **Matrice** *B*)

```
début
  Matrice C
  entier i
  pour i de 0 à 3 faire
    C[i] ← A[i] + B[i]
  retourner C
```

fin

Algorithmes de recherche

Les algorithmes permettant de rechercher un objet dans un ensemble utilisent des boucles "tant que": la recherche s'arrête dès que l'élément est trouvé

Exemple: recherche un entier dans un vecteur non trié; si la valeur recherchée est présente alors l'algorithme retourne son index, sinon il retourne la taille du vecteur

entier rechercher(**vecteur d'entiers** *tab*, **entier** *x*)

```
début
  booléen trouvé ← faux
  entier i ← 0
  tant que (i < longueur(tab)) et (non trouvé) faire
    trouvé ← tab[i] = x
    si non trouvé alors i ← i + 1
  retourner i
```

fin

Algorithmes de recherche dichotomique

Les algorithmes permettant de rechercher un objet dans un ensemble trié peuvent exploiter la dichotomie (beaucoup plus rapide).

entier dichotomie(**vecteur d'entiers** *tab*, **entier** *x*)

```
début
  entier milieu, gauche ← 0, droite ← longueur(tab)
  répéter
    milieu ← (gauche + droite)/2
    si x < tab[milieu] alors droite ← milieu - 1
    si x > tab[milieu] alors gauche ← milieu + 1
  jusqu'à (gauche > droite) ou (tab[milieu] = x)
  si gauche > droite alors milieu ← longueur(tab)
  retourner milieu
```

fin

Algorithmes de tri

Les algorithmes de tri simple (insertion, sélection) utilisent deux boucles "pour"

triParSelection(**vecteur d'entiers** *tab*)

```
début
  entier i, j
  pour i de 0 à longueur(tab) - 2 faire
    pour j de i + 1 à longueur(tab) - 1 faire
      si tab[j] < tab[i] alors
        permuter(tab[i], tab[j])
```

fin