ENSEIRB-Matmeca
2019-2020

1°re année - Filière Info
PG104

TD n°3 - Récursivité terminale

Exercice 1: Récursivité terminale

1. Écrire une fonction `plus` récursive terminale, qui appelée avec deux entiers a et b comme arguments, effectue la somme de a et b en utilisant l'idée que pour ajouter b, on additionne "b fois" la valeur 1.
2. Écrire une fonction `produit` récursive terminale, qui appelée avec deux entiers a et b comme arguments, effectue le produit de a par b en utilisant l'idée que $a \times b$ revient à faire $a + a + \ldots + a + a$ (b fois).

Pensez à tester vos fonctions, notamment avec des valeurs recouvrant l'ensemble des comportements possibles de vos fonctions.

Exercice 2: Suite de Fibonacci

Dans cet exercice, on se propose de calculer les valeurs de la suite de Fibonacci, définie par la récurrence classique :

- `fibo`$(0) = 1$ et `fibo`$(1) = 1$;
- `fibo`$(n) =$ `fibo`$(n-1) +$ `fibo`$(n-2)$.

1. Méthode 1 : Utiliser une récurrence double, à savoir deux appels récursifs à l'intérieur de la fonction.
   Quel est l'inconvénient majeur de cette méthode ?
2. Méthode 2 : Utiliser une fonction annexe pour réaliser le calcul avec une simple récurrence (cette fonction annexe effectue le calcul d'une suite de Fibonacci généralisée).

   Construire la fonction `fibo-gen` qui prend 3 arguments n, a et b et calculant le n-ème nombre de la suite de Fibonacci commençant avec les valeurs a et b.

   - `fibo-gen`$(0, a, b) = a$ et `fibo-gen`$(1, a, b) = b$;
   - `fibo-gen`$(n, a, b) =$ `fibo-gen`$(n-1, a, b) +$ `fibo-gen`$(n-2, a, b)$.

   Simplifier cette fonction de manière à ce qu'elle n'effectue plus qu'une simple récurrence.
5. Tester les deux versions avec la valeur 32. Que constatez-vous ? Pour confirmer cette impression, on peut utiliser la fonction `time` comme ceci :

```
(time (fibo 32))
(time (fibo-gen 32 1 1))
(time (fibo-gen 1000 1 1))
```

   Que pouvez-vous dire ?

6. Pour aller plus loin... Discuter / estimer la complexité de la Méthode 1 (double récurrence).

Exercice 3: Comparaison de `let` et `let*`

1. Formes `let` : déterminer les valeurs des formes suivantes :

```
(let ([x 5]
      [y (sqrt 9)]
      [z (* 4 (+ 3 (sqrt 4) (sqrt 9)))])
  (+ x y (* 5 z)))
```

```
(+ (let ([a 12]) (+ a a))
   (let ([b 30]) (+ b
                    (let ([b 20])
                      (+ b b)))))
```

2. Reprendre l'exercice sur les racines de trinôme de la feuille 1, et proposer une version utilisant des variables locales.
3. Considérer la session suivante ; déterminer les valeurs de `q1` et `q2` :

```
(define x 12)
(define y 5)
(define q1 (let ([x y]
                 [y x])
             (- x y)))
(define q2 (let* ([x y]
                  [y x])
             (- x y)))
```

4. Réécrire le code utilisant `let*` en n'utilisant que des instructions `let`.

Exercice 4: Formes spéciales

L'évaluation de fonctions en Scheme (à l'exception de l'évaluation des formes spéciales) se fait de la manière suivante : les arguments sont d'abord tous évalués dans un ordre quelconque puis la fonction leur est appliquée. Les expressions conditionnelles sont des formes spéciales et sont évaluées différemment. Les manipulations suivantes ont pour but d'expliquer comment se fait leur évaluation.

1. Écrire une fonction `new-if` à trois arguments `predicate`, `clause-then` et `clause-else` et faisant appel à la forme `if` avec les mêmes arguments pour réaliser un test.
2. Essayer la session suivante (la fonction `print` effectue un affichage à l'écran) :

```
(define a 0)
(new-if (zero? a) true false)
(new-if (zero? a)
        (print "a␣est␣nul")
        (print "a␣est␣non␣nul"))
```

Que constatez-vous ? Expliquer le mécanisme conduisant à ce résultat.

3. Écrire la fonction `new-factorial` en utilisant la fonction factorielle dans laquelle vous remplacez l'appel à `if` par un appel à `new-if`. Évaluez l'appel à `(new-factorial 3)`.
   Que constatez-vous ? Expliquer le mécanisme conduisant à cette erreur.

Exercice 5: Évaluation des expressions booléennes

Pour chacune des expressions conditionnelles indiquées ci-dessous, imaginer ce qui sera affiché à l'écran. Vérifier en évaluant cette expression au top-level puis en déduire les processus d'évaluation de `(and arg1 ... argn)` et `(or arg1 ... argn)`.

```
(define a 20)
(and (print 1) (= a 20) (print 2) (print 3))
(and (print 1) (= a 30) (print 2) (print 3))
(or  (print 1) (= a 20) (print 2) (print 3))
(or  (= a 30)  (= a 50) (print 1) (print 2))
```

Pour aller plus loin...

Exercice 6: Chous-chaînes

Étant donné une chaîne de caractères `s` et un prédicat `p` sur les chaînes de caractères (une fonction prenant en paramètre une chaîne et renvoyant un booléen), écrire une fonction calculant le nombre de sous-chaînes de `a` vérifiant `p`.

La solution doit être écrite de manière récursive terminale.

Exemple d'utilisation :

```
(substrings "abbbababbabbbabab" (lambda (x) (equal? x "ab")))           ;; → 6   (equal to "ab")
(substrings "abbbababbabbbabab" (lambda (x) (string-contains? x "bb"))) ;; → 112 (containing "bb")
```