CHAPITRE N°1 : Récursivité

PLAN :

Introduction

déf : objet récursif : défini à partir de lui-même
- une condition terminale (ou base de récursivité) évite la boucle infinie
- il est parfois possible de réécrire l'objet récursif autrement, sans récursion.

en maths :
axiomatique de Peano : déf de l'ens des entiers naturels $\mathbb{N}$
1 – 0 est un nbre entier
2 – si n nbre entier, n+1 aussi
3 – il n'existe pas de nbre x tel que x+1=0
4 – des nbres distincts ont des successeurs distincts
5 – si une prop est vraie (i) pour 0 (ii) pour le successeur de chaque entier, elle est vraie pour tous les entiers.

démo par récurrence :
Mq prop vraie pour 0
Mq prop vraie pour n+1 si elle est vraie pour n
alors elle est vraie pour tous les nbres.

2 notions :
- fonctions et procédures définies de manières récursives
- structures et données définies de manières récursives

I) Fonctions et procédures récursives

1) Définition et schéma général

déf de fonction récursive ssi elle contient des appels à la fonction elle-même.

ex : factorielle.

on a besoin d'une condition terminale (ne fait pas d'appel récursif)
et dans le cas général, on fait appel à la récursivité (qui doit déboucher sur la condition terminale).

Au lancement de la fonction, on a :
- la descente récursive (l'ordi mémorise les opérations à faire jusqu'à la cond terminale)
- la condition terminale
- la remontée récursive (l'ordi fait les opérations en partant de la cond terminale).

Schéma général d'une récurrence :

```
si cond alors TTER
        sinon TGEN.
```

cond = bool
si cond est vraie : cas terminal
si cond fausse : cas récursif

2) Autres schémas de récursion

- récursivité multiple (plusieurs appel récursif)
- récursivité mutuelle (les fonctions s'appellent les unes les autres)
- récursivité imbriquée (les paramètres de la fonction récursive sont eux-mêmes récursifs, comme Ackerman)

3) Principes et dangers de la récursivité

- intérêt :
moyen simple de trouver une solution
preuve de correction plus facile que pour une solution itérative
mécanisme de base pour les langages fonctionnels (LISP) et logiques (Prolog)
- inconvénients et difficultés
inefficace dans les langages non adaptés (mais on peut toujours « dérécursiver »)
garantie de terminaison : il faut retomber toujours sur une condition terminale
ordre bien fondé = suite des valeurs des arguments : strictement monotone et atteint toujours une valeur définie explicitement

4) Fonctions et procédures récursives

Comment résoudre un problème par récursion ?
1) Déterminer le paramètre portant la récursion
2) Résoudre les cas simples (ce sont les cas terminaux en général)
3) établir la récursivité : on suppose savoir résoudre le problème pour une (ou plusieurs) valeur du para strictement plus petite que la valeur passée en arguement
4) écrire le cas général (exprimer la solution cherchée en fonction d'une solution supposée connue)
5) écrire les conditions d'arrêt (vérifier que la récursion parviendra à ces valeurs dans tous les cas)

exemple :
les tours de Hanoï
- paramètres : n disques sur le piquet de départ, les piquets
- la récurrence se fera sur l'entier n
- comment résoudre le problème pour n disques quand on sait faire pour n-1 ?
La décomposition se fait entre le plus grand disque et les (n-1) plus petits
on veut écrire la procédure HANOI(N,DEP,ARR)
elle déplace N disque du piquet DEP vers le piquet ARR
on introduit la procédure DEPLACER(DEP,ARR)

elle déplace le disque de DEP vers ARR
- condition d'arrêt : quand il reste un seul disque,
on utilise déplacer_hanoi(1,X,Y) = deplacer(X,Y).

algorithme correspondant :

```
HANOI(n,a,b) :
si n=1 alors deplacer(a,b)
       sinon hanoi(n-1,a,c)
             deplacer(a,b)
             hanoi(n-1,c,d)
fsi
```

variante avec 0 comme cas terminal :

```
HANOI(n,a,b) :
si n <> 0 alors hanoi(n-1,a,c)
               deplacer(a,b)
               hanoi(n-1,c,b)
fsi
```

II) Structures de données récursives

déf : type récursif <=> objet construit à partir d'objets du même type.

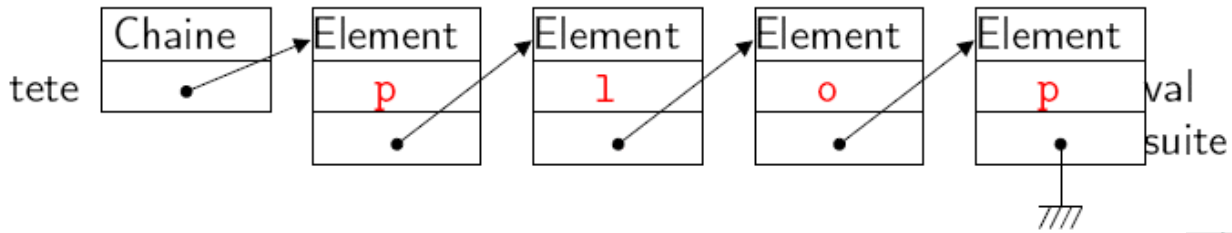
ex classiques : liste, arbre binaire

Exemple : type chaîne
chvide : chaîne vide
addT : chaîne x chaîne -> chaîne // ajout d'un cara en tête
addQ
premier : premier caractère
dernier
début : chaîne privée du dernier cara
fin
estvide : test si la chaîne est vide
Ce type est récursif car les chaînes sont construites à partir de chaînes

En java :
2 classes : Element : représentant une lettre et la chaîne dernière (ie, les chaînes non vides)
Claine : Classe représentant une chaîne (vide ou non)

pour le schéma mémoire :
on part de la chaîne, la tête pointe sur un élément : un caractère et une chaîne qui pointe sur un élément, etc... jusqu'à que la dernière chaîne pointe sur la chaîne vide.

ex pour la chaîne :



III) Quelques fonctions récursives classiques

1) Fibonacci

- on part d'un couple
- chaque couple produit un couple tous les mois
- un couple est productif après 2 mois
écriture mathématiques : $F_0 = 0$; $F_1 = 1$, et pour tt n : $F_n = F_{n-1} + F_{n-2}$.

2) 91 McCarthy

M(n) = n-10 si n > 100
M(M(n+11)) si n ≤ 100.

3) Syracuse

```
si n=0 ou n=1 alors 1
      sinon si n mod 2 = 0 alors syracuse(n/2)
            sinon syracuse(3xn+1)
        fsi
fsi.
```

IV) Preuves de fonctions récursives

2 choses à prouver :
- correction : respect des préconditions et postconditions
(preuve par récurrence (propagation des conditions entre étapes)
- terminaison : le cas terminal est toujours atteint

1) Correction

P(n) : précondition étape n ; $Q(n,r_n)$: postcondition étape n avec résultat r_n

Mq P(n) {TREC} $Q(n,r_n)$

on part de P(n) -> P(n-1) -> P(n-2) -...-> P(0) -> Q(0,r0) -..-> $Q(n-1,r_{n-1})$ -> $Q(n,r_n)$

P(0) -> $Q(0, r_{0)}$: cas terminal

cas général :
P(n) {si cond alors TTER sinon TGEN} Q(n,rn)
TGEN : r <- G(n,f(n_{int}))

TTER : r <- v(n)
avec n_{int} : valeur de l'appel récursif

f(x) : appel récursif
v(n) : fonction sans appel à f(n)
G(n,y) fonction sans appel récursif à f(n)
définie pour tout n paramètre, pour tout y

Cas simple
algo calculant r = f(n) :
si cond(n) alors r <- v(x)
sinon r <- G(n,f(n_{int}))

on doit prouver : P(n) ^ cond(n) => Q(n,r) (cas terminal)
P(n) ^ non cond(n) => P(n_{int}) (descente)

P(n) ^ non cond(n) ^Q(n_{int},n_{int}) => Q(n,r) (remontée)

Cas général :
on doit combiner ça avec les formules de preuves de terminaison vu dans les chapitres précédents.

2) Preuve de terminaison

- conditions suffisantes :
valeurs successives du paramètres x : suite strictement monotone
existence d'un extremum xg vérifiant la condition d'arrêt
- remarque : la suite de Syracuse semble se terminer sans ceci

V) Dérécursivation

1) Définition

actions réalisées lors d'un appel de fonction
1) création d'un cadre de fonction dans la pile
2) copie des valeurs des paramètres effectifs correspondants
3) exécution de la fonction
4) dépilement des paramètres formels (retour)
5) Destruction du cadre de fonction
La récursivité ne change rien à ce schéma

> *exemple : pour le pgcd, on parle de récursivité terminale (car pas d'opérations à la remontée)*
> *sinon, la récursivité est dite non-terminale (plus dur à dérécursiviser)*

définition : dérecursivation = transformer une fonction récursive en fonction itérative

2) Dérecursivation d'une fonction terminale

schéma :
f(x) :
si cond(x) alors TTER(x)
  sinon T(X) ; r <- f(x_int)

l'agorithme itératif suivant est équivalent :
f(x) :
u <- x
jusque à cond(u) faire
T(u)
u <- u_int = h(u)
fin jusqu'à
TTER(u)

3) Transformation en récursivité terminale

- soit f(n) fct récursive non-terminale
- si récursivité non-terminale, méthode précédente pas applicable
- on peut parfois définir une fct g() à récursivité terminale équivalente
  g() a plus de paramètres que f()
  on stocke les paramètres intermédiaires lors descente
  on évite ainsi la remontée

(on empile dans l'autre sens)
- f() doit avoir de bonnes propriétés (associativité, commutativité, ...)
- méthode : n opérations lors de la remontée => n paramètres supplémentaires
- il faut vérifier :
  que l'on peut trouver le résultat de cette manière
  que l'algorithme obtenu est récursif terminal

4) Passage directe à la version itérative

idée : les processeurs séquentiels exécutent toutes les fonctions récursives
=> il est toujours possible de dérécursiver
- le principe est de simuler la pile d'appels des processeurs

algo :
si cond(x) alors r <- g(x)
sinon T(x) ; r <- G(x,f(x_int))
donne :
p<-pileVide
a <- x //a : variable locale
//empilement (descente)

jusqu'à cond(a) faire
a <- h(a)
fin jusqu'à
r<- g(a) // cas terminal
// dépilements et calculs (remontée)
jusqu'à pileEstVide(p) faire
a <- sommet(p) ; dépiler(p) ; T(a)
r <- G(a,r)
fin jusqu'à

Condition d'arrêt : compter les appels

VI) Back-tracking

(pour la recherche exhaustive) (récursivité avec retour-arrière)

1) Caractérisation

- recherche de solution dans un espace donné de la sortie
  choix de solution partiel (en respectant des contraintes, en maximisant la fonction)
  appel récursif (pour le reste de la solution)
- certaines solutions construites sont des impasses
- retour arrière alors nécessaire pour un autre choix
- remarque : un appel récursif à l'intérieur d'une itération

2) Algorithme

- à chaque étape de la récursion, itération sur les différentes solutions
- chaque choix implique des impossibilités pour la suite
- pour chaque itération, une descente
- en cas de blocage, remontée (et descente dans l'itération suivante)

exemple : pseudo-code de n-reines :
si i > nbre de lignes : false
si une reine peut être placée en (i+1,0), alors si nbre de reines placées [k] = n alors vrai
sinon n-reine(i,j,k+1)
sinon n-reine(i-1,0,k)

3) Quelques principes

- examen « en profondeur » d'abord de l'arbre de décomposition
- lors d'un retour arrière, restaurer l'état avant le dernier choix
stratégie (ordre des branches) à établir
- construction progressive d'une fct booléenne
- si la fonction retourne faux, il n'existe pas de solution
- probabilité explosion combinatoire
=> nécessité d'heuristiques pour limiter le nombre d'essais

Conclusion

Outil algorithmique indispensable
- récursivité en informatique, récurrence en mathématiques
- les algorithmes récursifs sont fréquents car plus simples à comprendre
- les programmes récursifs sont légèrement moins efficaces, mais il est toujours possible de dérécursivier un programme
- preuve de correction : par récurrence
preuve de terminaison : paramètre = suite strictement monotone