Chapitre 1

Algorithmique

1.1 Principe de la description dans un langage informel

Les paragraphes 1.2 à 1.4 présentent quelques procédés de description des objets et des instructions les plus courants.
Les paragraphes 1.5 à 1.7 décrivent les structures qui permettent d'écrire tout algorithme.

Vocabulaire
Un *programme* est une suite d'*instructions* qui agissent sur des données. Les instructions sont reliées entre elles par des structures qui expliquent quelles instructions doivent être effectuées, combien de fois elles doivent être effectuées, ...
Une fonction est un type de programme particulier.
On lance l'exécution d'un programme ou d'une fonction en invoquant son nom.
Une instruction peut être un programme ou une fonction déjà défini. Un grand nombre de fonctions sont déjà définis dans le logiciel, par exemple $\sin, \exp, \ldots$
Un *algorithme* est la *description* d'une suite d'instructions qui va agir sur des données pour fournir un résultat en un *nombre fini* d'opérations.
La *programmation descendante* consiste à découper un problème (programme) complexe en plusieurs sous-problèmes élémentaires, que l'on traitera séparément. A chaque sous-problème on fait souvent correspondre un sous-programme ou une fonction. Chaque sous-problème peut lui même être découpé en sous-problèmes. En quelque sorte, la programmation descendante consiste à écrire le plan de résolution du problème. On commence par les titres de paragraphes, puis si nécessaire, on découpe chaque paragraphe en sous-paragraphe, ...

Méthode
Pour écrire un programme, on commence en général par rédiger l'*algorithme*. C'est-à-dire qu'on écrit la suite des instructions de manière informelle mais suffisamment précise pour que son codage dans un langage particulier ne soit plus qu'une traduction banale.

Dans la phase de recherche de l'algorithme, on peut se contenter de décrire un sous programme ou une fonction au moyen de son *cahier des charges* : on se contente de décrire son action.
Dans la phase de recherche de l'algorithme, il est contre-productif de se préoccuper du détail du codage.

Remarque
Dans les premiers exemples de ce manuel, les algorithmes seront très détaillés. Ils deviendront de plus en plus sommaires dans les derniers chapitres.

1.2 Affectation

Description intuitive
L'affectation consiste à ranger des données dans une *variable*.
On peut *imaginer* une variable comme un tiroir muni d'une étiquette portant le nom de la variable. La donnée qui est dans la variable est appelée sa *valeur* ou son *contenu* ou son *affectation*.
Lors de la première affectation d'une variable, elle est crée avec comme contenu sa première affectation. Cette phase est appellée l'*initialisation*.
Lors d'une affectation ultérieure d'une variable, son ancien contenu est effacé puis remplacé par la nouvelle affectation.

Notations
Une notation commode pour l'affectation de la valeur y à la variable X est $\boxed{X \leftarrow y}$
Pour indiquer d'affecter la valeur de la variable X à la variable Y, on note $\boxed{Y \leftarrow X}$
Lorsqu'une opération est décrite à droite de la flèche, le processeur exécute d'abord cette opération, puis affecte le résultat de l'opération.

Exemple 1.2.1
Ecrire un algorithme qui affecte à une variable X la valeur $\pi/5$, puis à une variable Y la valeur $\cos(\pi/5)$, et enfin à Y la valeur $\sin(\cos(\pi/5))$.
Solution :

$$\begin{vmatrix} X \leftarrow \pi/5 \\ Y \leftarrow \cos(X) \\ Y \leftarrow \sin(Y) \end{vmatrix}$$

Exemple 1.2.2 Algorithme d'échange
Ecrire un algorithme qui échange les valeurs de deux variables X et Y.
Solution :
On utilise une troisième variable T qui sert à stocker temporairement l'ancienne valeur de X. Si au début, la valeur de X est x et celle de Y est y, le tableau à droite de l'algorithme indique les valeurs des variables X, Y, T à chaque étape.

Algorithme
début
$T \leftarrow X$;
$X \leftarrow Y$
$Y \leftarrow T$

Instruction	*début*	$T \leftarrow X$	$X \leftarrow Y$	$Y \leftarrow T$
X	x	x	y	y
Y	y	y	y	x
T	non définie	x	x	x

1.3 Lecture, Ecriture, Chaînes, Commentaires

Lecture

Pour expliquer que l'algorithme doit lire une valeur (disons x_0 pour fixer les idées) et l'affecter à une variable X, on peut noter *lire(X)* au lieu de $X \leftarrow x_0$.
Pour expliquer que l'algorithme doit lire des valeurs et les affecter à des variables X et Y, il est commode de noter *lire(X,Y)*, ...

Ecriture

Pour expliquer que l'algorithme doit écrire la valeur d'une variable X, il est commode de noter *écrire(X)*.
Pour expliquer que l'algorithme doit dessiner un objet, disons pour fixer les idées le graphe d'une fonction f entre a et b, il est commode de noter :
tracer $(y = f(x), a \leq x \leq b)$.
La traduction de ces instructions dans la majorité des langages est directe.

Chaînes de caractères

Une suite de lettres s'appelle une chaîne de caractères. Par lettre on entend tous les caractères individuels (comme les chiffres, les symboles) que connait le processeur. Pour représenter un mot ou une phrase (suite de lettres) et ne pas le confondre avec [le nom d']une variable, on écrit les chaînes de caractères entre apostrophes.

Variable dont la valeur est une chaine de caractères

Limitons nous à donner un exemple.
L'affectation codée `S <-- 'AACTGTGATTAGCGT'` définit une variable dont :
— le nom est S,
— le contenu est la chaîne de caractères `'AACTGTGATTAGCGT'`.
Ici S peut représenter un segment d'ADN.
On notera dans la suite `S(k)` ou S_k le k-ième caractère (non compris l'apostrophe initiale) de S et $S(k_1 : k_2)$ la suite formée des lettres entre $S(k_1)$ et $S(k_2)$. Dans l'exemple, S_3 est la lettre `C` et `S(4 :6)` est le mot `TGT`.

Commentaires

Pour *expliquer* la signification d'une variable, l'action d'une instruction ou d'une suite d'instruction, il est commode de la décrire en français à droite de l'algorithme, en séparant *si nécessaire* au moyen d'un caractère spécial ; on utilisera la notation de MatLab qui est % ; celle de Scilab est //

Exemple 1.3.1
Dans le programme suivant, on crée une variable *hello* dont la valeur est 36, on demande l'affichage de *hello*, puis du texte formé des 5 lettres `hello`.

```
hello←36 ;
écrire(hello)          % écrit 36
écrire('hello')        % écrit hello
```

1.4 Variables logiques

En *logique classique*, on définit deux valeurs logiques : **`(V[rai],F[aux])`** .
En informatique, une *expression logique* est une phrase grammaticalement correcte qui fournit *en un temps fini* une des deux valeurs logiques **`(Vrai,Faux)`**.

Exemple 1.4.1
Si n est un nombre entier, l'expression $(n > 10)$ est une expression logique.
Lorsque n prend pour valeur 36, $(n > 10)$ renvoie la valeur **`Vrai`**,
lorsque n prend pour valeur -5, $(n > 10)$ renvoie la valeur **`Faux`**.

Opérateurs logiques
Les lettres $\mathcal{P}$, $\mathcal{Q}$ représentent des expressions logiques.
On définit les opérateurs : **`et, ou, non`** par le tableau ci-dessous

$\mathcal{P}$	$\mathcal{Q}$	$\mathcal{P}$ et $\mathcal{Q}$	$\mathcal{P}$ ou $\mathcal{Q}$	non($\mathcal{P}$)
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

Règles de calcul
Les lettres $\mathcal{P}, \mathcal{Q}, \mathcal{R}$ sont des expressions logiques. On a les règles suivantes :
non(non$\mathcal{P}$) $\Longleftrightarrow \mathcal{P}$
non($\mathcal{P}$ ou $\mathcal{Q}$) $\Longleftrightarrow \Big($(non$\mathcal{P}$) et (non$\mathcal{Q}$)$\Big)$
non($\mathcal{P}$ et $\mathcal{Q}$) $\Longleftrightarrow \Big($(non$\mathcal{P}$) ou (non$\mathcal{Q}$)$\Big)$
$\mathcal{P}$ et ($\mathcal{Q}$ ou $\mathcal{R}$) $\Longleftrightarrow \Big($ ($\mathcal{P}$ et $\mathcal{Q}$) ou ($\mathcal{P}$ et $\mathcal{R}$)$\Big)$
$\mathcal{P}$ ou ($\mathcal{Q}$ et $\mathcal{R}$) $\Longleftrightarrow \Big($ ($\mathcal{P}$ ou $\mathcal{Q}$) et ($\mathcal{P}$ ou $\mathcal{R}$)$\Big)$

Exemple 1.4.2
Soit n un entier.
L'expression : $(n < 5)$ et $(n \geq 1)$ prend pour valeur **`Vrai`** lorsque $n \in \{1, 2, 3, 4\}$
L'expression : $(n \geq 1)$ ou $(n \leq -1)$ prend pour valeur **`Vrai`** lorsque n est non nul, elle est donc équivalente à $(n \neq 0)$ ou encore à $\mathrm{non}(n = 0)$.

1.5 Alternatives `si ...alors ...`

Pour que l'exécution d'une instruction dépende du résultat d'un *test*, on utilise une *alternative* `si ...alors ...`
Dans la majorité des langages, il y a trois variantes.

Version courte

Si *test* est une *expression logique*, alors la structure : $\begin{cases} \texttt{si test, alors} \\ \quad \texttt{instruction} \\ \texttt{fin(si)} \end{cases}$
effectue *instruction* puis va après *fin(si)* lorsque *test* prend pour valeur `Vrai`
va directement après *fin(si)* lorsque test prend pour valeur `Faux`.

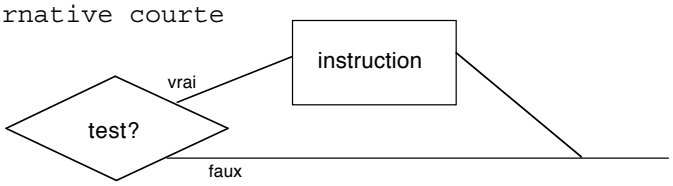
Exemple 1.5.1
Ecrire un algorithme qui demande un nombre X puis écrit `gagné` si ce nombre égal à 0.
solution :

```
lire(X)
si X=0, alors
  ecrire('gagne')
fin(si)
```

Organigramme

Un *organigramme* est un dessin schématique du programme.
Les langages modernes (aprés 1985) les rendent inutiles ; toutefois un organigramme *court* peut être commode et tenir lieu d'algorithme.
Les descriptions schématiques des processus techniques et industriels sont souvent des organigrammes.

`alternative courte`



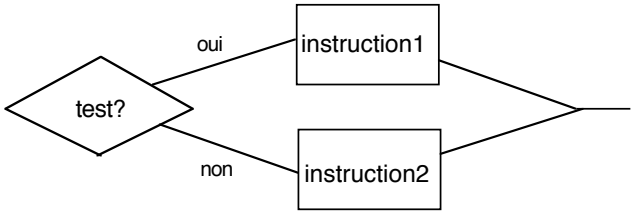
Version normale

Si `test` est une expression logique, alors la structure

```
si test alors
  instruction1
sinon
  instruction2
fin(si)
```

effectue `instruction1` puis va après `end` lorsque test prend pour valeur Vrai
effectue `instruction2` puis va après `end` lorsque test prend pour valeur Faux.

Organigramme de l'alternative `si...alors...sinon ...`



Exemple 1.5.2
Les nombres a et b étant déjà connus, écrire un algorithme qui discute et résout, en fonction de a, b, réels ou complexes, l'équation $aX + b = 0$ d'inconnue X.
Solution :
En math, pour $a, b \in \mathbb{R}$ ou $\mathbb{C}$, la discussion est :

$$\begin{cases} \textbf{si} \quad a = 0 \\ \qquad \textbf{alors} \quad \begin{cases} \textbf{si} \quad b = 0 \\ \qquad \textbf{alors} \quad \text{tout x est solution} \\ \qquad \textbf{sinon} \quad \text{pas de solution} \end{cases} \\ \qquad \textbf{sinon} \quad x = -b/a \end{cases}$$

Ce shéma rend un algorithme inutile, en effet la traduction est directe.
Un algorithme détaillé et inutile est :

```
si  a=0, alors
  si b=0, alors
     ecrire('tout  X  dans C est solution')
  sinon
     ecrire(' pas de solution')
  fin
sinon
  X <--  -b/a
fin
```

Structure : `si ...alors ...sinonsi ...`

Pour emboiter des `si`, on peut utiliser une des deux structures suivantes.

```
si TEST1 alors
   instruction1
sinon
   si TEST2 alors
      instruction2
   sinon
      instruction3
   fin(si interne)
fin(si externe)
```

```
si TEST1 alors
   instruction1
sinonsi TEST2
   instruction2
sinon
   instruction3
fin(si)
```

La structure avec `si ...sinonsi ...` est réputée un peu plus lisible.
Dans chaque version on peut supprimer les mots 'alors' ou 'fin' si l'algorithme devient plus clair et reste non ambigü.

Exemple
Voir exemple 2.10.3 (Matlab) ou 3.10.3 (Scilab).

1.6 Répétition `tant que ... faire ...`

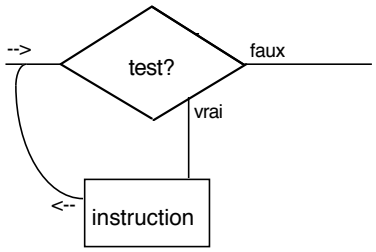
Pour répéter une instruction dont on ne connaît pas le nombre de répétitions, on utilise la structure `tant que ... faire ...`

Considérons la structure ci-contre :

```
tantque test faire
  instruction
fin(tantque)
```

Elle opère comme suit :
Si *test* prend pour valeur **`Faux`**, le programme va aprés `fin(tantque)`
Si *test* prend pour valeur **`Vrai`**, le programme exécute instruction puis revient au début de la structure . . . et ainsi de suite *tant que test* prend la valeur **`Vrai`**.

Organigramme
de la répétition
`tantque ...faire ...`



On peut supprimer les mots 'faire' ou 'fin' si l'algorithme devient plus clair et reste non ambigü.

Exemple 1.6.1
Ecrire un algorithme qui demande un réel strictement positif et réitère sa demande tant-que la valeur lue n'est pas strictement positive.
Solution :
L'idée de base est d'écrire :

```
tantque (il faut continuer)
  demander un reel
fin(tantque)
```

d'où l'idée d'introduire une variable logique `continuer` *qui prend la valeur* `vrai` *s'il faut continuer à demander un réel et* `faux`*, s'il faut arrêter.*

Chacun des deux algorithmes détaillés suivants fournit une solution correcte.
Noter qu'on s'est dispensé d'écrire "alors" et "faire".

```
continuer <-- Vrai
tantque continuer
  lire(X)
  si X>0
      continuer <-- Faux
  sinon
      continuer <-- Vrai
  fin(si)
fin(tantque)
```

```
continuer <-- Vrai
tantque continuer
   lire(X)
   si X>0
      continuer  <-- Faux
   fin(si)
fin(tantque)
```

Répétition `tantque test faire ... instruction` avec sortie en cours d'instruction

Matlab et Scilab disposent d'une instruction permettant de quitter "brutalement" une répétition ; le programme va alors juste après `fin(tantque)`. On peut noter cette instruction `quitter(tantque)`.

Exemple 1.6.2
Soient deux réel $a < b$, une fonction $f : [a,b] \to [a,b]$ et $\varepsilon > 0$. Ecrire un algorithme qui calcule les termes successifs de la suite $(u_n)_n$ définie par $u_0 = a$, $u_{n+1} = f(u_n)$ jusqu'à ce que $|u_{n+1} - u_n| < \varepsilon$ ou qu'il ait calculé u_{10}.
Si le programme s'arrête avant le calcul de u_{10}, il affiche le dernier u_n calculé, sinon il sort un message indiquant l'arrêt du calcul.

Si l'algorithme proposé vous paraît difficile à comprendre, construisez un tableau analogue à celuis de l'exercice 1.2.2, (haut de la page 7).

```
U <-- a;   V <-- f(U);  n <-- 1;        % c-a-d  V<--u(1)
tantque n<10
    n <-- n+1; U <-- V;  V <-- f(U);    % c-a-d  V<--u(n)
    si |U-V|<epsilon
        ecrire (V);  quitter(tantque)
    fin(si)
fin(tantque)
si n=10
   ecrire('10-ieme iteration atteinte')
fin(si)
```

Ce programme est équivalent à ce qui suit, où >= code $\geq$

```
U <-- a;    V <-- f(U);    n <-- 1
tantque (|U-V|>=epsilon et n<10)
   n <-- n+1;  U <-- V;  V <-- f(U);
fin(tantque)
si n=10,
    ecrire ('10-ieme iteration atteinte')
sinon
   ecrire(V)
fin(si)
```

1.7 Structure `pour k= M1 : h : M2 ...`

Si l'on connaît le nombre de répétitions d'une instruction, on utilise une *boucle*.
Lorsque `M1` et `M2` sont des réels et `h` un réel non nul, la structure

```
pour k = M1 :h:  M2
    instruction
fin
```