Programmation Fonctionnelle
Récurer du sol au plafond

Gilles Enée – LS4
Université des Antilles Guyane

Un peu de complexité

- Quelle est la complexité de (fac n) ?

```
(define (fac n)
  (if (zero? n)
    1
    (* n (fac (- n 1)))))
```

Un peu de complexité

- Si on choisit come granularité une opération entre deux nombres : C'est-à-dire qu'on considère qu'une opération entre deux nombres quelconques coûte 1 temps de calcul.
  - On note C_n la complexité de n!
  - Alors
    - $C_n = C_{n-1}$ + <Le coût de multiplier n par (n-1)!>
    - $\Leftrightarrow C_n = C_{n-1} + 1$
    - Sachant que $C_0 = 0$, nous pouvons en déduire que :
    - $C_n = 0+1+1+ \ldots + 1$ (n fois)
    - $\Leftrightarrow C_n = \theta(n)$

  On parle de complexité linéaire.

Un peu de complexité

- Observons les résultats obtenus sur la machine :
  - (time (fac 4000)) => 32
  - (time (fac 8000)) => 156
  - (time (fac 16000)) => 671
  - (time (fac 32000)) => 3015
- Est-ce que la progression est linéaire ?

Un peu de complexité

- Pourquoi ?
  - Parce que la granularité réelle est la multiplication de deux chiffres et pas de deux nombres !
  - Le nombre de multiplication de deux chiffres dans le produit de n-1! par n est de l'ordre de $\theta(n*\log^2(n))$. D'où :
    - $C_n = C_{n-1} + n \log^2(n)$
  - Finalement, on trouve grâce à la somme de Riemann C_n, que la complexité est de l'ordre :

    $\theta(n^2 \log^2(n))$

Un peu de complexité

- La théorie est-elle sauve ?
  - Passage de n = 4000 à n = 8000 :
    - x 4.6965 selon la théorie
    - x 4.875 selon nos mesures !
  - Passage de n = 8000 à n = 16000 :
    - x 4.6408 selon la théorie
    - x 4.3013 selon nos mesures…
  - Passage de n = 16000 à n = 32000 :
    - x 4.5933 selon la théorie
    - x 4.4933 selon nos mesures.

Peano

- Imaginez que Scheme ne possède comme primitives arithmétiques dans N que :

  (zero? n) (add1 n) (sub1 n)
- Interdiction donc d'utiliser les opérations + - * / quotient, etc… ni les prédicats = < <= etc…

Peano

- Définissons l'addition générale dans NxN de la manière suivante, par récurrence sur x :

```
(define (add x y)          ; x et y dans N
   (if (zero? x)
       y                   ; 0 + x = x
       (add1 (add (sub1 x) y))))
                           ; x+y = [(x-1)+y]+1
```

- Est-ce une itération ?

Peano

- Donnez une version itérative (addit x y).
- Mesurons la complexité de la fonction récursive.
  - Choisissons comme granularité les opérations de base add1, sub1. (i.e. ces opérations coûte « 1 »)
  - Soit $C_{x,y}$ le coût de (add x y)
  - On se propose de chercher un équivalent de cette quantité $C_{x,y}$

Peano

- En relisant l'addition, on déduit les équations suivantes :
  - $C_{0,y} = 0$
  - $C_{x,y} = C_{x-1,y} + 2$
- On en déduit aisément que Cx,y est proportionnel à x et indépendant de y
  - $C_{x,y} = kx$.
- La complexité reflète souvent le temps de calcul, mais dépend de l'unité de mesure.

Généralisez

- En passant à « l'ordre supérieur » !
- Les mathématiciens ont commencé à démontrer des théorèmes de géométrie en dimension 2, puis en dimension 3, puis… en dimension n.
- La généralisation étant une activité importante de la démarche scientifique, pourquoi ne pas s'en servir en programmation ?

Généralisez

- Souvent, il vous arrivera d'écrire un algorithme adapté à un problème particulier et de vous apercevoir qu'à peu de frais, il vous est possible d'en déduire un algorithme plus général qui resservira par la suite dans d'autres situations.
- Autre intérêt de la généralisation : résoudre des problèmes particuliers « difficiles ». C'est peut-être paradoxal mais il est souvent plus facile de résoudre des problèmes généraux que particuliers !

Généralisez

- Prenons l'exemple du calcul de la factorielle (fac n). Nous l'avions généralisé en (interfac a b), si dans le TD3 ! Et bien continuons cette généralisation. Pourquoi se borner à ne faire que des multiplications ?

Généralisez

- Remplaçons celle-ci par une opération quelconque binaire associative op de « neutre » e :

```
(define (accumulation op e a b)
    (if (> a b)
        e
        (op a (accumulation op e (+ a 1) b))))
```

Généralisez

- Exemple d'utilisation :

> (accumulation * 1 3 6) ; le produit 3*4*5*6
360
> (accumulation + 0 1 10) ; la somme de [1,10]
55

- Quelle est la définition de la factorielle en utilisant cette fonction ?

Généralisez

- Cette généralisation est insuffisante pour calculer la somme des carrés par exemple. Introduisez la fonction f (accumulation op e f a b) en paramètre qui sera appliquée à chaque élément de [a,b] rencontré :

> (accumulation + 0 (lambda (x) (* x x)) 100 200) ; la somme $100^2 + 101^2 + \ldots 200^2$

2358350

Généralisez

- Pourquoi nous limiter à aller de 1 en 1 ? Pourquoi pas de 2 en 2 ou de 5 en 5 ? Et pourquoi un pas constant ?
- Servons nous d'une fonction qui gère le pas (suiv a) qui calculera le suivant de a.
- Ecrivez la nouvelle version plus générale !

(accumulation op e f a suiv b)

La continuation (style CPS)

- Vous connaissez tous la factorielle maintenant.
- Soit la fonction auxiliaire suivante k-fac à deux paramètres :

  (k-fac n f) == (f (fac n))

  - Montrer que fac se définit à partir de k-fac.
  - En portant le texte de fac dans l'équation précédente, montrez que pour n non nul, on a :

    (k-fac n f) == (k-fac (- n 1) g)

  Où g est une fonction que l'on précisera.

  - En déduire une définition intrinsèque de k-fac.

La continuation (style CPS)

- (k-fac 5 id)

120

- (k-fac 5 add1)

121

- On lit cette dernière ligne : Pour calculer n! et continuer par f, je calcule n-1! Et je continue en multipliant par n et enfin en appliquant f''. On dit que le paramètre f est la continuation courante du calcul.
- Quel est l'intérêt de cette transformation de programme ?

La continuation (style CPS)

- Autre exemple avec Peano :

```
(define (mul a b)
    (if (zero? a)
            b
            (add b (mul (sub1 a) b))))
; a*b = b+[(a-1) * b]

(define (mul a b)
    (define (k-mul a b f)
            (if (zero? A)
                    (f b)
                    (k-mul (- a 1) b (lambda (r) (f (add b r))))))
    (k-mul a b id))
```

La continuation (style CPS)

- On peut ainsi obtenir des fonctions à plusieurs résultats sans utiliser de « structures ».
- Prenons comme exemple, le calcul des coeffs de Bezout de a et b entiers naturels. On cherche à calculer en même temps le pgcd g et deux entiers u et v tels que g = au + bv. On va alors utiliser des continuations à plusieurs variables !
- On raisonne par récurrence come dans le pgcd. On cherche un triplet de Bezout (G,U,V) pour G=g=b*u+r*v=b*u+(a-bq)*v=a*v+b*(y-q*v) d'où G=g, U=v et V=u-q*v :

La continuation (style CPS)

```
(define (bezout a b f) ; calcule (f g u v)
   (if (zero? b)
       (f a 1 0)
       (bezout b (modulo a b)
               (lambda (g u v)
                       (f g v (- u (* (quotient a b) v)))))))
> (bezout 13 5 (lambda (g u v) g))
1
> (bezout 13 5 list)
(1 2 -5)
```