Sommaire de la deuxième partie

Algorithmes impératifs

Il existe un autre style que la récursivité pour l'écriture des algorithmes : le style impératif.
Dans ce style la résolution d'un problème est décrit comme un enchaînement d'actions à exécuter les unes après les autres.

Exemple

Étant donné 2 nombres `a` et `b`, pour calculer le PGCD de `a` et `b` on pourrait procéder de la façon suivante :

1. On calcule `lDiva` la liste des diviseurs de `a`
2. On calcule `lDivb` la liste des diviseurs de `b`
3. On calcule `plusGrandCommun` le plus grand nombre appartenant à la fois à la liste `lDiva` et à la liste `lDivb`.

Le résultat (le PGCD de `a` et `b`) est alors la valeur de `plusGrandCommun`.

Nouvelles notions

Pour exprimer un algorithme en style impératif nous devons ajouter des nouveaux éléments au langage algorithmique :

- Le **corps d'un algorithme** est composé de 2 parties :
  - une **partie instruction** qui définit un ensemble d'actions à exécuter
  - une **partie résultat** qui définit la valeur du résultat calculée par l'algorithme
- La partie instruction est composée **d'instructions élémentaires**, que l'on appelle **affectations** ; une affectation consiste à calculer la valeur d'une expression et à affecter cette valeur à une **variable**.
- Il existe plusieurs façons de composer les instructions : en séquence, selon une condition, en répétition.

Variables, Environnement et Expression

Définition (Variable)

Une variable permet de mémoriser le résultat intermédiaire d'un calcul.
Une variable a un nom, un type, et éventuellement une valeur qui peut varier au cours de l'exécution de l'algorithme.
Les variables qui apparaissent dans le corps d'un algorithme doivent être déclarées au début du corps de l'algorithme. La déclaration d'une variable définit le nom et le type de la variable et s'écrit `nom : type`
Ex : **Variable :** `a: Nombre, b: Booléen, l: Liste de Nombres`
Lorsqu'une variable est déclarée on définit son nom et son type, pas sa valeur. La valeur d'une variable est définie/modifiée par une instruction d'affectation.

Définition (Environnement)

- On appelle **environnement** (de variables) un ensemble d'associations `nom-valeur`. Les noms sont les noms des variables déclarées.
- Un **environnement peut être modifié** en affectant une valeur à une variable.

Définition (Expression)

On ajoute une nouvelle forme d'expression. Une expression peut être :

- comme jusqu'à présent, une constante ou une expression conditionnelle ou l'application d'une opération, d'une fonction ou d'un algorithme
- **ou le nom d'une variable** déclarée.

Définition (Type d'une expression)

Si l'expression est le nom d'une variable, son type est celui défini lors de la déclaration de la variable.

Définition (Valeur d'une expression)

La valeur d'une expression est définie **dans un environnement**. Si l'expression est le nom d'une variable, sa valeur dans un environnement est :

- la valeur associée au nom de la variable dans l'environnement
- si le nom de la variable ou la valeur associée à ce nom est absent de l'environnement, il y a une erreur, l'expression n'a pas de valeur.

Le mécanisme d'évaluation des autres formes d'expression n'est pas modifié.

Exemple

Soit l'environnement composé des 4 variables `a, som, y, x`. Les variables `a, som, x` sont de type `Nombre`, la variable `y` est de type `Booléen`. Dans cet environnement, les variables `a, som, y` ont pour valeurs respectives 5, 9, true et la variable `x` n'a pas de valeur.
On représente cet environnement par le tableau

`Val(a)`	`Val(som)`	`Val(y)`	`Val(x)`
5	9	`true`	?

Dans cet environnement :

`exp`	Type de `exp`	`Val(exp)`
max((15-a),som)		
(3 < 8) et ((1 + a) = 7)		
y et ((2 * a) < 12)		
1 + x		
1 + y		
cond((2 *a) < som , a + 1 , som + 2)		

Affectation

Définition (Affectation)

La syntaxe d'une affectation est `nomVariable := expression`

Contraintes de type

La variable doit avoir été déclarée au préalable.
Le type de l'expression doit être correct et correspondre à celui de la variable.

Définition (Effets d'une affectation)

Les actions réalisées lors de son exécution sont :

1. On évalue `expression` dans l'environnement courant.
2. Si cette évaluation provoque une `Erreur`, l'affectation n'est pas exécutée. Sinon, soit `E` la valeur `Val(expression)`
3. L'environnement est modifié : la valeur associée à la variable devient `E`. La valeur associée aux autres variables n'est pas modifiée.

Remarque

Une affectation n'a pas de valeur. Son effet est de modifier l'environnement.

Exemple

`a` et `b` sont deux variables déclarées de type `Nombre`.

Environnement avant Val(a)	Val(b)	Affectation	Environnement après Val(a)	Val(b)
?	10	`a:= 4`		
2	10	`a:= (a + 1)`		
5	10	`a:= (b+2)`		
?	10	`a:= (a+1)`		
?	10	`a:= (b+2)`		
2	10	`a:= ((a+2)*b)`		
2	10	`a:= ((a et 2)*b)`		
2	10	`a:= ( a< b )`		

Algorithme impératif

Schéma d'un algorithme avec instructions

Un algorithme est constitué d'une partie **spécifications** et du **corps de l'algorithme**. Dans le corps de l'algorithme, la partie résultat peut être précédée d'une partie **déclaration des variables** et d'une partie **instruction**.

D'où le schéma :

Algorithme : nom de l'algorithme
Données : description des paramètres
Résultat : description du résultat

Variable : Déclaration des variables
Partie Instruction ;
Le résultat est : `Expression`
fin algorithme

La définition d'un algorithme doit être correcte du point de vue des types. Les erreurs de type qui peuvent se produire sont :

- Erreur de type d'une expression
- Erreur de type lors d'une affectation : le type de l'expression ne correspond pas au type de la variable affectée
- Erreur de type du résultat : le type de l'expression résultat ne correspond pas au type du résultat déclaré dans la partie spécification.

Application d'un algorithme

L'application d'un algorithme est une expression de la forme `nomAlgo(e1,...,en)` dont la valeur est calculée comme suit :

1. On évalue chaque expression `v1=Val(e1),... vn=Val(en)`
2. On substitue dans la partie instruction et dans l'expression résultat chaque `vi` au paramètre correspondant de l'algorithme.
3. On exécute la partie instruction ; l'environnement initial est constitué des variables déclarées ; aucune valeur ne leur est associée.
4. On évalue l'expression de la partie résultat dans l'environnement obtenu à la fin de l'exécution des instructions.
5. La valeur de `nomAlgo(e1,...,en)` est la valeur de cette expression

Si une erreur se produit lors de l'évaluation d'une expression, l'algorithme s'arrête : `nomAlgo(e1,...,en)` n'a pas de valeur et une erreur est indiquée.

Algorithme : monAlgo
Données : x : Nombre
Résultat : Nombre ..

Variable : v : Nombre
v := 2 * x + 1;
Le résultat est : `v*x`
fin algorithme

valeur de `monAlgo(4)`

1. substitution :
2. exécution de la partie instruction
3. évaluation de l'expression résultat :

Exécution de l'instruction

	Val(v)
avant instruction	
après instruction	

Ne pas confondre variable et paramètre

Lors de l'application d'un algorithme, les valeurs des paramètres ne varient pas. On ne peut pas leur affecter de nouvelles valeurs.

Algorithme : algoIncorrect
Données : p : Nombre
Résultat : Nombre ...

    p := 2 * p ;
    Le résultat est : `p`
fin algorithme

Cet algorithme est incorrect car, par exemple, lors de l'application de `algoIncorrect(3)`, `p` est remplacé par 3. On obtient alors le corps :

qui n'a pas de sens.

Séquence d'instructions

Dans la partie instruction d'un algorithme l'instruction de base est l'affectation.
On peut composer ces instructions pour définir de nouvelles instructions.
Il existe plusieurs façons de composer des instructions :
la séquence d'instructions, l'instruction conditionnelle, l'instruction itérative.

Définition (La séquence)

La séquence d'instructions `Inst1,Inst2,...,Instn` s'écrit

`Inst1; Inst2; ...; Instn`

L'exécution de cette instruction a pour effet d'exécuter `Inst1`, puis d'exécuter `Inst2` ..., enfin d'exécuter `Instn`.
Si l'exécution d'une instruction `Insti` provoque une erreur, aucune autre instruction n'est exécutée.

Exemple (Exécution d'une séquence d'instructions)

```
1 Variable s : Nombre, a : Nombre ;
2 s := 1;
3 a := 2;
4 s := s + a ;
5 a := a + 2 ;
6 s := s + a ;
```

En fin de ligne	Val(s)	Val(a)
1		
2		
3		
4		
5		
6		

Définition (Initialisation de variable)

La première affectation d'une variable est appelée son **initialisation**.
Dans l'exemple précédent `s := 1` et `a := 2` sont les initialisations des variables `s` et `a`.
On ne doit pas utiliser une variable dans une expression avant de l'avoir initialisée.

Les instructions conditionnelles

Il existe 2 formes d'instructions conditionnelles :

L'instruction *Si_simple*

si `ExpB` **alors**
    Inst
fin si

où `ExpB` est une expression de type booléen et `Inst` est une instruction.

Lors de son exécution, l'expression `ExpB` est évaluée. Si `Val(ExpB)=true`, l'instruction `Inst` est exécutée, sinon rien n'est exécuté.

L'instruction *Si_alors_sinon*

si `ExpB` **alors**
    Inst1
sinon
    Inst2
fin si

où `ExpB` est une expression de type booléen, `Inst1` et `Inst2` sont 2 instructions.

Lors de son exécution, l'expression `ExpB` est évaluée. Si `Val(ExpB)=true`, l'instruction `Inst1` est exécutée, sinon l'instruction `Inst2` est exécutée.

Exemples d'algorithmes avec instructions conditionnelles

Exemple (Si Alors simple)

Il s'agit de calculer le montant d'une commande d'un ensemble d'articles identiques connaissant le nombre d'articles et le prix unitaire d'un article.
Une réduction de 10% est appliquée lorsque le nombre d'articles dépasse 10.

Algorithme : montantCommande
Données : `nbArticle` : Nombre, `prixUnitaire` : Nombre
Résultat : Nombre, montant de la commande
Variable : montant : Nombre

Le résultat est : `montant`
fin algorithme

Ne pas confondre **instruction conditionnelle** et **expression conditionnelle**.

- L'expression conditionnelle est une expression. Elle a une valeur (et un type) et peut être composée avec d'autres expressions.
- L'instruction conditionnelle est une instruction. Elle n'a pas de valeur. Lors de son exécution elle a pour effet de modifier l'environnement (la valeur de certaines variables). Elle peut être mise en séquence avec d'autres instructions ou apparaître dans une autre instruction conditionnelle.

Exemple (montant d'une commande (2ème énoncé))

Cette fois-ci deux taux de réduction peuvent être appliqués :
15% si le nombre d'articles dépasse 20, 10% s'il est entre 10 et 20.

Algorithme : commande2
Données : `nbArt` : Nombre, `prixU` : Nombre
Résultat : Nombre
Variable : tauxReduc : Nombre

1

Le résultat est : `nbArt * prixU * (1 - tauxReduc)`
fin algorithme

Algorithme : commande2
Données : `nbArt` : Nombre,
`prixU` : Nombre
Résultat : Nombre
Variable : tauxReduc : Nombre

1
2 tauxReduc := 0;
si `nbArt > 10` **alors**
si `nbArt > 20` **alors**
3 tauxReduc := 0.15
sinon
4 tauxReduc := 0.10
fin si;
5 **fin si;**
Le résultat est :
`nbArt * prixU * (1 - tauxReduc)`
fin algorithme

`commande2(20,3)`

En fin de ligne	Val(tauxReduc)
1	
2	

`commande2(25,4)`

En fin de ligne	Val(tauxReduc)
1	
2	

Exemple (montant d'une commande ($2^{ème}$ énoncé))

Version équivalente avec une expression conditionnelle et sans instruction :

Algorithme : commande2
Données : `nbArt` : Nombre, `prixU` : Nombre
Résultat : Nombre, montant de la commande
Le résultat est :

```
cond( nbArt > 20, nbArt * prixU * 0.85,
cond( nbArt > 10, nbArt * prixU * 0.9,
nbArt * prixU) )
```

fin algorithme

Les instructions itératives

Exemple initial :

Algorithme : multiplierPar4
Données : n : Nombre
Résultat : Nombre, $4 \times n$, sans utiliser la multiplication.

solution 1

Le résultat est : `n+n+n+n`

Non généralisable.

solution 2

Variable : s : Nombre
s := 0;
s := s + n;
s := s + n;
s := s + n;
s := s + n;
Le résultat est : `s`

Non généralisable et lourd (multiplication par 100).

solution 3

Variable : s : Nombre
s := 0;
Iterer `4 fois`
s := s + n
fin itérer ;
Le résultat est : `s`

Généralisable si le nombre d'itérations est paramétré.

Itération Pour

Définition (*Itération Pour* – première forme)

L'instruction itérative `Pour` s'écrit :

pour `I de E1 à E2` **faire**
Inst
fin pour;

où `E1` et `E2` sont deux expressions de type nombre **entier**.
`I` est un nom. `I` est appelé **indice de boucle**

Exécution d'une *Itération Pour*

1. On évalue les expressions `E1` et `E2` (soit `V1=Val(E1)` et `V2=Val(E2)`).
2. L'indice de boucle `I` prend la valeur `V1`
3. On compare la valeur de l'indice de boucle à `V2`.
   Si la valeur de `I` est strictement supérieure à `V2`, l'itération s'arrête.
   Sinon (la valeur de `I` est inférieure ou égale à `V2`) :
   a) On exécute l'instruction `Inst`
   b) On augmente de 1 la valeur de l'indice de boucle `I`
   c) On recommence en 3.


```
Algorithme : multiplierPar4
Données : n : Nombre
Résultat : Nombre, 4 × n
Variable : s : Nombre
1
2     s := 0;
3     pour i de 1 à 4 faire
4         s := s + n
5     fin pour;
      Le résultat est : s
fin algorithme
```

Exécution de `multiplierPar4(3)`

En fin de ligne	Val(i)	Val(s)
1		
2		

Algorithme : multiplier
Données : `a` : Nombre, `b` : Nombre ; `b`$\in \mathbb{N}$
Résultat : Nombre, $a \times b$
Variable : s : Nombre

Le résultat est : `s`
fin algorithme

Exécution de `multiplier(2,3)`
Il y a d'abord substitution :
`a` est remplacé par 2, et `b` par 3 dans le corps, ce qui donne :

s := 0;
pour `i de 1 à 3` **faire**
  s := s + 2
fin pour;
Le résultat est : `s`


```
1
2 s := 0;
3 pour i de 1 à 3 faire
4     s := s + 2
5 fin pour;
  Le résultat est : s
```

Exécution de `multiplier(2,3)`

En fin de ligne	Val(i)	Val(s)
1		
2		

Convention.

Pour éviter toute confusion, on utilisera pour l'indice de boucle un nom différent de ceux des paramètres et des variables de l'algorithme.

Remarque

- Les expressions `E1` et `E2` sont évaluées une fois pour toute avant la première itération : les valeurs `V1` et `V2` ne varient pas au cours de l'exécution de l'itération.
- L'indice de boucle ne doit pas être déclaré.
- L'indice de boucle n'existe pas en dehors de l'itération `Pour`.
- L'indice de boucle peut être utilisé dans l'instruction itérée : il peut apparaître dans une expression. Par contre sa valeur ne peut pas être modifiée.

Exemple d'itération utilisant l'indice de boucle

Algorithme : somEntiers
Données : `n` : Nombre ; `n`$\in \mathbb{N}$
Résultat : Nombre, $\sum_{i=1}^{n} i$
Variable : som : Nombre

Le résultat est : `som`
fin algorithme

Exécution de `somEntiers(3)`

En fin de ligne	Val(i)	Val(som)
1		
2		

Définition (Itération pour – deuxième forme)

La deuxième forme d'instruction `Pour` s'écrit :

pour `I de E1 bas E2` **faire**
    Inst
fin pour;

où `I` est l'indice de boucle, `E1` et `E2` 2 expressions de type Nombre entier.

Exécution d'une *Itération Pour Bas*

1. On évalue les expressions `E1` et `E2` (soit `V1=Val(E1)` et `V2=Val(E2)`).
2. L'indice de boucle `I` prend la valeur `V1`
3. On compare la valeur de l'indice de boucle à `V2`.
   Si la valeur de `I` est strictement inférieure à `V2`, l'itération s'arrête.
   Sinon (la valeur de `I` est supérieure ou égale à `V2`) :
   a) On exécute l'instruction `Inst`
   b) On diminue de 1 la valeur de l'indice de boucle `I`
   c) On recommence en 3.

Itération sur les listes

Algorithme : liEntiers
Données : `n` : Nombre ; $n \in \mathbb{N}$
Résultat : `Liste de Nombre`, la liste croissante des `n` premiers entiers non nuls
Variable : li : Liste de Nombre
1

Le résultat est : `li`
fin algorithme

Exécution de `liEntiers(3)`

En fin de ligne	Val(i)	Val(li)
1		
2		

Itération *Tant que*

Les itérations `Pour` nécessitent de connaître le nombre de fois qu'il faut itérer. Dans certains cas ce nombre n'est pas connu a priori. On utilise alors une seconde forme d'itération.

Définition (Itération *Tant que*)

L'instruction itérative `Tant que` s'écrit :

tant que `ExpB` **faire**
    Inst
fin tq;

où `ExpB` est une expression de type booléen.

Exécution d'une itération *Tant que*

L'exécution d'une itération `Tant que` revient à :

1. évaluer `ExpB`. Soit `B` cette valeur.
2. si `B` est `false` l'itération s'arrête
3. sinon exécuter l'instruction `Inst` et recommencer en 1

Remarque

- Contrairement à l'itération `Pour`, le nombre d'itérations de l'itération `Tant que` dépend de l'instruction itérée.
- Dans l'itération `Tant que` l'instruction itérée doit modifier la valeur d'une variable figurant dans l'expression `ExpB`.
- L'exécution d'une itération `Tant que` peut ne pas s'arrêter !
- L'itération `Tant que` est plus générale que l'itération `Pour` : tout algorithme utilisant une itération `Pour` peut être réécrit en un algorithme utilisant une itération `Tant que` (voir TD). La réciproque est fausse.

Question : le nombre entier positif `p` est-il une puissance de 2 ?

Une méthode consiste à diviser `p` par 2 tant qu'il est pair.
Combien de fois doit-on le diviser par 2 ? On ne sait pas a priori.
On ne peut donc pas utiliser l'itération *Pour*. On utilise l'itération *Tant que*.

Tentative d'algorithme

Algorithme : puissance2
Données : `n` : Nombre ; $n \in \mathbb{N}$
Résultat : Booléen, `true` si `n` est une puissance de 2, `false` sinon
Variable : p : Nombre

```
1
2     p := n ;
3     tant que estPair(p) faire
4         p := p quo 2
5     fin tq;
      Le résultat est : ????
fin algorithme
```

Exécution de `puissance2(24)`

En fin de ligne	Val(p)	Val(estPair(p))

Quel est le résultat et pourquoi ?

Tentative d'algorithme

Algorithme : puissance2
Données : `n` : Nombre ; $n \in \mathbb{N}$
Résultat : Booléen, `true` si `n` est une puissance de 2, `false` sinon
Variable : p : Nombre

```
1
2     p := n ;
3     tant que estPair(p) faire
4         p := p quo 2
5     fin tq;
      Le résultat est : ????
fin algorithme
```

Exécution de `puissance2(16)`

En fin de ligne	Val(p)	Val(estPair(p))

Quel est le résultat et pourquoi ?

L'algorithme

Algorithme : puissance2
Données : `n` : Nombre ; $n \in \mathbb{N}^*$
Résultat : Booléen, `true` si `n` est une puissance de 2, `false` sinon
Variable : p : Nombre

```
1
2     p := n;
3     tant que estPair(p) faire
4         p := p quo 2
5     fin tq;
      Le résultat est :
fin algorithme
```

Exécution de `puissance2(6)`

En fin de ligne	Val(p)	Val(estPair(p))

Algorithmes itératifs sur les nombres : quelle itération ?

Compter dans un intervalle

Étant donnés 2 nombres entiers `a` et `b`, il s'agit de compter le nombre de nombres premiers dans l'intervalle `[a,b]`. On suppose disposer de l'algorithme `estPremier` testant si un nombre est premier.
On doit tester tous les éléments de l'intervalle `[a,b]`. On peut donc utiliser l'itération

Algorithme : nbPremier
Données : `a` : Nombre, `b` : Nombre ; $a, b \in \mathbb{N}$
Résultat : Nombre, le nombre de nombres premiers dans l'intervalle `[a,b]`.
Variable : cpt : Nombre

```
    cpt := 0;



    Le résultat est : cpt
fin algorithme
```

Tester l'existence d'un élément dans un intervalle

On cherche à tester si il existe au moins un nombre premier dans un intervalle.
Premier algorithme : on teste tous les éléments de l'intervalle `[a,b]`. On utilise pour cela l'itération

Algorithme : existePremier
Données : `a` : Nombre, `b` : Nombre ; `a,b`$\in \mathbb{N}$
Résultat : Booléen, `true` si l'intervalle `[a,b]` contient au moins un nombre premier, `false` sinon.
Variable : trouve : Booléen

trouve := false ;

Le résultat est : `trouve`
fin algorithme

Tester l'existence d'un élément dans un intervalle

Deuxième algorithme : on s'arrête au premier nombre premier trouvé.
On ne connaît pas a priori le nombre d'itérations.
On utilise donc l'itération

Algorithme : existePremier2
Données : `a` : Nombre, `b` : Nombre ; `a,b`$\in \mathbb{N}$
Résultat : Booléen, `true` si l'intervalle `[a,b]` contient au moins un nombre premier, `false` sinon.
Variable : trouve : Booléen, i : Nombre

trouve := false ;

Le résultat est : `trouve`
fin algorithme

Faire la somme d'éléments dans un intervalle

On veut calculer la somme des nombres premiers appartenant à un intervalle.
On doit tester tous les éléments de l'intervalle `[a,b]`. On utilise donc l'itération

Algorithme : sommePremier
Données : `a` : Nombre, `b` : Nombre ; `a,b`$\in \mathbb{N}$
Résultat : Nombre, la somme des nombres premiers appartenant à l'intervalle `[a,b]`.
Variable : som : Nombre

som := 0;

Le résultat est : `som`
fin algorithme

Rechercher le $n^{ième}$ élément vérifiant une propriété

On recherche le $n^{ième}$ nombre premier (ce nombre existe $\forall n > 0$).
On doit parcourir les entiers à partir de 0 et compter le nombre de nombres premiers rencontrés.
Combien de fois faut-il itérer ? On ne sait pas a priori. Donc on utilise l'itération

Algorithme : niemePremier
Données : `n` : Nombre, `n`$\in \mathbb{N}^*$
Résultat : Nombre, le $n^{ième}$ nombre premier.
Variable : cpt : Nombre, i : Nombre

cpt := 0 ; i := 0;

Le résultat est :
fin algorithme

Variables et affectation en C/C++

Variables

En C/C++ la déclaration d'une variable s'écrit

Type `nomVariable` **;**

Les déclarations de plusieurs variables de même type peuvent être regroupées :

Type `nomVariable1, nomVariable2, ..., nomVariablek` **;**

Affectation

L'instruction d'affectation est notée

`nomVariable = expression ;`

Exemple

Langage d'algorithmes

Variable : x : Nombre, y : Nombre
x := 2 ; y :=5 ;

C/C++

```
int x, y ;
x = 2 ; y = 5 ;
```


Algorithme avec instructions en C/C++

Langage d'algorithme

Algorithme : monAlgo
Données : x : Nombre
Résultat : Nombre
Variable : v : Nombre

v := 1;
v := 2 * x + v;
Le résultat est : `v * x`
fin algorithme

Langage C/C++

```
int monAlgo(int x)
{
  int v;
  v = 1;
  v = 2*x + v;
  return v*x;
}
```

Exemple

```
EVAL( monAlgo(4) );
   =>  Valeur de monAlgo(4) : 36
```


Instructions conditionnelles en C/C++

Traduction de l'instruction *Si_simple*

langage d'algorithmes

si `ExpB` **alors**
  Inst
fin si

s'écrit

C/C++

if (*ExpB* **)**
  Inst

Traduction de l'instruction *Si_alors_sinon*

langage d'algorithmes

si `ExpB` **alors**
  Inst1
sinon
  Inst2
fin si

s'écrit

C/C++

if (*ExpB* **)**
  Inst1
else
  Inst2

Langage d'algorithme

Algorithme : commande2
Données : `nbA` : Nombre,
`prix` : Nombre
Résultat : Nombre, montant de la commande
Variable : reduc :Nombre

reduc := 0;
si `nbA > 20` **alors**
  reduc := 0.15
sinon
  si `nbA > 10` **alors**
  reduc := 0.1
  fin si
fin si;
Le résultat est :
`nbA*prix*(1-reduc)`
fin algorithme

C/C++

```
float commande2(int nba,
                float prix)
{
  float reduc;
  reduc = 0;









}
```

```
EVAL( commande2(50,2) );
=> valeur de commande2(50,2):85
```

Attention

En C/C++ la fin des instructions conditionnelles n'est pas marquée.

C/C++

```
if ( ExpB )
    Inst1 ;
    Inst2 ;
```

correspond à

langage d'algorithmes

```
si ExpB alors
    Inst1
fin si;
Inst2 ;
```

C/C++

```
if ( ExpB )
    Inst1 ;
else
    Inst2 ;
    Inst3 ;
```

correspond à

langage d'algorithmes

```
si ExpB alors
    Inst1
sinon
    Inst2
fin si;
Inst3 ;
```


Bloc d'instructions

Lorsque plusieurs instructions figurent dans la partie *alors* ou bien dans la partie *sinon* d'une instruction conditionnelle, il faut en C/C++ former un **bloc d'instructions** en encadrant les instructions par une paire d'accolades { et }.

langage d'algorithmes

```
si ExpB alors
    Inst1 ; Inst2
fin si
```

s'écrit

C/C++

```
if ( ExpB )
  {
    Inst1 ; Inst2 ;
  }
```

langage d'algorithmes

```
si ExpB alors
    Inst1
sinon
    Inst2 ; Inst3
fin si
```

s'écrit

C/C++

```
if ( ExpB )
    Inst1 ;
else
  {
    Inst2 ; Inst3 ;
  }
```


Exemple

```
int exSi(int a)
{
  int v;
  v = a;
  if ( a>10 )
    v = v+1;
  else
    v = v+2;
  v = v+3;
  return v;
}


EVAL( exSi1(5) );

EVAL( exSi1(15) );
```

```
int exSi2(int a)
{
  int v;
  v = a;
  if ( a>10 )
    v = v+1;
  else
    {
      v = v+2;
      v = v+3;
    }
  return v;
}

EVAL( exSi2(5) );

EVAL( exSi2(15) );
```


Traduction en C/C++ de l'instruction *Pour*

langage d'algorithmes

```
pour i de E1 à E2 faire
    Inst
fin pour;
```

C/C++

```
for ( int i = E1 ; i <= E2 ; i++ )
    Inst
```

langage d'algorithmes

```
pour i de E1 bas E2 faire
    Inst
fin pour;
```

C/C++

```
for ( int i = E1 ; i >= E2 ; i-- )
    Inst
```

langage d'algorithmes

Algorithme : somEntiers
Données : `n` : Nombre ; $n \in \mathbb{N}$
Résultat : Nombre, $\sum_{i=1}^{n} i$
Variable : som : Nombre

som := 0;
pour `i de 1 à n` **faire**
  som := som + i
fin pour;
Le résultat est : `som`
fin algorithme

C/C++

```
int somEntiers(int n)
{
  int som;
  som = 0;


  return som;
}
```

```
EVAL (somEntiers(3) );
Valeur de somEntiers(3) : 6
```


langage d'algorithmes

Algorithme : liEntiers
Données : `n` : Nombre ; `n` est un entier naturel
Résultat : `Liste de Nombre`, liste croissante des `n` premiers entiers non nuls
Variable : li : Liste de Nombre

li := liVide;
pour `i de n bas 1` **faire**
  li := cons(i, li)
fin pour;
Le résultat est : `li`
fin algorithme

C/C++

```
list<int> liEntiers(int n)
{
  list<int> li;
  li = liVide<int>();


  return li;
}
```

```
EVAL( liEntiers(3) );
Val de liEntiers(3) ( 1 2 3 )
```


L'itération *Tant que* en C/C++

Traduction en C/C++ de l'instruction *Tant que*

langage d'algorithmes

tant que `ExpB` **faire**
  Inst
fin tq;

C/C++

while (`ExpB`)
  `Inst`

langage d'algorithmes

Algorithme : puissance2
Données : `n` : Nombre ; $n \in \mathbb{N}^*$
Résultat : Booléen, `true` si *n* est une puissance de 2, `false` sinon
Variable : p : entier

p := n;
tant que `estPair(p)` **faire**
  p := p quo 2
fin tq;
Le résultat est : `(p = 1)`
fin algorithme

C/C++

```
bool puissance2(int n)
{
  int p;
  p=n;


  return (p==1);
}
```

```
EVAL(puissance2(32));

EVAL(puissance2(48));
```

Lorsque plusieurs instructions doivent être itérées, il faut les regrouper dans un bloc d'instructions en les encadrant par une paire d'accolades.

langage d'algorithmes

Algorithme : existePremier
Données : `a` : Nombre, `b` : Nombre ; `a,b`$\in \mathbb{N}$
Résultat : Booléen, `true` si il existe un nb premier dans `[a,b]`
Variable : trouve : Booléen, i : Nombre

Le résultat est : `trouve`
fin algorithme

C/C++

```
bool existePremier(int a, int b)
{




}
```

```
EVAL( existePremier(24,30) );
```


Tableaux

Tableau

Un tableau est un nouveau type qui correspond à la notion de vecteur en mathématiques : un tableau est un regroupement de valeurs de même type.

- Le nombre d'éléments d'un tableau est fixe. Il est appelé **taille** du tableau
- Chaque élément d'un tableau est référencé par un **indice** qui est un entier de l'intervalle $[0..n-1]$ où n est la taille du tableau.

Exemple (Un tableau de 8 éléments de type Nombre)

0	1	2	3	4	5	6	7
5.4	7.2	0.8	−3.5	2.5	8.6	2.5	7.0

- l'élément d'indice 0 de ce tableau a pour valeur 5.4
- l'élément d'indice 1 de ce tableau a pour valeur 7.2
- ...
- l'élément d'indice 7 de ce tableau a pour valeur 7.0

Le type Tableau de `T`

- **Domaine** : `T` étant un type, le type `Tableau de T` désigne les tableaux dont les éléments sont de type `T`.
- **Opérations** :
  - fonction **taille** : Tableau de T $\longrightarrow$ $\mathbb{N}^*$
    tab $\longmapsto$ le nombre d'éléments de `tab`
  - [ ] est un opérateur permettant d'accéder à l'un des éléments du tableau que ce soit dans une expression pour obtenir sa valeur ou en partie gauche d'affectation pour modifier cette valeur.

Exemple : soit `exTab` le tableau de l'exemple précédent.

- `taille(extab)` vaut 8
- `exTab[0]` a pour valeur 5.4
- `exTab[0] := exTab[3] + 4` affecte la valeur 0.5 à l'élément d'indice 0.

Après cette affectation exTab est le tableau :

0	1	2	3	4	5	6	7
0.5	7.2	0.8	−3.5	2.5	8.6	2.5	7.0

Chaque élément du tableau `exTab` correspond donc à une variable de type `Nombre`.
Le tableau `exTab` est le regroupement de 8 variables de type `Nombre` : `exTab[0]`, `exTab[1]`, `exTab[2]`, ..., `exTab[7]`.

Variable de type Tableau

Pour déclarer une variable de type tableau, il faut indiquer **son nom**, **le type de ses éléments** et **sa taille**.

Exemple

Pour déclarer que `tab` est une variable de type `Tableau de Nombres` de taille `4`, on écrira :
Variable : tab : Tableau de 4 Nombres

La déclaration d'une variable de type tableau définit le nombre et le type des éléments du tableau, pas leurs valeurs.
Pour initialiser une variable de type tableau, il faut initialiser chacun de ses éléments.
On ne peut pas utiliser d'affectation sur une variable de type tableau, mais uniquement sur ses éléments.

Exemple

```
   Algorithme : ...
   Variable : tab : Tableau de 4 Nombre
1
2        tab[1] := 5 ;
3        tab[3] := 11 ;
4        tab[0] := tab[1] + 1 ;
5        tab[1] := tab[0] / 2;
6a       tab[3] := tab[2];
6b       tab[4] := 1;
       ...
   fin algorithme
```

En fin de ligne	Val(tab)
1	
2	
3	
4	
5	
6a	
6b	

Tableau en donnée d'un algorithme

```
   Algorithme : somTab
   Données : t : Tableau de Nombre
   Résultat : Nombre, la somme des
              éléments de t
   Variable : som : Nombre
1
2        som := 0;
3        pour i de 0 à taille(t)-1
         faire
4            som := som + t[i]
5        fin pour;
         Le résultat est : som
   fin algorithme
```

La donnée de l'algorithme est un Tableau de Nombre de taille quelconque

Supposons que `tab` soit le tableau :

0	1	2	3
6	3	3	11

Exécution de `somTab(tab)`

En fin de ligne	Val(i)	Val(som)

Tableau en résultat d'un algorithme

```
   Algorithme : tabEntiers
   Données : n : Nombre ; n∈ ℕ*
   Résultat : Tableau de Nombre,
   le tableau de taille n dont les
   éléments sont les n premiers entiers
   non nuls
   Variable : t :Tableau de n Nombre
1
2        pour i de 0 à n-1 faire
3            t[i] := i+1
4        fin pour;
         Le résultat est : t
   fin algorithme
```

La taille d'une variable de type tableau peut être une expression qui peut contenir des paramètres mais pas des variables.

Exécution de `tabEntiers(3)`

En fin de ligne	Val(i)	Val(t)

Tableau en donnée et résultat d'un algorithme

Un algorithme peut avoir en donnée un tableau, et en résultat un autre tableau.

```
   Algorithme : tabDouble
   Données : td : Tableau de Nombre
   Résultat : Tableau de Nombre,
   le tableau de même taille que td et
   dont les éléments sont les doubles
   des éléments de td
   Variable : tr : Tableau de taille(td)
   Nombre
1
2        pour i de 0 à
         taille(tr)-1 faire
3            tr[i] := 2 * td[i]
4        fin pour;
         Le résultat est : tr
   fin algorithme
```

Supposons que la valeur de `t` soit :

0	1
3	2

Exécution de `tabDouble(t)`

En fin de ligne	Val(i)	Val(tr)

Remarque (Différences entre Tableau et Liste)

`Liste de T` et `Tableau de T` représentent tous les deux des séquences de valeurs de type `T`. Ces deux types se différencient par le type d'opérations applicables aux séquences.

- La `Liste` permet une gestion *dynamique* de la séquence : on peut facilement ajouter un élément à une liste (en tête de liste) alors que la taille d'un `Tableau` est fixe.
- Le `Tableau` permet un accès direct aux éléments de la séquence (à l'aide des indices), alors qu'accéder aux éléments d'une `Liste` nécessite un calcul.
- Le `Tableau` permet facilement de modifier la valeur d'un élément de la séquence, alors que la modification d'un élément d'une `Liste` nécessite de reconstruire la `Liste`.

Le type `Tableau de T` en C/C++

Pour chaque type `T` le type `Tableau de T` est défini en C/C++ par :

- **Nom du type** : `vector<T>` (par exemple le type `Tableau de Nombres entiers` s'appelle en C/C++ `vector<int>`)
- **Domaine** : vecteurs de valeurs de type `T`.
- **Opérations** : `t` étant un tableau et `i` un entier

écriture en langage d'algorithme	écriture en C/C++
`taille(t)`	`taille(t)`
`t[i]`	`t.at(i)`

- **Déclaration d'une variable tableau**
  En C/C++, pour déclarer une variable de type tableau de `n` éléments de type `T` on écrit :

  vector<`T`> `nomVariable(n);`

  `n` est une expression dont la valeur est un entier strictement positif.

en C/C++ la valeur d'un tableau sera notée par la séquence de ces éléments encadrée par une paire de crochets ([et]).

Exemple (de session C/C++)

```
vector<int> t(3);
t.at(0)=3; t.at(1)=6;
t.at(2)=2;
EVAL(t);
EVAL(taille(t));
t.at(1)=t.at(0)+2;
EVAL(t);
t.at(3)=4;
EVAL(tableau({3,6}));
```


Exemple (Algorithme sur les tableaux en C/C++)

Algorithme : somTab
Données : `t` : Tableau de Nombre
Résultat : Nombre, la somme des éléments de `t`
Variable : som : Nombre

```
    som := 0;
    pour i de 0 à taille(t)-1 faire
        som := som + t[i]
    fin pour;
    Le résultat est : som
fin algorithme
```

```
int somTab(vector<int> T)
{
    int som;
    som=0;


    return som;
}
 EVAL( somTab(tableau({3,5,7})) );
=>  Valeur de somTab(tableau({3,5,7})) : 15
```

Exemple (Algorithme sur les tableaux en C/C++)

Algorithme : tabEntiers
Données : `n` : Nombre ; $n \in \mathbb{N}^*$
Résultat : Tableau de Nombre,
le tableau de taille `n` dont les éléments sont les `n` premiers entiers non nuls
Variable : t : Tableau de n Nombre
    pour `i de 0 à n-1` **faire**
        t[i] := i+1
    fin pour;
    Le résultat est : `t`
fin algorithme

```
vector<int> tabEntiers(int n)
{
  vector<int> t(n);


  return t;
}
 EVAL(tabEntiers(6));
=> Valeur de tabEntiers(6) : [ 1 2 3 4 5 6 ]
```


Exemple

Algorithme : tabDouble
Données : `td` : Tableau de Nombre
Résultat : Tableau de Nombre, le tableau de même taille que `t` et dont les éléments sont les doubles des éléments de `t`
Variable : tr : Tableau de taille(t) Nombre
    pour `i de 0 à taille(tr)-1` **faire**
        tr[i] := 2 * td[i]
    fin pour;
    Le résultat est : `tr`
fin algorithme

```
vector<int> tabDouble(vector<int> t)
{
  vector<int> tr(taille(t));


  return tr;
}
 EVAL(tabDouble(tableau({1,2,3})));
=> Valeur de tabDouble(tableau({1,2,3})) : [ 2 4 6 ]
```


Algorithmes de base sur les Tableaux

Minimum d'un tableau

Algorithme : minTableau
Données : `t` : Tableau de Nombre
Résultat : Nombre, la valeur minimum des éléments de `t`
Variable : m : Nombre

```
1     m := t[0];
2     pour i de 1 à taille(t)-1
      faire




      fin pour;
5     Le résultat est : m

fin algorithme
```

Soit `S` le tableau :

0	1	2	3
7	9	6	6

Exécution de `minTableau(S)`

ligne	Val(i)	Val(m)	Val(t[i] < m)

Attention aux initialisations !

Un algorithme Faux

Algorithme : minTabFaux
Données : `t` Tableau de Nombre
Résultat : Nombre, la valeur minimum des éléments de `t`
Variable : m : Nombre

```
1     m := 0;
2     pour i de 0 à taille(t)-1
      faire
3         si t[i] < m alors
4             m := t[i]
          fin si;
5     fin pour;
      Le résultat est : m
fin algorithme
```

Soit `S` le tableau :

0	1	2	3
7	9	6	6

Exécution de `minTabFaux(S)`

ligne	Val(i)	Val(m)	Val(t[i] < m)

Traduction en C/C++

Algorithme : minTableau
Données : `t` : Tableau de Nombre
Résultat : Nombre, la valeur minimum des éléments de `t`
Variable : m : Nombre
    m := t[0];
    pour `i de 1 à taille(t)-1` **faire**
        si `t[i] < m` **alors** m := t[i]
    fin pour;
    Le résultat est : `m`
fin algorithme

```
int minTableau(vector<int> t)
{




}
```


Recherche d'un élément dans un tableau

Algorithme : appTableau
Données : `x` : Nombre, `t` : Tableau de Nombre
Résultat : Booléen, `true` si `x` est la valeur d'un élément de `t`, `false` sinon
Variable : i : Nombre, trouvé : Booléen

1 i := 0 ; trouvé := false;
2 **tant que**
faire

fin tq;
4
Le résultat est : `trouvé`
fin algorithme

Soit `S` le tableau :

0	1	2	3
7	9	6	6

Exécution de `appTableau(6,S)`

ligne	Val(i)	Val(trouvé)

Le résultat est :

Traduction en C/C++

Algorithme : appTableau
Données : `x` : Nombre, `t` : Tableau de Nombre
Résultat : Booléen, `true` si `x` est la valeur d'un élément de `t`, `false` sinon
Variable : i : Nombre, trouvé : Booléen
    i := 0 ; trouvé := false;
    tant que `i<taille(t) et non(trouvé)` **faire**
        si `t[i]=x` **alors** trouvé := true **sinon** i := i+1 **fin si**
    fin tq;
    Le résultat est : `trouvé`
fin algorithme

```
bool appTableau(int x, vector<int> t)
{




 }
 EVAL(appTableau(2,tableau({4,3,7,1})));
=> Valeur de appTableau(2,tableau({4,3,7,1})) : false
```


Définition (Occurrence dans un tableau)

Une occurrence d'une valeur `v` dans un tableau `T` est associée à un indice `i` tel que `T[i] = v`.

Exemple

0	1	2	3
7	9	6	6

- Dans le tableau `S` l'entier 6 a 2 occurrences. Elles sont associées aux indices 2 et 3 du tableau S.
- Dans `S` l'entier 9 a une seule occurrence.
- Il n'y a pas d'occurrence de 5 dans le tableau `S`

Algorithme : nbOccTableau
Données : `v` : Nombre, `t` : un Tableau de Nombre
Résultat : Nombre, le nombre d'occurrences de `v` dans `t`
Variable : nbocc : Nombre

```
1   nbocc := 0 ;
2   pour i de 0 à taille(t)-1 faire
5   fin pour;
    Le résultat est : nbocc
fin algorithme
```

Soit `S` le tableau :

0	1	2	3
7	9	6	6

`nbOccTableau(6,S)`

ligne	Val(i)	Val(nbocc)	Val(t[i] = v)

Le résultat est

Tri d'un tableau

Le problème

Trier un tableau consiste à permuter les valeurs de ses éléments de sorte que la séquence des valeurs soit croissante.
Par exemple le résultat du tri du tableau | 4 | 9 | 2 | 4 | 3 |
est le tableau | 2 | 3 | 4 | 4 | 9 |

Les algorithmes de tri

Intérêt du problème : lorsqu'un tableau est trié les opérations sur les tableaux (calcul du minimum, test d'appartenance, calcul du nombre d'occurrences d'une valeur, ...) peuvent être réalisées par des algorithmes plus efficaces.
Il existe plusieurs algorithmes de tri.
L'un des plus simples d'entre eux (mais peu efficace) est le `tri à bulles`.

Le principe du tri à bulles

L'algorithme parcourt le tableau, et compare les couples d'éléments consécutifs. Lorsque deux éléments consécutifs ne sont pas dans l'ordre croissant, ils sont échangés. Après un premier parcours la valeur du dernier élément est la valeur maximum du tableau. Cette opération est répétée `n` fois pour trier le tableau.

Exemple

1er Parcours

4	**9**	2	4
4	**9**	**2**	4
4	2	9	4
4	2	**9**	**4**
4	2	4	9

2ème Parcours

4	**2**	4	9
2	4	4	9
2	**4**	**4**	9

3ème Parcours

2	**4**	4	9

Algorithme : triBulles
Données : `t` : tableau de Nombre
Résultat : Tableau de Nombre, le tableau `t` trié
Variable : aux : Nombre, tt : tableau de taille(t) Nombre

```
    pour i de 0 à taille(t)-1 faire
        tt[i] := t[i]
    fin pour;
    Le résultat est : tt
fin algorithme
```