Langage Algorithmique / Pseudo-Code / Langage Pivot

Florent Bouchez Tichadou

9 septembre 2021

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 tous 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
x : entier positif ou nul
y, z : entiers > 0
i : entier entre 0 et LMAX-1
ou bien
j : entier $\in [0 \dots \text{LMAX} - 1]$
b : booléen

r : réel
c : caratère
c' : caratère alphanumérique
str : chaine de caractères
t : tableau de n entiers

cpl : couple d'entiers
upl : 5-uplet de réels

constante PI $= 3,14159$
constante LMAX $= 1000$
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 }
type Cercle : { pos : Point, diam : réel }
type Séquence : { tab : tableau de n entiers, longueur : entier }
type Nombre : entier | réel

p : Point
c : Cercle
S : Séquence
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
triangle : Polygone
nbs : liste de Nombres
```

1.4 Opérations courantes

Add., mul., sous.

```
2 + 3
v * 5
12 − x
```

Divisions

```
7 / 3        /* 2,33... */
7 div 3          /* 2 */
7 mod 3          /* 1 */
```

Comparaisons

```
3 = 5        a ≠ 4
v < 0        r ⩽ 2,5
r > PI       y ⩾ 1
```

Booléens

```
b et b'
b ou b'
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
triangle ← { (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

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

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

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

```
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

faire n fois
  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 0 ⩽ r < 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()
```

```
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 les objets simples (entiers, flottants, booléens) sont copiés (sauf référence explicite, cf. ci-dessous), mais pour les objets plus complexes (tableaux, structures, etc.), nous préférons lever toute ambigüité et préciser *systématiquement* si un objet est copié intégralement ou si c'est simplement sa référence qui est copiée. Cela a un impact considérable sur l'efficacité des algorithmes. En particulier, il est important de préciser cela lors d'un appel et au retour d'une fonction.

Important

Attention ! Habituellement, les langages de programmation usuels ont leur propre convention. 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. Il est important de connaître les conventions du langage utilisé sous peine de faire face à des bugs en apparence incompréhensibles.

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.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.