Rappel : ce cours d'algorithmique et de programmation
est enseigné à l'Université Paris 7,
dans la spécialité PISE du Master MECI (ancien DESS AIGES)
par Christophe Darmangeat

Partie 12
Notions Complémentaires

« *Le danger, avec les ordinateurs, ce n'est pas tellement qu'ils deviennent aussi intelligents que les hommes, mais c'est que nous tombions d'accord avec eux pour les rencontrer à mi-chemin* » - Bernard Avishai

Une fois n'est pas coutume, ce chapitre ne sera l'objet d'aucun exercice. Cela ne veut pas dire pour autant que ce qui s'y trouve n'est pas intéressant.

Non mais des fois.

1. Programmation structurée

Petit retour sur une notion très rapidement survolée plus haut : celle de « programmation structurée ». En fait, nous avons jusqu'à présent, tels Monsieur Jourdain, fait de la programmation structurée sans le savoir. Aussi, plutôt qu'expliquer longuement en quoi cela consiste, je préfère prendre le problème par l'autre bout : en quoi cela ne consiste pas.

Dans certains langages (historiquement, ce sont souvent des langages anciens), les lignes de programmation portent des numéros. Et les lignes sont exécutées par la machine dans l'ordre de ces numéros. Jusqu'ici, en soi, pas de problème. Mais l'astuce est que tous ces langages, il existe une instruction de branchement, notée **aller à** en pseudo-code, instruction qui envoie directement le programme à la ligne spécifiée. Inversement, ce type de langage ne comporte pas d'instructions comme FinTantQue, ou FinSi, qui « ferment » un bloc.

Prenons l'exemple d'une structure « Si … Alors … Sinon »

```
Programmation Structurée
Si condition Alors
  instructions 1
Sinon
  instructions 2
FinSi

Programmation non structurée
1000 Si condition Alors Aller En 1200
1100 instruction 1
1110 etc.
1120 etc.
1190 Aller en 1400
1200 instruction 2
1210 etc.
1220 etc.
1400 suite de l'algorithme
```

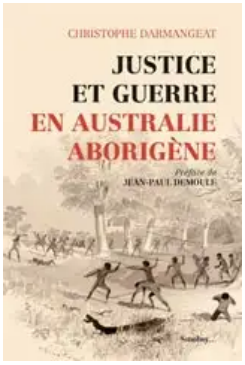
Vous voyez le topo : un programme écrit dans ce type de langages se présente comme **une suite de branchements emmêlés les uns dans les autres**. D'une part, on ne peut pas dire que cela favorise la lisibilité du programme. D'autre part, c'est une source importante d'erreurs, car tôt ou tard on oublie un « aller à », ou on un met un de trop, etc. A fortiori lorsqu'on complique un algorithme existant, cela peut devenir un jungle inextricable.

A l'inverse, la programmation structurée, surtout si l'on prend soin de rationaliser la présentation en mettant des lignes de commentaires et en pratiquant l'indentation, évite des erreurs, et révèle sa structure logique de manière très claire.

Très loin de l'informatique, pas tout près de l'économie :

Mon blog, la Hutte des Classes
À propos d'anthropologie sociale, de préhistoire et de marxisme.

Et mes livres...



Le danger est que si la plupart des langages de programmation utilisés sont structurés, ils offrent tout de même la plupart du temps la possibilité de pratiquer la programmation non structurée. Dans ce cas, les lignes ne sont pas désignées par des numéros, mais certaines peuvent être repérées par des noms (dits « étiquettes ») et on dispose d'une instruction de branchement.

> Une règle d'hygiène absolue est de programmer systématiquement de manière structurée, sauf impératif contraire fixé par le langage (ce qui est aujourd'hui de plus en plus rare).

Autrement dit, même quand un langage vous offre une possibilité de faire des entorses à la programmation structurée, il ne faut s'en saisir sous aucun prétexte.

Retour Haut de Page

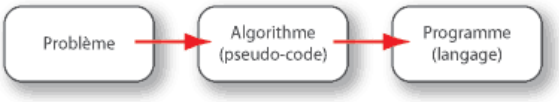
2. Interprétation et compilation

Avec ce paragraphe, on sort un peu de l'algorithmique proprement dite pour entrer dans le domaine plus technique de la réalisation pratique. Ou, si l'on préfère, ces dernières lignes sont l'apothéose, le bouquet final, l'extase ultime, la consécration grandiose, de ce cours.

En toute modestie, bien sûr.

Jusqu'ici, nous avons travaillé sur la première étape de la réalisation d'un programme : la rédaction de l'algorithme.

En fait, si l'algorithme est bien écrit, sans faute logique, l'étape suivante ne doit normalement poser aucun problème conceptuel. Il n'y a plus qu'à effectuer une simple traduction.



A partir de là, le travail du programmeur est virtuellement terminé (en réalité, il reste tout de même une inévitable phase de tests, de corrections, etc., qui s'avère souvent très longue). Mais en tout cas, pour l'ordinateur, c'est là que les ennuis commencent. En effet, aucun ordinateur n'est en soi apte à exécuter les instructions telles qu'elles sont rédigées dans tel ou tel langage ; l'ordinateur, lui, ne comprend qu'un seul langage, qui est un langage codé en binaire (à la rigueur en hexadécimal) et qui s'appelle le langage machine (ou assembleur).

C'est à cela que sert un langage : à vous épargner la programmation en binaire (une pure horreur, vous vous en doutez) et vous permettre de vous faire comprendre de l'ordinateur d'une manière (relativement) lisible.

C'est pourquoi tout langage, à partir d'un programme écrit, doit obligatoirement procéder à une **traduction** en langage machine pour que ce programme soit exécutable.

Il existe deux stratégies de traduction, ces deux stratégies étant parfois disponibles au sein du même langage.

- le langage traduit les instructions au fur et à mesure qu'elles se présentent. Cela s'appelle la **compilation à la volée**, ou **l'interprétation**.
- le langage commence par traduire l'ensemble du programme en langage machine, constituant ainsi un deuxième programme (un deuxième fichier) distinct physiquement et logiquement du premier. Ensuite, et ensuite seulement, il exécute ce second programme. Cela s'appelle la **compilation**

Il va de soi qu'un langage interprété est plus maniable : on peut exécuter directement son code - et donc le tester - au fur et à mesure qu'on le tape, sans passer à chaque fois par l'étape supplémentaire de la compilation. Mais il va aussi de soi qu'un programme compilé s'exécute beaucoup plus rapidement qu'un programme interprété : le gain est couramment d'un facteur 10, voire 20 ou plus.

Toute application destinée à un usage professionnel (ou même, tout simplement sérieux) est forcément une application compilée.

Retour Haut de Page

3. Une logique vicelarde : la programmation récursive

Vous savez comment sont les informaticiens : on ne peut pas leur donner quoi que ce soit sans qu'ils essayent de jouer avec, et le pire, c'est qu'ils y réussissent.

La programmation des fonctions personnalisées a donné lieu à l'essor d'une logique un peu particulière, adaptée en particulier au traitement de certains problèmes mathématiques (ou de jeux) : la programmation récursive. Pour vous expliquer de quoi il retourne, nous allons reprendre un exemple cher à vos cœurs : le calcul d'une factorielle (là, je sentais que j'allais encore me faire des copains).

Rappelez-vous : la formule de calcul de la factorielle d'un nombre n s'écrit :

N ! = 1 x 2 x 3 x … x n

Nous avions programmé cela aussi sec avec une boucle Pour, et roule Raoul. Mais une autre manière de voir les choses, ni plus juste, ni moins juste, serait de dire que quel que soit le nombre n :

n ! = n x (n-1) !

En bon français : la factorielle d'un nombre, c'est ce nombre multiplié par la factorielle du nombre précédent. Encore une fois, c'est une manière ni plus juste ni moins juste de présenter les choses ; c'est simplement une manière différente.

Si l'on doit programmer cela, on peut alors imaginer une fonction Fact, chargée de calculer la factorielle. Cette fonction effectue la multiplication du nombre passé en argument par la factorielle du nombre précédent. Et cette factorielle du nombre précédent va bien entendu être elle-même calculée par la fonction Fact.

Autrement dit, on va créer une fonction qui pour fournir son résultat, **va s'appeler elle-même un certain nombre de fois**. C'est cela, la récursivité.

Toutefois, il nous manque une chose pour finir : quand ces auto-appels de la fonction Fact vont-ils s'arrêter ? Cela n'aura-t-il donc jamais de fin ? Si, bien sûr, rassure-toi, ô public, la récursivité, ce n'est pas Les Feux de L'Amour. On s'arrête quand on arrive au nombre 1, pour lequel la factorielle est par définition 1.

Cela produit l'écriture suivante, un peu déconcertante certes, mais parfois très pratique :

```
Fonction Fact (N en Numérique)
Si N = 0 alors
  Renvoyer 1
Sinon
  Renvoyer Fact(N-1) * N
Finsi
Fin Fonction
```

Vous remarquerez que le processus récursif remplace en quelque sorte la boucle, c'est-à-dire un processus itératif. Et en plus, avec tous ces nouveaux mots qui riment, vous allez pouvoir écrire de très chouettes poèmes. Vous remarquerez aussi qu'on traite le problème à l'envers : on part du nombre, et on remonte à rebours jusqu'à 1 pour pouvoir calculer la factorielle. Cet effet de rebours est caractéristique de la programmation récursive.

Pour conclure sur la récursivité, trois remarques fondamentales.

- la programmation récursive, pour traiter certains problèmes, est **très économique pour le programmeur** ; elle permet de faire les choses correctement, en très peu d'instructions.

- en revanche, elle est **très dispendieuse de ressources machine**. Car à l'exécution, la machine va être obligée de créer autant de variables temporaires que de « tours » de fonction en attente.
- Last but not least, et c'est le gag final, **tout problème formulé en termes récursifs peut également être formulé en termes itératifs !** Donc, si la programmation récursive peut faciliter la vie du programmeur, elle n'est jamais indispensable. Mais ça me faisait tant plaisir de vous en parler que je n'ai pas pu résister… Et puis, accessoirement, même si on ne s'en sert pas, en tant qu'informaticien, il faut connaître cette technique sur laquelle on peut toujours tomber un jour ou l'autre.

Retour Haut de Page