Programmation Impérative : Méthodologie

Introduction

Le développement d'un programme dépend de la taille et de la complexité du problème. Nous identifions deux façons de procéder pour développer :

- **Programmation en petit**

  - Ici la taille du problème, d'une manière générale, est petite.
  - Le modèle de solution peut être exprimé par une action plus au moins complexe.
  - Il s'agit donc de développer un algorithme.

- **Programmation en large**

  - La taille du problème est importante ce qui conduit à une nécessité de spécifier les données complexes utilisées dans le problème.
  - Le programme est donc composé par une implémentation de ces données et des opérations qui les manipulent.
  - Une opération n'est qu'une action qui sera développée selon la programmation en petit.

Dans ce qui suit nous nous intéressons à introduire une démarche méthodologique pour développer des **programmes en petit**. Cette démarche consiste, pour un problème donné, à :

1. Comprendre le problème à automatiser

2. Spécifier le programme qui automatise ce problème

3. Proposer des modèles abstraits de solutions

4. Implémenter avec vérification le modèle de solution choisit

1. Comprendre le problème

Différents aspects sont à comprendre :

- Répondre à la question : est-ce que le problème est calculable sur un ordinateur ?

  Exemple de problèmes :
  non calculable :Ecrire un programme qui corrige toutes les fautes de dictée dans un texte.
  Calculable : Ecrire un programme qui, pour une valeur entier naturel N, calcule N!

- Etudier la logique du problème, ses constituants càd son domaine, ses acteurs et ses données et leurs relations.

  Exemple : Ecrire un programme qui, pour une valeur entier naturel N, calcule N!

  Domaine : arithmétique
  N! est définit pour N naturel : 1 si N=0 et N*(N-1)! si N>0
  Pour le programme, N doit être une entrée.

- Etudier les limitations physiques : N ! est calculable sur une machine et non sur une autre ! ça dépend de la capacité de la machine et de la valeur de N. Sur une machine où la plus grande valeur ne peut dépasser $2^8 - 1$ on peut calculer jusqu'à 6 A partir de N=7 on dépasse la capacité physique de la machine.

- Souvent pour comprendre le problème nous avons besoin d'étudier des exemples et d'essayer de décrire abstraitement les étapes de calcul. Le choix des exemples est souvent problématique et dépend de la compréhension de l 'application à programmer. Cette compréhension passe obligatoirement par :

  - Une bonne lecture (plusieurs lecture) du texte décrivant l'application
  - Une discussion, pour mieux préciser le but à atteindre, avec le client qui a posé le sujet.

2. Spécification du programme

On spécifie le programme comme une action qui traite une liste de valeurs en entrée et fournit, en sortie, des résultats sous forme d'une liste de valeurs. Il s'agit de décrire cette spécification en triplet :

/*Prédicat d'entrée*/ action(L_Entrée, L_Sortie) **/*Prédicat de sortie*/**

- **/*Prédicat d'entrée*/**

décrit sous forme relationnelle les hypothèses à satisfaire par l'utilisateur de l'action.

N : entier naturel

- **/*Prédicat de sortie*/**

décrit sous forme relationnelle les propriétés à satisfaire par l'exécution de l'action.

r=N !

Ces prédicats commentent le programme. Ainsi toute action utilisée pour développer le programme peut être spécifié par un triplet. Les prédicats sont exprimés en langage mathématique, afin de décrire les propriétés avec précision, et l'action en langage de programmation (Ada dans notre cas). Le langage mathématique utilisé doit nous permettre d'effectuer des raisonnements, en logique classique, en arithmétique et en théorie des ensembles. Ceci va nous permettre à vérifier le programme développé en s'appuyant sur des formalismes qui permettent un raisonnement rigoureux.

A partir d'une spécification en triplet et en utilisant des règles nous permettant de faire la liaison entre l'action et ses prédicats (traduisant l'action, écrit en Ada, en langage mathématique) nous allons pouvoir vérifier le programme comme une vérification mathématique.

La vérification de **/*Prédicat d'entrée*/** action(L_Entrée, L_Sortie) **/*Prédicat de sortie*/**
n'est que la vérification de l'implication suivante :

/*Prédicat d'entrée*/ → pfp(''action(L_Entrée, L_Sortie) '', **/*Prédicat de sortie*/**)

pfp signifie le plus faible prédicat dans lequel l'action s'exécute et satisfait le prédicat de sortie.

Pour calculer le pfp différentes règles seront introduites pour permettre une traduction en langage mathématique. Ensuite la vérification pourra s'appuyer sur les règles de raisonnement logiques, arithmétiques et ensemblistes.

Exemples

E1. Ecrire la spécification d'un programme qui pour une valeur N calcule N!

Spécification

/*N>=0*/ FACT(N, r) **/*r = N!*/**

E2. Ecrire la spécification d'un programme qui calcule la partie entière de la racine carré d'un entier N.

comprendre
N doit être de type entier >=0
N est une entrée
le résultat est fournit dans une variable a qui peut être >=0

Spécification
/*N >=0*/
PER(N,a*/
/* a^2<=N<=$(a+1)^2$ */

Par convention une constante ou une valeur en entrée est représentée en Majuscule tandis qu'une variable du programme est représentée en Minuscule. Dans les commentaires on suppose que les types des valeurs et des variables sont vérifiés implicitement.

3. Modèle de solution

Pour **E1** :

1. un modèle de solution abstrait peut être la solution fonctionnelle en récursif :

N ! → If (N= = 0) return 1 ; else return N*(N-1) ! ;

Deux difficultés :

- Exprimer un modèle de solution sous une forme fonctionnelle n'est pas toujours évident.
- A partir de ce modèle il faut trouver un moyen de le traduire en impératif, ce qui peut être assez compliqué.

On est censé, donc aussi, de proposer des modèles de solution sous d'autres formes (souvent d'une manière informelle et à travers des exemples) :

2. Pour calculer N! :

C'est 1 quand N=0 sinon il faut calculer N*N-1*N-2*…*4*3*2*1 càd il faut calculer, pour obtenir N!, (N-1)! et pour calculer (N-1)! Il faut (N-2)! Et ainsi de suite jusqu'au 0!.

0! = 1
1! = 1*0!
2! = 2*1!
: :
: :
(n-1)! = (n-2)!*(n-1)
n! = (n-1)!*n

4. Développer le programme avec preuve (E1)

Le processus de développement est un processus de raffinement. En partant de la spécification nous proposons une décomposition de l'action en actions plus élémentaires. Chaque action composante est spécifiée en triplet et reliée aux autres actions par des structures de contrôles imposées par le langage. On répète cette décomposition jusqu'à l'obtention de du programme composé seulement par des actions élémentaires définies dans le langage de programmation. Cette décomposition nous permettra de développer et de vérifier en même temps. Chaque étape dans le processus de raffinement respecte ce qui a été fait dans l'étape précédente. Les variables sont introduites au fur et à mesure dans le processus de développement.

1er niveau

/*N>=0*/ **-- à ce niveau on a introduit N et r**
FACT(N, r) -- N est considéré comme constante par rapport à l'exécution de
/*r = N!*/ -- FACT et r est une variable qui contiendra le résultat

2ème niveau

En s'inspirant du modèle de solution présenté auparavant, nous introduisons deux variables i et r.

i	r
0!	= 1
1!	= 1*0!
2!	= 2*1!
: :	
: :	
(n-1)!	= (n-2)!*(n-1)
n!	= (n-1)!*n

Quand i est à 0 le i ! vaut 1 donc r=1
Ensuite on répète jusqu'à i soit égal à N la mise à jour de i et de r :

A la fin de cette répétition r est égale à N !

/*N>=0*/

i =0 ; r =1 ; -- on peut vérifier avec les pfp

/*i>=0 et r=i !*/

on répète jusqu'à i soit égal à N

- mettre à jour (i,r)

/*r = N!*/

3ème niveau

```
        /*N>=0*/

 i =0 ; r =1 ;
        /*i>=0 et r=i !*/
while (i<N)
{       /*0<=i et r=i !*/
i =i+1 ;
r =r*i ;
        /*i>=0 et r=i !*/              /* à vérifier avec pfp*/
;}
/*r = N!*/                    /* à vérifier avec pfp*/
```

Cette méthode de raffinement sera utilisée tout au long du processus de développement d'un algorithme. Le fait de commenter l'algorithme par des assertions permettra de faire la vérification.

Mise en forme en programme C :

```
#include <stdio.h>
main()
{ int i, r, N ;
        printf(''tapez un nombre  entier positif\n'');
        scanf(''%d'', &N) ;

    /*N>=0*/
     i =0 ; r =1 ;
    /*i>=0 et r=i !*/
    while (i<N)
            {       /*0<=i et r=i !*/
                    i =i+1 ;
                    r =r*i ;
                    /*i>=0 et r=i !*/              /* à vérifier avec pfp*/
            ;}
    /*r = N!*/                    /* à vérifier avec pfp*/
```

```
printf(''factoriel de %d = %d \n'', N, r);
}
```

NB : Une fois le programme est développer, on passe à la vérification de la terminaison s'il contient des traitements répétitifs.