Chapitre II

Les objets de bases de l'algorithmique

Sommaire

I. Que retenir des séances de travail sur Guido ?

Nous avons vu dans les premières séances utilisant l'outil d'initiation *Guido van Robot* la plupart des éléments de bases des algorithmes[1] : les commandes élémentaires (avancer, tourner...), les *structures* (conditionnelles, boucles itératives ou conditionnelles) et les *fonctions*.

Nous allons voir dans ce premier chapitre les objets de base manipulés par les algorithmes : les *variables*. Nous verrons aussi de quelle manière l'ordinateur communique avec le monde extérieur.

II. Les commentaires, l'indentation du code

Du point de vue de leur action sur l'ordinateur, les commentaires ne servent à rien ! Le compilateur (ou l'interpréteur), lorsqu'il rencontre un commentaire, l'ignore superbement et recherche l'instruction suivante.

Pourquoi alors commencer par quelque chose qui ne sert à rien ? Tout simplement parce que pour nous pauvres humains, ces commentaires sont au contraire essentiels ! Voici un certain nombre de raisons :

- écrire un commentaire permet de s'assurer de la validité du code qu'on écrit ; si l'on ne sait pas pourquoi on écrit une ligne de code, c'est peut-être qu'il faut revoir l'algorithme dans son ensemble !

- si le code qu'on a écrit ne fonctionne pas, ou ne donne pas les résultats attendus, il va falloir le relire ; dans ce cas, il est toujours agréable d'avoir un balisage, surtout si le source à relire dépasse quelques dizaines de lignes ;

- si une autre personne utilise le code, et veut le corriger, l'étendre, ou simplement le comprendre, il est encore plus fondamental qu'il soit très précisément commenté ; il est très difficile de comprendre un code qu'on n'a pas écrit soi-même[2] ; donc dans un environnement de travail où plusieurs personnes doivent cohabiter, des commentaires précis doivent être ajoutés (il est d'ailleurs utile, dans ce cadre, de définit des règles de présentation et d'organisation du code communes) ;

- lorsque vous écrirez des algorithmes lors de vos évaluations, il vous sera demandé de commenter certains passages de votre code, pour montrer que vous avez compris ce que vous faites, ou pour justifier vos choix ; plus vous faciliterez la tâche du correcteur, meilleure sera votre note, toutes autres choses étant égales par ailleurs !

Algorithmique	Traduction Pascal
(* Un commentaire *)	*(* Un commentaire *)* *{ Un autre commentaire* *sur plusieurs lignes }* *{ (* un commentaire imbriqué* *dans un autre *) }*

Toujours pour des raisons de lisibilité, on s'attachera à *indenter* correctement le code. Nous avons vu avec les séances Guido que c'était une nécessité syntaxique en Python. En Pascal, cela n'a aucune importance syntaxique, cela en a par contre une grande pour la lisibilité. Comparez les deux présentations suivantes :

[1] à l'exception peut-être de la notion de *variable*, qui existe, mais est un peu cachée !

[2] C'est même un indice du fait que la programmation est plus un art qu'une science : à chaque programmeur son style, et sa façon d'interpréter la partition des spécifications qui lui sont imposé.

```
Program ToursDeHanoi;
Var disks:integer;
Procedure Hanoi(source, temp, destination: char;  n: integer);
begin if n>0 then begin
Hanoi(source,destination,temp,n-1);
writeln('Deplacer le disque ',n:1,' du pied ',source,' au pied ',destination);
Hanoi(temp,source,destination,n-1);
end;end;
begin
write('Entrer le nombre de disques : ');
readln(disks);
writeln('Solution :');
Hanoi('A','B','C',disks);
end.
```

Un code illisible

```
Program ToursDeHanoi;

Var disks : integer;

Procedure Hanoi(source, temp, destination: char;  n: integer);

begin
   if n > 0 then
   begin
      Hanoi(source, destination, temp, n - 1);
      writeln('Deplacer le disque ',n:1,' du pied ',source,' au pied ',destination);
      Hanoi(temp, source, destination, n - 1);
   end;
end;

begin
   write('Entrer le nombre de disques : ');
   readln(disks);
   writeln('Solution :');
   Hanoi('A','B','C',disks);
end.
```

Un code plus clair

III. Les entrées-sorties

Pour interagir avec son (ou ses) utilisateur, un programme doit pouvoir lire des données provenant du “monde extérieur”. Nous utiliserons essentiellement le clavier comme instrument d’entrée, même si on parlera des lectures de fichiers. On peut aussi envisager des lectures de données présentes sur la carte réseau, ou les actions d’une souris. Vous verrez tout ceci en détail avec vos professeurs d’informatique.

Nous utiliserons l’instruction `Lire` (ou `Lireln`) lorsque nous écrirons un algorithme. Sa traduction en Pascal est `read` ou `readln`[3].

Pour écrire le résultat d’un calcul, un programme doit de même pouvoir transmettre des données au monde extérieur. De la même façon, nous écrirons principalement nos résultats sur une *console* (un écran en mode texte). Nous parlerons plus tard d’écriture de fichiers, et vous verrez en cours d’informatique d’autres moyens de communication plus sophistiqués, comme l’écran graphique, le réseau...

[3]La différence entre les deux versions de la fonction est simplement un passage à la ligne après la dernière saisie pour la version `readln`.

La version algorithmique de l'opération d'écriture est `Ecrire` ou `Ecrireln`, et sa version Pascal est `write` ou `writeln`[4].

EXEMPLES :

1) Le programme le plus simple qu'on peut concevoir est le classique "Bonjour Monde" ("Hello World" pour les anglais) :

Algorithmique	Traduction Pascal

`Ecrire('Bonjour Monde !')`	`program HelloWorld;`
	
	`begin`
	`  writeln('Bonjour Monde !');`
	`  readln`
	`end.`

2) Voici un programme qui demande le prénom de l'utilisateur, et lui souhaite la bienvenue :

Algorithmique	Traduction Pascal

`Ecrire('Quel est votre prenom ? ')`
`Lire(prenom);`
`Ecrire('Bonjour, ',prenom,' !');`
	`program HelloWorld;`
	
	`var prenom : string;`
	
	`begin`
	`  write('Quel est votre prenom ? ');`
	`  readln(prenom);`
	`  writeln('Bonjour, ',prenom,' !');`
	`  readln`
	`end.`

IV. LES VARIABLES ET LES TYPES DE DONNÉES SIMPLES

A. Les variables

Un ordinateur n'a aucune "mémoire". Ou plutôt, il ne met pas automatiquement comme nous en mémoire les résultats de ses calculs. Si l'on veut qu'il se souvienne d'un résultat, il faut le lui indiquer explicitement, sous forme d'une *instruction d'affectation*.

En Pascal comme dans beaucoup d'autres langages, les variables doivent être *déclarées* avant d'être utilisées, comme indiqué ci-dessous :

Algorithmique	Traduction Pascal

`x <- 2+3`	`var x : integer;`
	`...`
	`x := 2+3;`

Les instructions ci-dessus font les choses suivantes :

- la déclaration `var` prépare de l'espace en mémoire pour une variable nommée `x`, qui sera de type `integer` (entier),
- le calcul à droite du symbole d'affectation `:=` est effectué,

[4] Même remarque, plus une autre : en informatique, si on veut écrire des accents dans les chaînes de caractères, on doit se préparer à quelques nuits blanches à lire des documentations, évidemment sans accents car en anglais ! Pour éviter des tracas (qui en plus ne sont pas transposables d'un environnement à l'autre), on évitera l'usage des accents dans le cours d'algorithmique. Mme Gallot vous dira certainement que c'est une bien piètre solution, mais je fais ce que je veux !

- le résultat de ce calcul est rangé dans la case mémoire dont l'adresse est référencée par le nom "x".

Ainsi, si l'on a besoin de cette valeur plus tard, il suffira d'utiliser le nom de la variable :

Algorithmique	Traduction Pascal
y <– x–1	**var** y : **integer**; ... y := x–1;

Ici, l'ordinateur commence par calculer "x–1", remplaçant x par la valeur 5 stockée dans la mémoire de nom x, puis il range le résultat (soit 4 dans la case mémoire de nom[5] y.

D'un point de vue pratique, il est important que les noms des variables soient correctement choisis : positionx est bien plus clair à relire que u ! Un indice de boucle sera en général noté i, j, k... Ne pas hésiter à commenter le code pour éclairer le lecteur sur la signification et l'utilité d'une variable.

EXERCICE : Comment implémenteriez vous la notion de variable dans l'environnement Guido[6] ? Cela vous semble-t-il pratique ?

B. Les types de données simples, et les opérateurs associés

Les types les plus simples[7] sont :

boolean : le type des booléens, les valeurs de vérités, notées par les constantes false (pour faux) et true (pour vrai). Les opérateurs utilisables avec ces variables sont :

Opérateur	Algorithmique	Traduction Pascal
négation	non	not
conjonction	et	and
disjonction	ou	or
ou exclusif	oubien	xor

integer : le type des entiers signés, appartenant au sous-ensemble $[-\texttt{maxint}, \texttt{maxint}]$, maxint étant une valeur prédéfinie dépendant de l'ordinateur utilisé. Pour un ordinateur 32 bits, $\texttt{maxint} = 2^{32} - 1 = 2\,147\,483\,647$. Les opérateurs utilisables avec ces variables sont :

Opérateur	Traduction Algorithmique/Pascal
addition	+
soustraction	–
multiplication	*
division entière	div
reste d'une division	mod

On remarquera qu'il n'y a pas d'opérateur d'exponentiation, et on se souviendra que l'opérateur div est l'opérateur de division *entière* : 5 div 2 donne pour résultat 2, et non pas 2.5 (qui n'est d'ailleurs pas un entier !).

real : le type des réels en virgule flottante. En dehors de 0, sont codés les nombres *décimaux* compris entre 10^{-37} et 10^{37}, et ayant 7 ou 8 chiffres après la virgule dans leur représentation en notation ingénieur (on parle de *chiffres significatifs*. On utilise la lettre E pour l'exposant :

[5] Notons qu'en général, le programmeur comme l'utilisateur du programme ne connaissent pas l'endroit de la mémoire pointé par les noms x et y. Ces "adresses" sont fournies par le système d'exploitation à la demande, et peuvent très bien changer d'une exécution du programme à l'autre. Ces problèmes de gestion de mémoire sont très intéressants, mais sortent largement du cadre de ce cours.

[6] Sachez qu'il existe une version étendue de GvRng qui inclue le concept de variable, et qui en fait est une véritable initiation à la programmation Python : c'est RUR-PLE (http://rur-ple.sourceforge.net/). Les plus "accros" à Guido y trouveront d'autres exercices facilement transposable à GvRng, et montrant qu'on peut demander à Guido d'apprendre à additionner des nombres !

[7] qui sont d'ailleurs ceux que peut renvoyer une fonction en Pascal

-123450 est noté `-1.2345E5`, ce qui signifie $-1{,}2345 \times 10^5$, 0.0001548 se note `1.548E-4` qui signifie $1{,}548 \times 10^{-4}$.

Les opérateurs utilisables avec ces variables sont :

Opérateur	Algorithmique	Traduction Pascal
addition	+	+
soustraction	−	−
multiplication	*	*
division	/	/
partie entière	ent	trunc
carré	$()^2$	sqr
racine carrée	$\sqrt{}$	sqrt
logarithme népérien	ln	ln
exponentielle	exp	exp
nombre aléatoire	rand	random

On remarquera encore une fois qu'il n'y a pas d'opérateur d'exponentiation. On verra dans le cours de mathématique comment en fabriquer un. D'autres fonctions existent, mais sont moins utiles, la documentation permet de les retrouver au cas par cas.

D'autre part, le générateur de nombre aléatoire doit être initialisé par un appel à la fonction `randomize`. `rand` renvoie un nombre pseudo-aléatoire x tel que $0 \leqslant x < 1$, et si `limite` est un entier positif, `rand(limite)` renvoie un entier y tel que $0 \leqslant y <$ `limite`.

char : c'est le type des *caractères*. Chaque caractère est codé sous la forme d'un octet représentant son *code ASCII*. On écrit les caractères usuels entre guillemets : `'T'`. On peut aussi écrire leur code ASCII précédé du symbole `#` : `#65` est équivalent à `'A'`. Enfin, on peut utiliser les caractères de contrôle en écrivant une lettre précédée d'un accent circonflexe : Ĝ est équivalent à `#7`, et permet d'émettre une sonnerie.

V. Les fonctions et procédures

Il est assez rare qu'une tâche puisse être réalisée d'un bloc. En général, le problème à résoudre sera découpé en sous-tâches plus simples, soit parce que cela simplifiera leur mise au point, soit parce que certaines sous-tâches seront utilisées à plusieurs endroit du programme.

Par exemple, un programme réalisant un crible d'Erathostene est très intéressant en lui-même, mais il est plus que probable que le programmeur ne veut pas juste contempler le tableau de nombres premiers qu'il produit, mais utiliser ces nombres premiers pour réaliser une autre tâche. On pourrait bien-sûr insérer le code de notre programme dans un programme plus vaste, mais ça n'est pas la bonne méthode.

Dans un autre programme, on peut avoir besoin de réaliser une tâche complexe (c'est-à-dire ne se réduisant pas à une ou deux instructions) à plusieurs endroits différents du code. On peut bien entendu écrire une fois le code, puis faire des copier-coller partout où on en a besoin. Mais là encore, ce n'est pas la bonne méthode. En effet, que se passera-t-il si l'on s'aperçoit qu'une erreur a été commise ? Ou bien si on veut modifier notre façon de procéder ? Il faudra alors chercher dans tout le programme le code à corriger, ce qui s'avèrera bien compliqué si le projet dépasse les quelques centaines de lignes.

Une bien meilleure méthode consiste à regrouper l'ensemble des instructions réalisant la sous-tâche dans une boîte, et se donner un moyen d'utiliser cette boîte là où on en a besoin. On appelle une telle boîte une *procédure*, ou une *fonction*, selon le mode d'utilisation.

Cette méthode de découpage des programmes en procédures et fonctions réalisant une partie de la tâche principale est très efficace pour analyser correctement un programme, le tester efficacement, et aussi pour diviser le travail entre plusieurs programmeurs. Il est alors nécessaire de bien préciser les spécifications de chacun des éléments, de manière à ce qu'ils s'emboîtent bien les uns dans les autres !

Par la suite, nous nous contenterons très souvent d'écrire des procédures et des fonctions, et ce sera votre travail que de les inclure dans des programmes pour les tester et les utiliser.

A. Procédures

Une procédure réalise une tâche à partir des arguments qui lui sont donné. Une procédure est aussi appelée un *sous-programme*. En pascal, elle est définie de la façon suivante :

```
procedure <nom>(<liste des paramètres>);

var <liste des variables internes>

begin
  <corps de la procedure>
end;
```

Ceci définit une procédure de nom `nom`. Les variables internes sont les variables qui seront utilisées pour réaliser le calcul. Ces variables sont créées au moment où la procédure est utilisée, et détruites la fin de son exécution. Elle ne sont pas *visibles* à l'extérieur de la procédure.

Lorsque la procédure est appelée, les instructions formant son corps sont exécutées avec les valeurs des paramètres. Voici un exemple d'une fonction `range` qui prend en argument deux entiers `x` et `y`, et qui place dans `x` la plus grande valeur et dans `y` la plus petite :

```
procedure range(var x,y : integer);

var u : integer;

begin
   if x<y then begin
(* si x<y,                    *)
      u := y;
      y := x; (* on echange les valeurs de x et y *)
      x := u
   end        (* sinon, il n'y a rien a faire !
*)
end; {range}
```

B. Fonctions

La seule différence notable entre une procédure et une fonction est que cette dernière renvoie une valeur.

```
function <nom>(<liste des paramètres>) : <type>;

var <liste des variables internes>

begin
  <corps de la fonction>
end;
```

`type` est le type de la valeur retournée, qui ne peut être qu'un type simple (une fonction ne peut pas renvoyer un tableau ou une chaîne de caractères). Dans le corps de la fonction, `nom` est utilisé comme une variable de type `type`, et sa valeur finale est retournée comme résultat du calcul.

Voici par exemple une fonction `sdiv` prenant en argument un entier `n`, et renvoyant la somme de ses diviseurs, ainsi qu'un programme l'utilisant pour trouver tous les *nombres parfaits*, c'est-à-dire les nombres dont le double est égal à la somme de leurs diviseurs :

```
program parfaits;

const N = 10000;
var k : integer;

function sdiv(n : integer) : integer;
var i : integer;
begin
   sdiv := 0;
   i := 1;
   while i <= n do begin
      if n mod i = 0 then sdiv := sdiv + i;
      i := i + 1
   end
end; {range}

begin
   for k := 2 to N do
      if sdiv(k)=2*k then writeln(k);
   readln
end.
```

C. Passage des paramètres

Comme on l'a vu plus haut, on peut passer les arguments d'une procédure de deux façons différentes[8], selon qu'on fait précéder le nom de la variable dans la déclaration du mot-clé `var` ou pas. Cela modifie la façon dont la fonction va influer sur ce paramètre.

Paramètres transmis par valeur : lorsque le nom de la variable n'est pas précédé de `var`, l'argument est transmis *par valeur*. Cela signifie que la procédure peut utiliser la valeur de l'argument (et éventuellement la modifier dans le cours du calcul), mais qu'à la fin de la tâche, la variable conserve la valeur qu'elle avait avant. En pratique, au moment de l'appel, le compilateur crée une nouvelle variable, dans laquelle il copie la valeur de l'argument, et c'est cette nouvelle variable qui est utilisée à l'intérieur de la procédure.

Paramètres transmis par adresse : lorsque le nom de la variable est précédé de `var`, l'argument est transmis *par adresse*. Cela signifie que la procédure travaille avec la variable originale, et non pas une copie. Toute modification de la valeur de cette variable dans le corps de la procédure perdurera après la fin de son exécution.

Voici deux exemple permettant de comprendre ce qui se passe :

[8] Une procédure ou une fonction peut agir d'une autre façon sur les paramètres du programme : c'est en modifiant les variables *globales*, qui ne font pas partie de ses paramètres d'appel. Même si ce procédé est parfaitement valide, il entraîne de nombreuses complications dans la correction et la mise au point des programmes, et doit donc être évité autant que faire se peut. Pour bien faire, les seuls paramètres globaux qu'une fonction devrait avoir le droit d'utiliser sont les *constantes* globales.

La *programmation fonctionnelle* est un paradigme de programmation dans lequel on se refuse à toute modification de variables globales. Sans aller jusqu'à cet extrémisme, on s'efforcera de minimiser l'usage des variables globales, et surtout leur modification par des procédures en dehors des transmission par adresse.

Passage par valeur	Passage par adresse
`program pass_valeur;`	`program pass_valeur;`
`var x : integer;`	`var x : integer;`
`procedure toto(u : integer);`	`procedure toto(var u : integer);`

`begin`
`  u := 3;`
`  writeln('Pendant l''appel, x=',u);`
`end; { toto }`
	`begin`
	`  u := 3;`
	`  writeln('Pendant l''appel, x=',u);`
	`end; { toto }`

`begin`
`  x := 2;`
`  writeln('Avant l''appel, x=',x);`
`  toto(x);`
`  writeln('Apres l''appel, x=',x);`
`  readln`
`end.`
	`begin`
	`  x := 2;`
	`  writeln('Avant l''appel, x=',x);`
	`  toto(x);`
	`  writeln('Apres l''appel, x=',x);`
	`  readln`
	`end.`

On voit que la seule différence entre ces deux programmes est la ligne d'en-tête de définition de la procédure `toto`. La différence se fait à l'exécution lors du troisième affichage : le programme de gauche montre que `x` vaut 2, celui de droite montre que `x` a changé de valeur et vaut maintenant 3.