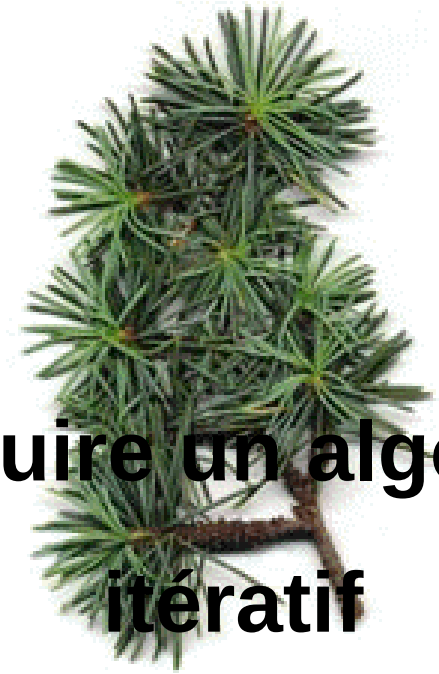
https://laurentbloch.net/BlogLB/Construire-un-algorithme-iteratif



Construire un algorithme itératif

- Zinformatiques - Cours de bioinformatique au CNAM -

Date de mise en ligne : lundi 4 octobre 2004

Quelle est la méthode pour construire un algorithme itératif ? Il n'y a pas de méthode générale, mais des idées générales qui aident à trouver la méthode particulière pour construire un algorithme particulier.

Certains d'entre vous se demandent quelle est la méthode pour construire un algorithme itératif. La réponse est qu'il n'y a pas de méthode générale, sinon l'informatique serait toute entière un problème résolu et ce cours n'aurait pas lieu d'être. Ceci dit il y a des idées générales qui d'expérience aident à trouver la méthode particulière pour construire tel ou tel algorithme particulier. Le présent article donne une première approche de ces idées. Vous pourrez aussi consulter un autre article qui donne une approche plus empirique.

Il est conseillé de déterminer un *invariant* du traitement à effectuer, c'est-à-dire une relation entre les variables qui doit rester vraie tout au long de l'exécution du programme.

Considérons l'exemple du calcul de **n !** par une méthode itérative. Nous avions écrit :

```
(define (fact n)
  (do ((i 1 (+ i 1))
       (f 1 (* i f)))
     ((> i n) f)))
```

Les variables `i` et `f` représentent à chaque *pas* du calcul l'état complet du calcul en cours. Plus précisément, à chaque pas du calcul nous pouvons vérifier la relation :

f = (i-1) !

par exemple en instrumentant notre programme pour lui faire produire des résultats intermédiaires :

```
(define (fact n)
 (do ((i 1 (+ i 1))
      (f 1 (* i f)))
     ((> i n) f)
     (print i " " f)))
```

dont l'exécution donne bien :

```
 1:=> (fact 6)
1 1
2 1
3 2
4 6
5 24
6 120
720
1:=>
```

Cette relation est *l'invariant* du calcul. Pour résumer, il faut donc déterminer les variables caractéristiques de l'algorithme et établir entre elles une relation invariante. Ceci peut souvent se faire en effectuant le calcul à la main et en notant sur une feuille de papier les valeurs significatives à chaque étape.

La boucle `do` du programme ci-dessus a un corps vide (sauf pour la version « instrumentée » qui a un corps limité au `print`) et un résultat limité... au résultat, le vrai travail est fait dans les expressions des liaisons (voir dans le *R5RS* la syntaxe de **do**).

Guy L. Steele Jr. et Richard P. Gabriel, grands maîtres de Scheme, écrivent : « La beauté de cette construction est qu'elle permet l'initialisation et l'incrémentation de plusieurs variables sans besoin de mots-clés en pseudo-anglais. Le revers de la médaille, c'est qu'elle utilise comme délimiteurs de nombreux niveaux de parenthèses, et qu'il faut les placer correctement sous peine de comportements étranges ; seul un Lispien endurci peut apprécier une telle syntaxe.

Au-delà du débat sur la syntaxe, il convient de se demander si une boucle qui n'incrémenterait qu'une variable ne serait pas singulièrement inutile, quel que soit le langage. Presque toujours, il faut une variable pour générer des valeurs successives tandis qu'une autre accumule les résultats. Si la syntaxe de la boucle n'incrémente que la première, alors il faut
incrémenter « à la main » celle qui accumule avec des affectations (comme en Fortran) ou quelque autre effet de bord. »

Vous pouvez aussi consulter un autre article qui donne une approche plus empirique.