Algorithme: Définition et Langage

- Objet : Rappel des définitions et de la syntaxe du langage algorithmique
- Niveau requis :
  débutant, avisé
- Commentaires : *Contexte d'utilisation du sujet du tuto.*
- Débutant, à savoir : Utiliser GNU/Linux en ligne de commande, tout commence là !.

Introduction : Notion d'objets et d'actions

Quelques éléments du langage algorithmique.

On utilise

- **caractères A…Z** ; **a…z** ;
- les **chiffres 0 1 2 3 …** ;
- des **signes _ = < > ' " ( ) [ ] + - * / , : . <espace>** ;
- les **identificateurs** : ce sont les noms donnés aux différents objets déclarés dans un algorithme.
  Ils peuvent être fait d'une suite de longueur quelconque de chiffres de lettre (min, maj) de soulignés.
- les **commentaires** : **//**explications sur les identificateurs, sur les étapes du programme d'algo

Quatre sortes d'objets

Il y a quatre sortes d'objets : les constantes, les variables, les fonctions et les procédures.

Les constantes

- Syntaxe de déclaration d'une constante :

```
nom_constante = valeur_de_la_constante  // On commente toujours le nom d'une constante.
```

- Exemple :

```
CONSTANTE
taille = '.' // caractère de terminateur
```

Les variables

Une variable est un emplacement mémoire qui change de valeur au cours d'un programme ou d'un d'un algorithme.

- **Syntaxe de déclaration d'une variable**

```
nom_variable :type
```

- Exemple :

```
Nbjeunes :entier // Nombre de jeunes interrogés
```

- **Affectation d'une variable**

Elle n'est possible que s'il y a eu déclaration de la variable.

On note le signe d'affectation **:=**.

```
variable i : entier // déclaration
...
i := 3 // affectation de i par la valeur 3

i := i + 1 // incrémentation de 1
...

i := 4 * i // i est multiplié par 4

...
```

FONCTION ET PROCEDURE

Fonction et procédure réalisent un travail, par exemple un calcul, et ne font rien d'autre que ce travail.

Elles sont utilisée par des programmes qui les appelle, et qui font d'autre chose, par exemple afficher le résultat calculé par une procédure ou une fonction.

Comparons une fonction et une procédure qui toute deux calculent le cube d'un nombre ; puis comment un programme utilise l'une ou l'autre.

Voir la structure générale d'un algorithme

Fonction cube

- Explication :

On imagine un tableau où on place 2, le nombre dont calcule la puissance de 3 ;
L'indice maximal de ce tableau est la puissance à laquelle on veut élever le nombre 2.

```
Pour i de 1 à 3 on fait :

   |   2   |   2   |    2   |
```

```
      |       |        |
   1 * 2      |        |
   = 2        |        |
              |        |
          2 * 2        |
          = 4          |
                       |
                   4 * 2
                   = 8
```

- Déclaration et mise en place de la fonction

```
Fonction fctcube (entrée x : entier):entier
  //cette fonction calcul la puissance trois d'un nombre
  // x est le nombre
  // la fonction retourne le cube du nombre

CONSTANTES

constante puissance = 3

VARIABLES

variable i : entier //comptage des puissances
variable y : entier // calcul intermédiaire


 début

   i := 1
   y := 1
      Tantque (i := puissance) FAIRE
        y := y * x
        i := i + 1
      FinTantque
   retourner (y)
 Fin
 fin
```

Utilisation de la fonction dans un programme

```
//Programme cube
// Programme qui calcule le cube d'un nombre et affiche le résultat

VARIABLES

nombre : entier // nombre à élever au cube

FONCTIONS

fonction fctcube (entrée x : entier):entier
```

```
  // cette procédure calcule le cube du nombre x et retourne le résultat

  Début
     lire(nombre)
     écrire ('le résultat est : ', fctcube(nombre))  // Appel de la
fonction, elle rend la résultat qu'on affiche avec écrire
            // utilisation du résultat
  Fin
```

Procédure cube

La différence avec un programme est l'en-tête.

```
procédure proccube (entrée x : entier, sortie y : entier)

  // Cette procédure calcule la puissance trois d'un nombre.
CONSTANTES

constante puissance = 3

VARIABLES

variable i :entier // comptage des puissances

   Début
        i := 1
        j := 1
           Tantque (i := puissance) FAIRE
              y := y * x
              i := i + 1
           FinTanque
   Fin
```

Utilisation de procédure dans un programme

La procédure fait le calcul du cube, le programme se moque de savoir comment, il a pour travail de calculer le cube au moyen d'une procédure et d'afficher le résultat fait par la procédure.

```
// Programme qui calcule le cube d'un nombre

VARIABLES
 variable nombre :entier //nombre à élever au cube
 variable resultat :entier // cube du nombre
```

```
PROCEDURES

procédure proccube (entrée x : entier , sortie y : entier)
      //Cette procédure calcule le cube du nombre x et met le résultat dans
y

  Début
   lire(nombre)
   proccube (nombre, resultat) // calcul du cube de nombre
   écrire('Le résultat est : ', résultat)
         // affichage du résultat du calcul
  Fin
```

Voir ci-dessous la structure en sous programme d'un algorithme complexe: algo-exo-constructions-d-algorithmes-de-procedure

Voir aussi : d'autres exemples simples d'algorithme de fonctions et de procédures : http://www.est-usmba.ac.ma/ALGORITHME/co/module_ALGORITHME_40.html

Les Types

Il y a quatre types prédéfinis : entiers ; réels ; booléens ; caractères.
Les tableaux sont des types de variables créés.

Les entiers

- Les entiers ne sont pas limités par leur taille.
- Les constantes entières sont toujours en base 10.

Les opérateurs de comparaison sur les entiers

Ils ont pour opérandes deux entiers et donnent un résultat booléen.

symboles	signification
=	Comparaison d'un égalité
<>	Comparaison d'une différence
>	Comparaison de plus grand que
<	Comparaison de plus petit que
>=	Comparaison de plus grand ou égal que
<=	Comparaison de plus petit ou égal que

Les opérateurs de calcul

Ils ont pour opérandes deux entiers et donnent un résultat entier

symboles	signification
+	addition
-	soustraction
*	multiplication
div	division (récupère le quotient)
mod	modulo (récupère le reste de la division

Exemples :

7 div 3 = 2
7 mod 2 = 1

Priorité de div et de mod sur + et - :

Les réels

Écriture

```
238.45    420.0
```

Opérateurs de comparaison sur les réels

symboles	signification
=	Comparaison d'un égalité
<>	Comparaison d'une différence
>	Comparaison de plus grand que
<	Comparaison de plus petit que
>=	Comparaison de plus grand ou égal que
<=	Comparaison de plus petit ou égal que

Le résultat de la comparaison est un booléen.

Les opérateurs de calcul sur les réels

symboles	signification
+	addition
-	soustraction
*	multiplication
/	division

Le résultat d'un calcul entre un entier et un réel se fait avec les opérateurs des réels et

donne toujours un réel comme résultat.

Les caractères

Ce sont tous les caractères imprimables de la tables ASCII ; 'a' et 'A' sont deux caractères différents.

On distingue '2' en tant que caractère et en tant qu'entier.

On n'utilise que les opérateurs de comparaison ; ils sont ordonnés selon la tables ASCII.

Voir : http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange

Les booléens

Comparaison et calcul sur les booléens

- Deux valeurs possibles : **VRAI FAUX**

Opérateurs de comparaison

Le résultat d'une comparaison est soit VRAI soit FAUSSE : le résultat d'une comparaison est toujours un booléen.

symboles	signification
=	**Comparaison d'un égalité**
<>	**Comparaison d'une différence**
>	Comparaison de plus grand que
<	Comparaison de plus petit que
>=	Comparaison de plus grand ou égal que
<=	Comparaison de plus petit ou égal que

Exemple de comparaison :

```
...
i := entier
bool : booléen // Déclaration d'une variable de type booléen
...

  SI (i > 5 ) ALORS
      bool := VRAI
   SINON
      bool := FAUX
  FINSI
```

La comparaison ci-dessus (SI ... SINON ... FINSI) se réduira ainsi :

```
bool := (i > 5)
```

Pour s'aider lire ainsi : **bool = VRAI pour (i > 5)**.

Opérations logiques

Trois opérations nous intéresse : **NON ET OU** (du plus prioritaire au moins prioritaire).

Il faut parenthèser les expressions pour la lisibilité.

Le ET et le OU algorithmiques ne sont pas le ET et le OU de la logique de bool.

- En algo, on comprends **ET** comme un **ET-alors** :

a	**b**	**condition a ET b**
FAUX	(non évalué)	FAUSSE (pas d'exécution du code)
VRAI	alors b évalué b = VRAI b = FAUX	expression = b qui doit être calculable a ET B = VRAI (condition exécutée) a ET b = FAUX (condition non exécutée)

⇒ Pour résumer, on tient compte de **b** seulement si a est vrai,
et la condition est exécutée si **a et b** sont vrais.

- On comprends le **OU** comme **OU-sinon** :

a	**b**	**condition a OU b**
VRAI	(non évalué)	VRAI
FAUX	(sinon) b évalué b = FAUX b = VRAI	expression = b qui doit être calculable a OU b = FAUX (condition non exécutée) a OU b = VRAI (condition exécutée)

⇒ Pour résumer, **b** est pris en compte seulement si a est FAUX,
et la condition est exécutée si **b** est vrai.

Rappel des tables de vérité

Voir : algèbre de bool surtout la loi de Morgan.

Les types de variables créés (tableaux)

Déclaration d'un tableau à une dimension

- Déclaration et affectation

```
type // On annonce qu'on crée un nouveau type de variable.
blabla = tableau[n] d'un type_pré-défini
```

- Mettre (affecter) la valeur d'un indice (n° de case du tableau) dans une variable (cible) :

```
cible := nom_tableau[i]
  |
 nom_variable <- valeur de tab[i]
```

- Mettre la valeur d'une variable (cible) dans une case de tableau (cible) :

```
nom_tableau[i] := cible
      |
 case tableau   <- valeur de variable
```

- Vérifier une condition :

```
SI nom_tableau[i] = cible ALORS ...
      |               |
  valeur case        valeur de variable
```

```
SI nom_tableau[i] = nom_tableau[i + 1] ALORS ...
          |                       |
    valeur de case        valeur de case + 1
```

- Exemples 1:

```
Type
nombres : tableau[35] d'entiers
```

- Exemple 2 :

```
Type
tab : tableau[35] d'entiers
```

Lire la valeur d'une case d'un tableau

```
lire(tab[2])
// enregistre en mémoire la valeur de la case 2 du tableau "tab".
```

Afficher la valeur d'une case d'un tableau

```
écrire(tab[5])
```

```
// afficher à l'écran la valeur de la case 5 du tableau "tab".
```

Déclaration d'un tableau à deux dimensions

```
...
variables  i: entier
           j: entier
           matcarré : matrice


  ...
matcarré [i][j] := 2 // matcarré est de type matrice
                     // matcarré[i] est de type tabent
                     // matcarré [i][j] est de type entier
```

matcarré [i][j] := 2 ou **matcarré [i][j] := 2**

Doc sur tableau :

- http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&ved=0CEAQFjAF&url=http%3A%2F%2F2igc.cours.free.fr%2Ftsttig%2F002I_Algo%2F02I_Algo02.doc&ei=1Lt6VLGUBIjdau32gbgG&usg=AFQjCNGBy4qN30W1FEHReM5fNcAHhdcQnQ&sig2=CYelZWE8ytMAn1tvjP7MRw&bvm=bv.80642063,d.d2s

Les actions

Il s'agit des instructions et des outils d'entrée et de sortie.

Les instructions alternatives

L'instruction SI

- Syntaxe :

```
SI <condition>  ALORS

   < instruction 1 >
         ...
   < instruction n >

FinSI
```

Exemple :

```
variable i : entier //déclaration d'un variable nommée i

  SI i > 4 ALORS

      i := i - 1

  SINON

      i := i + 1

  FinSI
```

L'instruction CHOIX

- Syntaxe :

```
 CHOIX sur < variable > FAIRE

    < valeur 1 > :    <instruction1 1>
                      <instruction1 2>
                       ...
                      <instruction1 n(1)>

     < valeur M > :   <instructionM 1>
                      <instructionM 2>
                       ...
                      <instructionM n(M)>
     AutreCas     :   <instruction (M+1) 1>
                       ...
                      <instruction (M+1) n(M+1)>

 FinCHOIX
```

> `AutreCas` est obligatoire, même dans le cas où il n'y en a pas.

- Exemple :

```
variable i :entier
...
CHOIX sur i faire

     1 : écrire(c'est bien)
     2 : écrire(c'est très bien)
     3 : écrire(c'est très nul)

     AutreCas : écrire('Les moyens') //cas tous regroupables
```

```
FinCHOIX
```

- enregistrer la valeur donnée par l'utilisateur d'une variable déclarée

```
age : entier //"age" c'est le nom de la variable
lire(age)    //
```

- afficher ce qu'on veut à l'écran :

```
écrire(blabla)    //
```

- afficher à l'écran la valeur d'une variable parmi un message :

```
age : entier
age := 25
écrire('La valeur de la variable age est :', age 'encore du
texte si on veut.')    //
```

Instructions itératives : TantQue

Instruction exécutée par la boucle tant que la condition est remplie ; il faut une condition d'arrêt.

La boucle est toujours exécutée au moins une fois.

- Syntaxe :

```
  TantQue < condition > FAIRE   // condition de continuation
          < instruction1 >
          < instruction2 >
          ...
  FinTQ
```

- Exemple :

```
variable i : entier
i := 1
  TantQue ( i < 3 ) FAIRE   // La boucle s'arrêtera quand i sera >= à 3.
                            // On met dans le commentaire la condition
contraire
                            // c'est-à-dire la condition de terminaison.

     i := i + 1   // incrémentation
     écrire ('coucou')    // Tant que i < 3 il s'affiche "coucou" à l'écran.

  FinTQ
```

```
  écrire('la variable i est égale à ', i) //
```

```
coucou
coucou
la variable i est égale à 3
```

Instructions itératives : Répéter

Elle permet d'exécuter une instruction jusqu'à ce qu'une condition soit remplie.

Attention si la condition n'est jamais remplie, l'instruction n'est jamais exécutée, car la conditiion est placée après l'instruction !

- Syntaxe :

```
  Répéter
        < instruction 1>
        < instruction n>
  Jusqu'à < condition >     // C'est une condition d'arrêt.
```

- Exemple :

```
variable i : entier
i := 1
  Répéter
        i := i + 1
        écrire('coucou')
  Jusqu'à ( i >= 3 )    // être égal à 3 est une condition d'arrêt.
écrire('la variable i est égale à ', i)
```

```
coucou
coucou
la variable i est égale à 3
```

> ATTENTION la boucle s'effectue toujours une fois

Dans le schéma “Tant que” la condition de poursuite est avant le traitement, dans le schéma “Répéter”, la condition de poursuite est après.

La boucle répéter est donc exécutée au moins une fois.

Avec une boucle répéter, on exécute l'instruction “ tant que la condition est fausse. ”

```
répéter
 Afficher "Saisir un nombre strictement positif "
 Saisir i
 Si i (<= 0) alors
  écrire("J'ai dit STRICTEMENT POSITIF !")
 Sinon
```

```
  écrire("Bravo")
 Finsi
jusqu'à i > 0
```

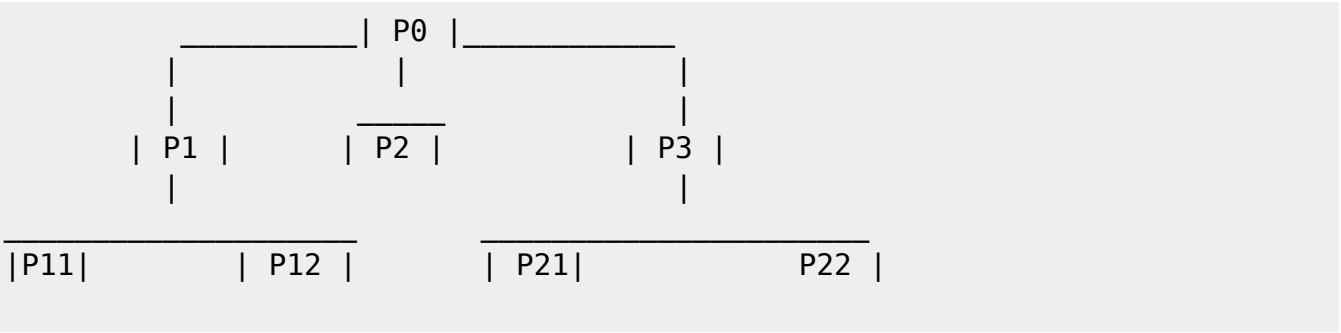
⇒ Si on entre -2 : on aura “J'ai dit STRICTEMENT POSITIF !”

Structure générale d'un algorithme

Un algorithme écrit d'un seul tenant devient difficile à comprendre et à gérer dès qu'il dépasse deux pages.

La solution consiste alors à découper l'algorithme en plusieurs parties plus petites. Ces parties sont appelées des sous-algorithmes.

Voici le schéma de l'algorithme d'un programme complexe, l'algorithme principal (P0) appelle les algorithmes de sous-programmes (P1, P2, P3), qui eux-mêmes en appellent d'autres ... :



Le sous-algorithme est écrit séparément du corps de l'algorithme principal et sera appelé par celui-ci quand ceci sera nécessaire.

Les différentes parties d'un algorithme (sous-algorithmes) sont appelées des blocs.

Il existe deux sortes de **sous-algorithmes** : **les procédures et les fonctions**.

Un objet définit dans un bloc n'est pas connu dans les blocs extérieurs.

Mais il doit être connu (défini) dans bloc et il est connu (re-défini) dans les sous-blocs de ce bloc.

Structure d'un bloc

Quelque soit le niveau où l'on se trouve dans l'arborescence, tous les blocs, y compris l'algorithme du programme principal, se déclare comme ceci :

```
Interface de bloc // nom du programme, de la procédure ou de la fonction.
// Commentaire sur le rôle du bloc

CONSTANTE // On définit les constantes.

TYPES     // Définitions des types du bloc

VARIABLES // Définitions des variables

PROCEDURES ... // Définitions des sous-procédures appelées par le bloc, ce
bloc
               // est alors un algorithme plus globale

FONCTIONS ... // Définitions des fonctions appelées par le bloc


// Définition des actions du bloc

  Début
      actions // actions effectuées lors de l'activation du bloc
  Fin
```

Les procédures dans algorithme

Une procédure est une boîte (un petit algorithme) qu'on insère dans un programme (gros algorithme) et à laquelle on donne les paramètres du programme. Ses **paramètres sont appelés paramètres formels**. Leurs valeurs ne sont pas connus lors de la création de la procédure.

*Dans un **programme principal qui utilise une procédure**, on n'aura pas à expliquer comment fonctionne la procédure utilisée, mais seulement ce à quoi elle sert dans ce programme, et ce qu'on lui donne en paramètre dans ce programme. Dans le programme les paramètres passés à la procédure sont réels ; ils sont appelés **paramètres effectifs**.*

Déclaration d'une procédure

- Syntaxe :

```
Procédure nom_proc(entrée nom_parmètre1_formel : typeDuParamètre
                          nom_paramètre2_formel : typeDuParamètre
                   entrée sortie nom_paramètre1 : typeDuParamètre
                   sortie nom_paramètre1_formel : typeDuParamètre)
```

> Cela est copier de l'étape n°4 de la présentation (algo-la-presentation-d-une-procedure

La procédure

Une procédure est une série d'instructions regroupées sous un nom, qui permet d'effectuer des actions par un simple appel de la procédure dans un algorithme ou dans un autre sous-algorithme.

- On y trouve

1. la déclaration ;
2. ces composants (constantes, type_créé , variables) ;
3. le programme qui permet à cette procédure de fonctionné

Appel d'une procédure dans un programme

Pour déclencher l'exécution d'une procédure dans un programme, il suffit de l'appeler. L'appel de procédure s'écrit en mettant le nom de la procédure, puis la liste des paramètres, séparés par des virgules. A l'appel d'une procédure, le programme interrompt son déroulement normal, exécute les instructions de la procédure, puis retourne au programme appelant et exécute l'instruction suivante.

```
...
Nom_procédure(liste de paramètres)
...
```

Exemple simple d'un algorithme appelant un sous-algorithme

Donnez l'algorithme qui affiche un carré d'étoile à l'écran.

L'exemple suivant ne prétend que mettre en avant l'appel de procédure dans un algorithme.
Cela est possible seulement parce qu'il s'agit d'un cas très simple.

```
// Algorithme carréEtoile                                  |
// Algorithme qui affiche un carré de 15 lignes
// et de 15 colonnes
Variables j : entier // nombre d'étoile sur une ligne
Procédure Etoile ( ) // pas de paramètre
                     //Procédure qui affiche une
                     // ligne de 15 étoiles.
 DébutAlgo
       Pour j DE 1 à 15 FAIRE
            // appel de la procédure   --------------->| Etoile(entrée i :
entier , sortie i :entier)
           Etoile( )                                   |  // Procédure qui
affiche une ligne d'étoile
                                                       |  // Et qui va à la
ligneFinProc
       FinPour                                         |
 FinAlgo                                               |
                                                       | Variables i :entier
                                                       |   DébutProc
                                                       |       Pour i DE 1 à
```

```
15
                                                                |
ecrire("*")
                                                                |       FinPour
                                                                |       ecrire("\n")
                                                                |   FinProc
```

Lors de problème plus complexes, on le divise en procédures et l'on cherche un algorithme
pour chaque procédure.

Les entrée sortie et entrée sortie

```
                +
 entrée     |       | sortie
variables   |       | c = 3
a = 1
b = 2
```

```
               entrée sortie
          |                       |
          | modifier une variable |
          |  qui est donnée       |
          | en entrée et qui sort |
          | avec une autre valeur |
```