Programmation procédurale

Yann **Régis-Gianas**
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

Plan

Introduction

Programmation par raffinements successifs

Raisonnement sur les programmes procéduraux

Comparaison avec la programmation orientée objet

Plan

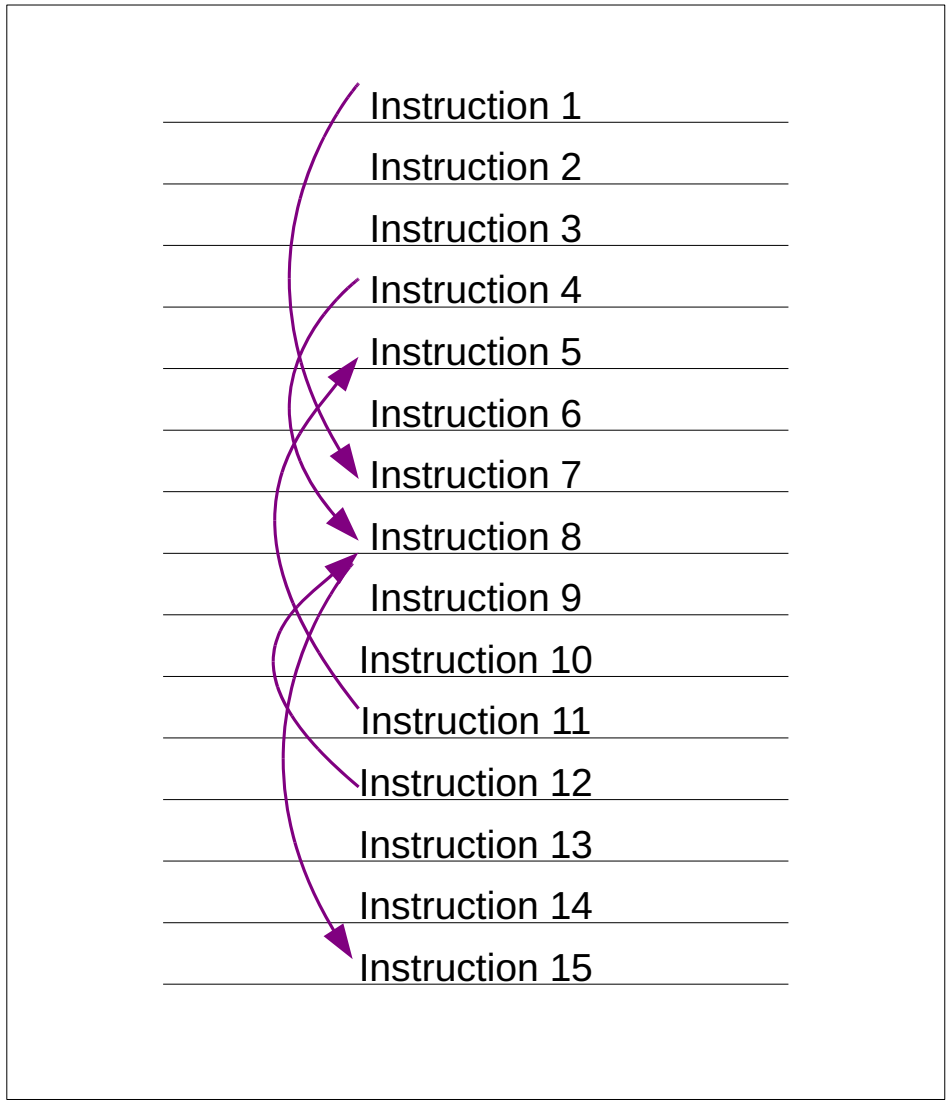
Introduction

Programmation par raffinements successifs

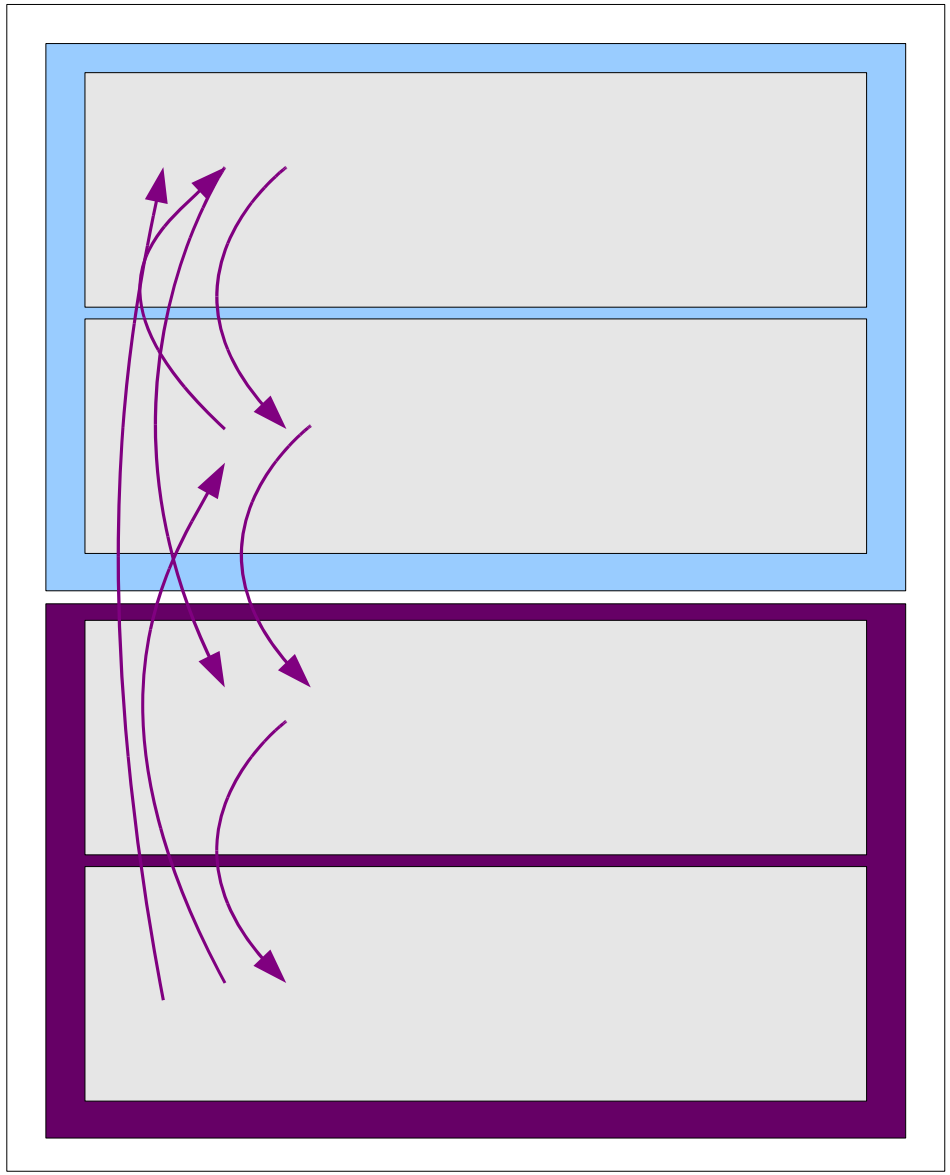
Raisonnement sur les programmes procéduraux

Comparaison avec la programmation orientée objet

De la programmation non structurée . . .



. . . à la programmation structurée.

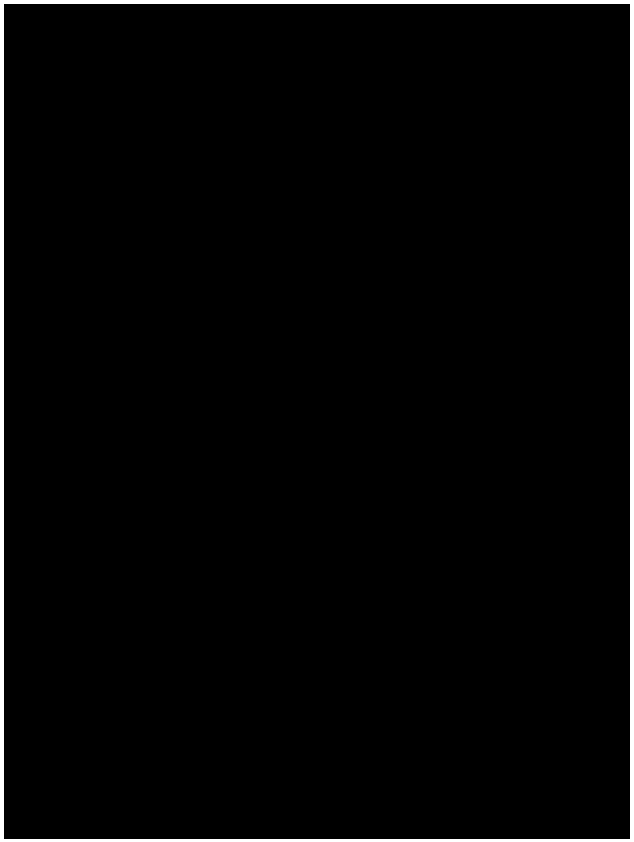


Question

Quels sont **vos** arguments en faveur et en défaveur
de la présence `goto` dans les langages de programmation ?

Donnez des exemples de programmes pour illustrer votre discours.

Un bataille historique



Dijkstra

Knuth

Langages procéduraux du premier ordre

Dans les années 60, la crise du logiciel a conduit à une réflexion autour de la méthodologie de la construction des programmes informatiques, aboutissant à l'émergence du génie logiciel. Du point de vue des langages de programmation, des mécanismes pour **structurer** la programmation et les programmes sont apparus :

- Structures de contrôle de haut-niveau : **while**, **if-then-else**,...
- Procédures et fonctions (de seconde classe).
- Suppression du **goto**
  (Un point unique de sortie et d'entrée pour les boucles.)
- Minimisation des variables globales au profit de variables locales.

Un article de recherche de l'époque explique pourquoi ces **restrictions** de l'expressivité des langages de programmation sont bénéfiques :

Go To Statement Considered Harmful – Edsger Dijkstra
http://www.ifi.uzh.ch/req/courses/kvse/uebungen/Dijkstra_Goto.pdf

Les arguments en faveur de la suppression du **goto**

Dès qu'une étiquette ℓ est présente dans un programme, elle devient la cible d'un nombre arbitraire de **goto**s. Il ne suffit donc pas de regarder le fragment de code qui se situe localement autour de cette étiquette pour savoir si ce fragment est valide : il faut *imaginer* toutes les exécutions qui peuvent mener à cette étiquette.

Sans **goto**, on peut répondre facilement à la question consistant à savoir *comment caractérisé le chemin d'exécution* qui a mené à un certain point du programme. En effet :

- Si cette instruction est précédée par une autre instruction dans le code source alors le contrôle provient de celle-ci.
- Si cette instruction se situe dans une branche d'une conditionnelle alors il su□t d'observer quelle est cette condition qui a permis de suivre cette branche.
- Si cette instruction se trouve en début d'une procédure alors il su□t de préciser quelle imbrication d'appels de procédure a pu conduire à l'appel de la procédure considérée.
- Si cette instruction se situe dans une boucle, alors il su□t de caractériser pour quelle itération de la boucle, l'instruction peut-être exécutée.

On obtient une sorte de *système de coordonnées* défini par la valeur des variables et l'enchaînement des appels de procédures.

Les arguments contre la suppression du **goto**

Dans un article de recherche de 1974 intitulé *Structured Programming with gotos statements*, D. Knuth tempère la thèse d'une suppression totale du **goto**.

Reconnaissant les clairs avantages de la programmation structurée pour mener à bien un raisonnement structuré sur les programmes à différents niveaux d'abstraction. D. Knuth note que l'on peut parfois donner du sens à un **goto**. Par exemple, un invariant peut être associé à une étiquette :

```
#define unix_call(X) X; if (errno != 0) goto unix_error;
void do_some_unix_stu() {
    unix_call (fd = fopen ("foobar.txt", "a"));
    return
unix_error:  // If the control flows here, an Unix error occured.
    // Report the error.
}
```

Par ailleurs, il explicite dans ce papier des algorithmes difficilement implémentables efficacement sans **goto**, tout en argumentant qu'en introduisant des structures de contrôle appropriées, ces problèmes disparaissent souvent.

Exemple tiré du papier

Soit une table A contenant des valeurs $A[0] \ldots A[m]$ et une table B servant à compter combien de fois l'élément x de A a été recherché. Lorsque l'on recherche un élément dans A et qu'il ne s'y trouve pas, on souhaite le rajouter.

```
for (i = 0; i ≤ m; ++i)
  if (A[i] == x) goto found;
not_found: i = m + 1; m = i; A[i] = x; B[i] = 0;
found: B[i] = B[i] + 1;
```

```
i = 0;
while (i ≤ m ∧ A[i] != x) i++;
if (i > m) { m = i; A[i] = x; B[i] = 0; }
B[i] = B[i] + 1
```

Pour vous, lequel de ces programmes est le plus lisible ? Le plus efficace ?

Parenthèse

Quel mécanisme des langages de programmation modernes permet d'écrire ce programme dans un style similaire à la première version ?

La programmation structurée

En plus de l'utilisation de langages de programmation procéduraux, cette programmation plus structurée est également caractérisée par une **méthologie de conception des programmes** par **raffinements successifs descendants**.

Dans le reste de ce cours, nous allons étudier cette méthode et la comparer aux méthodes de modélisation que vous connaissez déjà.

Pourquoi s'intéresser à la programmation procédurale ?

La programmation procédurale est encore très utilisée aujourd'hui. Le noyau LINUX est un exemple de programme important écrit (quasi exclusivement) dans ce style.

C'est aussi peut-être la programmation la mieux "comprise", tant par sa méthodologie que par les mécanismes calculatoires qu'elle met en jeu.

Plan

Introduction

Programmation par raffinements successifs

Raisonnement sur les programmes procéduraux

Comparaison avec la programmation orientée objet

La procédure

Procédure

Un **procédure**, routine ou sous-routine, est un fragment de code paramétré par des arguments formels.

Appel d'une procédure

Un **appel de procédure** est une instanciation particulière de ce fragment pour lequel des arguments effectifs à substituer aux arguments formels ont été fournis. Si une procédure possède des variables locales alors un jeu de variables différents est créé à chaque appel. L'appel de procédure est l'unique façon d'exécuter le fragment de code d'une procédure.

Calcul d'une procédure

Une procédure ne produit pas une valeur de résultat. Elle **modifie l'état** du programme. (Par exemple, *via* les variables globales du programme ou des primitives d'entrées/sorties.)

La programmation impérative

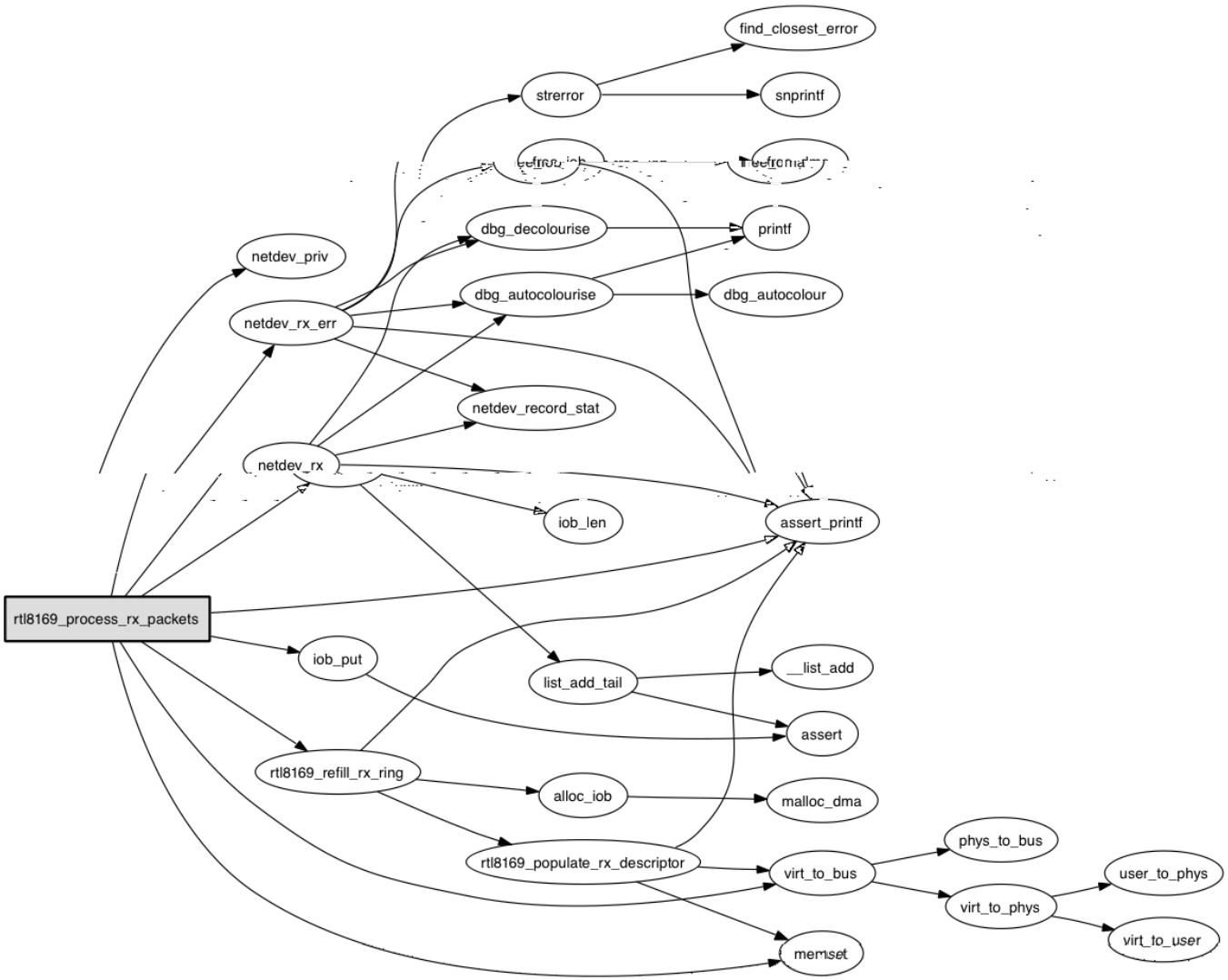
Programmation impérative

La programmation impérative est le paradigme de programmation qui consiste à voir un programme comme un **transformateur d'état** décrit comme une **imbrication séquentielle d'appels de procédures**.

L'exécution d'un programme impératif peut ainsi être décrite comme un **arbre d'appels de procédure** que l'on interprète suivant un parcours en profondeur transmettant l'état du programme d'un nœud à son fils, son frère ou son parent.

Graphe d'appels de procédures

Tiré de la documentation d'un drivers pour une carte réseau.

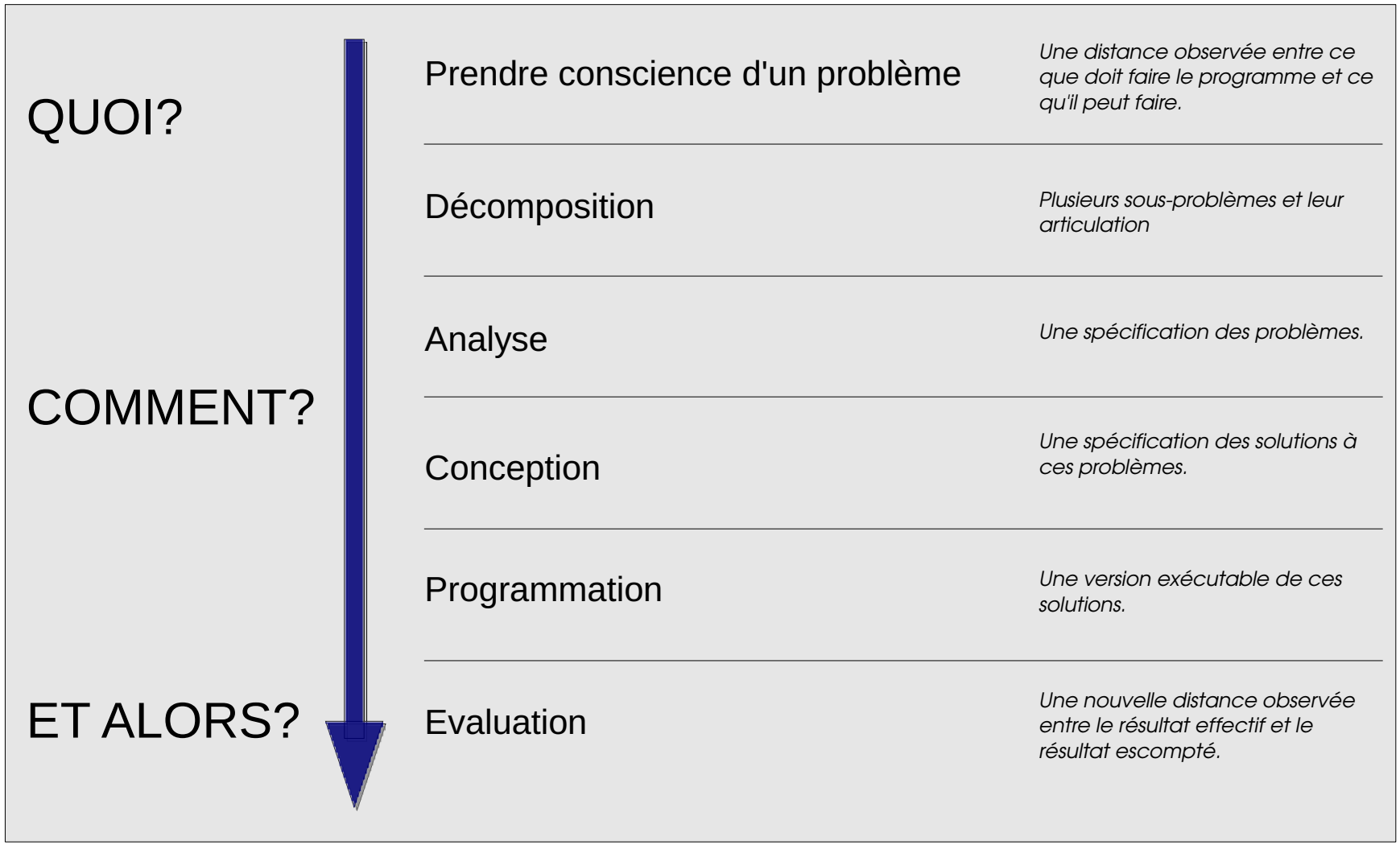


De quoi est formé l'état d'un programme ?

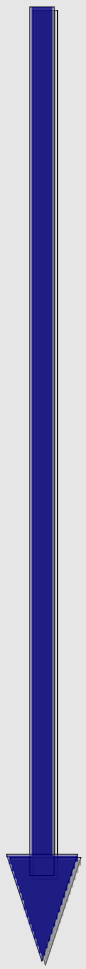
L'état d'un programme est essentiellement formé de la mémoire. Cette dernière peut êre plus ou moins structurée. En C, elle apparaît comme un ensemble de segments de données sur lesquels il est possible d'agir *via* des pointeurs dont la valeur est un emplacement arbitraire à l'intérieur du segment. Dans d'autres langages, comme le Pascal, de telles manipulations de bas-niveau ne sont pas autorisées.

Conceptuellement, la mémoire est le lieu où sont représentées des **structures de données** accessibles *via* des emplacements nommés. Ces emplacements nommés sont appelés variables mais ne sont pas exactement les variables que l'on trouve en mathématique par exemple. En effet, en mathématique, quand on définit la valeur d'une variable x à être un certain entier 42, cette définition est indépendante du temps. Or, dans un langage impérative, la valeur d'une variable x représente **le contenu** d'un emplacement mémoire et peut donc évoluer en fonction des actions effectuées par le programme.

Le développement logiciel



Le développement logiciel par programmation procédurale



Des données à traiter, à produire, à modifier...

Les traitements à effectuer sont hiérarchisés logiquement et temporellement.

Les propriétés des données dans l'état initial et dans l'état final puis dans les états intermédiaires.

Les structures de données du système, et leurs propriétés sont explicitées. Les traitements sont des algorithmes.

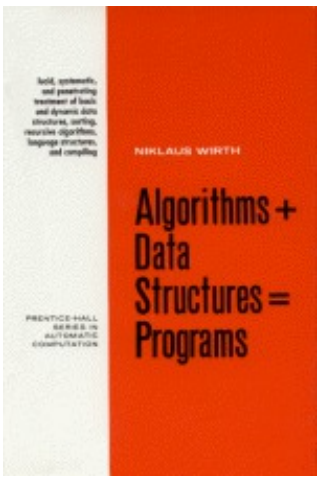
Un ensemble de routines et de sous-routines qui transforment l'état du système.

Les données d'entrées et produites correspondent aux données réelles et attendues.

La méthode de conception par raffinements

En programmation procédurale et impérative, on distingue clairement les algorithmes et les données. Ce point de vue est particulièrement frappant à la vue du titre d'un livre de référence sur ce style de programmation :

Data structures + *Algorithms* = *Programs*
Niklaus Wirth.



Data structures + Algorithms = Programs

Dans ce livre, l'auteur explique une méthode de conception **descendante** des programmes qui consiste à représenter le traitement à effectuer par un programme comme une procédure dont le code source contient des opérations **abstraites**, c'est-à-dire qui n'ont pas d'implémentation concrète, mais sur lequel on peut raisonner à un *certain niveau d'abstraction*, c'est-à-dire en mettant de côté certains détails. Une fois que l'on est convaincu que cette procédure réalise bien à sa spécification, on **raffine** les opérations abstraites en les remplaçant par des opérations plus concrètes faisant intervenir plus de détails d'implémentation et on recommence le raisonnement. Au bout de plusieurs raffinements successifs, on obtient un programme que l'on peut concrètement programmer sur machine.

Cette méthode est à la base de la méthode B, une méthode formelle de certification de logiciel.

Conception descendante d'un éditeur de ligne

Exemple tiré de

On souhaite développer un programme interactif d'édition permettant à l'aide d'un certain nombre de commandes de produire un nouveau texte y à partir d'un texte x.

Les commandes accessibles sont :

- I, m : Insère un texte après la m-ième ligne.
- D, m, n : Supprime les lignes de m à n.
- R, m, n : Remplace les lignes de m à n.
- E : Termine la session d'édition.

L'interaction se déroule *via* l'entrée standard du programme. m et n sont des nombres écrits en base décimale, et le texte à insérer pour les commandes I et R suivent immédiatement les instructions.

Les commandes d'édition sont appliquées séquentiellement sur le texte d'entrée en lisant ce dernier ligne par ligne.

Avez-vous déjà utilisé l'utilitaire UNIX `ed` ?

État du programme

Les données du programme sont :

- Le texte d'entrée x.
- Le numéro de la ligne courante ln.
- La ligne courante l.
- Le texte de sortie y.

À la fin du calcul, le texte de sortie doit contenir le texte d'entrée auquel on a appliqué les instructions fournies par l'utilisateur sur l'entrée standard à l'aide des commandes décrites plus haut.

Au début du calcul, le numéro de la ligne courante est 0. Comme on traite le texte x ligne par ligne, ce numéro ne fera que croître.

Enfin, la ligne courante devra correspondre à ligne numéro n du texte x tant que celle-ci n'a pas été traitée par l'instruction courante de l'utilisateur.

Éditeur à un haut niveau d'abstraction

Compte-tenu de la description précédente de l'état du programme et de la spécification, une première étape de l'analyse consiste à décomposer les traitements effectués par l'éditeur sous la forme d'une **boucle intéractive** :

```
program editor (x, y, input, output):
var n : integer
var ln : line
var x, y : text
begin
   read instruction;
   repeat
       interpret instruction;
       read instruction
   until instruction = 'E'
end
```

Il reste maintenant à raffiner les opérations abstraites `read instruction` et `interpret instruction`.

Lecture des instructions

La lecture des instructions nécessite une analyse syntaxique très basique qui va traiter les entrées de l'utilisateur et les communiquer à la suite du programme. Pour établir ce canal de communication, on introduit des variables :

```
var code, ch : char
var m, n : integer
```

Read instruction :

```
    read (code, ch) ;
    if ch = ',' then read (m, ch) else m ← ln ;
    if ch = ',' then read (n) else n ← m ;
```

Quelle spécification donneriez-vous à cette routine ?

Interprétation des instructions

En implémentant quasi-textuellement la spécification :

```
Interpret instruction:
    copy the lines of x from the current line to line number m into y;
    if code = 'I' then begin
       put current line from x into y;
       insert command text in y;
    end
    else if code = 'D' then skip current line to line number n
    else if code = 'R' then begin
       insert command text in y;
       skip current line;
    end
    else if code = 'E' then copy the rest of x into y
    else error
```

Encore une fois, on raffine les opérations abstraites.

Opérations de transfert par lignes

```
copy the lines of x from the current line to line number m into y:
   while ln < m do begin
      put line from x to y;
      get next line from x;
   end

skip:
   while ln < n do get next line from x

insert command text in y:
   read line from command text;
   while not at the end of command text do begin
      put line from x to y;
      read line from command text;
   end
   get next line from command

copy the rest of x into y:
   while not at the end of text x do begin
      put line from x to y;
      get next line from x;
   end;
   put line from x to y
```

Opérations de transfert par caractères

Le niveau final de détails détermine le traitement à effectuer caractère par caractère. Pour cela, il faut raffiner la structure de données représentant une ligne. On peut utiliser un tableau de 80 caractères, en supposant que c'est la taille maximale des lignes :

```
var l: array[1..80] of char
var i: integer
var L: integer
```

Une variable `i` va représenter un indice dans la ligne courante. La variable `L` va représenter la longueur de la ligne courante.

Opérations de transfert par caractères

```
get next line from x:
   i ← 0;
   ln ← ln + 1;
   while not at the end of the next line of x do begin
      i ← i + 1;
      l[i] ← read next character in the next line of x;
   end;
   L ← i;
   go to the next line in x

put line from x to y:
   i ← 0;
   while i < L do begin
      i ← i + 1;
      write l[i] in y
   end;
   go to the next line in y
```

`read line from command` est similaire à `put line from x to y`.

Exercices

1. Implémentez ce dernier raffinement dans le langage de votre choix.
2. À partir de la description du raffinement du tri fusion décrit page 87 du livre de Wirth, implémentez une version commentée de cet algorithme en vous assurant que vous comprenez chaque étape du raffinement.

Plan

Introduction

Programmation par raffinements successifs

Raisonnement sur les programmes procéduraux

Comparaison avec la programmation orientée objet

La procédure comme unité de raisonnement

Les procédures fournissent au programmeur un outil pour **hiérarchiser son raisonnement**. En effet, un appel de procédure peut être vu comme une boîte noire remplaçable par la spécification de la procédure en question.

On peut alors **mener des raisonnements à un niveau d'abstraction donné en ignorant les détails de l'implémentation** des procédures.

Mais, comment raisonner **rigoureusement** sur des programmes ?

La logique de Hoare

La logique de Hoare est une méthode pour prouver qu'un programme respecte sa spécification. Elle est utilisée par les méthodes formelles de certification des logiciels. Elle peut aussi être utilisée pour se donner un cadre pour raisonner sur les programmes impératifs de la vie de tous les jours. . . même si vous ne produisez pas une preuve formelle (niveau de complétude qui demande beaucoup de temps et d'énergie).

Au centre de la méthode de Hoare est un objet formel appelé **triplet de Hoare** :

$$\{P\}\ S\ \{Q\}$$

qui se lit ainsi :

> *"Si la précondition P est vérifiée sur l'état du programme avant l'exécution de S alors la postcondition Q sera vérifiée sur le programme après l'exécution de S."*

Un petit langage à boucle

Nous allons voir les règles de bonne formation des triplets de Hoare appliquées à un langage jouet dont la syntaxe est :

$$
\begin{array}{rcl}
S & ::= & \textbf{skip} \\
 & | & x := e \\
 & | & \textbf{if } b \textbf{ then } S \textbf{ else } S \\
 & | & S ; S \\
 & | & \textbf{while } b \textbf{ do } S \textbf{ done}
\end{array}
$$

où e est une expression arithmétique et b une expression booléenne.

Règle du **skip**

$$\frac{}{\{P\}\textbf{skip}\{Q\}}$$

Règle de l'affectation

$$\frac{}{\{P[x \mapsto e]\}x := e\{P\}}$$

Par exemple :

$$\frac{}{\{x + 21 = 42\}y := x + 21\{y = 42\}}$$

Règle du séquencement

$$\frac{\{P\}S_1\{R\} \qquad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

Règle du **if**

$$\frac{\{P \wedge B\} S_1 \{Q\} \qquad \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \textbf{if} B \textbf{ then } S_1 \textbf{ else } S_2 \{Q\}}$$

Règle du **while**

$$\frac{\{P \wedge B\} S \{P\}}{\{P\}\textbf{while } B \textbf{ do } S \textbf{ done } \{\neg B \wedge P\}}$$

La propriété P est l'**invariant** de la boucle. C'est une propriété qui est valide avant d'entrée dans la boucle, à chaque itération et à la fin de la boucle.

Exercice : Déterminez l'invariant d'une boucle servant à chercher l'entier le plus petit d'un tableau d'entiers non vide.

Règle de conséquence

$$\frac{P \Rightarrow P' \qquad \{P\}S\{Q\} \qquad Q' \Rightarrow Q}{\{P'\}S\{Q'\}}$$

Cette règle ne vous rappelle-t-elle pas une autre règle vue dans le cours POCA ?

Exercice : Retour sur le tri fusion

Annotez chaque étape de votre raffinement du tri fusion par les triplets de Hoare qui vous semblent valides.

Plan

Introduction

Programmation par raffinements successifs

Raisonnement sur les programmes procéduraux

Comparaison avec la programmation orientée objet

Comparez cette approche avec la POO

Comparez cette approche avec la POO

La conception descendante de la programmation procédurale

+ Raffinement continu des données et des algorithmes de leur formes abstraites de la spécification issue de l'analyse à la forme concrète de l'implémentation.
+ Facilite le raisonnement à l'aide de règles simples de la logique de Hoare. Le flot d'exécution est explicite et fixé : en particulier, ceci rend possible les raisonnements sur la complexité des programmes.
+ La compilation est plus simple que pour des langages d'ordre supérieur.
- La dépendance entre les structures de données et les algorithmes est très forte ce qui rend difficile les extensions *a posteriori*.
- La spécification des données peut être éloignée des données de la réalité car on suppose une connaissance *a priori* sur le domaine de définition et la structure des données du problème.

Une hypothèse importante pour la modularité du raisonnement

Les **interdépendances entre les procédures sont très fortes car elles peuvent communiquer à travers la totalité des variables globales**. Or, pour faciliter un raisonnement modulaire, il faut limiter au maximum les interactions *via* des effets de bord. Une technique consiste à lister et à garantir que seuls un nombre fixé de variables sont modifiées par une procédure. Ainsi, si une autre procédure utilise un jeu indépendant de variables, elles peuvent être utilisées côte à côte sans vérification supplémentaire.

Or, dès que les langages utilisent des données allouées dynamiquement, cette propriété est difficile à garantir.

Dans le prochain cours, nous aborderons la programmation fonctionnelle sous le jour des effets de bord et nous verrons **comment une restriction aussi forte que l'interdiction des effets de bord augmente significativement la déclarativité des langages de programmation**.