Langage Algorithmique / Pseudo-Code / Langage Pivot

Florent Bouchez Tichadou

2 août 2018

Un langage algorithmique est avant tout un outil de *communication* entre algorithmiciens. L'important est de s'assurer que la personne à qui l'on communique un algorithme a bien tout les détails nécessaires pour comprendre son fonctionnement ; c'est le cas quand vous discutez avec un autre étudiant, avec un enseignant, et encore plus sur un forum de discussion en ligne ou un site de questions/réponses comme www.stackoverflow.com.

Il n'y a donc pas à proprement parler de langage algorithmique universel, ni une syntaxe rigoureuse fixée dans le marbre, à l'inverse d'un langage de *programmation*. Cependant, un certains nombre de conventions existent, provenant parfois de langages de programmation et parfois du langage courant.

Ce document tente d'établir une base de communication suffisamment riche pour permettre d'exprimer tous les algorithmes étudiés en cours, et suffisamment générale pour qu'elle soit compréhensible par toute personne ayant fait de l'algorithmique sans pour autant avoir suivi ce cours. Il existe donc naturellement parfois plusieurs manières d'exprimer la même idée (exemple : « entier positif » ou « entier $\geqslant 0$ »). Dans la suite de ce document, nous présentons parfois plusieurs possibilités séparées par un « *ou bien* », mais ce n'est pas exhaustif !

Il est souvent possible d'omettre des détails inutiles (par exemple, le type d'un itérateur de boucle), par contre, *ne laissez jamais d'ambigüité* dans un algorithme.

1 Déclarations de variables et types

1.1 Types simples et constantes

v : entier	r : réel	cpl : couple d'entiers
x : entier positif	c : caratère	upl : 5-uplet de réels
y, z : entiers > 0	c' : caratère alphanumérique	
i : entier entre 0 et LMAX-1	str : chaine de caractères	constante PI $= 3,14159$
j : entier $\in [0 \ldots \text{LMAX} - 1]$	str' : chaine	constante LMAX $= 1000$
b : booléen	t : tableau de n entiers	constante nil : Nil /* *rien* */

1.2 Types construits

Pour décrire, par exemple, l'implantation bas-niveau de types de haut-niveau (cf. section suivante). On trouve le plus souvent des *structures* (ou *enregistrements* ou *types produits*), i.e., plusieurs éléments appelés *champs* regroupés ensembles. Nous définissons aussi les *unions* (ou *types sommes*) dont les variables peuvent être d'un type ou d'un autre.

type Point : { x, y : réels }	p : Point
type Cercle : { pos : Point, diam : réel }	c : Cercle
type Séquence : { tab : tableau de n entiers, longueur : entier }	S : Séquence
type Ensemble : Séquence	E : Ensemble
type Nombre : entier | réel	nb : Nombre

1.3 Types de haut niveau

Utilisables uniquement dans des algorithmes de « haut-niveau ». On peut regrouper en ensembles, séquences des élément de même type, y compris des types construits ou de haut niveau.

```
E : ensemble de réels
S : séquences d'entiers
L : liste d'entiers
p : ensemble de séquences de booléens
```

```
type Polygone : ensemble de couples d'entiers
tri : Polygone
nbs : liste de Nombres
```

1.4 Opérations courantes

Add., mul., sous.	Divisions	Comparaisons	Booléens
2 + 3	7 / 3 /* 2,33... */	3 = 5 a ≠ 4	b et b'
v * 5	7 div 3 /* 2 */	v < 0 r ⩽ 2,5	b ou b'
12 − x	7 mod 3 /* 1 */	r > PI y ⩾ 1	non b

1.5 Accès et assignations

```
v ← 12
i ← i+1
b ← vrai
b ← b ou faux
r ← 2,718
c ← 'A'
str ← "Hello world !"
```

```
cpl ← (3, 8)
upl ← (3, −1, 12, 5, 7)
(i,j) ← cpl
p.x ← 3.1
p.y ← r
c ← { pos=p, diam=10,5 }
nb ← 42  ou bien  nb ←2,1
tri← { (0, 0), (1, 2), (2, 0) }
```

```
t[0] ← 0   ou bien   t_0 ← 0
t ← [1, 5, 9]
S.longueur ← 4
S.tab[0] ← 9
S.tab[1] ← 3
S.tab[2] ← −5
S.tab_3 ← 0
```

Pour les structures de haut niveau, on peut aussi directement définir leur contenu.

```
E : ensemble d'entiers
E ← { 12, 4, 5, 9 }
soit e ∈ E
afficher (e)
```

```
S : séquence d'entiers
S ← ⟨ 2, 3, 5, 7, 11 ⟩
afficher (S_3)
```

1.6 Opérations d'entrées/sorties

Permettent d'« afficher à l'écran » et de « lire au clavier ». [1]

```
afficher ("Appuyer sur une touche pour continuer.")
afficher ("Valeur de x : ", x)
afficher ("Cercle de diamètre ", c.diam, " aux coordonnées ", c.pos)
afficher ("Entrez une valeur entière au clavier : ")
x ← lire()   ou bien   x ← lire_entier()
c ← lire()   ou bien   c ← lire_caractère()
str ← lire()   ou bien   str ← lire_ligne()
```

1.7 Opérations conditionnelles

```
si x > 0 alors
  afficher ("positif")

si E est vide alors
  rien   ou bien   ne rien faire
sinon  afficher (E)
```

1. « M'sieur, comment je branche mon clavier sur mon algorithme ? »

```
si S.longueur = n alors
  afficher ("Séq. pleine")
sinon si S.longueur = 0 alors
  afficher ("Séq. vide")
sinon  afficher ("mi-plein ?")
```

```
si nb est un entier alors
  afficher ("entier : ", nb)
sinon
  afficher ("réel : ", nb)
```

1.8 Structures de répétition

```
/* Parcours de haut niveau                    */
pour chaque e ∈ E faire
  afficher (e, " est dans l'ensemble")
afficher ("Séquence dans l'ordre :")
pour chaque x ∈ S faire
  afficher (" ", x)

/* Parcours de bas niveau                     */
pour i de 0 à n − 1 faire
  afficher ("Indice ", i, " : ", t[i]) ;
afficher ("Séquence en décroissant :")
pour i de S.longueur−1 à 0 faire
  afficher(" ", S.tab[i]) ;
```

```
x ← lire()
tant que x ≠ 0 faire
  afficher ("Erreur : vous devez entrer zéro")
  x ← lire()

répéter
  afficher ("Entrez un nombre : ")
  x ← lire()
jusqu'à  x = 0

do n times
  afficher ("I WILL NOT SKATEBOARD IN THE
    HALLS")
```

1.9 Multiples conditions

```
selon direction faire
  cas nord faire
    y ← y + 1
  cas ouest faire
    x ← x − 1
  autres cas faire
    afficher ("Direction interdite !")
```

```
selon r faire
  cas r < 0 faire r ← −r
  cas r ⩽ 0 < 1 faire r ← r²
  cas r ⩾ 1 faire rien
  autres cas faire  afficher ("Heu... ? ! ?")
```

1.10 Fonctions

```
debug (message : chaine)
  afficher ("Debug : ", message)

addition (x,y : entiers) : entier
  retourner x + y
z ← addition(2, 8)

gps () : Point
  p.x ← longitude() ; p.y ← latitude()
  retourner p

deux_valeurs () : couple d'entiers
  a ← lire() ; b ← lire()
  retourner (a,b)
(x, y) ← deux_valeurs()
```

Pour modifier un paramètre, il faut le passer par *référence* et non par *valeur* (cf. section suivante).

```
échange (x, y : références sur entiers)
  tmp : entier
  tmp ← x
  x ← y
  y ← tmp

a ← 3, b ← 5
échange (réf a, réf b)

/* Maintenant, a vaut 5 et b vaut 3.          */
```

1.11 Copies et références

Tous les langages de programmation ont des conventions qui indiquent dans quel cas une valeur est *copiée* lors d'une assignation, ou si la nouvelle variable est en fait une *référence* sur l'ancienne. Dans notre langage algorithmique, nous considérons que tous les objets sont *copiés* par défaut, y compris lors d'un appel et au retour d'une fonction, sauf si une référence est explicitement indiquée.

Attention ! C'est rarement le cas dans les langages de programmation usuels. Par exemple, les tableaux ou structures sont en général passés par référence (appel de fonction ou assignation de variable) tandis que les types simples (entiers, réels, etc.) sont copiés.

Copies à l'assignation

```
a ← 12
b ← a
b ← 18
/* a vaut 12 et b vaut 18                                   */

t ← [1, 5, 9]
u ← copie de t
u[1] ← 12
/* t = [1, 5, 9] et u = [1, 12, 9]                          */

p ← { nom="Gérard", age=77 }
q ← copie de p
q.nom ← "Eugène"
/* p et q sont différents                                   */

inc (x)
  x← x+1                          /* variable locale */
  retourner x
afficher (inc(a))
/* affiche 13 mais a vaut toujours 12                       */
```

Références à l'assignation

```
a ← 12
b ← référence sur a
b ← 18
/* a et b valent 18                                        */

t ← [1, 5, 9]
u ← réf t
u[1] ← 12
/* u et t valent [1, 12, 9]                                 */

p ← { nom="Bécassine", age=68 }
q ← réf p
q.nom ← "Germaine"
/* p.nom vaut aussi "Germaine"                              */

inc (x : référence)
  x← x+1
  retourner x
afficher (inc (réf a))
/* affiche 19 et a vaut maintenant 19 (b aussi
   d'ailleurs)                                              */
```

Note : nous n'utiliserons pas dans notre langage algorithmique directement de *pointeurs* (ou *adresses* mémoire) qui sont un mécanisme directement lié à un langage de programmation « bas-niveau » (par exemple le C). Les références en revanche sont un mécanisme générique permettant d'associer plusieurs noms (i.e., plusieurs variables) à une même donnée stockée en mémoire.

1.12 Listes chaînées

Il est possible à chaque fois de *copier* (une cellule ou un nœud) ou de faire des *références*. Il faut utiliser des références lors de manipulations de cellules existantes, par exemple lors d'un parcours, et d'utiliser des copies dans des fonctions qui « retournent » des variables locales.

Références

```
type Liste : Cellule | Nil
type Cellule : {
    val : entier,
    suivant : Liste
}
a, b : Cellule
a ← { val : 3, suivant : nil}
b.val ← 5 ;   b.suivant ← réf a ;   tête ← réf b

/* il y a maintenant une seule liste contenant la cellule
   b (= tête) suivie de la cellule a                        */
```

Copies

```
type Liste : Cellule | Nil
type Cellule : {
    val : entier,
    suivant : Liste
}
a, b, tête : Cellule
a ← { val : 3, suivant : nil}
b.val ← 5 ;      b.suivant ← a ;      tête ← b
/* il y a maintenant trois listes !                         */
/* a→nil,       b→a' (copie de a)→nil,                      */
/* et tête (= b', copie de b)→ a'→nil                       */
```

Parcours de liste : avec des références.

```
afficher_liste (tête : Liste)
    c : Cellule
    c ← réf tête
    tant que c ≠ nil faire
        afficher (c.val)
        c ← réf c.suivant
```

Ajout de cellule avec références pour modification de liens, et copie pour la nouvelle cellule.

```
ajoute_en_tête (tête : réf Liste, v : entier)
    c : Cellule
    c.val ← v
    c.suivant ← réf tête
    tête ← c   ou bien   tête ← copie c
```

1.13 Arbres

Idem, préférer les références ou expliciter les copies, par exemple pour des variables locales

```
type Arbre : Nœud | Nil
type Nœud : {
    gauche : Arbre,
    val : entier,
    droite : Arbre
}
g ← {gauche : nil, val : 5, droite : nil}
d ← {gauche : nil, val : 8, droite : nil}
racine ← {gauche : réf g, val : 6, droite : réf d}
```

```
ajouter (n : réf Nœud, v : entier)
    tmp : Nœud
    tmp.val ← v
    /* Ici on va copier tmp car c'est une variable locale
       qui serait sinon perdue à la sortie de la fonction.
       */
    si n.gauche = nil alors
        n.gauche ← copie tmp
    sinon si n.droite = nil alors
        n.droite ← copie tmp
    sinon
        Erreur ("Impossible d'ajouter ", v)
racine : Nœud ; racine.val ← 6
ajouter (réf racine, 5)
ajouter (réf racine, 8)
```