Programmation applicative - TP4

G.Artignan {artignan@lirmm.fr} M.Lafourcade {lafourcade@lirmm.fr}
S.Daudé {sylvain.daude@univ-montp2.fr} A.Chateau {chateau@lirmm.fr}
B.Paiva Lima da Silva {bplsilva@gmail.com} C.Dony {dony@lirmm.fr}

1 Généralisation

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 ? 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 pourra être réutilisé par la suite dans d'autre situations.

Autre intérêt de la généralisation : résoudre des problèmes particuliers. C'est peut être surprenant, mais il est souvent plus facile [en programmation comme en maths] de résoudre des problèmes généraux que des problèmes particuliers !

Essayons de généraliser la fonction factorielle. Pourquoi se borner à ne faire que des multiplications ? Remplaçons celle-ci par une opération binaire associative et commutative `combiner` d'élément neutre `null-value` tel que `(combiner x null-value)` = `(combiner null-value x)` = valeur de `x`. La fonction d'accumulation est :

```
(define (accumulate combiner null-value a b)
  (if (> a b)
      null-value
      (combiner a (accumulate combiner null-value (+ a 1) b))))
```

```
> (accumulate * 1 10 20)      ; calcule le produit: 10 × 11 × ··· × 20
⇒ 6704425728000
```

Exercice 1

1. *Écrivez une définition de la fonction factorielle* `fact-accu` *qui utilise cette fonction d'accumulation.*
2. *Écrivez une définition de la fonction* `somme-accu` *qui à entier n associe la somme des entiers de 1 à n, en utilisant cette fonction d'accumulation.*
3. *Écrivez une définition de la fonction* `liste-accu` *qui à entier n associe la liste des entiers de 1 à n, en utilisant cette fonction d'accumulation.*

Exercice 2 *Cette généralisation est insuffisante pour calculer une somme de carrés par exemple. On veut pouvoir indiquer le terme de la fonction à "accumuler".*

1. *Introduisez une fonction* `term` *en paramètre qui s'appliquera à chaque élément de l'intervalle* $[a, b]$*, et écrivez la fonction plus générale*

`(accumulate-term combiner null-value term a b)` *telle que :*

```
> (accumulate-term + 0 square 10 20) ; calcule la somme: 10^2 + 11^2 + ··· + 20^2
⇒ 2585
```

2. *Redéfinissez les fonctions* `fact-accu, somme-accu, liste-accu` *avec cette nouvelle version de la fonction d'accumulation.*
3. *Calculez une valeur approchée de la constante* $e = 2.718\cdots$ *en exprimant le développement limité* $1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{15!} \cdot + \frac{1}{n!} \cdots$. *Comparez avec la valeur de* `(exp 1)` *où* `exp` *est prédéfini en Scheme. Pour quelle valeur de* n *approche-t-on* e *avec une précision inférieure à* 10^{-11} *?*
4. *Calculez le produit de tous les nombres impairs de* $[1, 50]$.
5. *Calculez le produit de tous les nombres pairs de* $[1, 50]$.
6. *Calculez le produit de tous les nombres égaux à 1 modulo 3 de* $[1, 50]$.

Exercice 3 *La question précédente soulève la question : pourquoi se limiter à aller de 1 en 1 dans les appels de la fonction* `accumulate` *? Nous pourrions parcourir l'intervalle* $[a, b]$ *2 en 2, ou de 5 en 5, etc. Et d'ailleurs, pourquoi avec un pas constant ? Utilisons plutôt une fonction* `next` *pour calculer le suivant d'un nombre dans l'intervalle.*

1. *Écrivez* `(accumulate-term-next combiner null-value term a next b)` *plus générale encore où* `next` *est une fonction qui à partir de a calcule le suivant de a dans l'intervalle* $[a, b]$.
2. *Redéfinissez les fonctions* `fact-accu, somme-accu, liste-accu` *avec cette nouvelle version de la fonction d'accumulation.*
3. *Calculez une valeur approchée de la somme* $S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \cdots$. *Pour une grande valeur devinez-vous vers quoi tend la limite ?*

Exercice 4 *Enfin, toujours à l'aide de la fonction d'accumulation, écrivez une fonction* `pi-product` *qui calcule le produit :* $\frac{2\times2\times4\times4\times6\times6\times8\cdots\times2n\times2n}{1\times3\times3\times5\times5\times7\times7\cdots\times(2n-1)\times(2n+1)}$ *qui converge vers* $\frac{\pi}{2}$. *Précisez pour quelle valeur de* n *la convergence s'effectue avec une précision inférieure à* 10^{-3}. *Comparer avec l'approximation de* e *faite précédemment. Remarque : pour avoir accès à la valeur de* π, *on peut avoir besoin d'inclure la librairie mathématique avec la commande* `(require (lib "math.ss"))`.

Exercice 5 *Tout ca pour ca : écrivez la fonction d'accumulation la plus générale en forme récursive terminale.*

Exercice 6 *Redéfinir les fonctions* `fact-accu, somme-accu, liste-accu` *en forme récursive terminale avec cette fonction d'accumulation générale récursive terminale. Comparer à l'aide de* `time` *les temps d'exécutions entre la forme itérative et la forme récursive pour ces trois fonctions.*