Paradigmes de la programmation

Exercices 1

1 le retour des coroutines

Les coroutines sont un concept de programmation qui permet de faire collaborer des unités de programme (fonction, procédure, ...) qui se passent tour à tour le flot de contrôle. Ce concept était utilisé avant les communications inter-processus et les threads et relève du multi-tâche collaboratif (c'est à dire non préemptif).

Les coroutines sont une généralisation de la notion de fonction ou de procédure (c'est à dire une fonction qui ne retourne pas de valeur, nous voyons ici les procédures comme un cas particulier des fonctions). En plus de pouvoir terminer et retourner sa valeur à la fonction qui l'a appelé, une coroutine f peut suspendre son exécution et passer le flot de contrôle à une autre coroutine g en lui passant éventuellement une valeur. Cette autre coroutine n'est pas nécessairement celle qui a appelé f.

En Python, l'instruction `yield` sert à créer des générateurs, qui sont une forme limitée de coroutine où le flot de contrôle retourne toujours à la fonction appelante. Pour le dire vite, une coroutine à la façon de Python est une fonction qui retourne plusieurs fois une valeur, sans pouvoir choisir la fonction à qui elle passe cette valeur.

L'appel se fait à l'aide la méthode `next()`. L'instruction `yield` prend une paramètre optionnel qui sera la valeur passée au retour de l'appel. la méthode `send` des générateurs permet de passer explicitement une valeur à la coroutine.

Plus récemment, les versions 3.3, 3.4 et 3.5 de Python ont achevé d'intégrer le concept dans le langage en utilisant le concept de futures (async/await), dont nous reparlerons en Scala.

1.1 Générateur

Dans le code suivant on crée un générateur et on l'appelle jusqu'à la fin. Essayez le et voyez à quoi sert chaque instruction. Comment peut-on faire pour passer une valeur au générateur ?

```
def copain(nom, n):
    for i in range(n):
         yield ('%s : %d' %(nom, i))
    yield ('Bob : bye')

def main():
    generateur = copain('Bob', 5)
    print generateur.next()
    generateur.send('salut bob') # cette valeur sera retournee par yield
    for i in generateur:
        print i

main()
```

1.2 Inverser l'inversion du contrôle

Lorsqu'on programme avec des *callbacks* dans une boucle événementielle on se retrouve avec une inversion du contrôle au sens où le code que l'on écrit ne maîtrise plus le moment où il sera appelé. En général, cela se produit en utilisant un framework ou une bibliothèque qui fournit la boucle événementielle. Habituellement on écrirait un programme qui effectue des opérations en faisant appel aux fonctions de la bibliothèque, mais lorsqu'on utilise une boucle événementielle on écrit du code qui sera appelé par la bibliothèque. C'est en sens qu'il y a inversion du contrôle.

Dans les générateurs, on inverse également le contrôle de la même façon : au lieu de maîtriser la boucle jusqu'au bout la fonction `copain` ci-dessus en délègue le contrôle à une autre fonction.

Que se passe t'il lorsqu'on inverse l'inversion en créant une callback à la façon d'un générateur retrouve t'on un style de programmation impératif ? Retrouve t'on un cadre de programmation standard ?

Nous allons tester en écrivant un programme Python dans lequel une fonction est appelée en boucle une fois par seconde en recevant un argument qui représentera un événement. Pour réaliser la fonction appelée commencer par écrire son code comme si elle avait le contrôle et qu'elle recevait les événements comme retour d'appels à yield, puis la transformer en générateur pour créer la callback.

2 Récursion, récursion terminale et continuation (CPS)

2.1 Ocaml.

Écrire la fonction factorielle en ocaml, puis en écrire une version récursive terminale. Écrire la fonction fibonacci.

2.2 Scala.

Vous ferez la même chose en Scala.

2.3 CPS.

Comment rendre Fibonacci récursive terminale ? Une solution générale qui va consister en troquer de l'espace sur la pile d'appel en espace alloué sur le tas est d'utiliser les continuations, en faisant comme si chaque fonction recevait une fonction qui représente la suite du calcul à effectuer...