De l'usage de `break`, `continue` et `return`

Jean-Christophe Filliâtre

28 mai 2020

Résumé

On rencontre régulièrement des enseignants qui se refusent catégoriquement à utiliser les constructions `break` et `continue` ou encore à sortir d'une boucle avec une construction `return`. C'est bien dommage. Je donne ici des arguments allant au contraire en faveur de l'utilisation de ces trois constructions.

Tout ce que je vais dire ci-dessous s'applique autant à la sortie anticipée de boucle avec `break`, qu'à l'utilisation de `continue` et qu'à la sortie anticipée de fonction avec `return` (y compris depuis l'intérieur d'une boucle). Voici mes arguments en faveur de l'utilisation de ces trois constructions.

1. Quand on utilise un langage, et surtout quand on l'enseigne, il faut le faire **idiomatiquement**. C'est vraiment desservir les étudiants que de ne pas leur montrer la façon idiomatique de faire, car c'est celle qu'ils trouveront dans la bibliothèque standard et dans le code écrit par d'autres, mais aussi parce que c'est probablement celle qui est compilée le plus efficacement (sinon, elle ne serait pas l'idiome).
2. Se passer de `break`/`continue`/`return` quand on écrit des boucles oblige à des **contorsions**, à base de variables booléennes le plus souvent, qui
   — obscurcissent le code, pour le programmeur comme pour le lecteur ;
   — augmentent considérablement le risque d'erreur.
   Au-delà de ces deux phénomènes, le code est généralement plus long, ce qui est regrettable.
3. Dans le même esprit, ces trois constructions permettent de **simplifier le flot de contrôle en le conservant le plus linéaire possible**. Ainsi, un `return` permet de se passer d'un `else` :

   ```
   def f(x):
     if x == 0: return 1
     ...
   ```

   (J'utilise ici Python à des fins d'illustrations mais le langage importe peu.) Nul besoin d'indenter tout ce qui vient après ce `return`, ce qui est d'autant appréciable que le reste de la fonction est gros. Notez que c'est la même chose avec une exception :

   ```
   def f(x):
     if x < 0: raise ValueError
     ...
   ```

Il en va de même pour un `continue`. Ainsi, il est agréable d'écrire

```
while len(q) > 0:
  x = q.pop()
  if x in vus: continue
  ...
```

plutôt que d'utiliser tout un nouveau bloc pour y mettre le contenu de `...`, qui peut être arbitrairement gros.

4. Enfin, les constructions `break`/`continue`/`return` ont une **sémantique simple**. (Quand on s'intéresse à la compilation, on comprend rapidement que ce sont mêmes là les constructions les plus simples à compiler : un simple saut vers un endroit statiquement connu, facilement identifié.) Les langages de programmation contiennent tous de très nombreuses subtilités et je comprends qu'un enseignant puisse délibérément écarter les aspects les plus sordides (je le fais moi-même quand j'enseigne à des débutants). Mais pourquoi écarter des constructions simples, qui rendent de plus le code plus élégant ?

À propos des langages fonctionnels. Si les constructions `break`/`continue`/`return` sont présentes dans la plupart des langages impératifs, on constate généralement leur absence dans les langages fonctionnels comme Haskell, OCaml, Standard ML, F#, etc.

Ceci peut s'expliquer techniquement par le fait que, dans les langages fonctionnels, il est courant d'utiliser une fonction récursive plutôt qu'une boucle. Dès lors, la sortie anticipée est immédiate : il suffit de ne pas faire d'appel récursif. Ainsi, en OCaml, je peux chercher l'indice du premier 0 dans un tableau avec cette fonction récursive :

```
let rec cherche a i =
  if i = Array.length n then raise Not_found;
  if a.(i) = 0 then i else cherche a (i + 1) in
cherche a 0
```

Les langages fonctionnels optimisent typiquement l'appel terminal (c'est le cas d'OCaml), ce qui fait que cette fonction `cherche` va se retrouver compilée **exactement** comme la boucle

```
i = 0
while i < len(a):
  if a[i] == 0: return i
  i += 1
raise NotFound
```

qui contient une sortie anticipée. Dit autrement, les langages fonctionnels n'ont pas besoin de telles constructions. (Cela étant, j'adorerais disposer de `break`/`continue`/`return` en OCaml !) Dans les langages impératifs, en revanche, il n'est pas usuel de recourir à des fonctions récursives à la place de boucle et surtout (peut-être pour cette raison) les appels terminaux sont rarement optimisés (jamais en Java et Python, par exemple), ce qui expose au débordement de pile.

Merci à Alexandre Casamayou de m'avoir amené à rédiger et publier cette note.