La récursivité

Principe

Utilisation

Exemples

Le principe de récursivité

Tout objet est dit récursif s'il se définit à partir de lui-même

Ainsi, une fonction est dite récursive si elle comporte, dans son corps, au moins un appel à elle-même

De même, une structure est récursive si un de ses attributs en est une autre instance

Correspondance mathématique

Principe de récurrence

Exemple : définition des entiers

- 0 est un entier
- Si *n* est un entier, alors *n*+1 est un entier

Exemples de fonctions récursives

Calcul de la somme des entiers de 1 à *n*

- On calcule la somme jusqu'à *n*-1
- Puis on ajoute *n*

Idem avec le produit (fonction factorielle)

Un peu de vocabulaire

Pour une fonction récursive, on parlera :

- De récursivité terminale si aucune instruction n’est exécutée après l’appel de la fonction à elle-même
- De récursivité non terminale dans l’autre cas

Exemple

Terminale

```
void f(int n){

  if(n==0) System.out.println("Hello");
  else f(n-1);}
```

Non terminale

```
void f(int n) {

  if(n>0) f(n-1);
  System.out.println("Hello");
}
```

Récursivité directe

Lorsque *f* s'appelle elle-même, on parle de récursivité directe

Lorsque *f* appelle *g* qui appelle *f*, il s'agit aussi de récursivité

- On l'appelle alors indirecte

Exemples de structures récursives

Liste récursive

- Le premier élément
- Et le reste de la liste (qui est aussi une liste)

Une expression arithmétique est :

- Soit une valeur
- Soit une expression, un opérateur et une autre expression

Implémentation

Comment programmer une
fonction récursive ?
Quels sont les pièges à éviter ?

Programmer une fonction récursive

Il suffit de la faire s'appeler elle-même

```
int f(int n){
   return f(n-1);
}
```

La fonction f est récursive : elle s'appelle elle-même

Mais *f* n'a-t-elle pas un problème ?

Une obbligation : s’arrêter

La fonction *f* telle qu’elle est écrite ne s’arrête pas :

- Appel : `f(2)`
- Appel : `f(1)`
- Appel : `f(0)`
- Appel : `f(-1)`
- Appel : `f(-2)`
- Etc...

Comment y parvenir

Première étape : la condition terminale

- *Obligatoirement au début de toute fonction récursive*
- Une condition : le cas particulier
- Pour ce cas, pas d'autre appel à la fonction : la chaîne d'appels s'arrête

Exemple

```
void f(int n){
  if(n==0)
    System.out.println("Hello");
  else f(n-1);
}
```

Ici quand *n* vaut 0, on s’arrête

Problème : arrive-t-on à *n* = 0 ?

Terminaison de la fonction

Il faut que la fonction s'arrête

La condition terminale ne sert à rien si elle ne devient jamais vraie

Exemple avec la fonction précédente :

- `f(-2)` provoque une pile d'appels infinie
- Probablement d'autres tests à faire (si n<0, envoyer une exception par exemple)

Une bonne solution

```
void f(int n) {
  if(n<0) exit(-1);
  if(n==0)
        printf("Hello");
  else f(n-1);
}
```

Pourquoi ça marche ?

Si *n* est négatif : on s’arrête sur une exception

Si *n* est nul : c’est le cas d’arrêt (« Hello »)

Si *n* est positif : on appelle *f* avec la valeur *n*-1

- Chaine d’appels avec des valeurs entières strictement décroissantes de 1 en 1
- On arrive forcément à 0
- On affiche « Hello »
- On remonte la pile des appels (sans rien faire, ici la récursivité est terminale)

Théorème de Gödel

Il n'existe pas de moyen automatique pour savoir si un programme termine ou pas

Conclusion

Il faut regarder cas par cas, et à la main

Même si aucune méthode n'est générale, le principe de récurrence aide souvent

En résumé

Une fonction récursive doit comporter :

- Un cas d'arrêt dans lequel aucun autre appel n'est effectué
- Un cas général dans lequel un ou plusieurs autres appels sont effectués

La chaîne d'appel doit conduire au critère d'arrêt

- Optionnellement, des cas impossibles ou incorrects à traiter par des exceptions

Quelques exemples

Récursification facile ;
récursivité obligatoire ?

Les boucles `for`

Très bonne candidate

Toute boucle `for`peut se transformer en une fonction récursive

Principe :

- Pour faire des choses pour un indice allant de 1 à *n*
  - On les fait de 1 à *n*-1 (même traitement avec une donnée différente)
  - Puis on les fait pour l'indice *n* (cas particulier)

Traduction

```
void f(int n) {
  for(int i=0; i<=n;
  i++)
     traiter(i);
}
```

```
void f(int n){
  if(n==0)
     traiter(0);
  else {
     f(n-1);
     traiter(n);
  }
}
```

Exemple : fonction factorielle

```
int fact(int n) {
  int res = 1;
  for(int i=1; i<=n;
  i++)
     res = res*i;
  return res;
}
```

```
intfact(intn){
  if(n==0)
   return 1;
  else
   return
   fact(n-1)*n;
}
```

Appel de `fact(5)` récursif

Phase de descente récursive

Appel à **`fact(5)`**

Appel à **`fact(4)`**

Appel à **`fact(3)`**

Appel à **`fact(2)`**

Appel à **`fact(1)`**

Appel à **`fact(0)`**

Condition terminale

- Retour de la valeur 1

Suite

Phase de remontée (après l'appel à fact(n-1) on multiplie par n : la récursivité n'est pas terminale)

Retour de la valeur 1

Retour de la valeur 2

Retour de la valeur 6

Retour de la valeur 24

Retour de la valeur 120

Quelques conséquences

La plupart des traitement sur les tableaux peuvent se mettre sous forme récursive :

- Tris (sélection, insertion)
- Recherche séquentielle (attention: pas dichotomique)
- Inversion
- Problème des huit reines
- Etc...

Une constatation

L’écriture sous forme récursive est toujours plus simple que l’écriture sous forme itérative

Une question

Une même fonction est-elle plus efficace sous forme récursive ou sous forme itérative ? (Ou, sous une autre forme, y a-t-il un choix optimal généralisable ?)

La réponse est non. La réponse à la question inverse est non. Il n’y a pas de généralité

En revanche

La plupart des traitements itératifs simples sont facilement traduisibles sous forme récursive (exemple du `for`)

L'inverse est faux

Il arrive même qu'un problème ait une solution récursive triviale alors qu'il est très difficile d'en trouver une solution itérative

La fonction d'Ackermann

Ack(m, n) vaut :

- n+1 si m=0
- Ack(m-1,1) sinon et si n=0
- Ack(m-1, Ack(m,n-1)) autrement

Remarque : on finit bien car max(m,n) est strictement décroissant sur les appels (à l'exception de Ack(1,0) qui finit trivialement)